Empirical Software Engineering Journal manuscript No.
(will be inserted by the editor)

Evaluating refactorings for disciplining `#ifdef` annotations: An eye tracking study with novices

José Aldo Silva da Costa · Rohit Gheyi · Márcio Ribeiro · Sven Apel · Vander Alves · Baldoino Fonseca · Flávio Medeiros · Alessandro Garcia

Abstract The C preprocessor is widely used in practice. Conditional compilation with `#ifdef` annotations allows developers to flexibly introduce variability in their programs. Developers can use disciplined annotations, entirely enclosing full statements with preprocessor directives, or undisciplined ones, enclosing only parts of the statements. Despite some debate, there is no consensus on whether a developer should use exclusively disciplined annotations. While one prior study found undisciplined annotations more time-consuming and error-prone, another study found no difference between disciplined and undisciplined annotations regarding task completion time and accuracy. In this article, we evaluate whether three fine-grained refactorings to discipline `#ifdef` annotations correlate with improvements in code comprehension and visual effort with an eye tracker. We conduct a controlled experiment with 64 human subjects who were majoritarily novices in the C programming language. We observed statistically significant differences for two refactorings to

J.A. Silva da Costa and R. Gheyi
Federal University of Campina Grande
E-mail: josealdo@copin.ufcg.edu.br, rohit@dsc.ufcg.edu.br

M. Ribeiro and B. Fonseca
Federal University of Alagoas
E-mail: marcio@ic.ufal.br, baldoino@ic.ufal.br

S. Apel
Saarland University, Saarland Informatics Campus
E-mail: apel@cs.uni-saarland.de

V. Alves
University of Brasília
E-mail: valves@unb.br

F. Medeiros
Federal Institute of Alagoas
E-mail: flavio.medeiros@ifal.edu.br

A. Garcia
Pontifical Catholic University of Rio de Janeiro
E-mail: afgarcia@inf.puc-rio.br

discipline annotations with respect to the analyzed metrics (time, fixation duration, fixation count, and regressions count) in the code regions changed by each refactoring.

Keywords refactoring · `#ifdefs` · eye tracking · code comprehension · disciplined annotations · undisciplined annotations

1 Introduction

The C preprocessor is widely used in practice, such as in *Linux* [15]. It provides mechanisms to implement variability through conditional compilation [15]. Conditional compilation allows developers to conditionally include selected blocks of source code by annotating the code using directives, such as `#ifdefs`. There are two types of annotations, *undisciplined* (or incomplete), and *disciplined* (or complete) [25]. Although both achieve the same purpose, they differ in terms of whether they align with the syntactic structure of the code. Disciplined annotations enclose only whole syntactical units while undisciplined annotations do not, for example, wrapping only an opening bracket of a statement but not the closing one [18]. Even though no study has formally specified undisciplined annotations, we aligned our definition with previous studies [25,15,37]. Relying on this definition, Liebig et al. [25] 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 [45,26,28,27]. Code with either disciplined or undisciplined annotations may affect program comprehension. However, empirical knowledge on the influence of annotation discipline is still scarce. Medeiros et al. [28] proposed a catalogue 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 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 [26,37]. For instance, Malaquias et al. [26] 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. [37] found no differences between using disciplined and undisciplined annotations regarding task completion time and accuracy. Fenske et al. [16] conducted a survey study with 521 developers regarding annotations in the C language and found that their perception and their performance are different.

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., it is attached to one or a few statements). Opinions and conventional measures may not reveal important nuances on the comprehension of disciplined versus undisciplined annotated code, which may also help to better explain the benefits and drawbacks of annotation discipline. As a consequence, 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.

This article reports an eye tracking study that evaluates whether and how three fine-grained refactorings to discipline `#ifdef` annotations affect code comprehension. We conduct a controlled experiment with 64 human subjects majoritarily novices. We consider all the participants 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 [26,37]. Fixation duration, fixation count, and regressions count have been associated before with visual attention and effort in code comprehension scenario [8,4,39]. We measure these metrics in the code region in which both code versions differ after applying the refactorings, referred to as main Area of Interest (AOI). For this study, we selected the three refactorings most preferred by developers to discipline annotations according to Medeiros et al. [28]. The three refactorings differ in various ways: Refactoring 1 ⟨wrapping function call⟩ (R1) duplicates a token in a function call to wrap only entire statements with preprocessor directives. Refactoring 2 ⟨undisciplined `if` conditions⟩ (R2) resolves undisciplined directives surrounding boolean expressions by defining a fresh variable to maintain the statement's conditions. Refactoring 3 ⟨alternative `if` statements⟩ (R3) uses an alternative `if` statement also defining a fresh variable to keep the statement's condition. We explain these differences with examples in Section 4.5.

In our study, we found that after applying R1 or R3, the total time spent in the AOI, fixation duration, fixation count, and regressions count were statistically significantly reduced. After applying R3, also the number of answers submitted to solve the tasks statistically significantly reduced. 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 Lines of Code (LOC), but it correlated 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 R1 or R3, she solves the task faster and with less visual effort. In addition, applying R3 correlated with improvements in the accuracy of her answers.

In summary, this study makes the following key contributions:

- We present a controlled experiment using eye tracking with 64 novices in the C programming language to evaluate three refactorings that discipline `#ifdef` annotations in C programs.
- Moreover, we discuss methodological perspectives and findings not observed in previous studies.

This article is organized as follows: Section 2 provides a motivating example. Section 3 presents the study definition, then Section 4 presents the study methodology. Section 5 presents the obtained results, and Section 6 discusses a qualitative interview with the novices. Section 7 discusses the threats to validity, Section 8 relates our work to others, and finally, Section 9 concludes the study.

2 Motivating Example

The C preprocessor is often used by developers to deal with code portability and variability. Preprocessor directives such as `#ifdef` and `#endif` annotations are used as a mechanism for conditional compilation so that a piece of software can operate in different application scenarios [27]. Annotations with preprocessor directives can be disciplined or undisciplined.

For instance, Figure 1(a) shows a code snippet containing an undisciplined annotation. The annotation starts at Line 1 and wraps only the `if` statement in Lines 2 and 4 without their opening and closing brackets. Figure 1(b) presents the same code snippet but using a disciplined annotation. Liebig et al. [25] 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 [28,18]. For instance, there is a refactoring [28] that allows us to convert the code snippet presented in Figure 1(a) to Figure 1(b).

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 Figure 1 have almost the same values for code metrics such as LOCs. Malaquias et al. [26] recommend avoiding undisciplined annotations because they are more time-consuming and error-prone. Schulze et al. [37] conclude that there is no difference between disciplined and undisciplined annotations regarding time and accuracy. Fenske et al. [16] conclude that there is a difference between developers' perception and performance regarding annotations in C language.

To shed light on this issue, we aim to use an eye tracking camera to analyze how disciplining annotations impacts code comprehension using a different perspective not considered in other studies [26,37,18]. An eye tracking camera

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

(a) With undisciplined annotation

```
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) With disciplined annotation

Fig. 1: Code snippets adapted from *OpenSSL* with undisciplined and disciplined annotations.

consists of equipment tracking participants' eyes movements while they are performing a task [20]. It has been used in a variety of fields including source code reading and comprehension [14,7,48]. The data captured by the camera allow researchers to access where, when, and for how long a subject is looking at a screen, thus, the most common metrics evaluate how much subjects fixate and how they switch between distinct areas [32,7,39]. Researchers can then infer how much visual attention is given to specific elements on a screen and thus study the visual effort in code comprehension [14,13,4]. This way, in addition to time and accuracy, in our study, we investigate the impact of these annotations on the visual effort by measuring the fixation duration, fixation count, and regressions count.

3 Study Definition

In this section, we present the study definition following the Goal-Question-Metrics approach [2]. We analyze three refactorings for C programs that discipline `#ifdef` annotations for the purpose of understanding whether disciplined annotations correlate 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 total time duration novices need to solve a "specify the correct output" task with three evaluated refactorings. In addition, we measure the time the participants 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 task accuracy?** 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 number of answer submissions.

- **RQ_3: To what extent do disciplined annotations affect visual effort?** In the code domain, longer fixations have been associated with substantial increase in demands of attentiveness [8]. Crosby et al. [13] have shown that there is a correspondence between fixations and attention focus, suggesting validity of immediacy and eye mind theory, also in code domain. The results of those studies imply that longer fixations indicate more attention and consequently more visual effort. 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 associates with cognitive and visual effort. For instance, higher number of fixation indicates longer processing time to understand code phrases [4], more attention to complex code [13], and more visual effort to recall the name of identifiers [41]. Moreover, regarding eye tracking metrics, we may have backward eye movements of any length over the stimuli called regressions [7]. According to Sharafi et al. [39], regressions can be used to measure visual effort. The linearity of natural language and code reading have been measured before using the regression rate in code domain [7].
  - **$RQ_{3.1}$: To what extent do disciplined annotations affect fixation duration?** To answer this question, we measure the fixation duration of the novices while solving the task with three evaluated refactorings. Our null hypothesis ($H_{3.1}$) is that there is no difference between disciplined and undisciplined annotations regarding fixation duration.
  - **$RQ_{3.2}$: To what extent do disciplined annotations affect fixations count?** To answer this question, we measure the fixation count of the novices while solving the task with three evaluated refactorings. Our null hypothesis ($H_{3.2}$) is that there is no difference between disciplined and undisciplined annotations regarding fixation count.
  - **$RQ_{3.3}$: To what extent do disciplined annotations affect regressions count?** 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 ($H_{3.3}$) is that there is no difference between disciplined and undisciplined annotations regarding regressions count.

4 Methodology

In this section, we present the methodology of our study. We present the pilot study (Section 4.1), experiment phases (Section 4.2), participants (Section 4.3), treatments (Section 4.4), evaluated refactorings (Section 4.5), tasks

(Section 4.6), eye tracking instrumentation (Section 4.7), and finally the analysis (Section 4.8).

4.1 Pilot Study

Before conducting the actual experiment (see Section 4.2), we conducted a pilot study with four participants 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 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 participant to complete all phases. Next, we describe these phases in detail.

4.2 Experiment Phases

First, we chose a quiet room to minimize distractions and with typical indoor fluorescent bulbs for the experiment. As the participants 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 participants 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 participant, but we did not encourage conversation while the participant was performing a task. Second, we presented a tutorial on variability implementation explaining how `#ifdefs` 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 participants.

Third, we illustrated the nature of the experiment through a simple warm-up task in which we asked the participants 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 participants got comfortable with the equipment and the study setup. We asked the participant to close 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 participants to verbally provide the output of the code, which is an approach adopted by other studies as well [42, 43, 21]. We provided real-time feedback by emitting a distinct sound corresponding to whether the answer was correct. If the answer was incorrect, participants could choose to keep

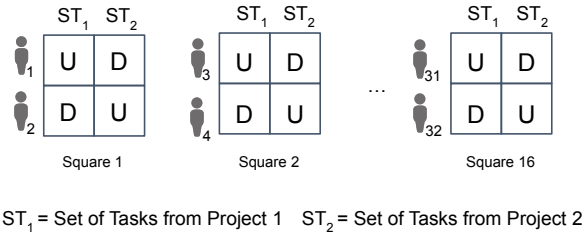


Fig. 2: Design of experiment with Latin Squares for first phase with 32 participants with projects P1 and P2. U and D refer to undisciplined and disciplined annotation tasks, respectively. The second phase follows the same design with other 32 participants and projects P3 and P4.

trying submitting 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 [6] to ensure that every participant was exposed to each treatment only once and to ensure that the same participant answered the same task only once, avoiding learning effect. Thus, we randomly assigned participants to treatments in the cells of each square as depicted in Figure 2. The comparison in further analysis occurs across the squares by gathering all participants who answered the same task.

Fifth, once a participant finished all the tasks, we conducted a semi-structured interview to obtain qualitative feedback on how they approached the tasks. We asked the participants 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 participants to point out in the task 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 were capable of getting the data corrected. Camera calibration consists of an automatic procedure in which the participant is asked to look at specific locations on the screen and, during that, the camera's integrated system customizes captured data according to each participant's eye characteristics. The camera indicates when calibration is successfully done.

In addition, we were careful with environment aspects and the swivel function of the chair, so that participants' 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 participants' aspects. Thus, we had to perform data correction by slightly shifting chunks of fixations up or down. We discuss this strategy and their effects in the threat to validity section (see Section 7.1). We provide a replication package with the data collected, tasks, and other materials [1].

4.3 Participants

We performed the study with 64 participants divided in two phases of 32 participants each. In total, we had 42 undergraduates, 11 MSc. students, 8 PhD. students, and 3 postdocs. Regarding experience with programming languages, 40 participants 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 participants who know how to program but have little experience specifically with C programming language, which corresponds to all participants in the study except for 10. All participants 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. Participants were invited mainly through e-mails and text messages that suggested them to respond by communicating their availability. In addition, we also met some participants in person and invited them.

4.4 Treatments

We expose each participant to three disciplined (D) and three undisciplined (U) annotated tasks as seen in Figure 3, which results in six tasks (T1-T6) for each participant. The same participant does not solve the same disciplined and undisciplined annotated task to avoid learning effect. For that, we have two projects (P1 and P2) which 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. Figure 3 depicts the first phase. The second phase is similar with the same refactorings but two other projects. We consider this study as between-subjects in the sense that the same participant does not solve the same task in both disciplined and undisciplined annotations [10]. In Figure 4, we present an example of the distribution of the participants, tasks, and refactorings according to the projects.

In the first phase, 32 participants solve three tasks with disciplined annotations from three distinct refactorings of P1, and three tasks with undisciplined

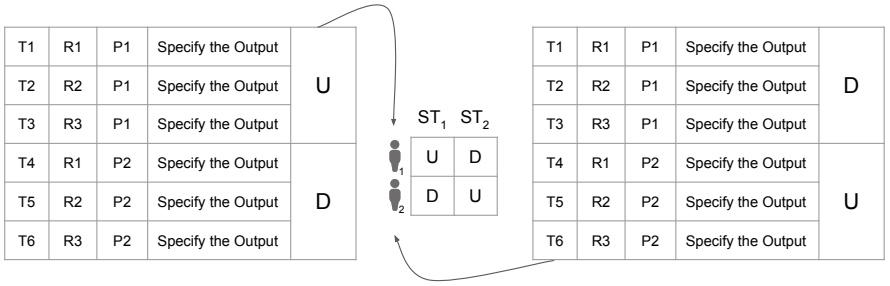


Fig. 3: Structure of the experiment in terms of experimental units for the first phase of the study. There are six tasks (T1–T6) distributed in two sets of tasks (ST_1 and ST_2), with R1 ⟨wrapping function call⟩, R2 ⟨undisciplined `if` conditions⟩, and R3 ⟨alternative `if` statements⟩, and two projects (P1 and P2).

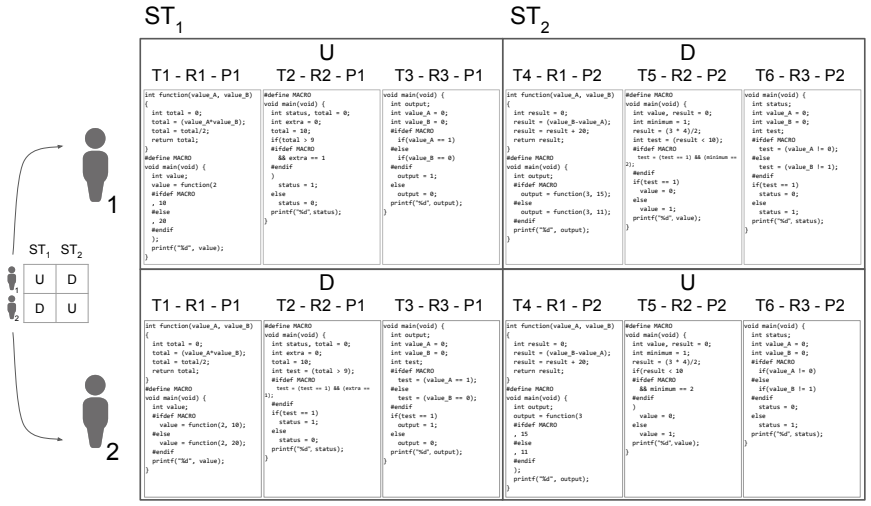


Fig. 4: Distribution of participants, tasks, and refactorings in two projects in the first phase of the study. The structure of the tasks in projects P1 and P2, before and after applying the refactoring, is similar but involves distinct elements.

annotations from P2. Thus, each participant solves two distinct tasks, one without applying the refactoring (undisciplined version), and another with the refactoring applied (disciplined version). In the second phase, other 32 participants solve six tasks with three refactorings, three with undisciplined and three with disciplined annotations, but from P3 and P4. In all them, the participants had the task of specifying the correct output. We present an open-ended question so that the participant could read the entire code and find the

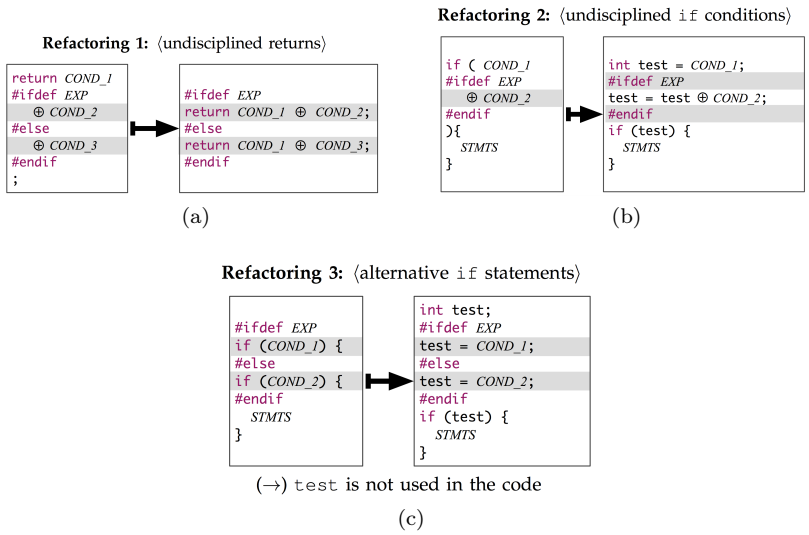


Fig. 5: Refactorings R1, R2, and R3 to discipline `#ifdef` annotations evaluated in this study.

output for themselves. The undisciplined versions are our baseline group, and the disciplined ones are the treatment group.

4.5 Evaluated Refactorings

In Figure 5, we present three refactorings to discipline annotations proposed by Medeiros et al. [28] 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 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.

Medeiros et al. [28] surveyed 246 developers to access their preference regarding disciplined or undisciplined annotated code. We selected the top three refactorings which developers most preferred to discipline annotations. Moreover, Medeiros et al. [28] 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. [28] 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. [26] 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.

4.6 Tasks

The code snippets were selected as a result of mining code repositories for commits that showed an opportunity to apply the refactorings evaluated [28]. Thus, all tasks have a template associated with real projects. For instance, R1 was applied to a task with template associated with *Vim's* source code, R2 was applied to a task with template associated with *Libpng's* source code, and R3 was applied to a task with template associated with *OpenSSL*. Malaquias et al. [26] 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 same number of lines of code. Even though R1 in Figure 5(a) involves undisciplined returns, according to Medeiros et al. [28], 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 participants 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. [33] 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 [33], "understanding" consists of one level of the dimension of interpretation. Most activities such as code trace and inspections performed by subjects occur in "understanding" level followed

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 - P1 - 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 - P2 - 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 - P4 - Undisciplined to disciplined annotations

Fig. 6: Examples of six tasks from projects P1, P2, and P4, before and after applying R1, R2, and R3.

by "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 size of the screen. All the products of the configurable system could be compiled with no syntactic errors. We had tasks with macro enabled and disabled, and they presented distinct outputs depending on whether macro was enabled or disabled. However, 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 6, we present three undisciplined annotated tasks and their refactored versions. For instance, in Figure 6(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 participants 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 participants are looking at the opposite statement when macro is enabled.

4.7 Fixation Instrumentation

Fixations can be defined as the stabilization of the eye on part of a stimulus for a period of time [36, 22]. The duration threshold typically depends on the tasks processing demands. According to Salvucci and Goldberg [36], the duration threshold can be between 100 and 200 ms, while according to Rayner [35], our eyes remain relatively still during fixations for about 200–300 ms. Commonly applied in practice, we applied a Dispersion-Based algorithm to generate the fixations. Particularly, we used the Dispersion-Threshold Identification (I-DT) to classify gaze samples into fixations [36]. It classifies gaze samples as belonging to a fixation if the samples are located within a spatially region of approximately 0.5 degrees [31]. The I-DT algorithm requires two parameters: the dispersion threshold and the duration threshold [31]. 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 [36]. 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.

4.8 Analysis

Of all 64 participants, resulting in 384 tasks, one participant 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 completed, which corresponds to less than 1% of the total of tasks. We included those two participants 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 [23]. This approach performs better when the sample size is sufficiently large [24], 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 with Shapiro [38]. 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 [44, 40]. 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 [44]. 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 [44,40]. 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 [17], 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 [11] to yield the effect size. Since in most cases our data do not follow a normal distribution, Cohen's effect size would not be appropriate. So, we use Cliff's Delta. Cohen has made widely accepted suggestions on what constitutes small and large effects [12]. 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 on the treatment group (disciplined annotations) are greater than the control group (undisciplined annotations).

5 Results

In Sections 5.1–5.6, 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 5.7, we summarize the results for all research questions.

Table 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	10^{-5}	-0.44	**29.1**	**15.4**	↓46.9	10^{-5}	-0.44	**6.3**	**3.1**	↓51.1	$7x10^{-7}$	-0.49	**3.1**	**1.2**	↓59.4	$2x10^{-9}$	-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

5.1 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 1). We observed statistically significantly reductions by 23.8% and 46.9% in the time they spent in AOI, respectively. Thus, they spend less time in AOI in Figure 6(a) and Figure 6(c) after applying R1 or R3. After applying R3, we observed a statistically significant reduction by 42.4% in the time they spent in whole code. Applying R3 correlated with a reduction in the time novices spent in both activated and in deactivated areas by 51.1% and 59.4%, respectively. They spend less time in AOI Activated and AOI Deactivated areas in Figure 6(b), both right and left-hand sides. 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 6(b), right-hand side. It also correlated with a slowdown in their task completion by increasing the time they spent on whole code by 24.6% with R2 applied. Thus, applying R1 or R3 correlated with improvements in task completion time for novices. However, after applying R2, we cannot observe the same effect.

Combined, after applying R1, R2, and R3, novices exhibited faster task completion (see Table 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 also correlated with a reduction in the time they spent in the whole code by 12.5%. Thus, applying R1, R2, and R3 combined correlated 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 2: Summarizing the results for accuracy (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

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.

5.2 RQ_2: To what extent do disciplined annotations affect task accuracy?

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 7(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 2), the mean number of submissions decreased from 1.25 to 1.20. Thus, applying R3 correlated with improvements in the accuracy of the answers submitted by the novices. Combined, we did not find differences in accuracy 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.

5.3 $RQ_{3.1}$: 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 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 6(a) and Figure 6(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 correlated with a reduction in the fixation duration in the AOI for novices.

Table 3: Summarizing the results for duration of fixations ($RQ_{3.1}$). 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	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

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 correlated with a reduction in the duration of the fixations in the whole code by 20.8%. Thus, applying them combined correlated with a reduction in the fixation duration both in the AOI and in the whole code for novices.

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.

5.4 $RQ_{3.2}$: 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 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 6(a) and Figure 6(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 correlated 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 correlated with a reduction in the number of fixations in the whole code by 22.4%. Thus, applying them combined correlated with a reduction in the fixation count both in the AOI and in the whole code for novices.

Table 4: Summarizing the results for fixation count ($RQ_{3.2}$). 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	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

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.

5.5 $RQ_{3.3}$: 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, 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 transition 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). We observed statistically significant reductions by 33.3% and 60.5% in the number of the regressions in AOI after applying R1 and R3, respectively. It correlates with improvements in the number of regressions in the AOI in Figure 6(a) after applying R1, and in Figure 6(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 in the regressions count after applying R3. We observed a statistically significant reduction by 50% in the number of regressions. Thus, applying R3 correlated 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 correlated with a reduction in the number of regressions in the whole code by 24%. Thus, applying them combined correlated with alleviating the need of going back to the same or to

Table 5: Summarizing the results for regressions count ($RQ_{3.3}$). 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				
	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

previous lines of code in the whole code.

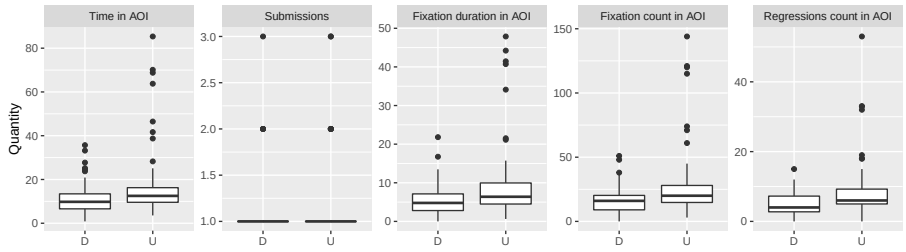
Finding 5: In our study, after applying R1 or R3 in isolation, the novices exhibit a reduction in the regressions count in the AOI. A reduction in the regressions count in the AOI is also exhibited by the novices after applying R1, R2, and R3 in combination.

Table 6: Summary of the null-hypotheses' statuses in isolated refactorings in the AOIs. The effect sizes are an approximation based on Cohen's description [12].

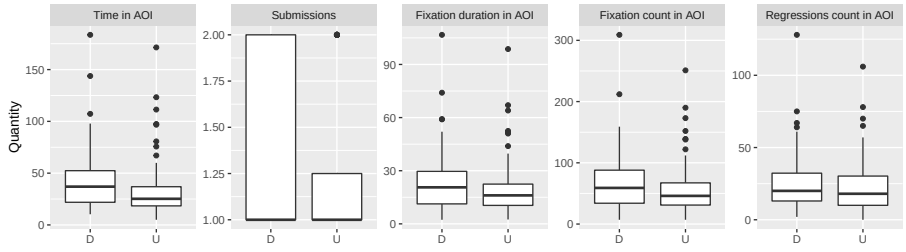
RQ	**Refact.**	**Null-Hypothesis**	***p*-value**	**Status**	**Effect size**
RQ_1	R1	No difference in time between treatments	< 0.05	Rejected	Small
RQ_2	R1	No difference in accuracy between treatments	> 0.05	Not Rejected	—
$RQ_{3.1}$	R1	No difference in fixation duration between treatments	< 0.05	Rejected	Small
$RQ_{3.2}$	R1	No difference in fixation count between treatments	< 0.05	Rejected	Small
$RQ_{3.3}$	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 accuracy between treatments	> 0.05	Not Rejected	—
$RQ_{3.1}$	R2	No difference in fixation duration between treatments	> 0.05	Not Rejected	—
$RQ_{3.2}$	R2	No difference in fixation count between treatments	> 0.05	Not Rejected	—
$RQ_{3.3}$	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 accuracy between treatments	< 0.05	Rejected	Small
$RQ_{3.1}$	R3	No difference in fixation duration between treatments	< 0.05	Rejected	Medium
$RQ_{3.2}$	R3	No difference in fixation count between treatments	< 0.05	Rejected	Medium
$RQ_{3.3}$	R3	No difference in regressions count between treatments	< 0.05	Rejected	Medium

5.6 RQ_3: To what extent do disciplined annotations affect visual effort?

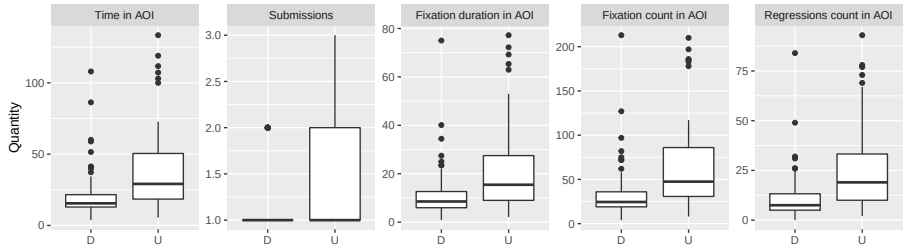
In our study, the visual effort has been measured by three eye tracking based metrics, namely, fixation duration, fixation count, and regressions count. Considering the results of the three research questions addressing each of them,



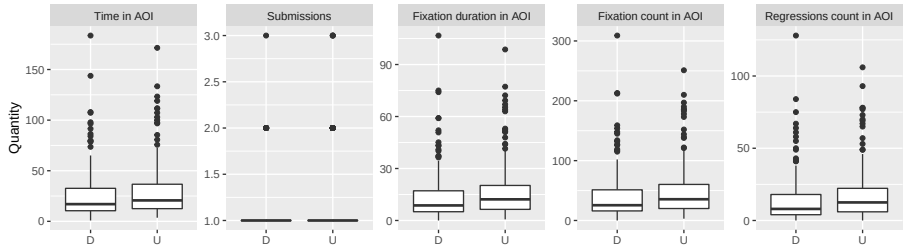
(a) Comparing Statistics for R1



(b) Comparing Statistics for R2



(c) Comparing Statistics for R3



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

Fig. 7: Comparison between: Disciplined (D) and Undisciplined (U) annotations for R1, R2, and R3 from an isolated and combined perspective involving P1–P4 together.

we conclude that after applying R1 or R3, the novices exhibit less visual effort in AOI. Less visual effort in AOI is also exhibited by novices after applying R1, R2, and R3 in combination. Thus, applying R1 or R3 correlated with alleviating the visual effort in the AOI for novices.

Finding 6: In our study, after applying R1 or R3 in isolation, the novices exhibit less visual effort in the AOI. Less visual effort in the AOI is also exhibited by the novices after applying R1, R2, and R3 in combination.

5.7 Summary

In Table 6, we present the confirmation/rejection of the original null-hypotheses. Figures 7(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 P1–P4 combined. Figure 7(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, $RQ_{3.1}$, $RQ_{3.2}$, $RQ_{3.3}$ 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 refactorings, 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 number of correct answers submitted. We did not observe differences in these metrics after R2 applied.

6 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 participants to describe 1) their approach used to solve the tasks, 2) their perception on how difficult were the tasks, and 3) what difficulties they had, if any. The participants 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, accuracy, fixation duration, fixation count and regressions count could be better explained through a triangulation of the data.

Based on Corbin and Strauss [46], we adopted the following approach to qualitatively code the interview: In Step 1, the first author played the interviewer role analyzing each whole sentence spoken by the participant 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 in 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.

Regarding our first question of the interview, we observed that the most common approach adopted consisted of first checking whether the macro was defined (37 participants). In addition, 13 participants mentioned looking at `#ifdefs` directives, and, by doing so, 6 participants 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 participants mentioned also first looking at the function that was on the top of the code, 8 participants mentioned starting to read the code from the beginning, and 9 participants mentioning reading the code in a top-down fashion. Furthermore, 9 participants mentioned a sequential reading pattern, whereas 7 participants mentioned looking at the end of the code, specifically to the output, and 2 participants mentioned a bottom-up fashion. Regardless of the order in which they were mentioned, participants mentioned looking at key parts such as variables (14 participants) and their assigned values (7 participants). 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 participants 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 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 participants), boolean expressions (11 participants), broken lines (9 participants), confusion regarding the interaction between commands from directives and language constructs (7 participants),

and confusion with `#ifdefs` specifically (5 participants). 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 participants 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 participants 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 6(a), which may have helped them to solve tasks faster. Similarly, the participants 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 6(c), which may have contributed to a 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, we observe that the data seem more scattered in the boxplot regarding number of submissions in Figure 7(c). Separating an `if` statement with `#ifdef` annotations and placing `else` close to the `#else` seemed to confuse the participants. Separating these terms by adding an alternative `if` statement, participants seemed less confused about the correct output.

According to $RQ_{3.1}$ and $RQ_{3.2}$, after applying R1 or R3, novices exhibited less attention in AOI and in the whole code by reducing both the number of fixations and their duration. These reductions imply less effort of jumping from one place to another in the code, which translates to less visual effort. For instance, participants 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 higher number of fixations. They also mentioned confusion with `else` and `#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.

7 Threats to Validity

We discuss potential threats to validity: internal validity (Section 7.1), external validity (Section 7.2), and construct validity (Section 7.3).

7.1 Internal Validity

The environment location may have influenced the participants' attention. Given 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 first author's presence may have unintentionally influenced the data because participants may have felt being observed. The author may also have influenced the participants to achieve certain outcomes. To mitigate these threats, the author minimized the interaction with the participants 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 equipment. For instance, in some cases, 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. In those specific 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 participants in order to verify whether it best matched their visual intent. In addition, two of the authors later on discussed this strategy. To minimize the threat of the equipment, we performed a correction of the eye tracking data in the aforementioned 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 participants 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 [1].

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

The duration of the experiment may have influenced the visual effort of the participants. The six tasks for each participant 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 participant spent on a task was 5 min and 2 seconds, and the median time for all the tasks of all participants was 36 seconds.

We gave the option to the participants 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 participant answers incorrectly in the first tentative, 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 in 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 [23]. Regarding the reliability of this method, PMM generally performed better when sample size was sufficiently large [24], 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 under 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 understand the effects of the evaluated refactorings.

7.2 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. [28] 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 participants 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 [9]. We had a total of 64 participants in our study, out of which only 10 were experienced participants. For our analysis and reported results, we did not filter out these 10 experienced participants. 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 participants to the results of considering only the 54 novice participants. We found that the results from both groups

of 64 (all participants) and 54 (only novices) participants are the same. In the future, we need to conduct further studies with more experienced participants 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 participants 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 participant 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 new features. The font size or font style may have influenced the participant'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 participants, 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.

7.3 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 [30,7,43,3]. For the purpose of investigating code comprehension, time and accuracy have been used in isolation [37,26] and in combination with visual effort [42]. The visual effort has been measured before by separate eye-tracking based metrics such as fixation duration and fixation count [42,4]. In addition to fixations-based metrics, regressions have been associated with visual effort [39].

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 participants about the precise goals of the study to avoid hypothesis guessing, but we informed that their eyes were being tracked, which may have influenced where or how much they have looked at some regions of the code.

8 Related Work

In this section, we provide an overview of the related work. In Section 8.1, we discuss in detail the related work that is closest to ours, and, in Section 8.2, we discuss in detail the related work that is second closest to ours and, in Section 8.3, we discuss the other relevant work.

8.1 Comparison with Medeiros et al. [28]

Medeiros et al. [28] conducted a survey with 246 experienced developers to access their perception on 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 participants 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 participants 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 participants 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.

Thus, our study and the one conducted by Medeiros et al. [28] are distinct in the following characteristics: research questions, experience of the developers, tasks used, empirical method, metrics, and threats to validity. These differences are summarized in Table 7. The differences shown in Table 7 may explain the differences in the conclusions. However, we need to conduct further studies to better understand the reasons for some differences.

Besides the survey, Medeiros et al. [28], 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 correlated with improvements in time and visual effort in our study.

8.2 Comparison with Fenske et al. [16]

Fenske et al. [16] 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 6). Their comprehension tasks are distinct from ours comprising larger snippets with more directives. In addition, multiple choices

Table 7: Summarizing the comparison between the study conducted by Medeiros et al. [28] and our study.

	Medeiros et al. [28]	**Our study**
Common RQs	—	—
Distinct RQs	What is the number of possibilities to apply the refactorings in practice? What opinion do developers have on the catalog of refactorings in practice? Do the refactorings of the catalog preserve program behavior?	To what extent do disciplined annotations affect task completion time? To what extent do disciplined annotations affect task accuracy? To what extent do disciplined annotations affect visual effort?
Common Findings	Developers prefer applying R1 Developers prefer applying R3	Applying R1 correlated with improvements in time and visual effort Applying R3 correlated with improvements in time, accuracy, and visual effort
Distinct Findings	Developers prefer applying R2	Applying R2 did not correlate 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 participants not being observed	Controlled experiment with participants 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

are presented to the participants. For instance, among multiple statements about the code, the participants had to select the correct one. Moreover, the participants 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 participants worked on code with refactored directives. However, on their perception, the refactored code was more comprehensible and easier to work. In contrast, we have presented smaller snippets with one directive to the participants. We

have configured the directive by enabling or disabling the macro. Then, we asked the participants an open-ended question regarding the correct output of the snippet. In addition, we have performed a controlled experiment using eye tracking with novices.

Thus, our study and the one conducted by Fenske et al. [28] 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 8 and may explain the differences in the conclusions. However, we need to conduct further studies to better understand the reasons for some differences.

Table 8: Summarizing the comparison between the study conducted by Fenske et al. [16] and our study.

	Fenske et al. [16]	**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

8.3 Comparison with other Related Work

Medeiros et al. [27] 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 to comprehend 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 on the accuracy after applying R3.

Schulze et al. [37] conducted a controlled experiment to analyze the effect of disciplined and undisciplined annotations on program comprehension. The participants were undergraduates 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 participant'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 correlated with improvements in time, fixation duration, fixation count, and regressions count.

Malaquias et al. [26] 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. [37]. For R1 or R3, the results of Malaquias et al. [26] align with ours in the sense that disciplined annotations correlate with improvements in task completion time. In addition, disciplined annotations correlate 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.

Aiming to understand how developers debug code in the presence of code variability, Melo et al. [30] carried out an experiment by using an eye tracker predominantly with graduate students. All participants had Java programming experience, and several of them had industrial experience. 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 have focused on refactorings to discipline annotations to understand how novices specify the

correct output. We put extra effort in minimizing potential threats regarding eye tracker camera usage. For instance, we systematized program style with fewer number of lines and larger size, to easy reproducibility, as well as avoided chairs with swiveling capability, which showed potential to impact data quality.

Melo et al. [29] presented a controlled experiment predominantly with graduate students. All participants 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 bug 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.

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 [34]. For instance, using an eye tracker, Crosby and Stelovsky [14] observed that there are differences in reading source code and reading prose. However, they did not investigate refactorings. We analyzed how the 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 [32]. For instance, Sharafi et al. [43] 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. [4] 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. [47] 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 to 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. [5] studied the effect of identifier length on the ability of programmers to recall. Their experiment's 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 9 summarizes these works.

Table 9: Other related works. In column "Eye," we refer 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 participants 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. [27]	No	—	Yes	—	Access developers' perception on C preprocessor usage (∗)	Despite the criticism of C preprocessor, they use it nonetheless
Schulze et al. [37]	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. [26]	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. [30]	Yes	D	Yes	Time, accuracy, fixations and saccades	Study how developers debug with variability (†)	Debugging time increases with variability
Melo et al. [29]	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 correlates with reduction in time, fixation duration, fixation count, and regressions count.

9 Conclusions

In this article, we reported on a controlled experiment using an eye tracker camera with 64 subjects who were novices in C language to evaluate the influence of three refactorings that discipline `#ifdef` annotations. In our results,

applying R1 ⟨wrapping function call⟩ or R3 ⟨alternative `if` statements⟩ correlated with improvements in the time and visual effort. In addition, applying R3, specifically, correlated 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 correlated 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 [28].

As future work, we intend to evaluate other refactorings proposed by Medeiros et al. [28]. We aim at performing experiments considering more participants, experienced developers, as well as exploring different settings, other types of tasks that add functionalities to the code and fix bugs, higher number of macros, and other types of annotations. We also intend to explore larger source code files, which can also be studied with eye tracking with the addition of a proper tool such as iTrace [19]. 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.

Acknowledgments

We would like to thank the anonymous reviewers, Rafael Mello, and Rodrigo Bonifácio for their insightful comments. This work was partially supported by CAPES (117875 and 175956), FAPEAL (60030.0000000462/2020), CNPq (426005/2018-0, 421306/2018-1, 309844/2018-5, 311442/2019-6, 310757/2018-5, 427787/2018-1), and FAPDF (00193-00000926/2019-67).

References

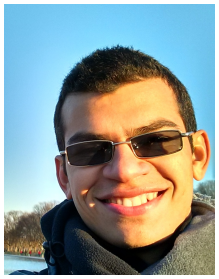
1. José Aldo, Rohit Gheyi, Márcio Ribeiro, Sven Apel, Baldoino Fonseca, Vander Alves, Flávio Medeiros, and Alessandro Garcia. Evaluating refactorings for disciplining #ifdef annotations using eye tracking with novices (artifacts). At `https://github.com/josealdo/EMSE21-ifdefs-with-eye-tracking`, 2021.
2. Victor Basili, G. Caldiera, and H. Rombach. The goal question metric approach. *Encyclopedia of software engineering*, pages 528–532, 1994.
3. 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.
4. 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.
5. Dave Binkley, Dawn Lawrie, Steve Maex, and Christopher Morrell. Identifier length and limited programmer memory. *Science of Computer Programming*, 74(7):430–445, 2009.
6. George Box, J. Stuart Hunter, and William G. Hunter. *Statistics for experimenters*. Wiley-Interscience, 2005.

7. 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.
8. 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*, pages 1–9, 2011.
9. Teresa Busjahn, Carsten Schulte, Sascha Tamm, and Roman Bednarik. Eye movements in programming education II: Analyzing the novice's gaze. 2015.
10. 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.
11. Norman Cliff. Dominance statistics: Ordinal analyses to answer ordinal questions. *Psychological bulletin*, 114(3):494, 1993.
12. Jacob Cohen. *Statistical power analysis for the behavioral sciences*. Academic press, 2013.
13. 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.
14. Martha Crosby and Jan Stelovsky. How do we read algorithms? A case study. *Computer*, 23(1):25–35, 1990.
15. Michael Ernst, Greg Badros, and David Notkin. An empirical analysis of C preprocessor use. *IEEE Transactions on Software Engineering*, 28(12):1146–1170, 2002.
16. 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.
17. Niels Galley, sDirk Betz, and Claudia Biniossek. Fixation durations: Why are they so highly variable. *Advances in visual perception research*, pages 83–106, 2015.
18. Alejandra Garrido and Ralph E. Johnson. Embracing the C preprocessor during refactoring. *Journal of Software: Evolution and Process*, 25(12):1285–1304, 2013.
19. 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.
20. 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.
21. Frouke Hermens and Sunčica Zdravković. Information extraction from shadowed regions in images: An eye movement study. *Vision research*, 113:87–96, 2015.
22. 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.
23. 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.
24. 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.
25. 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.
26. 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.
27. 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.

28. 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.
29. Jean Melo, Claus Brabrand, and Andrzej Wąsowski. 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.
30. Jean Melo, Fabricio Batista Narcizo, Dan Witzner Hansen, Claus Brabrand, and Andrzej Wąsowski. Variability through the eyes of the programmer. In *Proceedings of the 25th International Conference on Program Comprehension*, ICPC'17, pages 34–44, Piscataway, NJ, USA, 2017. IEEE Press.
31. 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.
32. 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):5, 2018.
33. 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.
34. Keith Rayner. Eye movements in reading and information processing. *Psychological bulletin*, 85(3):618, 1978.
35. Keith Rayner. Eye movements in reading and information processing: 20 years of research. *Psychological bulletin*, 124(3):372, 1998.
36. 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.
37. Sandro Schulze, Jörg Liebig, Janet Siegmund, and Sven Apel. Does the discipline of preprocessor annotations matter?: A controlled experiment. In *Proceedings of the 12th International Conference on Generative Programming: Concepts & Experiences*, GPCE '13, pages 65–74. ACM, 2013.
38. Samuel Sanford Shapiro and Martin B. Wilk. An analysis of variance test for normality (complete samples). *Biometrika*, 52(3/4):591–611, 1965.
39. 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.
40. 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.
41. 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.
42. 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.
43. 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.
44. David J Sheskin. *Handbook of parametric and nonparametric statistical procedures*. crc Press, 2020.
45. Henry Spencer and Geoff Collyer. #ifdef considered harmful, or portability experience with C news. In *USENIX Summer. USENIX Association*, pages 185–197, 1992.
46. Anselm Strauss and Juliet Corbin. *Basics of qualitative research techniques*. Citeseer, 1998.
47. 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.

48. 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.



José Aldo Siva da Costa is a PhD student at the Federal University of Campina Grande, Paraíba, Brazil. He received his master's degree in Computer Science in 2016 from the same university. His research interests include program comprehension in the context of refactorings with eye tracking.

Rohit Gheyi is a professor in the Department of Computer Science at Federal University of Campina Grande. His research interests include refactorings, formal methods, and software product lines. He holds a Doctoral degree in Computer Science from the Federal University of Pernambuco.

Márcio Ribeiro is an Associate Professor in the Computing Institute at Federal University of Alagoas. He holds a Doctoral degree in Computer Science from the Federal University of Pernambuco (2012). He also holds the ACM SIGPLAN John Vlissides Award. His PhD thesis has been awarded as the Best in Computer Science of Brazil in 2012. His main research interests are Configurable Systems, Empirical Software Engineering, Software Testing, and Refactoring. He created the Aglomerações app, an app to register and track agglomerations. The app pioneered the idea in which the user registers agglomerations during the COVID-19 pandemic in Brazil.

Sven Apel holds the Chair of Software Engineering at Saarland University & Saarland Informatics Campus, Germany. Prof. Apel received his Ph.D. in Computer Science in 2007 from the University of Magdeburg. His research interests include software product lines, software analysis, optimization, and evolution, as well as empirical methods and the human factor in software engineering.

Vander Alves is Associate Professor at the Computer Science Department of the University of Brasilia, Brazil. He conducts research on Software Product Lines, Command and Control, and Model-Driven Development. Previously, he was a CAPES-Humboldt Fellow at the University of Passau, Germany, and he also held research and development positions at Fraunhofer IESE, Germany, Lancaster University, England, and IBM Silicon Valley Lab, USA.

Baldoino Fonseca is an associate professor at Federal University of Alagoas (UFAL), heading Engineering and Systems Group (EASY). His main research interests include software maintenance and machine learning.

Flávio Medeiros is a professor in the Federal Institute of Alagoas, Brazil. His research interests include configurable systems with a high amount of variability, refactoring and software product lines. He received his Doctoral degree in Computer Science from the Federal University of Campina Grande, Brazil, in 2016.

Alessandro Garcia received the PhD degree in informatics from the Pontifical Catholic University of Rio de Janeiro. He is an associate professor with Informatics Department of Pontifical Catholic University of Rio de Janeiro. His research focuses on software modularity, software metrics, exception handling, and empirical software engineering. He is a member of the IEEE.