What Drives the Reading Order of Programmers? An Eye Tracking Study

Norman Peitek
Leibniz Institute for Neurobiology
Magdeburg, Germany

Janet Siegmund
Chemnitz University of Technology
Chemnitz, Germany

Sven Apel
Saarland University
Saarbrücken, Germany

ABSTRACT

Background: The way how programmers comprehend source code depends on several factors, including the source code itself and the programmer. Recent studies showed that novice programmers tend to read source code more like natural language text, whereas experts tend to follow the program execution flow. But, it is unknown how the *linearity of source code* and the comprehension strategy influence programmers' *linearity of reading order*.
Objective: We replicate two previous studies with the aim of additionally providing empirical evidence on the influencing effects of linearity of source code and programmers' comprehension strategy on linearity of reading order.
Methods: To understand the effects of linearity of source code on reading order, we conducted a non-exact replication of studies by Busjahn et al. and Peachock et al., which compared the reading order of novice and expert programmers. Like the original studies, we used an eye-tracker to record the eye movements of participants (12 novice and 19 intermediate programmers).
Results: In line with Busjahn et al. (but different from Peachock et al.), we found that experience modulates the reading behavior of participants. However, the linearity of source code has an even stronger effect on reading order than experience, whereas the comprehension strategy has a minor effect.
Implications: Our results demonstrate that studies on the reading behavior of programmers must carefully select source code snippets to control the influence of confounding factors. Furthermore, we identify a need for further studies on how programmers should structure source code to align it with their natural reading behavior to ease program comprehension.

CCS CONCEPTS

• **Human-centered computing** → *Empirical studies in HCI*; *HCI design and evaluation methods.*

KEYWORDS

program comprehension, eye tracking, source code linearity, linearity of reading order, programmer experience

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICPC '20, October 5–6, 2020, Seoul, Republic of Korea
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-7958-8/20/05…$15.00
https://doi.org/10.1145/3387904.3389279

ACM Reference Format:
Norman Peitek, Janet Siegmund, and Sven Apel. 2020. What Drives the Reading Order of Programmers? An Eye Tracking Study. In *28th International Conference on Program Comprehension (ICPC '20), October 5–6, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3387904.3389279

1 INTRODUCTION

In the past decades, much research has focused on how programmers comprehend source code, which is a central activity in software development [26, 52]. The underlying cognitive process, *program comprehension*, is a prerequisite for all subsequent programmer activities, such as testing, debugging, and maintenance. Past research theorized on two main strategies of how programmers comprehend software. *Bottom-up comprehension* is used when programmers lack domain knowledge, experience, or context to efficiently understand source code [39]. Instead, they have to understand individual source code lines and statements and integrate their semantic meaning to eventually build an overarching understanding (i.e., chunking [46]). *Top-down comprehension* is used when programmers take advantage of previous experience or domain knowledge for an efficient hypothesis-driven comprehension process [51], for example, guided by variable identifiers [10].

Although there is some evidence for the validity of these existing comprehension models, there are still knowledge gaps, such as when and how programmers are able to apply top-down comprehension. Program comprehension is an internal cognitive process and as such inherently difficult to measure [47]. Conventional methods, such as think-aloud protocols or measuring task efficiency, cannot provide deep insights into the underlying cognitive processes of program comprehension.

One important aspect of program comprehension is observing the way programmers *read* source code. Eye tracking has proved useful to observe programmers reading source code and answer such fundamental research questions on program comprehension (e.g., [13, 44, 53]). For example, Sharif and Maletic replicated a conventional study with eye tracking and found that naming style affects program comprehension in that programmers are able to read under_score style faster than camelCase style [8, 44].

Previous research suggested that the *linearity of the reading order* could be an indicator of how efficient programmers comprehend source code [13]. Busjahn et al.'s seminal study described several eye-gaze measures to gauge linearity of reading order. They showed that programmers read source code less linear than natural text and also that expert programmers read source code less linearly than novices [13]. This study indicates that comprehending source code is a skill that needs to be developed and honed with experience. A replication of Busjahn et al.'s study by Peachock et al. supports the adequacy of the developed eye-gaze measures and partially corroborated Busjahn's study results with student participants [38].

In this paper, we further dig into the role of the linearity of reading order for program comprehension. Specifically, we aim at understanding how programmers' comprehension strategy and linearity of source code itself affect programmers' reading behavior. Understanding all factors that influence programmers' linearity of reading order is critical to more accurately measure program comprehension with eye tracking. To this end, we conducted a non-exact replication of the studies by Busjahn et al. and Peachock et al. with novice and intermediate programmers. Our study differs in the following details: First, based on the two studies, we further refined the materials with a more systematically varied source code linearity. Second, we investigated in addition the interaction between comprehension strategy (i.e., top-down comprehension or bottom-up comprehension) with the linearity of reading order.

In short, we make the following contributions:

- We report on a non-exact replication of Busjahn et al.'s eye-tracking study on linearity of reading order.
- We provide further evidence that more experienced programmers read source code less linearly than novices.
- We present data that indicate that top-down and bottom-up comprehension affect linearity of reading order.
- We propose a method to systematize source code linearity and demonstrate that differences between source code snippets can substantially influence programmers' reading order.

2 ORIGINAL STUDY AND REPLICATION

In this section, we briefly summarize the original study by Busjahn et al. as well as the replication study by Peachock et al.

2.1 Original Study (Busjahn et al.)

Busjahn et al. conducted a novel study on programmers' linearity of reading order [13]. They compared the linearity of reading order of novice and expert programmers as well as the novices' linearity of reading order for natural text and for source code. They observed the eye movements of 14 students while reading source code as well as natural text in their weekly Java beginners course. The natural text were short English passages of four to five lines. In addition, they asked 6 professional programmers to comprehend the source code and observed their eye movements. Due to the novelty of this research question, they also described appropriate eye-gaze measures to quantify the *linearity of reading order.*

Experiment Design. Ultimately, Busjahn et al. ran 17 trials of novices reading natural text, 101 trials of novices reading source code, and 21 trials of experts reading source code. As the novices were still learning programming, their snippets were simpler than the snippets for the expert participants. Only two snippets had to be comprehended by both participant groups. For all snippets, participants were randomly asked one of three possible tasks: write a summary of the source code, compute the output, or answer a multiple-choice question. To observe eye movements Busjahn et al. used a SMI RED-m remote eye-tracker with a sample rate of 120 Hz.

Participants. 7 of the 14 novices were females. They were between 19 and 33 years old, had, at most, little programming experience, and all had, at least, a medium English proficiency (while German being the native language). The experts were all professional programmers with, at least, 5 years of programming experience, and were between 26 and 49 years old. One of the 6 experts was female.

Variables. The study of Busjahn et al. had two independent variables: programmer experience (novice or expert, between-subject) and, for novices, whether the presented stimuli were source code or natural text (within-subject). Busjahn et al. analyzed the data in two steps: First, they contrasted how novices read source code versus natural text (within-subject). Second, they contrasted linearity of reading order between experts and novices (between-subject).

Dependent Variables. To quantify the participants' linearity of reading order, Busjahn et al. describe a set of six eye-gaze measures, which we summarize in Table 1 and which we will also use for our data analysis.[1] In essence, Busjahn et al.'s eye-gaze measures abstract a fixation sequence of (x,y) coordinates on the screen to higher-level concepts, such as *regressions.* Regressions are a sign that a participant had to revisit a previous part, which could be due to an insufficient understanding or following a snippet's execution (e.g., loop structures).

Needleman-Wunsch Algorithm. In addition to the six fixation-based eye-gaze measures, Busjahn et al. analyzed the order in which each source code line was fixated and contrasted it with (a) the "story order" and (b) the execution order of a source code. The story order of a source code snippet is the sequence of each line from top to bottom, similar to natural text (e.g., 1, 2, 3, 4). The execution order of a source code snippet is the sequence of lines in which the code is executed, which may differ significantly from the story order (e.g., 3, 4, 2, 1, 2, 4).

The presented source code snippets contained up to 30 lines of source code. To effectively compare long sequences of line numbers, Busjahn et al. relied on the Needleman-Wunsch (N-W) algorithm, which was originally designed for molecular comparisons of proteins [34] and later applied for use in eye-tracking research by Cristino et al. [16]. The N-W algorithm computes the similarity between two sequences. In this study, it can be interpreted as how similar an observed linearity of reading order is to a line-by-line reading order or a computer's execution order of the source code. Furthermore, as programmers often cannot comprehend a piece of source code in a single read, Busjahn et al. added a dynamic version of the N-W algorithm that tolerates multiple reads. In our data analysis, we will also use both versions of the N-W algorithm to assess the linearity of reading order of our participants.

Results. Busjahn et al. reported two main findings: First, novice programmers read source code less linearly than natural text. Second, expert programmers read source code less linearly than novice programmers.

[1]Busjahn et al. also describe an *element coverage* measure, which we did not include due to technical limitations of our experiment setup.

Table 1: Overview of gaze-based measures taken from Table 1 from Busjahn et al. [13]. In each trial, F is the set of all recorded fixations. F_i (where $i = 1, ..., n$) is the fixation recorded at time index i. $L(F_i)$ is the line number of the fixation at index i. In each trial, W is the set of word indices in the text. $W(F_i)$ is the word number of the fixation at index i.

Measure	Definition	Computation
Vertical Next Text	% of forward saccades that either stay on the same line or move one line down	% of all F_i, where $L(F_i) - L(F_{i+1}) \in \{0, -1\}$
Vertical Later Text	% of forward saccades that either stay on the same line or move down any number of lines	% of all F_i, where $L(F_i) \leq L(F_{i+1})$
Horizontal Later Text	% of forward saccades within a line	% of all F_i, where $L(F_i) = L(F_{i+1}) \wedge W(F_i) \leq W(F_{i+1})$
Regression Rate	% of backward saccades of any length	% of all F_i, where $W(F_i) > W(F_{i+1})$
Line Regression Rate	% of backward saccades within a line	% of all F_i, where $L(F_i) = L(F_{i+1}) \wedge W(F_i) > W(F_{i+1})$
Saccade Length	Average Euclidean distance between every successive pair of fixations	$\frac{\sum_{i=1}^{n-1} \text{Distance}(F_i, F_{i+1})}{\lvert F \rvert - 1}$
Story Order	N-W alignment score of fixation order with linear text reading order	Alignment($L(F)$, story-order pattern)
Execution Order	N-W alignment score of fixation order with the source code's execution order	Alignment($L(F)$, execution-order pattern)

2.2 Replication Study (Peachock et al.)

Experiment Design. Peachock et al. replicated Busjahn et al.'s original study [38]: It was also a mixed-model experiment with two independent variables (programmer experience, between-subject) and stimuli (source code or natural text, within-subject). They also invited student programmers (33 overall, 18 male, 15 female) and asked them to comprehend seven short source code snippets and three pieces of natural language text. They used the same natural language snippets as Busjahn et al., but different source code snippets in C++. The source code contained some variety in complexity, but all on a rather low level. Similar to Busjahn et al., Peachock et al. asked three random comprehension questions after each task to ensure that participants fulfill the given task. Novice participants had no or only little contact with programming. The "expert" participants already had some programming experience, but were still undergraduate students.

There were some differences in their experiment design to the original study. Specifically, Peachock et al. used C++ (instead of Java) snippets; the natural language content was the same, but due to the different participant pool, it was in their native language (English). Unlike Busjahn et al., they did not invite expert programmers, but advanced undergraduate students, which they refer to as "non-novice" participants. Peachock et al. used a Tobii X60 eye-tracker with a sample rate of 60 Hz.

Analysis. Peachock et al. analyzed the dependent variable eye movements in terms of the Busjahn et al.'s linearity measures.

Results. Peachock et al. reported the following results:

- Programmers read source code less linear than natural text (replicated),
- There is no significant difference in linearity between novice and expert participants (not replicated), and
- There are significant differences between natural text and source code in terms of linearity of reading order based on the N-W score (replicated).

2.3 Implications for Future Research

Busjahn et al. developed a tested way to quantify program comprehension: How *linear* do programmers read source code? With their methodology, researchers can tackle further research questions about source code readability and programmer education with affordable and reliable eye tracking. The two presented studies by Busjahn et al. and Peachock et al. show consistently that programmers read source code less linear than natural text.

As a next step, we were interested in how the linearity of the source code itself may affect programmers' linearity of reading order. Since the two studies did not explicitly manipulate the source code linearity, we conducted a non-exact replication.

3 EXPERIMENT DESIGN

The overarching goal of our study is to gain a deeper understanding of how source code, programmer experience, and comprehension strategy affect linearity of reading order. We provide a replication package, which includes all stimuli, acquired data, and analysis scripts.[2] Specifically, we pose the following research questions:

RQ1: Can we resolve the contradicting results of Busjahn et al. and Peachock et al. regarding whether more experienced programmers read source code less linear than novice programmers?

RQ2: Does the comprehension strategy, that is, bottom-up and top-down comprehension, affect linearity of reading order?

RQ3: Does the linearity of source code affect programmers' linearity of reading order?

To evaluate RQ1, we conducted a non-exact replication with novice programmers as well as more experienced programmers that can be classified as *intermediate* programmers according to Dreyfus' taxonomy of skill acquisition [18, 32]. Since we presented the same source code snippets to both groups, we can reduce the risk of a confounding factor arising from within the source code snippets.

[2]https://github.com/brains-on-code/eyetracking-linearity-replication/

To address RQ2, we need to control the programmers' comprehension strategy. We operationalized the comprehension strategy by using meaningful versus obfuscated identifier names in the source code snippets, which has been shown to induce top-down or bottom-up comprehension (cf. Section 3.2, [50]).

To address RQ3, we measure the linearity of source code. For this purpose, we have developed and validated a metric that quantifies source code execution order, which we describe next.

3.1 Source Code Linearity i

To investigate how source code linearity affects linearity of reading order, we define source code linearity i as follows:

$$i = \frac{\Delta}{\bar{\Lambda}}, \text{ where } \Delta = \sum_{i=1}^{|M|} \delta_{m_i} \text{ and } \bar{\Lambda} = \frac{1}{|M|} \times \sum_{i=1}^{|M|} \lambda_{m_i} \text{ for } m_i \in M$$

with

$$\lambda_m := \text{length of a method}$$
$$M := \{m \mid m \in P,\ m \text{ is a method})$$
$$\delta_m := |C_m - D_m|$$
$$D_m := \iota_e \text{ for } e := \text{Declaration}(m)$$
$$C_m := \iota_e \text{ for } e := \text{Call}(m)$$
$$\iota_e := \text{index}(e) \text{ for } e \in P$$
$$P := \Omega_{\text{Program}}$$

The linearity i of a source code snippet is the relation between the distances between *jumps* Δ and the average method length $\bar{\Lambda}$. A jump δ_m for a method m is the absolute distance between line ι where it is called (C_m) and the line where it is declared (D_m).

When programmers comprehend source code, they may follow its execution flow. When they encounter a method call while reading source code, their eyes may "jump" to the declaration of that method. Throughout the process of a thorough understanding of a source code snippet, over time this adds up to an overall "jump distance" Δ, which depends on the number of jumps and the distance of each jump. The distance of a jump is substantially influenced by method length, that is, when programmers have to jump to the subsequent method, they have to skip the entire length of the current method. Thus, we need to normalize the overall "jump distance" by the average length $\bar{\Lambda}$ for each method λ_m of all methods in a given source code snippet.

For example, the snippet `Calculation` in Listing 1 contains two methods, one with 7 and one with 4 lines. Thus, this source code snippet has an average method length $\bar{\Lambda}$ of 5.5 lines. The snippet contains only one method jump from line 11 to 2 (i.e., overall jump distance Δ is 9 lines). We divide the overall jump distance Δ by $\bar{\Lambda}$ and obtain a result of $i = 1.64$, a rather low value indicating a fairly linear snippet. We visualize the three large jumps of the less linear snippet `Student` in Listing 2.

Prerequisites. For these definitions to work, we assume that a given source code is a syntactically correct Java program, containing the declaration of a package, a class, and a static `main` function. Although only evaluated for Java source code, this principle can also be applied to other programming languages.

Snippet Comparability with i. The source code linearity i allows us to compare source code snippets with higher sensitivity than other metrics (e.g., number or size of methods) with each other. The lower the source code linearity i, the more linear a source code snippet is (i.e., the flow of calls follows the order from the top of the screen to the bottom). The higher i is, the less linear is a source code snippet (i.e., the methods in the source code are not located in a position corresponding to the sequence of their calls).

Validation Study. To evaluate whether the source code linearity i captures the intuitive notion of linearity, we conducted a validation study with 10 advanced graduate students. In the validation study, participants compared two snippets regarding their perceived linearity. To this end, we selected 20 snippets with a wide range of linearity, as reflected in the source code linearity i. Then, we divided the snippets into three categories: linear, medium, and non-linear, and asked participants to compare two snippets of different categories (e.g., a linear and a medium snippet) and of the same category (e.g., two linear snippets). This way, we evaluated whether differences and similarity in linearity predicted by our source code linearity i are also reflected in programmers' intuitive notion of linearity.

We found that, for most of the snippets, the linearity metric reflected the perception of participants well. For the few snippets in which both judgements were different, we excluded such cases from the actual study to avoid having different estimations of linearity. It would be interesting to refine the definition of source code linearity i in future studies.

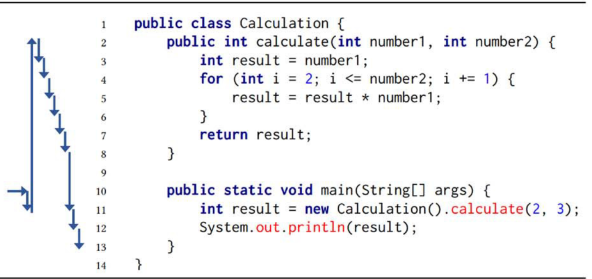
3.2 Material

In line with our research goals, we used source code snippets that both novices and intermediate programmers are able to understand. To understand the effect of source code linearity, we calculated i for all candidate snippets and assigned them to a category (A, B, C, D, or E). The category of a snippet indicates to which 20% percentile its linearity i belongs. For example, a linearity $i = 1.64$ belongs to the 30% percentile and thus is part of category B.

We selected 10 snippets, 2 from each category (cf. Table 2). We re-used 8 Java snippets from the original study by Busjahn et al. (including both snippets that were shown to novices and experts). In addition, we created 2 new snippets (`CheckIfLetters`, `SumArray`), comparable in complexity and content, to balance our snippets along our linearity i metric. All snippets are rather short with, at most, 30 lines of code. The source code snippets implement algorithms commonly used in computing education (e.g., insertion sort). For example, Listing 1 shows a snippet that calculates the cubed number of 2 ($i = 1.64$, Category B).

Each snippet contains a single class with one `main` method, up to 4 helper methods, and, at least, one `System.out.print()` statement composing the snippet's result.

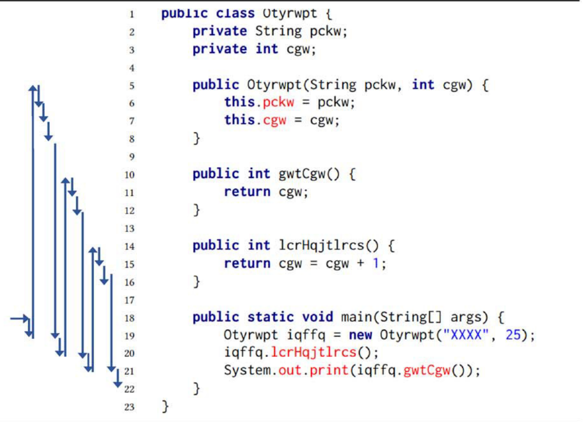
Obfuscated Snippets. After selecting the 10 snippets, we created a second version of all snippets, in which we removed semantic cues by obfuscating all identifier names. This method was initially used by Siegmund et al. to force participants to apply bottom-up comprehension [48]. The motivation to distinguish bottom-up and top-down comprehension, and to observe them separately, is



```java
public class Calculation {
    public int calculate(int number1, int number2) {
        int result = number1;
        for (int i = 2; i <= number2; i += 1) {
            result = result * number1;
        }
        return result;
    }

    public static void main(String[] args) {
        int result = new Calculation().calculate(2, 3);
        System.out.println(result);
    }
}
```

Listing 1: Source code snippet that elicits top-down comprehension with meaningful identifier names, which calculates the mathematical result of 2^3. The source code can be largely read along its linear execution order from top to bottom (visualized with →).



```java
public class Otyrwpt {
    private String pckw;
    private int cgw;

    public Otyrwpt(String pckw, int cgw) {
        this.pckw = pckw;
        this.cgw = cgw;
    }

    public int gwtCgw() {
        return cgw;
    }

    public int lcrHqjtlrcs() {
        return cgw = cgw + 1;
    }

    public static void main(String[] args) {
        Otyrwpt iqffq = new Otyrwpt("XXXX", 25);
        iqffq.lcrHqjtlrcs();
        System.out.print(iqffq.gwtCgw());
    }
}
```

Listing 2: Source code snippet with obfuscated identifiers that prints a student's age after a birthday. The snippet requires programmers' eyes to vertically jump between methods to follow execution flow (visualized with →).

two-fold: First, bottom-up comprehension reduces the advantage of prior programming experience [39] as done by previous studies [3, 23, 24, 27, 28, 33, 48]. Second, the direct contrast between meaningful and obfuscated identifier names allows us to investigate how eye movements change depending on the comprehension strategy (which is similar to the fMRI study by Siegmund et al. [50]). We show an example of an obfuscated snippet that computes the age after a birthday in Listing 2.

Table 2 lists all snippets, their linearity values, and an overview of the behavioral results. All snippets in both versions, meaningful and obfuscated, along with their solutions are available in our replication package.

3.3 Task

We asked participants of both experience levels and both snippet versions to enter the result of the final `print` statement for all presented snippets. For example, for the snippet of Listing 1, the correct output is "8". This is a simplified version from the original study, where Busjahn et al. randomly selected between computing output, a comprehension summary, or multiple-choice questions. The rationale of fixing the task to computing the output is that we aimed to eliminate the chance that the kind of task affects participants' comprehension strategy (in addition to the source code linearity and snippet obfuscation).

3.4 Independent Variables

Our study design contains three independent variables:

- Programmer experience (novice vs. intermediate programmers, between-subject)
- Top-down vs. bottom-up comprehension (meaningful vs. obfuscated identifier names, within-subject)
- Linearity of snippets (5 categories of A, B, C, D, E, with A being most linear and E being least linear, within-subject, cf. Section 3.1)

3.5 Dependent Variables

We consider two dependent variables: behavioral data (i.e., response time and correctness) and eye gaze. We define response time as the time from a participant first viewing a snippet until the time they submit their answer. The raw data of the observed eye gaze is a stream of (x,y) coordinates on the screen, which we used to compute the measures of the study by Busjahn et al. (cf. Table 1, [13]).

3.6 Participants

We recruited participants for both groups from three universities, which we detail in Table 3. We offered to participate in a raffle for a 20€ Amazon gift card, see a visualized export of their eye-tracking data, and receive results of the study as compensation.

Novice participants had a fundamental understanding of Java and object-oriented programming (i.e., passed, at least, an introductory programming class). Our intermediate programmers were advanced graduate students of computer science (i.e., higher-level master or PhD students in computer science or a related field). We verified our selection with small programming questionnaire during the experiment [49]. We summarize our participants' demographics and programming experience in Table 4. The experience of our intermediate programmers lies between the two previous studies' "expert" groups. We classified our participant groups as novice and intermediate programmers according to Dreyfus' taxonomy of skill acquisition [18, 32].

Due to the requirements of the eye-tracker, only programmers without eye-vision conditions (e.g., strabismus; corrective glasses or lenses were acceptable) were eligible to participate in our study.

3.7 Experiment Procedure

Eye-Tracker. We used a Tobii EyeX eye-tracker with a sample rate of 60 Hz. Since we collected data at three different universities, different screen sizes and resolutions were in use: 1920×1200, 1920×1080, and 1680×1050. We scaled all analyses according to the respective screen resolution.

Data Collection. The experimenter led participants through an explanation of the experiment, all program comprehension tasks, and finally demographic and eligibility questions.

We assigned most participants 10 snippets, except for the first 3 novices, who got only 7 snippets. The first three participants were faster than expected and reported little exhaustion from the

Table 2: All snippets used in the study, their metric values, and experimental results. For the metrics columns, darker shading indicates higher values. Unless noted with *, all snippets were part of the study by Busjahn et al. [13]. How often a snippet had to be solved by a group is unbalanced due to a randomized presentation of snippets regarding linearity and variant.

Snippet	Metrics			Variant	Novices		Intermediate Programmers	
	LOC	# Methods	Linearity *i* (Category)		Correctness	Response Time [$\bar{s}$]	Correctness	Response Time [$\bar{s}$]
MoneyClass	8	1	0.00 (A)	Meaningful	7/7	37.6	11/12	31.0
				Obfuscated	5/5	55.3	7/7	44.2
SumArray*	12	1	0.00 (A)	Meaningful	5/5	38.7	14/15	35.1
				Obfuscated	3/4	42.7	4/4	47.3
CheckIfLetters*	21	2	1.56 (B)	Meaningful	7/7	81.2	9/10	64.9
				Obfuscated	2/2	15.1	8/9	101.4
Calculation	14	2	1.64 (B)	Meaningful	6/8	72.4	12/14	47.4
				Obfuscated	4/4	78.4	5/5	99.8
InsertSort	29	3	3.24 (C)	Meaningful	2/9	163.0	3/12	153.9
				Obfuscated	–	–	4/7	218.1
Vehicle	26	3	4.42 (C)	Meaningful	7/7	86.1	16/16	54.1
				Obfuscated	5/5	107.4	3/3	85.8
Student	23	4	8.27 (D)	Meaningful	8/9	29.1	12/12	29.2
				Obfuscated	2/3	64.1	7/7	46.1
SignChecker	26	3	9.90 (D)	Meaningful	8/8	78.6	17/17	56.7
				Obfuscated	2/4	107.1	2/2	68.7
Street	21	4	10.57 (E)	Meaningful	6/6	26.4	14/14	30.1
				Obfuscated	5/6	172.6	5/5	45.7
Rectangle	29	5	20.00 (E)	Meaningful	9/9	94.6	11/11	59.9
				Obfuscated	2/3	77.2	8/8	90.6
Overall				Meaningful	65/75 (87%)	76.3 ± 54.0	120/133 (90%)	55.0 ± 39.9
				Obfuscated	30/36 (83%)	96.9 ± 55.7	53/57 (93%)	89.6 ± 60.4

Table 3: Participant recruitment universities.

	University Passau	University Weimar	University Magdeburg
Novices	5	2	5
Intermediate Programmers	0	17	2

Table 4: Demographic data of our participants. Our intermediate programmers tend to be older, but also have more experience specific to Java and general programming.

	Novices (n=12)	Intermediate Programmers (n=19)
Male	9 (75%)	18 (95%)
Female	3 (25%)	1 (5%)
Age (in Years)	21.4 ± 2.3	29.9 ± 4.6
Years of Programming	3.3 ± 1.8	12.2 ± 6.1
Years of Java Programming	3.0 ± 2.3	6.8 ± 5.5

meaningful snippets. Thus, we increased the number of meaningful snippets from 4 to 7. For all subsequent participants, we presented 3 obfuscated (bottom-up comprehension) snippets and 7 meaningful (top-down comprehension) snippets, which led to an imbalance between the number of the two comprehension strategies (cf. Table 2). We pseudo-randomized the order and the selection in which snippets were presented (i.e., we ensured the split between obfuscated and meaningful snippets, but besides that, everything else was random).

Execution. We obtained 15 eye-tracking data sets from novices and 19 from intermediate programmers. We excluded three data sets from novices, because the eye-tracker failed to track more than one minute of data due to a setup issue. Therefore, all subsequent data analyses (including behavioral data) are based on 12 novice participants.

Deviation. Three obfuscated snippets contained (by mistake) an error that would prevent compilation. We discuss an interesting observation on how programmers with different experience levels handle this issue in Section 6.2.

4 DATA ANALYSIS

4.1 Preparation and Preprocessing

Behavioral Data. To decide whether an answer was *semantically* correct, we manually evaluated each response. We interpreted responses with only minor formatting inaccuracies as semantically correct (e.g., if a participant responded with a value of "1.4" instead of "1.40").

Eye-Tracking Data. The eye-tracker provides a stream of (x,y) coordinates. We applied several preprocessing steps to ensure high data quality and reliability.

First, we smoothed the stream of (x,y) coordinates with a Savitzky-Golay filter (window length of 5, polynomial order of 3) [35]. Next, we applied a velocity-based algorithm to detect fixations and saccades from the eye gaze. We used a velocity threshold of 150 pixel in 100 milliseconds. If the velocity was below the threshold, it was interpreted as a fixation, otherwise as a saccade [22].

Second, we created areas-of-interest (AOIs) for each line and block of each snippet. The AOIs allow us to compute the measures that describe the linearity of reading order as described by Busjahn et al. For all subsequent analyses based on AOIs, we filtered out all fixations outside of defined AOIs (e.g., participants looking around the room). Following Busjahn et al., we included fixations with a maximum of a 100 pixel horizontal deviation (≈ 7–8 characters), as small AOIs can otherwise be easily missed (e.g., a closing bracket) and may distort the results.

4.2 Analysis Procedure for Eye-Tracking Data

After preprocessing, we computed the measures capturing linearity of reading order developed by Busjahn et al. (cf. Table 1) for each participant group and experimental factor.

Since we have more than one independent variable, we compute a linear mixed regression model, which allows us to also detect possible interaction effects [30]. This analysis differs from Busjahn et al., who only had one independent variable and used Mann-Whitney-U tests to test for significant differences. For each of the Busjahn et al.'s measures described in Table 1, we computed a linear mixed regression model with three factors: programmer experience (novice or intermediate programmer), comprehension mode (top-down or bottom-up), and source-code linearity (in five categories: A, B, C, D, E). We used the R `lme4` package, version 1.1.21, to compute the linear model [2]. Data across all measures yielded in a converged model, indicating that the provided factors can explain the observed variance. We subsequently tested the fitted model for statistical significance with the `car` package [19], version 3.0.6, which internally uses a type II Wald chi-square test.

To avoid an inflated probability of the type-I error (i.e., incorrect rejection of a null hypothesis) due to multiple statistical testing, we adjusted the significance threshold with a false-discovery-rate (FDR) correction [6]. This resulted in an adjusted p-level significance threshold of 0.033. Thus, we consider only results with a p-level below 0.033 as statistically significant.

5 RESULTS

In this section, we report the results of the behavioral and eye-tracking analysis, followed by our interpretation in Section 6.

5.1 Behavioral Data

We show a summary of the behavioral results between the two participant groups in Table 5 and a detailed version for each snippet, variant, and group in Table 2. While novices achieved a similar correctness rate as the intermediate programmers (86% vs. 91%, Mann-Whitney-U test[3]: $U = 11121, p = 0.072$), they were across all snippets significantly slower, on average (83 sec vs. 65 sec, $U = 8001, p = 0.000$). Intermediate programmers showed a faster comprehension when snippets contained meaningful identifier names, which facilitate top-down comprehension (76 sec vs. 55 sec, $U = 3562, p = 0.000$). But, when we obfuscated identifier names enforcing bottom-up comprehension, intermediate programmers fall back close to the speed of novices (97 sec vs. 90 sec, $U = 889, p = 0.142$).

Unlike intermediate programmers, most novices found the accidental compiler errors in the obfuscated snippets (cf. Section 3.7). However, this difference is non-significant ($U = 52, p = 0.079$).

Table 5: Behavioral data separated by programming experience. Intermediate programmers are faster, but miss more compiler errors. Gray font color marks non-significance.

	Novices (n=12)	Intermediate Programmers (n=19)
Correct Responses (All)	86%	91%
Correct Responses (Meaningful)	87%	90%
Correct Responses (Obfuscated)	83%	93%
Response Time (All, in sec)	83.0 ± 55.1	65.4 ± 49.4
Response Time (Meaningful)	76.3 ± 54.0	55.0 ± 39.9
Response Time (Obfuscated)	96.9 ± 55.6	89.6 ± 60.4
Compiler Errors Detected	7 of 11 (64%)	2 of 7 (28%)

5.2 Eye-Tracking Data

In Table 6, we provide an overview of the eye-tracking results for all three RQs. In essence, we can *partially replicate Busjahn et al.*'s results that intermediate programmers read source code less linear and *contradict Peachock et al.*'s negative result (RQ1). Specifically, we found evidence that intermediate programmers show significantly more *vertical next* and *vertical later* eye movements, that is, their eye gaze jumps to the next or a source code line further down (cf. Table 1). In contrast to Busjahn et al., we observed that intermediate programmers use more *vertical regressions*, that is, that the percentage of their eye gaze movements going upwards in source code is larger than that of novices. We cannot confirm Busjahn et al.'s findings with the *N-W algorithm* in our data set, which yielded non-significant results. In other words, novice and intermediate programmers' reading order is not different for our sample.

We observed mixed results about whether and how programmers' comprehension strategy affects their linearity of reading order (RQ2). First, programmers using top-down comprehension show percentage-wise less *vertical regressions*, but more *horizontal later* eye movements than in bottom-up comprehension. In other words, programmers' eyes seem to move within a line more during top-down comprehension. Although programmers use longer saccades during top-down comprehension, this effect is not significant (after FDR correction). The *N-W scores* for bottom-up comprehension are significantly smaller than for top-down comprehension, indicating

[3] t tests are inappropriate as Shapiro-Wilk tests [42] showed non-normality for response correctness and times.

Table 6: Overview of the three studies' eye-tracking results. Inequality symbol > indicates in which direction the linearity of reading order is influenced. Gray font color marks non-significance. Text highlighted with green indicate replicated results, while purple indicates that our results are different from Busjahn et al. We did not find any significant interaction effects.

	Busjahn [13]	Peachock [38]	Our Study		
	Experience*	Experience*	Experience**	Comprehension Strategy**	Source Code Linearity i (A, B, C, D, E)**
Vertical Next	$p < 0.001$, E > N	$p = 0.406$	$p = 0.031$, E > N	$p = 0.148$, TD > BU	$p = 0.000$, A > B > C > D > E
Vertical Later	$p < 0.010$, E > N	$p = 0.461$	$p = 0.003$, E > N	$p = 0.059$, TD > BU	$p = 0.000$, D > E > C > B > A
Vertical Regressions	$p < 0.001$, N > E	$p = 0.453$	$p = 0.010$, E > N	$p = 0.000$, BU > TD	$p = 0.000$, D > E > C > B > A
Horizontal Later	$p < 0.001$, E > N	$p = 0.487$	$p = 0.008$, N > E	$p = 0.001$, TD > BU	$p = 0.000$, A > B > C > D > E
Horizontal Regressions	$p = 0.970$, E > N	$p = 0.973$	$p = 0.044$, N > E	$p = 0.450$, TD > BU	$p = 0.001$, A > B > C > D > E
Saccade Length	$p < 0.001$, E > N	*not provided*	$p = 0.903$, N > E	$p = 0.046$, TD > BU	$p = 0.000$, D > E > A > C > B
Story Order (Naïve)	$p < 0.001$	*not provided*	$p = 0.909$, N > E	$p = 0.000$, TD > BU	$p = 0.000$, A > D > E > B > C
Story Order (Dynamic)	$p < 0.0001$	*not provided*	$p = 0.557$, N > E	$p = 0.000$, TD > BU	$p = 0.000$, A > D > B > E > C
Exec. Order (Naïve)	*not provided*	*not provided*	$p = 0.809$, N > E	$p = 0.000$, TD > BU	$p = 0.000$, A > D > B > E > C
Exec. Order (Dynamic)	*not provided*	*not provided*	$p = 0.932$, N > E	$p = 0.000$, TD > BU	$p = 0.000$, A > B > D > E > C

* Matt-Whitney U Tests ** Linear Mixed Model with Wald Chi-Square Test (Significance Threshold 0.033 after FDR Correction)
N = Novice, E = Expert/Intermediate Programmers BU = Bottom-Up, TD = Top-Down

that bottom-up comprehension is closer to a top-to-bottom reading order, whereas top-down comprehension expresses itself in more wandering eye movements.

In our study, source code linearity i significantly affects all observed linearity measures (RQ3). In general, a higher linearity score i (indicating source code with many large vertical jumps in its execution order) leads to longer vertical eye movements, but fewer short eye movements (i.e., within one or two neighboring source code lines). While the order of the linearity categories A, B, C, D, E appear sensible for vertical and horizontal eye movements, they are mostly inconsistent for *saccade length* and the *N-W scores*. In other words, source code linearity does not seem to influence the average eye jump distance nor how similar the reading order is to execution or story order.

6 DISCUSSION

6.1 Behavioral Data

We expected that more experienced programmers are generally faster [31, 45], but when bottom-up comprehension is enforced, the differences may vanish based on a contradictory result from studies of Soloway and Ehrlich [51] and Gilmore and Green [21]. We indeed observed that intermediate programmers are significantly faster than novices when meaningful identifier names facilitate top-down comprehension. While the performance gap is significantly reduced during bottom-up comprehension, novices are still slower. We thus classify our result in between Soloway and Ehrlich's vanishing effect [51] and Gilmore and Green's result that experienced programmers stay faster [21].

6.2 Spotting Compiler Errors in Snippets

Three of the obfuscated snippets (i.e., `Street`, `SignChecker`, `CheckIfLettersOnly`) mistakenly contained function and variable identifiers that were undefined (e.g., the variable `cjviij`*1* in Listing 3 should be `cjviij`). This results in a compiler error and therefore not to a determined output. Interestingly, many novices spotted this error, while most intermediate programmers did not.

```
1  Cjviij cjviij = new Cjviij(5);
2  cjviij.cijTqmniv(15);
3  System.out.print(cjviij1.wijTqmniv());
```

Listing 3: Part of the obfuscated snippet `Street`, which contains a compiler error. Most intermediate programmers miss the non-initialized variable in Line 3, while novices tend to notice it.

An early study of Shneiderman and Mayer showed that experts focus on semantic aspects of source code, while novices concentrate on syntax [46]. We discussed this phenomenon with three of our intermediate programmers. They generally confirmed that these early results still hold true in modern times, especially with IDE support. They reported that, due to their daily work, they are used to an IDE highlighting basic compiler errors. One reported that "you have to look for compiler errors to find them", which due to the missing IDE highlighting (e.g., red underline) may not be on the mind of a programmer. In addition, one reported that "at that spot, you recognize the object based on the name without checking every individual character". Novices appear to be less likely to take such mental shortcuts and need to learn to focus on semantic aspects of source code.

6.3 Eye-Tracking Data

RQ1: Comparing Study Results Regarding Linearity of Reading Order. The two studies by Busjahn et al. and Peachock et al. did not draw a clear picture of whether there is a distinguishable difference in linearity of reading order with increased programming experience:

Busjahn et al.'s expert programmers were significantly different across several reading linearity measures, while Peachock et al.'s "non-novices" were not. It is interesting to learn that our data partially replicate Busjahn's results with an intermediate programmer group, which brings us closer to learn with what experience level programmers change their reading order.

Unlike Busjahn et al., we find that intermediate programmers use significantly more eye movements across *all three vertical* measures. This is consistent with the notion that experienced programmers' eyes jump through source code, looking for "beacons" [10, 43]. Intuitively, it also makes sense that all three measures capturing vertical eye movements point in the same direction (in our case, intermediate programmers use more vertical eye movements).

Busjahn et al. also observed that experts more often stay on a single line and read it from left to right, which, however, appears contradictory to the interpretation that novices read source code more "book like" from top to bottom and left to right, while experts' eyes jump around more. Our data support the view that more experienced programmers use less horizontal eye movements than novices. This is plausible, because novices use more bottom-up comprehension, which leads to reading entire lines of source code from left to right. Our result is in general agreement with the notion that experts apply a more erratic but intentional search through source code and novices a more repetitive gaze pattern [4].

Again, in contrast to the results of Busjahn et al., we did not observe significant differences in saccade length or reading order between the two participant groups. We suspected that our results diverge from Busjahn et al., because we showed both participant groups the same snippets. The average saccade length for both participant groups was the same, while Busjahn et al. showed longer snippets to experts. However, when we normalize the observed saccade length with the snippet length, this actually reverses, such that longer snippets lead to shorter (normalized) saccades. Thus, our results are inconclusive, and we cannot be certain how snippet length, saccades, and experience interact.

RQ1: Overall, we were able to partially corroborate Busjahn et al.'s result and contradict Peachock et al.'s result. Intermediate programmers read source code less linear than novices.

RQ2: Effect of Comprehension Strategy on Linearity of Reading Order. We aimed at understanding whether the comprehension strategy, that is, top-down or bottom-up comprehension, leads to a significant difference in programmers' eye movements (RQ2). We expected that programmers using top-down comprehension show more vertical eye movements and less horizontal eye movements, as top-down comprehension is rather a hypothesis-driven comprehension strategy, whereas bottom-up comprehension requires building up a source code snippet's meaning by reading every line.

However, our eye-tracking results are inconclusive: The differences in the fixation-based linearity measures were only significant for two measures, which do not converge to an apparent interpretation. All other measures were non-significant, leading to an overall unclear picture about whether top-down comprehension entail more vertical movements.

Interestingly, the reading order based on the *N-W scores*, which did not discriminate between our novice and intermediate programmers, shows highly significant differences between bottom-up and top-down comprehension. In other words, on average, the reading patterns during bottom-up comprehension are closer to a snippet's story order, whereas reading patterns during top-down comprehension are closer to a snippet's execution order. But, we note that all scores are rather low (smaller than -100), indicating that there still are large differences between expected reading order and actual eye movements during program comprehension.

RQ2: While we partly uncover a less linear reading order during top-down comprehension, most measures are inconclusive. Thus, a difference of eye-movement patterns between bottom-up and top-down comprehension is unsupported by our data.

RQ3: Effect of Source Code Linearity on Linearity of Reading Order. One goal of our study was to understand whether the linearity of source code significantly affects programmers' eye movements. The linearity of source code showed a strong effect on reading order, for both novice and intermediate programmers and both comprehension strategies. In other words, linearity of source code seems to be the major driving factor that determines the reading order, whereas experience and the comprehension strategy play a more minor role. While this may not be surprising, we have provided empirical evidence that this actually is the case.

The two original studies paved the way to study programmers' reading order. We took one further step and varied source code linearity in a systematic way to understand its influence. With our setup, we found that there are two separate effects that influence the reading order: On the one hand, efficient program comprehension by more experienced programmers or top-down comprehension leads to larger vertical eye movements, because programmers search for certain features to quickly verify their hypothesis of a snippet's purpose. These are intentional eye movements that are necessary to adeptly comprehend source code. On the other hand, less linear source code forces programmers' eyes to make unnecessary large vertical jumps when they try to follow a snippet's method call chain. Thus, while the former are eye movements initiated by internal cognitive processes of the programmer to make the comprehension process efficient, the latter are eye movements that are imposed by external factors. Both types of eye movements may also interact, such that a less linear structure makes it more difficult for the programmer to make the intentional, hypothesis-confirming eye movements. In other words, linearly structured source code could reduce the required eye movements and make the comprehension process more efficient, if it supports the programmer's hypothesis-confirming eye movements.

However, our less linear snippets tend to be more complex, so we cannot draw robust conclusions with the conducted study. Instead, we call for further dedicated studies that specifically contrast source code with different internal structures but implementing the same algorithm to understand how programmers shall organize methods in a class [20]. Our source code linearity measure i, although incomplete for some special cases, can be a starting point to systematically operationalize the linearity of a source code snippet.

RQ3: The linearity of source code strongly affects programmers' reading order: Less linear source code leads to large unnecessary vertical eye movements.

7 THREATS TO VALIDITY

7.1 Construct Validity

Several of our used eye-movement measures are based on matching a fixation to a source code line. This leaves some room for interpretation, as participants may use peripheral vision and do not need to focus exactly on a source code line [36]. Like Busjahn et al., we interpreted a fixation to be on a source code line if it was horizontally less than 100 pixels away.

Similarly, there is room for interpretation when computing the execution order of source code. For example, how to interpret lines that only contain a closing bracket (`}`) is debatable or whether class definition code (`public class Example`) should be considered as part of the execution order. We included all brackets and boilerplate code in the execution flow (as an interpreter would step through the code). This technical interpretation may divert from how humans read code and likely contributed to low N-W scores.

To operationalize linearity, we developed a source code linearity metric and validated it with intuitive notions of linearity of programmers. Although there are some deviations of the intuitive perception and the linearity metric, this did not pose a threat to construct validity, as we excluded such cases from our study.

7.2 Internal Validity

There are several threats to validity arising from our participant sample. First, we have a skewness in gender distribution, which, however, is close to the actual population in computer science for our universities. Second, we have to question whether our participant group division was reasonable: Are our intermediate programmers actually sufficiently experienced programmers? To ensure a correct assignment, we asked participants a few questions regarding their experience, based on a questionnaire developed by Siegmund et al. [49]. Furthermore, the behavioral data indicate that our assignment was reasonable.

Finally, regarding our eye-tracking data: We did not apply a manual correction of the scan paths, which can be error prone [37]. A visual exploration of the obtained data showed reasonable preciseness, so we do not consider our results threatened without a manual correction. We also did not apply a drift correction [22], as our experiment was comparably short, so we do not expect meaningful drift.

7.3 External Validity

Our study exhibits the typical threats of having small Java programs and recruited students. Our results can only be carefully generalized to other contextual factors. Reading behavior of larger snippets with higher control-flow complexity may show different results [17, 25]. Nevertheless, our setting targets an important population.

8 RELATED WORK

Original Study and Replications. In addition to the original study and its first replication introduced in Section 2, Blascheck and Sharif conducted another replication, albeit focused on introducing a new methodology of visualizing the linearity of reading order [9].

Eye Tracking on Program Comprehension. In addition to the original study and its replications, there are numerous studies that used eye tracking to observe program comprehension. For example, Turner et al. used eye tracking to investigate the difference in bug searching tasks between C++ and Python source code, which yielded no significant difference [53]. Binkley et al. investigated the difference in identifier styles (under_score vs. camelCase) and found that it is mostly a matter of preference, with experts' comprehension being more affected by the identifier style [7, 44]. There are several other studies related to program comprehension covering method summarization [1, 41], syntax highlighting [5], code review [43, 54], and programmer education and expertise [14, 40]. They all investigate one specific aspect of program comprehension, but did not focus on reading order.

Source Code Metrics and Readability. While there is a plethora of source code metrics [55], we are not aware of one that is designed to specifically capture the linearity of source code. We consider the linearity of source code as an element of *code readability*, which captures all syntactic factors that affect programmers. For example, Buse and Weimer asked programmers how readable source code is and, based on the results, build a predictive model to estimate source code readability [11, 12]. However, these readability studies work on individual methods rather than (small) classes. On a class-level, many studies focus on *maintainability* [15] or *complexity* [29], rather than fundamental readability.

9 CONCLUSION

Inspired by Busjahn et al.'s seminal study [13], and the replication study by Peachock et al. [38], we set out to investigate the effect that source code linearity, programmer experience, and comprehension strategy have on reading order of source code. Our results indicate the linearity of source code is a major driving factor that determines programmers' reading order, while experience and comprehension strategy seem to play more minor roles. With our intermediate programmers' experience level lying between the two previous studies, we seem to have found a turning point when programmers switch from a linear reading order to a reading order following the execution order. The strong effect of linearity implicates that the structure of the source code should be matched to programmer's expectations to avoid unnecessary eye movements, which may make program comprehension more efficient. In future studies, we intend to dig deeper into the size of the effect of source code linearity and programmer expectation, and the experience level.

ACKNOWLEDGMENTS

We thank all participants of our study, Jennifer Bauer and Sandro Schulze for supporting data acquisition, Johannes Hofmeister for his contributions to the linearity metric, and Susanne Mayr for fruitful discussions. Siegmund's work is supported by DFG grant SI 2045/2-2 and the Bavarian State Ministry of Education, Science and the Arts in the framework of the Centre Digitisation.Bavaria (ZD.B). Apel's work has been supported by the German Research Foundation (AP 206/6).

REFERENCES

[1] Nahla Abid, Bonita Sharif, Natalia Dragan, Hend Alrasheed, and Jonathan Maletic. 2019. Developer Reading Behavior While Summarizing Java Methods: Size and Context Matters. In *Proc. Int. Conf. Software Engineering (ICSE)*. IEEE, 384–395.

[2] Douglas Bates, Martin Mächler, Ben Bolker, and Steve Walker. 2015. Fitting Linear Mixed-Effects Models Using lme4. *Journal of Statistical Software* 67, 1 (2015), 1–48.

[3] Jennifer Bauer, Janet Siegmund, Norman Peitek, Johannes Hofmeister, and Sven Apel. 2019. Indentation: Simply a Matter of Style or Support for Program Comprehension?. In *Proc. Int. Conf. Program Comprehension (ICPC)*. IEEE, 154–164.

[4] Roman Bednarik. 2012. Expertise-Dependent Visual Attention Strategies Develop over Time during Debugging with Multiple Code Representations. *Int'l Journal of Human-Computer Studies* 70, 2 (2012), 143–155.

[5] Tanya Beelders and Jean-Pierre du Plessis. 2016. The Influence of Syntax Highlighting on Scanning and Reading Behaviour for Source Code. In *Proc. Annual Conf. of the South African Institute of Computer Scientists and Information Technologists (SAICSIT)*. ACM, 1–10.

[6] Yoav Benjamini and Yosef Hochberg. 1995. Controlling the False Discovery Rate: A Practical and Powerful Approach to Multiple Testing. *Journal of the Royal Statistical Society. Series B (Methodological)* 57, 1 (1995), 289–300.

[7] Dave Binkley, Marcia Davis, Dawn Lawrie, Jonathan Maletic, Christopher Morrell, and Bonita Sharif. 2013. The Impact of Identifier Style on Effort and Comprehension. *Empirical Software Engineering* 18, 2 (2013), 219–276.

[8] David Binkley, Marcia Davis, Dawn Lawrie, and Christopher Morrell. 2009. To CamelCase or Under_score. In *Proc. Int. Conf. Program Comprehension (ICPC)*. IEEE, 158–167.

[9] Tanja Blascheck and Bonita Sharif. 2019. Visually Analyzing Eye Movements on Natural Language Texts and Source Code Snippets. In *Proc. Symposium on Eye Tracking Research & Applications (ETRA)*. ACM, 14:1–14:9.

[10] Ruven Brooks. 1983. Towards a Theory of the Comprehension of Computer Programs. *Int. Journal of Man-Machine Studies* 18, 6 (1983), 543–554.

[11] Raymond Buse and Westley Weimer. 2008. A Metric for Software Readability. In *Proc. Int'l Symposium on Software Testing and Analysis*. ACM, 121–130.

[12] Raymond Buse and Westley Weimer. 2010. Learning a Metric for Code Readability. *IEEE Transactions on Software Engineering* 36, 4 (2010), 546–558.

[13] Teresa Busjahn, Roman Bednarik, Andrew Begel, Martha Crosby, James H Paterson, Carsten Schulte, Bonita Sharif, and Sascha Tamm. 2015. Eye Movements in Code Reading: Relaxing the Linear Order. In *Proc. Int. Conf. Program Comprehension (ICPC)*. IEEE, 255–265.

[14] Teresa Busjahn, Carsten Schulte, Bonita Sharif, Andrew Begel, Michael Hansen, Roman Bednarik, Paul Orlov, Petri Ihantola, Galina Shchekotova, and Maria Antropova. 2014. Eye Tracking in Computing Education. In *Proc. Conf. on International Computing Education Research*. ACM, 3–10.

[15] Don Coleman, Dan Ash, Bruce Lowther, and Paul Oman. 1994. Using Metrics to Evaluate Software System Maintainability. *Computer* 27, 8 (1994), 44–49.

[16] Filipe Cristino, Sebastiaan Mathôt, Jan Theeuwes, and Iain Gilchrist. 2010. ScanMatch: A Novel Method for Comparing Fixation Sequences. *Behavior Research Methods* 42, 3 (2010), 692–700.

[17] Martha Crosby and Jan Stelovsky. 1990. How Do We Read Algorithms? A Case Study. *Computer* 23, 1 (1990), 25–35.

[18] Hubert Dreyfus and Stuart Dreyfus. 1986. Mind over Machine: The Power of Human Intuition and Expertise in the Era of the Computer. *The Free Press* (1986).

[19] John Fox and Sanford Weisberg. 2019. *An R Companion to Applied Regression* (third ed.). Sage, Thousand Oaks CA.

[20] Yorai Geffen and Shahar Maoz. 2016. On Method Ordering. In *Proc. Int. Conf. Program Comprehension (ICPC)*. IEEE, 1–10.

[21] David Gilmore and Thomas Green. 1988. Programming Plans and Programming Expertise. *The Quarterly Journal of Experimental Psychology* 40, 3 (1988), 423–442.

[22] Kenneth Holmqvist, Marcus Nyström, Richard Andersson, Richard Dewhurst, Halszka Jarodzka, and Joost Van de Weijer. 2011. *Eye Tracking: A Comprehensive Guide to Methods and Measures.* OUP Oxford.

[23] Yoshiharu Ikutani, Takatomi Kubo, Satoshi Nishida, Hideaki Hata, Kenichi Matsumoto, Kazushi Ikeda, and Shinji Nishimoto. 2020. Expert Programmers have Fine-Tuned Cortical Representations of Source Code. *bioRxiv 2020.01.28.923953* (2020).

[24] Yoshiharu Ikutani and Hidetake Uwano. 2014. Brain Activity Measurement during Program Comprehension with NIRS. In *Proc. Int. Conf. Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*. IEEE, 1–6.

[25] Ahmad Jbara and Dror Feitelson. 2017. How Programmers Read Regular Code: A Controlled Experiment Using Eye Tracking. *Empirical Software Engineering* 22, 3 (2017), 1440–1477.

[26] Thomas LaToza, Gina Venolia, and Robert DeLine. 2006. Maintaining Mental Models: A Study of Developer Work Habits. In *Proc. Int. Conf. Software Engineering (ICSE)*. ACM, 492–501.

[27] Seolhwa Lee, Danial Hooshyar, Hyesung Ji, Kichun Nam, and Heuiseok Lim. 2018. Mining Biometric Data to Predict Programmer Expertise and Task Difficulty. *Cluster Computing* 21, 1 (2018), 1097–1107.

[28] SeolHwa Lee, Andrew Matteson, Danial Hooshyar, SongHyun Kim, JaeBum Jung, GiChun Nam, and HeuiSeok Lim. 2016. Comparing Programming Language Comprehension between Novice and Expert Programmers Using EEG Analysis. In *Proc. Int. Conf. Bioinformatics and Bioengineering (BIBE)*. IEEE, 350–355.

[29] Stefan Lessmann, Bart Baesens, Christophe Mues, and Swantje Pietsch. 2008. Benchmarking Classification Models for Software Defect Prediction: A Proposed Framework and Novel Findings. *IEEE Transactions on Software Engineering* 34, 4 (2008), 485–496.

[30] Mary Lindstrom and Douglas Bates. 1988. Newton-Raphson and EM Algorithms for Linear Mixed-Effects Models for Repeated-Measures Data. *Journal of the American Statistical Association* 83, 404 (1988), 1014–1022.

[31] Steve McConnell. 2011. What Does 10x Mean? Measuring Variations in Programmer Productivity. *Making Software, O'Reilly* 16 (2011).

[32] Jerry Mead, Simon Gray, John Hamer, Richard James, Juha Sorva, Caroline St Clair, and Lynda Thomas. 2006. A Cognitive Approach to Identifying Measurable Milestones for Programming Skill Acquisition. *ACM SIGCSE Bulletin* 38, 4 (2006), 182–194.

[33] Takao Nakagawa, Yasutaka Kamei, Hidetake Uwano, Akito Monden, Kenichi Matsumoto, and Daniel M. German. 2014. Quantifying Programmers' Mental Workload During Program Comprehension Based on Cerebral Blood Flow Measurement: A Controlled Experiment. In *Proc. Int. Conf. Software Engineering (ICSE)*. ACM, 448–451.

[34] Saul Needleman and Christian Wunsch. 1970. A General Method Applicable to the Search for Similarities in the Amino Acid Sequence of Two Proteins. *Journal of Molecular Biology* 48, 3 (1970), 443–453.

[35] Marcus Nyström and Kenneth Holmqvist. 2010. An Adaptive Algorithm for Fixation, Saccade, and Glissade Detection in Eyetracking Data. *Behavior Research Methods* 42, 1 (2010), 188–204.

[36] Pavel Orlov. 2017. Ambient and Focal Attention During Source-Code Comprehension. In *Proc. Int'l Workshop on Eye Movements in Programming (EMIP)*. 12–13.

[37] Christopher Palmer and Bonita Sharif. 2016. Towards Automating Fixation Correction for Source Code. In *Proc. Symposium on Eye Tracking Research & Applications (ETRA)*. ACM, 65–68.

[38] Patrick Peachock, Nicholas Iovino, and Bonita Sharif. 2017. Investigating Eye Movements in Natural Language and C++ Source Code - A Replication Experiment. In *Augmented Cognition. Neurocognition and Machine Learning*. Springer International Publishing, 206–218.

[39] Nancy Pennington. 1987. Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs. *Cognitive Psychology* 19, 3 (1987), 295–341.

[40] Cole Peterson, Jonathan Saddler, Tanja Blascheck, and Bonita Sharif. 2019. Visually Analyzing Students' Gaze on C++ Code Snippets. In *Proc. Int'l Workshop on Eye Movements in Programming (EMIP)*. IEEE, 18–25.

[41] Paige Rodeghero and Collin McMillan. 2015. An Empirical Study on the Patterns of Eye Movement During Summarization Tasks. In *Proc. Int. Symp. Empirical Software Engineering and Measurement (ESEM)*. IEEE, 1–10.

[42] Samuel Shapiro and Martin Wilk. 1965. An Analysis of Variance Test for Normality (Complete Samples). *Biometrika* 52, 3/4 (1965), 591–611.

[43] Bonita Sharif, Michael Falcone, and Jonathan Maletic. 2012. An Eye-Tracking Study on the Role of Scan Time in Finding Source Code Defects. In *Proc. Symposium on Eye Tracking Research and Applications (ETRA)*. ACM, 381–384.

[44] Bonita Sharif and Johnathon Maletic. 2010. An Eye Tracking Study on camelCase and under_score Identifier Styles. In *Proc. Int. Conf. Program Comprehension (ICPC)*. IEEE, 196–205.

[45] Ben Shneiderman. 1977. Measuring Computer Program Quality and Comprehension. *Int. J. Man-Machine Studies* 9, 4 (1977), 465–478.

[46] Ben Shneiderman and Richard Mayer. 1979. Syntactic/Semantic Interactions in Programmer Behavior: A Model and Experimental Results. *Int. J. Parallel Programming* 8, 3 (1979), 219–238.

[47] Janet Siegmund. 2016. Program Comprehension: Past, Present, and Future. In *Proc. Int. Conf. Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 13–20.

[48] Janet Siegmund, Christian Kästner, Sven Apel, Chris Parnin, Anja Bethmann, Thomas Leich, Gunter Saake, and André Brechmann. 2014. Understanding Understanding Source Code with Functional Magnetic Resonance Imaging. In *Proc. Int. Conf. Software Engineering (ICSE)*. ACM, 378–389.

[49] Janet Siegmund, Christian Kästner, Jörg Liebig, Sven Apel, and Stefan Hanenberg. 2014. Measuring and Modeling Programming Experience. *Empirical Software Engineering* 19, 5 (2014), 1299–1334.

[50] Janet Siegmund, Norman Peitek, Chris Parnin, Sven Apel, Johannes Hofmeister, Christian Kästner, Andrew Begel, Anja Bethmann, and André Brechmann. 2017. Measuring Neural Efficiency of Program Comprehension. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*. ACM, 140–150.

[51] Elliot Soloway and Kate Ehrlich. 1984. Empirical Studies of Programming Knowledge. *IEEE Transactions on Software Engineering* 10, 5 (1984), 595–609.

[52] Rebecca Tiarks. 2011. What Programmers Really Do: An Observational Study. *Softwaretechnik-Trends* 31, 2 (2011), 36–37.
[53] Rachel Turner, Michael Falcone, Bonita Sharif, and Alina Lazar. 2014. An Eye-Tracking Study Assessing the Comprehension of C++ and Python Source Code. In *Proc. Symposium on Eye Tracking Research and Applications (ETRA)*. ACM, 231–234.
[54] Hidetake Uwano, Masahide Nakamura, Akito Monden, and Ken-ichi Matsumoto. 2006. Analyzing Individual Performance of Source Code Review Using Reviewers' Eye Movement. In *Proc. Symposium on Eye Tracking Research & Applications (ETRA)*. ACM, 133–140.
[55] Alberto Salvador Núñez Varela, Hector G. Pérez-González, Francisco E. Martínez-Perez, and Carlos Soubervielle-Montalvo. 2017. Source Code Metrics: A Systematic Mapping Study. *Journal of Systems and Software* 128 (2017), 164–197.