Bachelor's Thesis

Beyond Syntax: An Eye Tracking Analysis of Java Method Ordering Strategies

Sami Naim

March 20, 2024

Advisors:
Dr. Norman Peitek Chair of Software Engineering
Anna-Maria Maurer Chair of Software Engineering

Examiners:
Prof. Dr. Sven Apel Chair of Software Engineering
Prof. Dr. Antonio Krüger Deutsches Forschungszentrum für Künstliche Intelligenz

Chair of Software Engineering
Saarland Informatics Campus
Saarland University

UNIVERSITÄT
DES
SAARLANDES

Erklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken,_____ _____

(Datum/Date) (Unterschrift/Signature)

Abstract

Source code should be written in a manner that guarantees consistency and efficiency while reading, for it not only affects the code revision of the developer but additionally eases the comprehension process for other developers reading the code. Without a generally established guideline to follow, most projects apply different strategies for organizing the source code. While this allows the developer the flexibility to write code following their intuition, the lack of consistency may provoke confusion in subsequent source code comprehension by others. Therefore, code-writing guidelines that guarantee reading efficiency are needed. In this thesis, we focus on one of the flexible aspects of programming languages, i.e., the order in which methods within a class are ordered. We analyze three method ordering strategies in the programming language Java using behavioral methods, eye tracking, and the subjective preference of participants. In the first step, we observe and analyze quantitative metrics, namely task correctness, response time, fixation count, fixation time, and saccade length. Then we compare the quantitative results to the subjective preference of participants on the best method ordering strategy out of the three. Based on our results, we conclude that the "Calling" strategy is subjectively preferred among the participants. The "Calling" strategy revolves around ordering methods following the call hierarchy within a class, i.e., if a method calls another method, then the called method is positioned below the caller method. However, the subjective preference does not fully adhere to the quantitative results, which do not show a significant advantage of the "Calling" strategy, although further research might provide more insights. Furthermore, we demonstrate the application of eye tracking to offer further analysis of participants' reading processes, which can be used in future work to explore a larger set of method ordering strategies and expand to other programming languages.

Contents

List of Figures

List of Tables

Listings

Acronyms

OGAMA	Open Gaze And Mouse Analyzer
StCS	StyleCop Strategy
CaS	Calling Strategy
CoS	Cohesion Strategy
RaS	Random Strategy
fMRI	functional Magnetic Resonance Imaging
EEG	Electroencephalography
JCC	Java Code Conventions

1 Introduction

1.1 Goal of this Thesis

Software has become an undeniably essential part of the everyday life of each individual. It provides us with functionalities that assist us in our daily lives and enable us to connect with people all around the globe. An analysis from March 2023 [14] shows that there are over 5.7 million apps on the Google Play Store and the Apple App Store combined. Unfortunately, an app's development lifecycle does not end after the initial release; rather, apps require daily maintenance and improvements based on the users' feedback [21]. Given that each software contains many files with possibly hundreds of lines of source code, it is crucial to guarantee that the developer comprehends the source code in a reasonable time. One of the aspects that greatly influences code readability is consistency [43].

Keeping consistency throughout the code of a project is pivotal for understanding source code quickly and efficiently [18], especially considering the high writing flexibility offered by a lot of programming languages. Although most programming languages have some agreed-upon conventions to keep source code consistent, these conventions are very minimal and still leave a lot of room for deviation. An example of a widely used convention is the "one class per file" convention, which states that each file must contain a single class with the goal of reducing confusion while browsing project files [17].

In contrast, one of the aspects of programming languages without a unified consensus is method order. Several programming languages allow any arbitrary method order within a class while still maintaining the logic. Since the class only serves as a container for methods, the order of methods does not affect the execution or output of source code. We can observe this concept in programming languages such as Java [26] and Python [30], where methods are defined as objects belonging to a class and only referenced through that class. In other words, classes consist of a set of methods that implement certain functionalities and are exclusively accessible through that class, which renders the ordering of methods obsolete. Nevertheless, programmers must decide on the visual order of the methods. As mentioned above, consistency is pivotal. Hence, choosing arbitrary method ordering for each class would lead to inefficient source code comprehension.

In this thesis, we will introduce three method ordering strategies and compare them in terms of task correctness, response time, and eye movements. The three strategies, which are defined later in Chapter 2, are "StyleCop," "Calling," and "Cohesion." Considering the vast number of possible method ordering strategies, the aim of this thesis is not to conclude which strategy is the best overall, but rather, which of the three method ordering strategies chosen is the most promising for enhancing program comprehension. Additionally, we

hope that the methodology applied in this thesis will pave the way for future research with a larger set of method ordering strategies and expand to other programming languages. To achieve this goal, we employ a combination of behavioral and eye tracking measures. The behavioral measures provide a basis to compare method ordering strategies based on participants' task correctness and response time. Furthermore, the eye tracking measures allow us to observe the visual attention of participants during source code comprehension, thus developing a deeper understanding of their reading process.

1.2 Motivational Example

To illustrate the problem of method ordering, we will observe two source code snippets found on GitHub[1] from the Spring Boot repository[2], which is an open-source project that aids in creating production-grade spring applications with minimal configurations. For Listing 1.1, we observe that none of the three method ordering strategies are applied in this class. The access modifiers are scrambled, which excludes the "StyleCop" strategy. The *clear()* method calls *getSesssionFile()*, which is located above it. This excludes the "Calling" strategy. The closest method ordering strategy is "Cohesion." Although, this should imply the usage of "Cohesion" throughout the project for consistency purposes. However, the method order in Listing 1.2 does not conform to the "Cohesion" strategy completely due to the fact that *refreshApplicationListeners()* is separated from other methods accessing the field *application*. Interestingly enough, it fully adheres to the "StyleCop" strategy. The approach used by Spring Boot developers leaves us with a method order incompatible with the three strategies.

1 GitHub: https://github.com/
2 Spring Boot: https://github.com/spring-projects/spring-boot

Listing 1.1: A class skeleton from the Spring Boot repository

```
class FileSessionPersistence implements SessionPersistenceManager {
    private final File dir;
    FileSessionPersistence(File dir) {
        this.dir = dir;
    }
    public void persistSessions() {
        // Calls getSessionFile() and saveF()
    }
    private void saveF() {
        // Calls saveS()
    }
    private void saveS() {
    }
    public Map<String, PersistentSession> loadSessionAttributes() {
        // Calls getSessionFile() and loadF()
    }
    private Map<String, PersistentSession> loadF() {
        // Calls loadS()
    }
    private Map<String, PersistentSession> loadS() {
        // Calls readSession()
    }
    private Map<String, SerializablePersistentSession> readSession() {
    }
    private File getSessionFile() {
    }
    public void clear() {
        // Calls getSessionFile()
    }
}
```

Listing 1.2: Another class skeleton from the Spring Boot repository

```
class EventPublishingRunListener {
    private final SpringApplication application;
    private final String[] args;
    private final SimpleApplicationEventMulticaster initialMulticaster;
    EventPublishingRunListener(SpringApplication application, String[] args) {
        // Initializes all fields
    }
    public int getOrder() {
    }
    public void starting() {
        // Accesses the fields application and args
    }

    // ...
    // Five public methods that also access the fields application and args
    // ...

    public void failed() {
        // Accesses the fields application, args, and initialMulticaster
    }
    private void multicastInitialEvent(ApplicationEvent event) {
        // Accesses the fields initialMulticaster
    }
    private void refreshApplicationListeners() {
        // Accesses the fields application and initialMulticaster
    }
}
```

1.3 Overview

In the background section, in Chapter 2, we introduce the method ordering strategies and program comprehension. Following that, in Chapter 3, we present related work, its significance, and how this thesis builds upon it. After establishing background and motivation, in Chapter 4, we explain the methodology used for the experiment. The results of the experiment are evaluated in Chapter 5, along with discussion and threats to validity. We summarize the findings of this thesis and provide thoughts on future work in Chapter 6.

2

Background

In this chapter, we address several concepts that are essential for the topic at hand. First, we introduce code readability and its connection to method ordering strategies. Then we briefly define program comprehension, followed by an elucidation on eye tracking.

2.1 Code Readability

Code readability concerns the difficulty of comprehending source code and is hence directly related to the maintainability of programs [7]. It has been the main topic in many literatures published, each of which provides different views and guidelines for writing source code.

Martin [22] and McConnell [23] offer with their books "Clean Code" and "Code Complete" two perspectives on code design and maintenance based on their personal experience. Martin [22] also inspired one of the method ordering strategies that we analyze in this thesis. Although both books suggest different code-writing styles, the authors agree that source code should be written with the goal of making the code more readable, maintainable, and testable.

Furthermore, code readability has been extensively explored and investigated by numerous papers [12, 43]. For example, Sedano [36] conducts a study on 21 programmers who follow four code readability sessions to improve the readability of their code. Each session consists of two phases. During the first phase, an expert programmer reads aloud the code written by one of the 21 programmers. In the second phase, the expert programmer discusses the code with the programmer who wrote it. The researcher reports an improvement in code readability after only three sessions. Additionally, the researcher identifies common fixes to improve readability, for example, changing variable and method names, and replacing repeated code with a new method.

Sasaki et al. [34] highlight the connection between code readability and the order of statements. They propose a new technique for ordering statements within methods and they test it in the programming language Java. The technique revolves around the idea of maintaining the smallest possible distance between a variable and its reference. Through an experiment with 215 methods and 44 subjects, they reveal that the reordered statements enhanced the readability considerably.

Analogous to the latter work, we aim to investigate the reordering of methods to enhance source code comprehension. Although we do not observe one technique but rather focus on comparing different method ordering strategies. In the following section, we present papers that aim to explore different method ordering strategies applied by developers.

2.2 Method Ordering Strategies

Biegel et al. [6] are one of the first to tackle the issue of ordering fields and methods within classes. They explore a wide range of specific ordering strategies, such as grouping methods that call each other together, grouping fields of the same type together, etc., and if they are implemented in Java projects. They focus on the strategies provided by the official Java Code Conventions (JCC) [24], however they detect multiple strategies that deviate from JCC. In addition to that, they find out through a small survey of 52 developers that 87% of them regard the ordering of fields and methods as meaningful or important. The JCC are not included as a strategy in our set of method ordering strategies because the researchers conclude that they are implemented as a primary ordering criterion in all projects in their study. Concerning the alternative strategies found throughout the study, their descriptions are insufficient for a complete method ordering strategy.

In another paper, which we elaborate on in Chapter 3, Geffen and Maoz [16] compare three method ordering strategies in the programming language Java and test them on task correctness and response time of participants. Two of the method ordering strategies that Geffen and Maoz [16] investigate, namely the "StyleCop" and "Calling" strategies, are included in our set. However, the combined strategy "Calling+Connectivity" is left out to keep the strategies simple and concise without worrying about the attributes of different strategies colliding or contradicting each other. The "Connectivity" strategy, which is inspired by JCC, states that methods calling each other must be adjacent to one another. This strategy is excluded because of its high similarity to the "Calling" strategy (defined below). This similarity stems from their reliance on method calls alone.

In the previous two examples, the researchers chose a set of method ordering strategies to conduct their experiment on since one cannot test all possible method orders. In this thesis, we focus on three method ordering strategies that were inspired by the related work and literature on code readability. In the following, we will introduce the three ordering strategies:

1. StyleCop Strategy (StCS): Using this strategy, the methods are ordered mainly by their access modifiers. Similar to the convention of inserting getters and setters to access private variables outside a class, add and remove methods are responsible for adding and removing elements of lists, sets, or other types of collections. The add and remove methods are positioned at the start of the class, with the add methods stationed above their respective remove methods. They are followed by the getters and setters of the class, with getters stationed above their respective setters. All other methods are then ordered by their access modifier. The order is as follows: public, then protected, then private methods. Additionally, static methods are positioned above instance methods. Since the documents provided do not mention the order for the rest of the class, any methods left without an order are ordered alphabetically. This strategy originates from Microsoft's StyleCop [1], which is an open-source static code analysis tool. An example is provided in Listing 2.1.

1 StyleCop: https://github.com/StyleCop/StyleCop

Listing 2.1: An example of a class skeleton ordered with StCS

```
public class Foobar {
    public static void ...() {
        // Does something
    }
    public void ...() {
        // Does something
    }
    protected void ...() {
        // Does something
    }
    private void ...() {
        // Does something
    }
}
```

2. Calling Strategy (CaS): The methods in this strategy are ordered based on call dependency. A method calling another method inside the class should always be positioned above the called method. In other words, the caller always points downward to the callees. Callee methods are in the same order they are called in. Methods that are neither callers nor callees are pushed to the bottom of the class. Once ordered, if two methods share the same place in the hierarchy, for example, two methods that call other methods but are not called by any method themselves, they are ordered alphabetically. We added the latter rule of alphabetical order because the details on how to further order methods were absent. This strategy is described under the name "Vertical Ordering" in the book "Clean Code" by Robert C. Martin [22] and is provided in Listing 2.2.

Listing 2.2: An example of a class skeleton ordered with CaS

```
public class Foobar {
    public void caller1() {
        // Calls callee1() and callee2()
    }
    public void callee1() {
        // Calls callee2()
    }
    private void callee2() {
        // Does something
    }
    public void caller2() {
        // Calls callee3()
    }
    private void callee3() {
        // Does something
    }
}
```

3. Cohesion Strategy (CoS): This strategy is inspired by the types of method cohesion explored by Athanasopoulos and Zarras [2]. Communicational and functional cohesion, which are explained below, form the basis of this strategy. We excluded the sequential cohesion since it overlaps with CaS. This strategy revolves around the idea of grouping methods together that either operate on the same data or perform similar tasks. The constructors and the main method are positioned at the top of the class. Then, methods are divided into three groups:
   - Communicational Cohesion: a communicational group of methods contains methods that operate on the same data within the class. If a method operates on multiple variables, the dominant variable is the one to be considered. A variable in a method is dominant if it appears more frequently than others within the method. Alternatively, if all variables appear with the same frequency, then the one that comes first alphabetically is considered dominant. Belonging to this group are, for example, methods that act on a list within a given class, specifically, add elements, remove elements, empty the list, etc.
   - Functional Cohesion: a functional group of methods contains methods that operate similarly and are not part of a communicational cohesion group. This includes, for example, methods that increase their input by some amount since they do not operate on data within the class.
   - Other Cohesion: all other methods, whether they belong to a cohesion group not mentioned above or are completely incoherent, belong to this final group.

   The groups are inserted in the same order they appear in the enumeration above, i.e., first, communicational cohesion, followed by functional cohesion, and at the end, all other methods. Each group can incorporate multiple subgroups, considering that methods operating on two different variables within a class would both be considered communicational cohesion groups, albeit two different ones. The communicational cohesion groups are ordered alphabetically by the variable name they operate on. Seeing that functional cohesion groups do not share an unequivocal answer to which attribute to order them by, we chose to order them by the method name that is first alphabetically within each group. As for the other cohesion, since there are no groups, the methods are ordered alphabetically. Next, the methods within each subgroup are also ordered alphabetically. All alphabetical orderings mentioned above are added by us to reinforce distinguishability within the different method ordering strategies. An example is shown in Listing 2.3.

Listing 2.3: An example of a class skeleton ordered with CoS

```
public class Foobar {
    private File file;
    private List list;
    // Group 1
    public void saveFile() {
        // Accesses file
    }
    public void loadFile() {
        // Accesses file
    }
    public void deleteFile() {
        // Accesses file
    }
    // Group 2
    public void addToList() {
        // Accesses list
    }
    public void removeFromList() {
        // Accesses list
    }
}
```

2.3 Program Comprehension

Program comprehension is the process of perceiving given source code and then interpreting it based on prior computer science knowledge [45]. It is an important cognitive process for software developers since it usually takes up most of their time [41], which only solidifies the necessity of further research to optimize source code for fast and efficient comprehension. Since program comprehension requires reading the source code, the readability of the program might have a positive or negative effect on it [28].

Research surrounding program comprehension can be traced back to the late 60s, where the researchers Sackman et al. [33] were one of the first to tackle this subject matter. They compare online and offline debugging, where online refers to the debugging process while the program is executing and offline refers to debugging source code without observing the runtime behavior of the program. They conclude that online debugging is better performance-wise, although this result is not as important as the researchers describing the methodology and problems that occurred during the experiment design since there was no prior research to base it on.

Several decades later, in 1995, the researchers Von Mayrhauser and Vans [45] compare six program comprehension models, namely the Letovsky model, the Shneiderman and Mayer model, the Brooks model, top-down model, bottom-up model, and the integrated metamodel. They dive deep into the differences and similarities of the models and provide a detailed overview of their evaluation of each model. This work has been cited in numerous

papers within the field of program comprehension, which highlights the level of detail and importance of this work.

That being said, the field of program comprehension has insufficient research compared to other fields. Starting in the mid-90s, there was a huge setback in conducting research, which is suspected to be caused by the loss of appreciation for empirical evaluations of program comprehension, where papers would be rejected several times [41]. Considering how rapidly the computer science industries are evolving and the countless programming languages, IDEs, etc. that are emerging [10, 19], the mentioned setback made it near impossible to check whether modern technologies and enhancements have positive or negative effects on program comprehension. Hence, conducting research on state-of-the-art programs requires highly fine-tuned study scenarios to assure measuring desired parameters. In light of this information, we elaborate on methods utilized by researchers to measure program comprehension.

Siegmund [41] describes multiple behavioral methods to measure program comprehension, for example, think-aloud protocols, memorization, and comprehension tasks. Think-aloud protocols are verbalized thoughts of participants captured via audio or video while reading source code. Memorization is based on the idea of participants recalling source code. The degree of comprehension is directly related to the correctness of the recall. Lasty, comprehension tasks are measured in terms of task correctness and response time, which is an integral part of our thesis. Comprehension tasks are closer to real-world scenarios than the other two techniques since they rely on comprehending source code to solve a problem. In contrast, thinking aloud and memorizing source code are not directly related to developer activities [32].

Measuring program comprehension using a neural approach varies greatly from study to study because of the different methods that exist, for example, Electroencephalography (EEG) and functional Magnetic Resonance Imaging (fMRI) [41]. There is not a single method that can be regarded as the best; rather, each method has its own suitable use cases. fMRI is usually applied to study brain activity patterns during program comprehension [40]. It measures changes in the oxygen levels of blood caused by brain activity in different regions of the brain. In order to measure brain activity without the time and cost consumption of fMRI, one could also opt for EEG instead. EEG is proven especially useful to capture cognitive performance of developers during program comprehension, for example, drawing conclusions about the link between expertise and programming language comprehension [11].

Eye tracking has been utilized in numerous papers [38] to assess visual attention. Researchers analyze eye tracking data with respect to specific regions on the screen that they assign, known as areas of interest. In this thesis, we are interested in exploring the visual perception of the participants; hence, we will be utilizing eye tracking to measure program comprehension. In the following section, we will elaborate on eye tracking as a method of analyzing participants' eye movements.

2.4 Eye Tracking

Reading source code can be divided into perceiving the source code and then comprehending it [9]. Eye tracking provides the necessary medium to observe and analyze the perception of source code [27].

Eye tracking has been proven useful in numerous scenarios concerning program comprehension. For example, Uwano et al. [44] utilized eye tracking to characterize the performance of participants during code review. They identify a specific eye movement pattern in some participants who are able to find defects faster than others. However, eye tracking studies in program comprehension are not only limited to source code comprehension but also include, for example, UML diagram comprehension. In the paper by Yusuf et al. [48], they conclude that experts rely on coloring and layout to achieve more efficient exploration and navigation of class diagrams.

There are multiple metrics one could consider when working with eye tracking, each providing a different insight into perception. The metrics can be categorized according to the number and duration of fixations, saccades, and scanpaths [38], where fixation is the act of stabilizing the eye gaze on a specific location within the screen.

One of the metrics in the category of number of fixations is fixation count, which indicates the total amount of fixations in each area of interest. Sharafi et al. [39] analyze the fixation count of participants during code reading. They conclude that higher fixation count correlates with higher cognitive load, i.e., the effort spent to answer a question was significantly higher.

One of the metrics in the category of duration of fixations is fixation time, which equals the sum of the durations of all fixations in an area of interest. Bednarik [5] experiments on two groups of programmers with two distinct levels of experience while they debug a program. Through analyzing fixation time, similar to fixation count, a higher fixation time indicated more effort in searching for bugs.

Lastly, the saccades describe the rapid eye movements in between fixations and the scanpath is a chronological order of fixations that indicates some distinct pattern. The two metrics, regression rate and scanmatch, that belong to saccades and scanpath respectively, are the main focus of the paper by Busjahn et al. [8]. After comparing novices and experts via the two metrics, the researchers discovered that novices read source code almost equivalent to reading natural language, whereas experts show signs of tracing code execution to comprehend more efficiently, which is not a possibility in natural language.

3

Related Work

In this chapter, we introduce research related to the topic at hand. We split the research into two sections: the first section includes a study on method ordering strategies with regard to program comprehension, and the second section includes a study on eye tracking and its evaluation.

3.1 Method Order and Program Comprehension

Geffen and Maoz [16] investigate the effect of method ordering on program comprehension in the programming language Java. They examine the usage of four different ordering strategies in open-source projects and conduct a user study on task correctness and response time utilizing the different ordering strategies. The setup of the user study is almost identical to our setup, as they analyze the multiple-choice answers of the participants by observing extracted graphs on task correctness and response time. The study shows that the ordering strategies have little to no effect on task correctness, but some reduce the response time. They do not mention any statistical tests applied while analyzing the data, which leads us to the conclusion that they based their results solely on the graphs. We improve upon this by employing statistical tests on the behavioral data. Additionally, in this thesis, we introduce eye tracking, which they suggest as future work. The eye tracking addition should provide visual insight into participants' source code comprehension, for example, the real-time eye movements of the participants, to assess methods of high interest.

3.2 Eye Tracking in Program Comprehension

Kővári et al. [20] provide a quantitative evaluation of eye tracking metrics using Open Gaze And Mouse Analyzer (OGAMA), which is elucidated in Chapter 4. The eye tracker records participants' eye movements during a debugging task, which revolves around exploring and correcting an incorrectly functioning algorithm. The participants are divided into two groups based on whether they solve the debugging issue via testing multiple modifications or whether they think about the algorithm and then solve it within one or a few modifications. The researchers first analyze the attention maps, i.e., the areas on the screen with the most attention from the participants. Since attention maps are visual representations, the researchers only emphasize the areas with the highest attention. Following the attention maps, the researchers focus on the quantitative measures, more specifically, the metrics: fixation count, fixation duration, and average saccade length. For

each metric, the data of both groups is initially tested for normality via the Shapiro-Wilk test. After that, the data of both groups is compared through a two-sample t-test, which shows no significant difference for the fixation duration and the average saccade length. In contrast, the test shows a significant difference in terms of fixation count, in which the group with participants needing multiple modifications exhibits a higher average fixation count than the other group. Following the latter, they draw the conclusion that developers who tend to think more about the code before editing it debug errors more efficiently than developers who follow a trial-and-error strategy. The methodology and analysis applied by Kővári et al. [20] provide a framework for our methodological approach.

4

Methodology

In this chapter, we establish the research questions and the methodology we applied to answer them. The methodology sections range from the participant recruitment and snippet selection to the main experiment design, ending with experiment variables (dependent and independent) and analysis.

4.1 Research Questions

The primary goal of this thesis is to deduce the method ordering strategy for Java methods within a class that best aligns with human comprehension. To achieve this goal, we formulate five research questions that we hope to answer through the methodology described in the following sections. The five research questions are divided into three quantitative questions and two qualitative questions. As hinted on in Chapter 1, we use a combination of behavioral methods and eye tracking to compare the different method ordering strategies. Subsequently, the first three research questions are about comparing the three method ordering strategies based on task correctness, response time, and eye movements. The choice of dividing the quantitative analysis into three questions is based on the fact that each question explores different phases of the experiment. As we elucidate in the following question descriptions, each question concerns either one or two phases of the experiment; for example, the task correctness is analyzed for only the first question and for all questions together. The first and second questions are aimed at validating the findings of Geffen and Maoz [16]. The third question incorporates the new method of eye tracking to analyze method ordering strategies. Additionally, we include the subjective preference of the participants in our analysis, although it is separate from the quantitative aspects. Thus, the last two research questions aim to connect the quantitative findings with the subjective preference of participants. Specifically, the research questions we will be addressing in this thesis are:

- **RQ1**: Does one of the three method ordering strategies ease program comprehension in terms of task correctness?

  Task correctness refers to the number of correct answers per method ordering strategy. We differentiate between the correct answers for only the first question, and for all questions together, which we elaborate on in Chapter 5.

- **RQ2**: Does one of the three method ordering strategies ease program comprehension in terms of response time?

  Response time refers to the time spent reading the snippets, and the time spent reading the snippets and answering questions. We added the time of only reading

to the analysis to assess the time of participants comprehending the snippets with different strategies.

- **RQ3**: Does one of the three method ordering strategies improve the eye movements in terms of reducing fixation count, fixation time, and saccade length during program comprehension?

  The three eye tracking metrics are illustrated in Chapter 2. We analyze the three metrics during snippet reading.

- **RQ4**: Do the participants' subjective preferences regarding the order of methods align with the objective results of the behavior?

- **RQ5**: Do the participants' subjective preferences regarding the order of methods align with the objective results of the eye movements?

By answering these research questions, we aim to deliver valuable insights into the significance of ordering strategies in program comprehension and to provide a basis for further studies on this topic.

4.2 Participant Recruitment

We selected participants who fulfilled one of two criteria. Each participant should either have a minimum of one year of Java experience or have at least attended the Programming 2 course, which introduces Java concepts and requires students to undertake three small Java projects. We chose these criteria to ensure that the participants comprehend the source code in the snippets provided.

The recruitment of participants was accomplished by advertising on social media and asking students who had completed Java-related courses. A total of twelve participants were recruited. Eleven of the participants are bachelor's and master's students attending Saarland University, and the twelfth participant is a master's graduate who works as a researcher at the DFKI[1]. The participants consisted of one female and eleven males, aged between 20 and 26. The semesters of the students span from the third until the eleventh semester, although the average semester is the sixth. Except for three participants that do not program in their free time, nine participants program on average six hours per week on personal projects.

The Java experience of the participants spans from one to six years, where the average is 2.6 (SD = 1.36) years. When asked to rate their programming skills compared to those of their fellow students, the participants chose, on average, three out of five on the scale (SD = 0.87), which implies that no participant considers themselves a beginner nor an expert. Based on their subjective opinion, half of the participants claim to have project experience in Java, i.e., they have worked on Java projects before, and the other half claim to have regular or basic knowledge. Almost every participant mentioned Python and C as another programming language they know well, which could be interesting to note for future work. An overview of the demographics is provided in Table 4.1.

1 German Research Centre for Artificial Intelligence: https://www.dfki.de/web

Table 4.1: Demographics data of the study population

Category	Sub-Category	Frequency	Percentage
Age	Between 20-23	6	50%
	Between 24-26	6	50%
Gender	Male	11	92%
	Female	1	8%
Highest Education	High School Diploma	9	75%
	Bachelor	2	17%
	Master	1	8%
Java Experience	1 Year	2	17%
	2 Years	3	25%
	3 Years	5	41%
	More than 3 Years	2	17%

4.3 Snippet Selection

The programming language used in the source code is set to be Java, since it is a widely used language [29]. Furthermore, multiple courses at the Saarland University, where the participants are recruited from, have Java as the main programming language throughout the lecture. Source code snippets were collected from multiple GitHub repositories. The snippets are named based on their original repositories, namely "Board" [35], "Buffer" [47], "String" [13], "Snake" [31], and "Palindrome" [1]. During snippet selection, the attributes we searched for were: average class difficulty, which was assessed in the pilot studies and by gathering opinions from programmers, presence of public and private methods for StCS, and methods that call each other for CaS. The methods of each snippet are ordered by the three method ordering strategies StCS, CaS, and CoS with an additional fourth Random Strategy (RaS), which we include to compare it to the other method ordering strategies. RaS assigns each snippet a random method order that does not collide with any of the three method ordering strategies. However, the "Palindrome" snippet, which is shown in the subjective preference section, is only ordered using the three method ordering strategies: StCS, CaS, and CoS. All snippets with the different orderings are provided in Appendix A.

We modified the snippets to ensure the same number of methods in each snippet, which is seven methods. There are two reasons that led us to conclude that the number seven is the optimal number of methods for our experiment. First, we wanted the number of methods in each snippet to be equal to or bigger than the smallest number of methods observed in the paper by Geffen and Maoz [16], which is five. Additionally, the design of our experiment relied on showing the code on one screen without scrolling. After testing out multiple numbers, seven methods sufficed for these two conditions. The modification included removing methods, removing fields that are only accessed by deleted methods, and adding a main method to each snippet. The latter was deemed necessary to ensure that

each snippet has an execution cycle and an output, thus underpinning the need to read all the methods. The "Snake" snippet without a specific order is provided in Listing 4.1.

Listing 4.1: "Snake" code snippet used in the experiment [31]. The snippet does not have a specific order.

```
public char direction = 'R';
private int x = 0;
private int y = 0;

public void update() {
    move();
    if (checkCollision()) die(); }

private void move() {
    if (direction == 'U') y -= 1;
    if (direction == 'D') y += 1;
    if (direction == 'L') x -= 1;
    if (direction == 'R') x += 1; }

private boolean checkCollision() {
    if (x < 0 || x > 20 || y < 0 || y > 20) return true;
    return false; }

public void setDirection(char direction) {
    if (this.direction != getOppositeDir(direction)) {
        this.direction = direction; }}

private char getOppositeDir(char dir) {
    if (dir == 'U') return 'D';
    else if (dir == 'L') return 'R';
    else if (dir == 'D') return 'U';
    else return 'L'; }

private void die() {
    System.out.println("Game Over!"); }

public static void main(String[] args) {
    Example example = new Example();
    example.update();
    example.setDirection('U');
    example.update(); }
```

4.4 Experiment Design

Before initiating the experiment, the participants undergo a warmup session and answer a questionnaire to assess their demographic and their Java experience. In the warmup session, the participants familiarize themselves with the eye tracking equipment and the

general procedure to ensure seamless execution. The snippet used in the warmup session is provided in Listing A.1. Next, they fill out the demographic questionnaire, which entails questions about age, gender, and education level. Additionally, there are questions included on Java programming experience, such as, "How many years of programming experience do you have in total?" and "How do you rate your programming skills compared to your fellow students? (1 to 5)." The questionnaire with the possible answers is provided in Table A.5.

The actual experiment consists of comprehending four snippets and answering three multiple-choice questions per snippet, followed by a question on the subjective preference for ordering the "Palindrome" snippet. The experiment follows a within-subject design. Each participant is presented with the four snippets that we collected, however the method ordering strategy for each snippet differs between participants, i.e., the first snippet could have StCS for the first participant but CaS for the second. If the first snippet always had the same order, for example, CoS, then the risk of learning effect increases.

The four snippets are displayed one by one along with the multiple-choice questions. The participants have seven minutes to read and comprehend the snippets before viewing the questions, however we provide a button in the lower right corner to manually open the questions before the seven minutes pass. We refrain from initially displaying the questions alongside the snippets to direct the participants' focus solely towards the code. This allows us to analyze the response time during reading only and to observe the eye movements while participants comprehend the snippet. After that, the participants have two minutes per question to choose one of the answers. The multiple-choice questions are shown one by one alongside the code. Each snippet has three multiple-choice questions with one correct answer, two wrong answers, and an "I do not know" option. The first question in each snippet asks for the output of the main method. The second and third questions are snippet-specific and aim to assess participants' general comprehension of the snippets. The difficulty of the questions was tested in the pilot studies, wherein the participants reported that the difficulty appeared consistent across the questions. The number of multiple-choice questions and the choice of answers are inspired by the related work in Chapter 3. To illustrate the questions' structure, we show the questions for the "Snake" snippet in Listing 4.1:

1. What is the output of the main method?
   - ☐ "Game Over!"
   - ☐ Nothing
   - ☐ Two times "Game Over!"
   - ☐ I do not know
2. If the movement were to be rendered, where would the starting point be on a grid?
   - ☐ Top-left corner
   - ☐ Centered
   - ☐ Bottom-left corner
   - ☐ I do not know
3. What happens if the main method sets the direction to 'L' instead of 'U'?
   - ☐ An exception is raised

- ☐ The direction changes to 'L'
- ☐ The direction stays 'R'
- ☐ I do not know

All questions are provided in Appendix A.

The eye movements of the participants are collected with a Tobii EyeX Eye Tracker[2] during code reading and question solving. The eye tracker tracks the user's eyes at a 60 Hz frequency, as specified by the manufacturer.

Following the comprehension phase, the participants provide a subjective opinion on the order of the "Palindrome" snippet. The experiment, including the pre-experiment phase, is set to last around 45 minutes per participant.

After the experiment, we gather feedback on the difficulty of the tasks and suggestions for improvements. Additionally, we check the validity of the data collected during the experiment, especially eye tracking data. We verify that the files concerning the demographic questionnaire, snippet answers, and eye tracking data have been successfully generated. Furthermore, we scan the files to ensure the inclusion of all answers from the demographic questionnaire and snippets, as well as the availability of eye tracking data.

4.5 Variables

In the following, we introduce the independent and dependent variables of our study.

4.5.1 Independent Variables

The method ordering strategies used in the experiment are explained in Chapter 2 with the addition of RaS. There are four sequences of the method ordering strategies that repeat three times to cover the twelve participants. the sequences are presented in Table 4.2. The selection of sequences relied purely on having each strategy in each possible position, i.e., each strategy is in the first, second, third, and fourth positions in three iterations. Though ideally, testing all possible permutations of positions, i.e., the 24 possible permutations of the four strategies, would exclude any influence other strategies could have on the strategies after. For example, there is a possibility that having CoS before CaS might influence CaS either positively or negatively. However, this would impose 24 different sequences that require a minimum of 24 participants, which is not possible within the scope of this thesis.

2 https://help.tobii.com/hc/en-us/articles/212818309-Specifications-for-EyeX

Table 4.2: Sequences of method ordering strategies used in the experiment

Sequence Name	Sequence
CaCoRaSt	CaS, CoS, RaS, then StCS
CoRaStCa	StCS, CaS, CoS, then RaS
RaStCaCo	RaS, StCS, CaS, then CoS
StCaCoRa	CoS, RaS, StCS, then CaS

Moreover, the five snippets shown to the participants constitute an independent variable. The sequence of the four snippets during the comprehension phase is identical for all participants, namely the "Board" snippet followed by "Buffer," "String," and then "Snake." The reason behind the consistent sequence is the limited scope of the study. If the snippets exhibited a similar sequence variation as the method ordering strategies, we would need to account for snippets having an impact on each other in terms of comprehension. Considering the low number of participants, we decided on a single sequence of snippets. The "Palindrome" snippet is excluded from the sequence since it is presented in the subjective preference phase.

4.5.2 Dependent Variables

The dependent variables of the study can be derived from the research questions proposed at the beginning of this chapter. The first research question concerns the task correctness. The task correctness will be measured by the multiple-choice answer submissions of the participants. We analyze the correctness across all questions and additionally, the correctness of the first question. Although there are three possible answers (correct, wrong, and "I do not know"), only the correct answers are of significant importance to us since they confirm that the participant has comprehended the code fully.

The second research question is about the response time of each code snippet. We record the time while participants read the code snippet, in addition to the overall time spent reading and solving the multiple-choice questions. We did not deem the overall response time sufficient in our case since the time of comprehension could also yield insights into the efficiency of each method ordering strategy.

The third research question does not comprise of only one dependent variable, but rather a set of dependent variables all related to eye tracking. The three main metrics we are analyzing are fixation count, fixation time, and saccade length, which are explained in Chapter 2.

Both the fourth and fifth research questions are related to the subjective preference of participants. We measure the preference by the number of participants who choose one of the three method ordering strategies: StCS, CaS, or CoS for the "Palindrome" snippet.

4.6 Analysis

In the following sections, we will describe the approach to analyze the data extracted from the experiment. All visual and statistical analysis scripts are available in the GitHub repository "Method Ordering" [25]. The library used to conduct the statistical tests is SciPy[3]. We chose $\alpha = 0.05$ as the significance level for all tests.

4.6.1 Behavioral Methods

Bauer et al. [4] present an empirical study on the various levels of indentation and its effect on program comprehension through eye tracking and behavioral methods. Although the topic is not directly relevant to the topic at hand, the researchers provide detailed description of analysis methods they apply to assess correctness and response time. They check the data for normal distribution via Shapiro-Wilk test and then transform it accordingly. For the statistical analysis part, they apply one-way repeated-measures ANOVA test, which compares means of groups that are differentiated by one factor, and Friedman test, which is an alternative to ANOVA in case the data violates ANOVA's assumptions. Based on their methodology, we describe in the following the statistical tests used to assess task-correctness and response time.

The task correctness of the ordering strategies will be compared using one-way repeated-measures ANOVA test [42] to compare the means or Friedman test [15] to compare the medians of the method ordering strategies. The null hypothesis of both tests states that the different groups are equal. Hence, if the p-value is greater than 0.05, then the strategies are not significantly different. The decision between the two tests depends on whether the data satisfies ANOVA's assumptions, which state that data must be normally distributed and has approximately equal variances across all groups. To test the assumptions, we apply the Shapiro-Wilk test [37] to assess the normality, and Bartlett's test [3] to check for variance homogeneity. For the Shapiro-Wilk test, the null hypothesis states that the sample comes from a normal distribution. In other words, if the p-value of the Shapiro-Wilk test is greater than 0.05, then the data is normally distributed. For Bartlett's test, the null hypothesis states that all variances of the samples are equal, i.e., a p-value greater than 0.05 implies that all variances are equal.

Similar to the task correctness, the response time will be analyzed by one-way repeated-measures ANOVA test or Friedman test.

4.6.2 Eye Tracking

The eye tracking metrics analyzed in this thesis are fixation count, fixation time, and saccade length, which are explained in Chapter 2. The raw data from the eye tracker is inserted into the open-source software OGAMA[4] to analyze various aspects, especially the three metrics mentioned. OGAMA provides a medium to create a slideshow from the snippets. After

3 SciPy: Scientific Library for Python version 1.7.3
4 OGAMA version 5.1: http://www.ogama.net/node/3

importing the raw data into the slideshow, the program calculates the fixations and displays them on the corresponding snippet. The maximum distance in pixels that a data point may vary from the average fixation point and still be considered part of the fixation is set to twelve pixels. Additionally, the minimum number of data points that can be considered a fixation is four. The fixations and their respective times can be exported as a text file. An example of OGAMA displaying fixation count is provided in Figure 4.1. Furthermore, we are able to export the saccade length in pixels as a text file. The results generated via OGAMA are interpreted similarly to the results provided by papers mentioned in Chapter 3. We apply ANOVA and Friedman test to assess equality between the strategies.

Figure 4.1: An example plot of eye tracking data from the experiment. The circles represent fixations of the participant.

5 Evaluation

5.1 Results

In this chapter, we show the results of the study by addressing the research questions and discussing them thereafter. The behavioral and eye-movement results are provided in Table 5.1 and Table 5.8 respectively. All raw data collected throughout the experiment can be found in Appendix A.

Table 5.1: Behavioral results of the study grouped by snippet and method ordering strategy. The standard deviation of response time is provided in parentheses next to the mean value.

Snippet	Strategy	Correctness	Correctness (first question)	Time $[\bar{s}]$	Time $[\bar{s}]$ (only reading)
Board	StCS	8/9	2/3	235 (110)	163 (93)
	CaS	6/9	3/3	258 (112)	174 (98)
	CoS	5/9	0/3	352 (57)	184 (131)
	RaS	8/9	3/3	226 (97)	112 (79)
Buffer	StCS	6/9	1/3	332 (97)	200 (81)
	CaS	2/9	0/3	395 (103)	252 (67)
	CoS	3/9	1/3	251 (161)	157 (114)
	RaS	2/9	0/3	431 (45)	297 (46)
Snake	StCS	3/9	1/3	257 (14)	124 (21)
	CaS	7/9	3/3	196 (15)	96 (18)
	CoS	7/9	2/3	238 (77)	103 (7)
	RaS	5/9	2/3	151 (65)	84 (58)
String	StCS	3/9	0/3	161 (54)	76 (53)
	CaS	6/9	2/3	293 (66)	204 (58)
	CoS	8/9	2/3	172 (30)	109 (18)
	RaS	8/9	2/3	241 (49)	147 (24)

5.1.1 RQ1: Task Correctness Comparison

We compare the method ordering strategies based on the task correctness of all three questions per class. Additionally, we look at the correctness of only the first question, with the assumption that the method ordering strategies might affect cognitive effort on the initial read and become irrelevant thereafter, as stated by Geffen and Maoz [16]. Table 5.1 reports the correctness results for both cases. The results of the "Buffer" snippet suggest an advantage of StCS over the other strategies, although this pattern does not repeat in other snippets. The answers for only the first questions appear to show an advantage of CaS over the other method ordering strategies. A visualization of the distribution is provided in Figure 5.1 and Figure 5.2.

After applying the Shapiro-Wilk test to assess the normality of correct answers, the method ordering strategies exhibit higher p-values than the significance level. The p-values measured in the Shapiro-Wilk test are provided in Table 5.2.

Table 5.2: P-values of the Shapiro-Wilk test per method ordering strategy for the task correctness

Method ordering strategy	All questions	First question
StCS	0.86	0.161
CaS	0.103	0.272
CoS	0.798	0.406
RaS	0.272	0.683

To meet the ANOVA assumptions, we also check for variances' homogeneity via Bartlett's test, which also indicates higher p-values than the significance level (all questions: $p = 0.97$; first question: $p = 0.812$). This excludes the necessity of comparing the times using the Friedman test since we can directly apply one-way ANOVA. In both cases of observing all questions, and first question, the one-way ANOVA test indicates no significant difference between the method ordering strategies (all questions: $F = 0.093$, $p = 0.962$; first question: $F = 0.645$, $p = 0.601$). An overview of the task correctness per method ordering strategy is provided in Table 5.3 and Table 5.4.

To answer the first research question, the correctness results of all three questions do not show a definite advantage of one strategy over the others. Similarly, the tests show no significant difference between the correct answers of the first question.

Table 5.3: Task correctness of participants for all three questions

Method ordering strategy	Correct	Wrong	I do not know
StCS	20	11	5
CaS	21	14	1
CoS	23	8	5
RaS	23	8	5

Table 5.4: Task correctness of participants for only the first question

Method ordering strategy	Correct	Wrong	I do not know
StCS	4	6	2
CaS	8	4	0
CoS	5	3	4
RaS	7	4	1

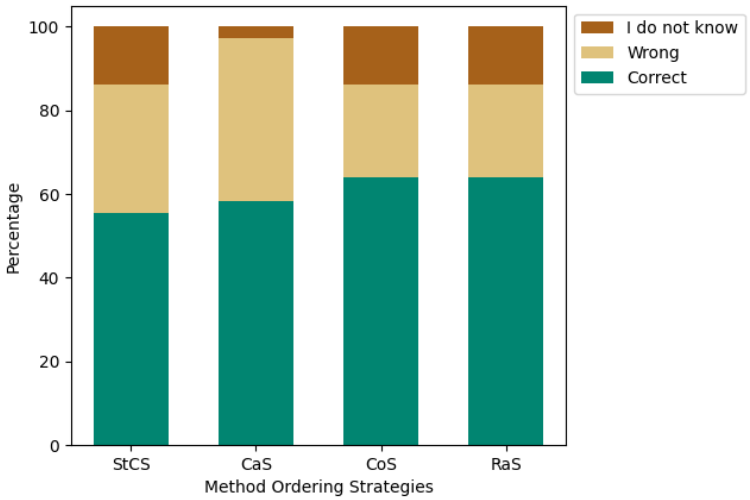

Figure 5.1: Task correctness for all the questions, showing the percentage of correct, wrong, and "I do not know" answers

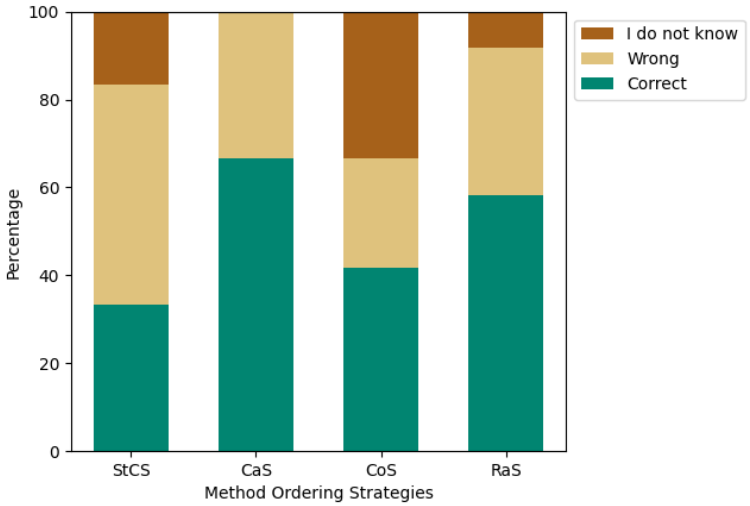

Figure 5.2: Task correctness for only the first question, showing the percentage of correct, wrong, and "I do not know" answers

5.1.2 RQ2: Response Time Comparison

We compare the method ordering strategies based on the task response time while participants read the classes and solve their respective questions. Additionally, we look at the response time of only reading the classes. Based on Table 5.1, the data shows varied times between strategies for reading and solving. The data does not suggest an advantage of any strategy. During reading, CoS seems to score better in terms of response time compared to the other strategies, although this pattern does not repeat combined with the time spent on solving the questions. CaS, on the other hand, appears to have average to high response times in all snippets. The data is visually presented in Figure 5.3 and Figure 5.4.

Using the Shapiro-Wilk test to assess the normality of response time data, the method ordering strategies exhibit higher p-values than the significance level. The p-values measured in the Shapiro-Wilk test are provided in Table 5.5.

Table 5.5: P-values of the Shapiro-Wilk test per method ordering strategy for the response time

Method ordering strategy	Reading and solving	Only reading
StCS	0.394	0.353
CaS	0.813	0.39
CoS	0.644	0.327
RaS	0.971	0.676

To meet the ANOVA assumptions, we also check for variances' homogeneity via Bartlett's test, which also indicates higher p-values than the significance level (reading and solving: $p = 0.919$; only reading: $p = 0.937$). This excludes the necessity of comparing the times

using the Friedman test since we can directly apply one-way ANOVA. In both cases of observing reading and solving, and only reading, the one-way ANOVA test indicates no significant difference between the method ordering strategies (reading and solving: $F = 0.261$, $p = 0.853$; only reading: $F = 0.539$, $p = 0.658$). An overview of the response time per method ordering strategy is provided in Table 5.6 and Table 5.7.

To answer the second research question, the response time results of reading and solving the questions do not show a definite advantage of one strategy over the others. The statistical tests indicate no significant difference.

Table 5.6: Response time (in seconds) of participants while reading and solving questions

Method ordering strategy	Mean	Standard deviation	Min	Max
StCS	246	99	83	468
CaS	286	110	108	483
CoS	253	115	28	413
RaS	263	123	66	484

Table 5.7: Response time (in seconds) of participants while reading only

Method ordering strategy	Mean	Standard deviation	Min	Max
StCS	141	82	2	304
CaS	182	88	35	323
CoS	138	94	5	328
RaS	160	99	4	348

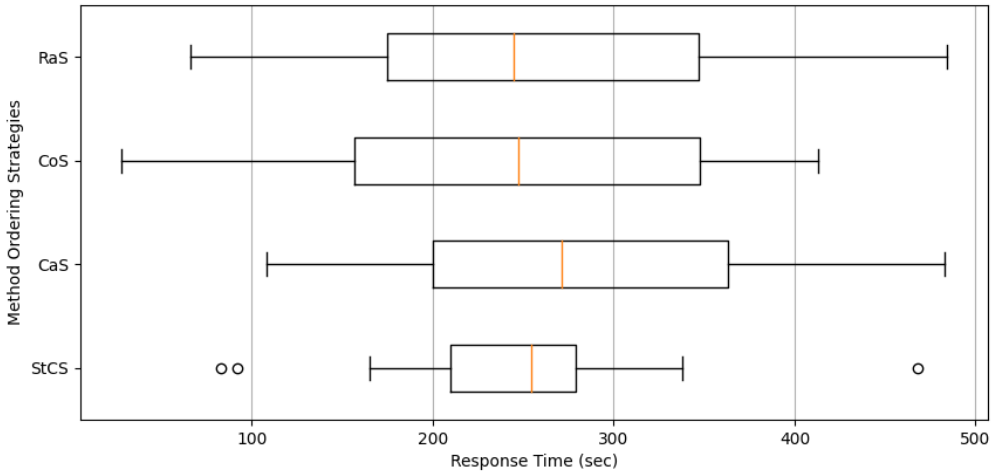

Figure 5.3: Response time per method ordering strategy of participants during reading and solving the questions

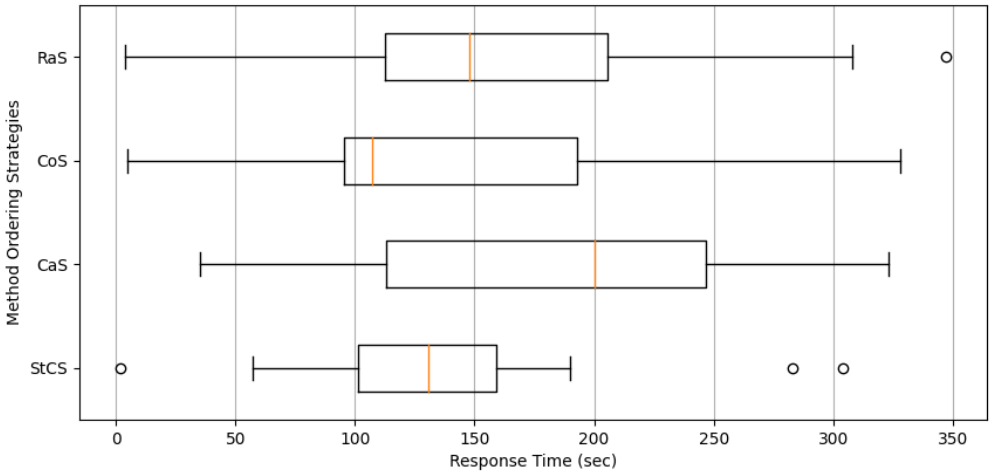

Figure 5.4: Response time per method ordering strategy of participants only during reading

Table 5.8: Eye tracking results of the study grouped by snippet and method ordering strategy. The standard deviation of all metrics is provided in parentheses next to the mean value.

Snippet	Strategy	Average Fixation Count	Fixation Time $[\overline{ms}]$	Saccade Length $[\overline{px}]$
Board	StCS	563 (312)	110 (17)	100 (15)
	CaS	720 (218)	104 (4)	116 (17)
	CoS	285 (192)	130 (21)	155 (21)
	RaS	625 (169)	111 (8)	123 (10)
Buffer	StCS	718 (384)	110 (3)	138 (10)
	CaS	818 (330)	154 (55)	111 (17)
	CoS	622 (107)	109 (7)	126 (11)
	RaS	688 (467)	130 (22)	136 (17)
Snake	StCS	230 (170)	148 (30)	234 (169)
	CaS	325 (90)	104 (2)	124 (17)
	CoS	312 (78)	144 (46)	101 (10)
	RaS	188 (148)	89 (13)	173 (58)
String	StCS	251 (77)	97 (3)	171 (32)
	CaS	558 (406)	138 (19)	174 (62)
	CoS	360 (78)	108 (5)	158 (8)
	RaS	487 (130)	120 (10)	120 (23)

5.1.3 RQ3: Effect on Eye Movements

For the analysis of eye movements, we investigate the data of eleven participants. The twelfth participant is excluded due to insufficient data quality, i.e., raw data containing little to no fixations per class. The eye tracker stopped recording after capturing the first snippet. Analogous to analyzing the response time, we check the following three eye tracking metrics separately on ANOVA assumptions, and then apply one-way ANOVA or Friedman test to assess differences while also observing the respective box plots.

Based on Table 5.8, the strategy CaS seems to have a higher average fixation count compared to the other three strategies. The results are visualized in Figure 5.5. Using the Shapiro-Wilk test together with Bartlett's test, the fixation count data is proven to be normally distributed and consist of homogeneous variances with a significance level (Bartlett's test: $p = 0.164$). The p-values measured in the Shapiro-Wilk test are provided in Table 5.9. Thus, we apply the one-way ANOVA test, which shows no significant difference between the fixation counts of the four strategies ($F = 0.922$, $p = 0.439$). An overview of the fixation count per method ordering strategy is provided in Table 5.10.

Contrary to the results of fixation count, the results of fixation time do not appear to be different. The results are visualized in Figure 5.6. Using the Shapiro-Wilk test, the fixation time data is proven to be not normally distributed with a significance level. The p-values measured in the Shapiro-Wilk test are provided in Table 5.9. Thus, we apply the Friedman test, which shows no significant difference between the fixation times of the four strategies ($F = 1.444$, $p = 0.695$). An overview of the fixation time per method ordering strategy is provided in Table 5.11.

Lastly, the average saccade length in pixels does not seem to show an advantage of one strategy over the others. The results are visualized in Figure 5.7. Using the Shapiro-Wilk test, the saccade length data is proven to be not normally distributed with a significance level, similar to the fixation time. The p-values measured in the Shapiro-Wilk test are provided in Table 5.9. Thus, we apply the Friedman test, which shows no significant difference between the average saccade lengths of the four strategies ($F = 0.709$, $p = 0.871$). An overview of the saccade length per method ordering strategy is provided in Table 5.12.

To answer the third research question, in terms of eye tracking metrics, the results of the statistical tests do not favor one method ordering strategy over the others.

Table 5.9: P-values of the Shapiro-Wilk test per method ordering strategy for the eye tracking metrics

Method ordering strategy	fixation count	fixation time	saccade length
StCS	0.2	0.022	3.74e-5
CaS	0.29	6.16e-4	1.26e-3
CoS	0.646	1.14e-3	0.876
RaS	0.393	0.375	4.39e-3

Table 5.10: Number of fixations per participant

Method ordering strategy	Mean	Standard deviation	Min	Max
StCS	458	346	4	1255
CaS	595	356	43	1141
CoS	374	176	21	753
RaS	525	330	7	1334

Table 5.11: Average fixation time (in milliseconds) per participant

Method ordering strategy	Mean	Standard deviation	Min	Max
StCS	118	26	90	177
CaS	127	38	99	231
CoS	124	31	101	207
RaS	115	21	72	161

Table 5.12: Average saccade length (in pixels) per participant

Method ordering strategy	Mean	Standard deviation	Min	Max
StCS	160	104	79	473
CaS	133	44	89	260
CoS	136	28	86	185
RaS	135	39	95	244

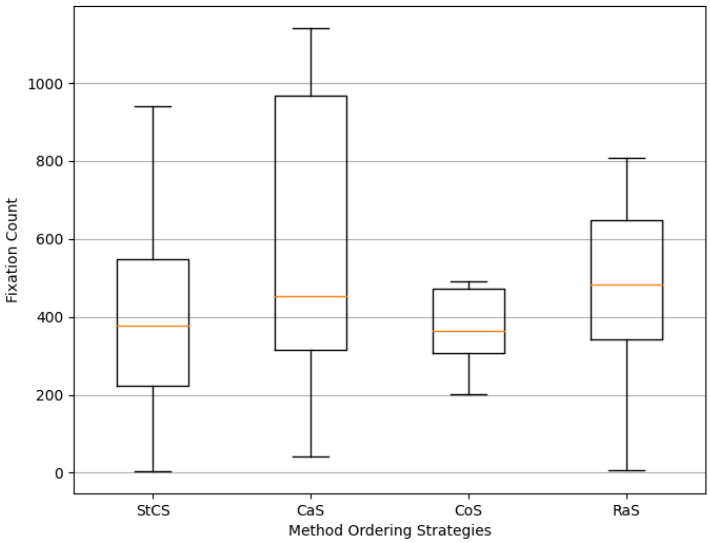

Figure 5.5: Fixation count of participants for each method ordering strategy

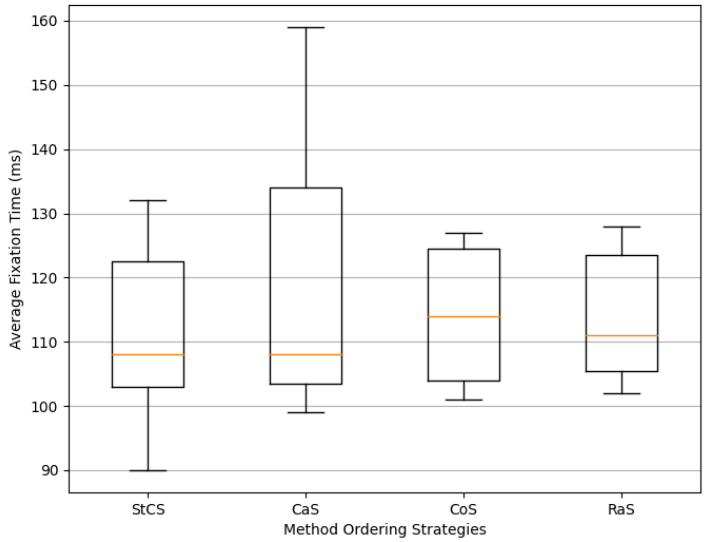

Figure 5.6: Average fixation time of participants for each method ordering strategy

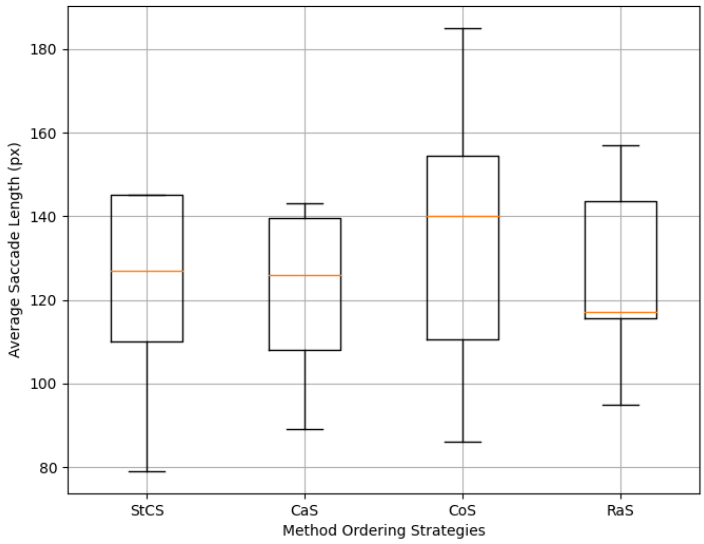

Figure 5.7: Average saccade length of participants for each method ordering strategy

5.1.4 RQ4: Subjective Preference and Behavior

The subjective results in Table 5.13, which are also plotted in Figure 5.8, show CaS having the majority of votes. This does not align with the results of behavioral methods, which do not indicate an advantage of any method ordering strategy.

To answer the fourth research question, the subjectively best method ordering strategy CaS does not demonstrate significantly different results from the other method ordering strategies in terms of task correctness and response time.

5.1.5 RQ5: Subjective Preference and Eye Movements

CaS is voted the highest among the participants, however it does not score differently than other strategies in terms of eye tracking metrics.

To answer the fifth research question, the subjectively best method ordering strategy CaS does not demonstrate significantly different results from the other method ordering strategies in terms of fixation count, fixation time, and saccade length.

Table 5.13: Subjective preference of participants on the subjectively preferred method ordering

Method ordering strategy	Number of participants
StCS	0
CaS	8
CoS	4

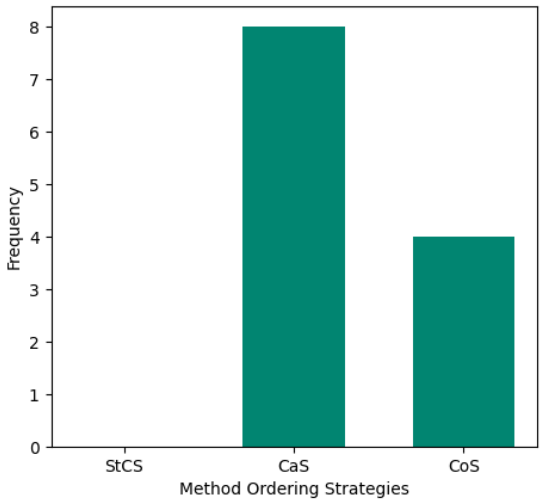

Figure 5.8: Participants' preference regarding optimal method ordering strategy for readability

5.2 Discussion

We have shown that in our case of classes with seven methods each, we could not conclude the best method ordering strategy based on behavioral and eye tracking methods. However, through subjective preference of participants, one method ordering strategy, namely CaS, was preferred more than the others.

The insignificant results throughout the analysis could have multiple reasons. Factors such as number of participants, number of methods, number and difficulty of snippets and questions might greatly affect the results. Depending on the Java experience, participants might have perceived the snippets and questions as too difficult, so the method order had little influence on comprehension. Similarly, the number of methods may have limited the impact of their order on code comprehension, whether positively or negatively. Furthermore, we recruited twelve participants and included eleven of them in the eye tracking analysis, which could be a major reason for the insignificance.

Concerning the main method, all participants agree that the main method belongs at the top or bottom of the class, and it would disturb their reading flow if it was anywhere in between. While observing the reading pattern of participants, it also becomes evident that participants move their eye gaze to the main method immediately. However, it should be mentioned that main methods mostly exist only one time throughout the whole source code, since it starts the program. One could argue that the results would have been different if we excluded the main method, although this is not possible in the scope of this thesis since we show only code snippets and not whole applications.

The method ordering strategy StCS was not prevalent throughout the analysis, except for acquiring zero votes. When asked why they did not choose StCS, the participants claimed that they never looked at the scope of methods while reading. It did not concern them whether the method was public or private. Although this could also be true for bigger

projects, testing out code snippets might have affected the results of StCS because of the lack of other classes trying to access it.

RaS having neutral scores throughout all metrics might indicate that either the sample size of participants was too small, or that the number of methods per class was too little to have a significant effect on program comprehension. Seeing that all other method ordering strategies had the main method at the very top based on their definitions, participants reading the main method might have moved their eye gaze from the main method to the methods called within it constantly leading to a lengthy saccade.

5.3 Threats to Validity

5.3.1 Internal Validity

First, some multiple-choice questions might vary in difficulty across different snippets, which would influence the performance of the method ordering strategies. To avoid this issue, we tried to keep the difficulty of the questions consistent. Other than that, we apply each question to all four method ordering strategies through different sequences, which implies that no strategy had a question that was not presented in all other strategies.

Second, to avoid performance bias of the participants, we added the random ordering strategy (RaS), such that the results of the three method ordering strategies can be compared to a random order' performance. Although, since the study consists of only twelve participants, the results might not be enough to consider the random ordering strategy as a control variable.

Third, we cannot guarantee that all participants comprehended the snippets before opening the questions and solving them. Although we emphasized this point prior to the experiment, some participants might have opened the questions immediately, which would explain the low response times and eye tracking metrics in the plots.

Fourth, the response time during reading might not reflect a real programming scenario since the participant could simply open the questions without having comprehended the snippet completely. To mitigate this issue, we provided the participants with more time to read the snippets compared to the time to answer the questions. However, some participants might have pressed the "show questions" before fully comprehending the snippets.

Lastly, the chosen snippet sequences might have influenced the results, as we did not try every possible sequence. We ensured that every method ordering strategy had a different position within each sequence, although strategies may have an effect on each other based on the sequence. Similarly, the snippets were presented to the participants identically; however, different orders of snippets could have different effects on comprehension.

5.3.2 Construct Validity

First, the algorithms used to order the snippets may not reflect the original intention for each method ordering strategy. Since the definitions of the method ordering strategies do

not fully entail every possible case, we expanded them with alphabetical ordering. We applied the alphabetical order to every missing case to ensure that every strategy had the same expansion. Although investigating other techniques for complementing the strategies might lead to different results.

Second, the eye tracker had to be adjusted depending on the height of the participants since some participants could not be detected by the eye tracker otherwise. To mitigate this issue, we shifted the fixations of the participants in OGAMA to fit the snippets and ensure the eye tracking data is similar across all participants.

5.3.3 External Validity

First, we chose Java as the programming language for the snippets. The results might differ when analyzing other programming languages; hence, the results of this study are not directly applicable to other languages. Nevertheless, considering the popularity and wide usage of Java, it proves suitable as the first programming language to analyze.

Second, the code snippets consisted of seven methods each, which is an adequate number of methods in a class, although classes and their methods within big projects vary largely. Using snippets with more methods or varied method numbers might yield different results.

Third, the participants gathered are students with none of them rating themselves as an expert Java programmer, but rather beginner to average level. The results may have different distributions if we had recruited expert programmers.

6 Concluding Remarks

6.1 Conclusion

In this thesis, we present a study on the effect of method ordering strategies on behavior and eye movements in the programming language Java. We have defined three method ordering strategies, in addition to a fourth random strategy, and examined their performance on twelve participants solving multiple-choice questions of four Java snippets, each ordered with a different ordering strategy.

Through analyzing task correctness, response time, and eye tracking metrics, such as fixation count, fixation time, and saccade length, we have concluded that the method ordering strategies do not show any statistically significant differences in the measured metrics. However, it is noteworthy that the participants' subjective preference for the best method ordering strategy showed CaS acquiring the most votes.

These results suggest that while our objective analysis did not yield any significant results, the subjective opinions we discussed hint at CaS being preferred among programmers. Thus, highlighting the importance of further research to explore method ordering strategies in a wider scope.

6.2 Future Work

In future research, we suggest expanding the set of method ordering strategies and choosing different programming languages. For instance, comparing method ordering strategies between Java and Python. The set of method ordering strategies could also incorporate combined strategies. By broadening the method ordering analysis, we can deepen our understanding of attributes influencing source code comprehension. This could lead to unified conventions across languages to ease comprehension or optimized strategies tailored for each explored programming language.

Furthermore, the cohesion types analyzed in the experiment are communicational and functional cohesion, although there are other cohesion types, namely logical, temporal, and procedural cohesions [46]. We do not consider coincidental cohesion because it is based on arbitrary ordering. We excluded sequential cohesion from CoS since it overlaps with CaS. However, future research could incorporate all six cohesion types to classify them based on efficiency in terms of program comprehension. Following that, we could derive insights into the programmer's cohesion strategy, i.e., which cohesion type is applied the most

during comprehension. Moreover, combining all cohesion types into a single strategy might eliminate the necessity of the alphabetical ordering we add.

Additionally, we apply four sequences of the four ordering strategies in the experiment. Nonetheless, as mentioned in Chapter 4, there are 24 possible sequences out of the four strategies. Conducting a study with all possible sequences would provide insights into the effects of method ordering strategies on each other with regards to the position within the sequence. Combining this approach with the expansion mentioned at the start of this section, future research can cover all possible combinations with sufficient participants.

Lastly, it would also be interesting to explore the effect of the method ordering strategy on the cognitive load via fMRI or EEG. For example, the spatial localization provided by fMRI might offer mappings of method ordering strategies to specific regions in the brain. Subsequently, future research could relate method ordering strategies to cognitive processes such as memory or attention, which would help identify suitable ordering strategies based on the strengths and weaknesses of programmers. Additionally, the eye tracking methodology can be expanded upon to include more metrics and a larger pool of participants to yield significant eye movement results.

Appendix

A.1 Code Snippets

Listing A.1: The warmup code snippet used at the start of the experiment.

```
public static boolean taskWarmUp(int[] input) {
    boolean understood = true;
    for (int i = 0; i < input.length - 1; i++) {
        if (input[i] > input[i + 1]) {
            understood = false;
            break; }}
    return understood; }

public static void main(String[] args) {
    int[] input = {1, 2, 3, 4};
    boolean result = taskWarmUp(input);
    System.out.println(result); }
```

Listing A.2: "Palindrome" code snippet used in the subjective preference [1]. The snippet is ordered with StCS.

```
public class LowestBasePalindrome {

    public static boolean isPalindromicInBase(int number, int base) {
        checkNumber(number);
        checkBase(base);
        if (number <= 1) { return true; }
        if (number % base == 0) { return false; }
        return isPalindromic(computeDigitsInBase(number, base)); }

    public static int lowestBasePalindrome(int number) {
        int base = 2;
        while (!isPalindromicInBase(number, base)) { ++base; }
        return base; }

    public static void main(String[] args) {
        System.out.println(LowestBasePalindrome.isPalindromicInBase(2222, 4));
        System.out.println(LowestBasePalindrome.lowestBasePalindrome(2222222)); }

    private static void checkBase(int base) {
        if (base <= 1) { throw new IllegalArgumentException("base must be greater than
             1."); }}

    private static void checkNumber(int number) {
        if (number < 0) {
            throw new IllegalArgumentException("number must be nonnegative."); }}

    private static ArrayList<Integer> computeDigitsInBase(int number, int base) {
        var result = new ArrayList<Integer>();
        while (number > 0) {
            result.add(number % base);
            number /= base; }
        return result; }

    private static boolean isPalindromic(ArrayList<Integer> list) {
        for (int pos = 0; pos < list.size() / 2; ++pos) {
            if (list.get(pos) != list.get(list.size() - 1 - pos)) { return false; }}
        return true; }

}
```

Listing A.3: "Palindrome" code snippet used in the subjective preference [1]. The snippet is ordered with CaS.

```
public class LowestBasePalindrome {

    public static void main(String[] args) {
        System.out.println(LowestBasePalindrome.isPalindromicInBase(2222, 4));
        System.out.println(LowestBasePalindrome.lowestBasePalindrome(2222222)); }

    public static int lowestBasePalindrome(int number) {
        int base = 2;
        while (!isPalindromicInBase(number, base)) { ++base; }
        return base; }

    public static boolean isPalindromicInBase(int number, int base) {
        checkNumber(number);
        checkBase(base);
        if (number <= 1) { return true; }
        if (number % base == 0) { return false; }
        return isPalindromic(computeDigitsInBase(number, base)); }

    private static void checkNumber(int number) {
        if (number < 0) {
            throw new IllegalArgumentException("number must be nonnegative."); }}

    private static void checkBase(int base) {
        if (base <= 1) { throw new IllegalArgumentException("base must be greater than
             1."); }}

    private static boolean isPalindromic(ArrayList<Integer> list) {
        for (int pos = 0; pos < list.size() / 2; ++pos) {
            if (list.get(pos) != list.get(list.size() - 1 - pos)) { return false; }}
        return true; }

    private static ArrayList<Integer> computeDigitsInBase(int number, int base) {
        var result = new ArrayList<Integer>();
        while (number > 0) {
            result.add(number % base);
            number /= base; }
        return result; }

}
```

Listing A.4: "Palindrome" code snippet used in the subjective preference [1]. The snippet is ordered with CoS.

```
public class LowestBasePalindrome {

    public static void main(String[] args) {
        System.out.println(LowestBasePalindrome.isPalindromicInBase(2222, 4));
        System.out.println(LowestBasePalindrome.lowestBasePalindrome(2222222)); }

    private static void checkBase(int base) {
        if (base <= 1) { throw new IllegalArgumentException("base must be greater than
             1."); }}

    private static void checkNumber(int number) {
        if (number < 0) {
            throw new IllegalArgumentException("number must be nonnegative."); }}

    private static ArrayList<Integer> computeDigitsInBase(int number, int base) {
        var result = new ArrayList<Integer>();
        while (number > 0) {
            result.add(number % base);
            number /= base; }
        return result; }

    private static boolean isPalindromic(ArrayList<Integer> list) {
        for (int pos = 0; pos < list.size() / 2; ++pos) {
            if (list.get(pos) != list.get(list.size() - 1 - pos)) { return false; }}
        return true; }

    public static boolean isPalindromicInBase(int number, int base) {
        checkNumber(number);
        checkBase(base);
        if (number <= 1) { return true; }
        if (number % base == 0) { return false; }
        return isPalindromic(computeDigitsInBase(number, base)); }

    public static int lowestBasePalindrome(int number) {
        int base = 2;
        while (!isPalindromicInBase(number, base)) { ++base; }
        return base; }

}
```

Listing A.5: "Board" code snippet used in the experiment [35]. The snippet is ordered with StCS.

```
public int currentP = 1;
private int[][] board = new int[3][3];

public static void main(String[] args) {
    Example example = new Example(); // Creates an instance of this class
    example.makeMove(2,0);
    example.makeMove(1,0);
    System.out.println("Player " + example.currentP +
        ((example.isWinner()) ? " Won!" : " Lost!")); }

public boolean isWinner() {
    return checkR() || checkC() || checkD(); }

public void makeMove(int row, int col) {
    board[row][col] = currentP;
    currentP = (currentP == 1) ? 2 : 1;}

private boolean checkC() {
    for (int i = 0; i < 3; i++) {
        if (checkLine(board[0][i], board[1][i], board[2][i])) {
            return true; }}
    return false; }

private boolean checkD() {
    return (checkLine(board[0][0], board[1][1], board[2][2]) ||
            checkLine(board[0][2], board[1][1], board[2][0])); }

private boolean checkLine(int a, int b, int c) {
    return (a != 0 && a == b && b == c); }

private boolean checkR() {
    for (int i = 0; i < 3; i++) {
        if (checkLine(board[i][0], board[i][1], board[i][2])) {
            return true; }}
    return false; }
```

Listing A.6: "Board" code snippet used in the experiment [35]. The snippet is ordered with CaS.

```
public int currentP = 1;
private int[][] board = new int[3][3];

public static void main(String[] args) {
    Example example = new Example(); // Creates an instance of this class
    example.makeMove(2,0);
    example.makeMove(1,0);
    System.out.println("Player " + example.currentP +
        ((example.isWinner()) ? " Won!" : " Lost!")); }

public void makeMove(int row, int col) {
    board[row][col] = currentP;
    currentP = (currentP == 1) ? 2 : 1;}

public boolean isWinner() {
    return checkR() || checkC() || checkD(); }

private boolean checkR() {
    for (int i = 0; i < 3; i++) {
        if (checkLine(board[i][0], board[i][1], board[i][2])) {
            return true; }}
    return false; }

private boolean checkC() {
    for (int i = 0; i < 3; i++) {
        if (checkLine(board[0][i], board[1][i], board[2][i])) {
            return true; }}
    return false; }

private boolean checkD() {
    return (checkLine(board[0][0], board[1][1], board[2][2]) ||
            checkLine(board[0][2], board[1][1], board[2][0])); }

private boolean checkLine(int a, int b, int c) {
    return (a != 0 && a == b && b == c); }
```

Listing A.7: "Board" code snippet used in the experiment [35]. The snippet is ordered with CoS.

```
public int currentP = 1;
private int[][] board = new int[3][3];

public static void main(String[] args) {
    Example example = new Example(); // Creates an instance of this class
    example.makeMove(2,0);
    example.makeMove(1,0);
    System.out.println("Player " + example.currentP +
        ((example.isWinner()) ? " Won!" : " Lost!")); }

private boolean checkC() {
    for (int i = 0; i < 3; i++) {
        if (checkLine(board[0][i], board[1][i], board[2][i])) {
            return true; }}
    return false; }

private boolean checkD() {
    return (checkLine(board[0][0], board[1][1], board[2][2]) ||
            checkLine(board[0][2], board[1][1], board[2][0])); }

private boolean checkR() {
    for (int i = 0; i < 3; i++) {
        if (checkLine(board[i][0], board[i][1], board[i][2])) {
            return true; }}
    return false; }

public void makeMove(int row, int col) {
    board[row][col] = currentP;
    currentP = (currentP == 1) ? 2 : 1;}

private boolean checkLine(int a, int b, int c) {
    return (a != 0 && a == b && b == c); }

public boolean isWinner() {
    return checkR() || checkC() || checkD(); }
```

Listing A.8: "Board" code snippet used in the experiment [35]. The snippet is ordered with RaS.

```
public int currentP = 1;
private int[][] board = new int[3][3];

private boolean checkC() {
    for (int i = 0; i < 3; i++) {
        if (checkLine(board[0][i], board[1][i], board[2][i])) {
            return true; }}
    return false; }

public static void main(String[] args) {
    Example example = new Example(); // Creates an instance of this class
    example.makeMove(2,0);
    example.makeMove(1,0);
    System.out.println("Player " + example.currentP +
        ((example.isWinner()) ? " Won!" : " Lost!")); }

private boolean checkLine(int a, int b, int c) {
    return (a != 0 && a == b && b == c); }

private boolean checkD() {
    return (checkLine(board[0][0], board[1][1], board[2][2]) ||
            checkLine(board[0][2], board[1][1], board[2][0])); }

public void makeMove(int row, int col) {
    board[row][col] = currentP;
    currentP = (currentP == 1) ? 2 : 1;}

public boolean isWinner() {
    return checkR() || checkC() || checkD(); }

private boolean checkR() {
    for (int i = 0; i < 3; i++) {
        if (checkLine(board[i][0], board[i][1], board[i][2])) {
            return true; }}
    return false; }
```

Listing A.9: "Buffer" code snippet used in the experiment [47]. The snippet is ordered with StCS.

```
private final List<Integer> integers = new ArrayList<>();

public static Example initialize(List<Integer> integers) {
    Example example = new Example();
    for (int i : integers) { example.integers.add(i); }
    return example; }

public static void main(String[] args) {
    Example example = Example.initialize(new ArrayList<>(Arrays.asList(2, 3, 9, 1)));
    Example example2 = example.nextExample(4);
    System.out.println(example.next() + " "
            + example.next() + " " + example2.fold()); }

public Integer fold() {
    int i = 0;
    while (hasNext()) { i += next(); }
    return i; }

public Integer next() {
    if (hasNext()) return integers.remove(0);
    else return null; }

public Example nextExample(int size) {
    List<Integer> integers = nextS(size);
    return Example.initialize(integers); }

private boolean hasNext() {
    return !integers.isEmpty(); }

private List<Integer> nextS(int size) {
    List<Integer> integers = new ArrayList<>();
    while (size > 0) {
        if (hasNext()) integers.add(this.integers.remove(0));
        else break;
        size--; }
    return integers; }
```

Listing A.10: "Buffer" code snippet used in the experiment [47]. The snippet is ordered with CaS.

```
private final List<Integer> integers = new ArrayList<>();

public static void main(String[] args) {
    Example example = Example.initialize(new ArrayList<>(Arrays.asList(2, 3, 9, 1)));
    Example example2 = example.nextExample(4);
    System.out.println(example.next() + " "
            + example.next() + " " + example2.fold()); }

public Example nextExample(int size) {
    List<Integer> integers = nextS(size);
    return Example.initialize(integers); }

public static Example initialize(List<Integer> integers) {
    Example example = new Example();
    for (int i : integers) { example.integers.add(i); }
    return example; }

public Integer fold() {
    int i = 0;
    while (hasNext()) { i += next(); }
    return i; }

public Integer next() {
    if (hasNext()) return integers.remove(0);
    else return null; }

private List<Integer> nextS(int size) {
    List<Integer> integers = new ArrayList<>();
    while (size > 0) {
        if (hasNext()) integers.add(this.integers.remove(0));
        else break;
        size--; }
    return integers; }

private boolean hasNext() {
    return !integers.isEmpty(); }
```

Listing A.11: "Buffer" code snippet used in the experiment [47]. The snippet is ordered with CoS.

```
private final List<Integer> integers = new ArrayList<>();

public static Example initialize(List<Integer> integers) {
    Example example = new Example();
    for (int i : integers) { example.integers.add(i); }
    return example; }

public static void main(String[] args) {
    Example example = Example.initialize(new ArrayList<>(Arrays.asList(2, 3, 9, 1)));
    Example example2 = example.nextExample(4);
    System.out.println(example.next() + " "
            + example.next() + " " + example2.fold()); }

private boolean hasNext() {
    return !integers.isEmpty(); }

public Integer next() {
    if (hasNext()) return integers.remove(0);
    else return null; }

public Example nextExample(int size) {
    List<Integer> integers = nextS(size);
    return Example.initialize(integers); }

private List<Integer> nextS(int size) {
    List<Integer> integers = new ArrayList<>();
    while (size > 0) {
        if (hasNext()) integers.add(this.integers.remove(0));
        else break;
        size--; }
    return integers; }

public Integer fold() {
    int i = 0;
    while (hasNext()) { i += next(); }
    return i; }
```

Listing A.12: "Buffer" code snippet used in the experiment [47]. The snippet is ordered with RaS.

```
private final List<Integer> integers = new ArrayList<>();

public Integer next() {
    if (hasNext()) return integers.remove(0);
    else return null; }

public Integer fold() {
    int i = 0;
    while (hasNext()) { i += next(); }
    return i; }

public static void main(String[] args) {
    Example example = Example.initialize(new ArrayList<>(Arrays.asList(2, 3, 9, 1)));
    Example example2 = example.nextExample(4);
    System.out.println(example.next() + " "
            + example.next() + " " + example2.fold()); }

private List<Integer> nextS(int size) {
    List<Integer> integers = new ArrayList<>();
    while (size > 0) {
        if (hasNext()) integers.add(this.integers.remove(0));
        else break;
        size--; }
    return integers; }

private boolean hasNext() {
    return !integers.isEmpty(); }

public static Example initialize(List<Integer> integers) {
    Example example = new Example();
    for (int i : integers) { example.integers.add(i); }
    return example; }

public Example nextExample(int size) {
    List<Integer> integers = nextS(size);
    return Example.initialize(integers); }
```

Listing A.13: "Snake" code snippet used in the experiment [31]. The snippet is ordered with StCS.

```
public char direction = 'R';
private int x = 0;
private int y = 0;

public static void main(String[] args) {
    Example example = new Example();
    example.update();
    example.setDirection('U');
    example.update(); }

public void setDirection(char direction) {
    if (this.direction != getOppositeDir(direction)) {
        this.direction = direction; }}

public void update() {
    move();
    if (checkCollision()) die(); }

private boolean checkCollision() {
    if (x < 0 || x > 20 || y < 0 || y > 20) return true;
    return false; }

private void die() {
    System.out.println("Game Over!"); }

private char getOppositeDir(char dir) {
    if (dir == 'U') return 'D';
    else if (dir == 'L') return 'R';
    else if (dir == 'D') return 'U';
    else return 'L'; }

private void move() {
    if (direction == 'U') y -= 1;
    if (direction == 'D') y += 1;
    if (direction == 'L') x -= 1;
    if (direction == 'R') x += 1; }
```

Listing A.14: "Snake" code snippet used in the experiment [31]. The snippet is ordered with CaS.

```
public char direction = 'R';
private int x = 0;
private int y = 0;

public static void main(String[] args) {
    Example example = new Example();
    example.update();
    example.setDirection('U');
    example.update(); }

public void update() {
    move();
    if (checkCollision()) die(); }

public void setDirection(char direction) {
    if (this.direction != getOppositeDir(direction)) {
        this.direction = direction; }}

private void move() {
    if (direction == 'U') y -= 1;
    if (direction == 'D') y += 1;
    if (direction == 'L') x -= 1;
    if (direction == 'R') x += 1; }

private boolean checkCollision() {
    if (x < 0 || x > 20 || y < 0 || y > 20) return true;
    return false; }

private char getOppositeDir(char dir) {
    if (dir == 'U') return 'D';
    else if (dir == 'L') return 'R';
    else if (dir == 'D') return 'U';
    else return 'L'; }

private void die() {
    System.out.println("Game Over!"); }
```

Listing A.15: "Snake" code snippet used in the experiment [31]. The snippet is ordered with CoS.

```
public char direction = 'R';
private int x = 0;
private int y = 0;

public static void main(String[] args) {
    Example example = new Example();
    example.update();
    example.setDirection('U');
    example.update(); }

private void move() {
    if (direction == 'U') y -= 1;
    if (direction == 'D') y += 1;
    if (direction == 'L') x -= 1;
    if (direction == 'R') x += 1; }

public void setDirection(char direction) {
    if (this.direction != getOppositeDir(direction)) {
        this.direction = direction; }}

private boolean checkCollision() {
    if (x < 0 || x > 20 || y < 0 || y > 20) return true;
    return false; }

private void die() {
    System.out.println("Game Over!"); }

private char getOppositeDir(char dir) {
    if (dir == 'U') return 'D';
    else if (dir == 'L') return 'R';
    else if (dir == 'D') return 'U';
    else return 'L'; }

public void update() {
    move();
    if (checkCollision()) die(); }
```

Listing A.16: "Snake" code snippet used in the experiment [31]. The snippet is ordered with RaS.

```
public char direction = 'R';
private int x = 0;
private int y = 0;

private boolean checkCollision() {
    if (x < 0 || x > 20 || y < 0 || y > 20) return true;
    return false; }

private char getOppositeDir(char dir) {
    if (dir == 'U') return 'D';
    else if (dir == 'L') return 'R';
    else if (dir == 'D') return 'U';
    else return 'L'; }

public static void main(String[] args) {
    Example example = new Example();
    example.update();
    example.setDirection('U');
    example.update(); }

private void die() {
    System.out.println("Game Over!"); }

public void setDirection(char direction) {
    if (this.direction != getOppositeDir(direction)) {
        this.direction = direction; }}

public void update() {
    move();
    if (checkCollision()) die(); }

private void move() {
    if (direction == 'U') y -= 1;
    if (direction == 'D') y += 1;
    if (direction == 'L') x -= 1;
    if (direction == 'R') x += 1; }
```

Listing A.17: "String" code snippet used in the experiment [13]. The snippet is ordered with StCS.

```
public static void main(String[] args) {
    Example example = new Example(); // Creates an instance of this class
    String test = "i did not eat anything today.!:";
    test = example.removeE(test);
    test = example.removeC(test);
    test = example.removeP(test);
    test = example.cap(test);
    System.out.println(test); }

public String cap(String input) {
    if (input == null || input.trim().equals("")) {
        return input; }
    input = input.trim().toLowerCase();
    return input.substring(0, 1).toUpperCase() + input.substring(1); }

public String removeC(String input) {
    return remove(input, ':'); }

public String removeE(String input) {
    return remove(input, '!'); }

public String removeP(String input) {
    return remove(input, '.'); }

private boolean endsWith(String input, char end) {
    return input.charAt(input.length() - 1) == end; }

private String remove(String input, char end) {
    if (input == null || input.trim().equals("")) { return ""; }
    if (endsWith(input, end)) { return input.substring(0, input.length() - 1); }
    return input; }
```

Listing A.18: "String" code snippet used in the experiment [13]. The snippet is ordered with CaS.

```
public static void main(String[] args) {
    Example example = new Example(); // Creates an instance of this class
    String test = "i did not eat anything today.!:";
    test = example.removeE(test);
    test = example.removeC(test);
    test = example.removeP(test);
    test = example.cap(test);
    System.out.println(test); }

public String removeE(String input) {
    return remove(input, '!'); }

public String removeC(String input) {
    return remove(input, ':'); }

public String removeP(String input) {
    return remove(input, '.'); }

private String remove(String input, char end) {
    if (input == null || input.trim().equals("")) { return ""; }
    if (endsWith(input, end)) { return input.substring(0, input.length() - 1); }
    return input; }

private boolean endsWith(String input, char end) {
    return input.charAt(input.length() - 1) == end; }

public String cap(String input) {
    if (input == null || input.trim().equals("")) {
        return input; }
    input = input.trim().toLowerCase();
    return input.substring(0, 1).toUpperCase() + input.substring(1); }
```

Listing A.19: "String" code snippet used in the experiment [13]. The snippet is ordered with CoS.

```
public static void main(String[] args) {
    Example example = new Example(); // Creates an instance of this class
    String test = "i did not eat anything today.!:";
    test = example.removeE(test);
    test = example.removeC(test);
    test = example.removeP(test);
    test = example.cap(test);
    System.out.println(test); }

public String removeC(String input) {
    return remove(input, ':'); }

public String removeE(String input) {
    return remove(input, '!'); }

public String removeP(String input) {
    return remove(input, '.'); }

public String cap(String input) {
    if (input == null || input.trim().equals("")) {
        return input; }
    input = input.trim().toLowerCase();
    return input.substring(0, 1).toUpperCase() + input.substring(1); }

private boolean endsWith(String input, char end) {
    return input.charAt(input.length() - 1) == end; }

private String remove(String input, char end) {
    if (input == null || input.trim().equals("")) { return ""; }
    if (endsWith(input, end)) { return input.substring(0, input.length() - 1); }
    return input; }
```

Listing A.20: "String" code snippet used in the experiment [13]. The snippet is ordered with RaS.

```
public String removeC(String input) {
    return remove(input, ':'); }

public String removeP(String input) {
    return remove(input, '.'); }

public String cap(String input) {
    if (input == null || input.trim().equals("")) {
        return input; }
    input = input.trim().toLowerCase();
    return input.substring(0, 1).toUpperCase() + input.substring(1); }

public static void main(String[] args) {
    Example example = new Example(); // Creates an instance of this class
    String test = "i did not eat anything today.!:";
    test = example.removeE(test);
    test = example.removeC(test);
    test = example.removeP(test);
    test = example.cap(test);
    System.out.println(test); }

private String remove(String input, char end) {
    if (input == null || input.trim().equals("")) { return ""; }
    if (endsWith(input, end)) { return input.substring(0, input.length() - 1); }
    return input; }

private boolean endsWith(String input, char end) {
    return input.charAt(input.length() - 1) == end; }

public String removeE(String input) {
    return remove(input, '!'); }
```

a.2 Snippet and Demographic Questions

Table A.1: Questions for the "Board" snippet. The fourth answer "I do not know" is excluded because of repetition.

Question	1. Answer	2. Answer	3. Answer
What is the output of the main method?	Player 2 Lost!	Player 1 Lost!	Player 2 Won!
What does the checkLine() method check for?	Checks whether a, b, and c are either 1 or 2	Checks whether only a is not equal to 0	Always returns true
How is the player switched?	The main method	The makeMove() method	The player is static

Table A.2: Questions for the "Buffer" snippet. The fourth answer "I do not know" is excluded because of repetition.

Question	1. Answer	2. Answer	3. Answer
What is the output of the main method?	2 3 15	null null 15	2 3 10
Is there a way to add integers to an Example instance?	Yes, by calling nextS()	No, there is not	Yes, by calling nextExample()
What happens if the main method declares example2 by calling Example.initialize similar to example?	The output is "2 3 10"	The output is "2 3 15"	Nothing changes

Table A.3: Questions for the "Snake" snippet. The fourth answer "I do not know" is excluded because of repetition.

Question	1. Answer	2. Answer	3. Answer
What is the output of the main method?	"Game Over!"	Nothing	Two times "Game Over!"
If the movement were to be rendered, where would the starting point be on a grid?	Top-left corner	Centered	Bottom-left corner
What happens if the main method sets the direction to 'L' instead of 'U'?	An exception is raised	The direction changes to 'L'	The direction stays 'R'

Table A.4: Questions for the "String" snippet. The fourth answer "I do not know" is excluded because of repetition.

Question	1. Answer	2. Answer	3. Answer
What is the output of the main method?	I did not eat anything today	I did not eat anything today.!	I Did Not Eat Anything Today
What is the purpose of the remove() method?	Removes all ending characters that equal "end"	Removes only the last character if it is equal to "end"	Removes the last occurrence of "end"
What happens if the main method calls cap() twice?	Nothing	The second word is capitalized	The first word returns to lowercase

Table A.5: The questions and possible answers presented in the demographic questionnaire. The questions are divided into: personal, general programming, and Java-related.

Question	Possible answers
What year were you born?	Between 1924 and 2024
What is your gender?	Male, female, prefer not to say
What is your main occupation at the moment?	Student, employed at the university (no HiWi), no job, other
What is your highest degree?	High school diploma, apprenticeship, bachelor, master, doctorate, no Degree, other
Which semester are you currently in?	0 to 50
How many hours a week do you spend on your own projects?	0 to 168
How do you rate your programming skills compared to your fellow students?	1: very inexperienced to 5: very experienced
How many years of experience do you have with Java?	Zero to hundred
How well do you know Java?	Basic knowledge, project experience, regular, expert
What other programming languages do you know?	Free-response

Bibliography

[1] The Algorithms. *The Algorithms - Java*. https://github.com/TheAlgorithms/Java/blob/master/src/main/java/com/thealgorithms/others/LowestBasePalindrome.java. 2023.

[2] Dionysis Athanasopoulos and Apostolos V. Zarras. "Fine-Grained Metrics of Cohesion Lack for Service Interfaces." In: *2011 IEEE International Conference on Web Services*. 2011, pp. 588–595. DOI: 10.1109/ICWS.2011.27.

[3] M. S. Bartlett. "Properties of Sufficiency and Statistical Tests." In: *Proceedings of the Royal Society of London. Series A, Mathematical and Physical Sciences* 160.901 (1937), pp. 268–282. ISSN: 00804630. URL: \url{http://www.jstor.org/stable/96803} (visited on 03/15/2024).

[4] Jennifer Bauer, Janet Siegmund, Norman Peitek, Johannes C. Hofmeister, and Sven Apel. "Indentation: Simply a Matter of Style or Support for Program Comprehension?" In: *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. 2019, pp. 154–164. DOI: 10.1109/ICPC.2019.00033.

[5] Roman Bednarik. "Expertise-dependent visual attention strategies develop over time during debugging with multiple code representations." In: *Int. J. Hum.-Comput. Stud.* 70 (Feb. 2012), pp. 143–155. DOI: 10.1016/j.ijhcs.2011.09.003.

[6] Benjamin Biegel, Fabian Beck, Willi Hornig, and Stephan Diehl. "The Order of Things: How developers sort fields and methods." In: *2012 28th IEEE International Conference on Software Maintenance (ICSM)*. 2012, pp. 88–97. DOI: 10.1109/ICSM.2012.6405258.

[7] Raymond P.L. Buse and Westley R. Weimer. "Learning a Metric for Code Readability." In: *IEEE Transactions on Software Engineering* 36.4 (2010), pp. 546–558. DOI: 10.1109/TSE.2009.70.

[8] 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: *2015 IEEE 23rd International Conference on Program Comprehension*. 2015, pp. 255–265. DOI: 10.1109/ICPC.2015.36.

[9] Teresa Busjahn, Carsten Schulte, and Andreas Busjahn. "Analysis of code reading to gain more insight in program comprehension." In: *Proceedings of the 11th Koli Calling International Conference on Computing Education Research*. Koli Calling '11. Koli, Finland: Association for Computing Machinery, 2011, pp. 1–9. ISBN: 9781450310529. DOI: 10.1145/2094131.2094133.

[10] K. R. Chowdhary. "On the evolution of programming languages." In: *CoRR* (2020). DOI: \url{https://doi.org/10.48550/arXiv.2007.02699}.

[11] Igor Crk and Timothy Kluthe. "Toward using alpha and theta brain waves to quantify programmer expertise." In: *2014 36th Annual International Conference of the IEEE Engineering in Medicine and Biology Society, EMBC 2014* 2014 (Aug. 2014), pp. 5373–6. DOI: 10.1109/EMBC.2014.6944840.

[12] Sarah Fakhoury, Devjeet Roy, Adnan Hassan, and Vernera Arnaoudova. "Improving Source Code Readability: Theory and Practice." In: *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. 2019, pp. 2–12. DOI: 10.1109/ICPC.2019.00014.

[13] Fewlaps. *PrettyStrings*. https://github.com/fewlaps/PrettyStrings/blob/master/src/main/java/com/fewlaps/prettystrings/PrettyString.java. 2017.

[14] Jack Flynn. *40 Fascinating Mobile App Industry Statistics [2023]: The Success Of Mobile Apps In The U.S.* https://www.zippia.com/advice/mobile-app-industry-statistics. 2023.

[15] Milton Friedman. "The Use of Ranks to Avoid the Assumption of Normality Implicit in the Analysis of Variance." In: *Journal of the American Statistical Association* 32.200 (1937), pp. 675–701. ISSN: 01621459. URL: \url{http://www.jstor.org/stable/2279372}.

[16] Yorai Geffen and Shahar Maoz. "On method ordering." In: *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*. 2016, pp. 1–10. DOI: 10.1109/ICPC.2016.7503711.

[17] Todd Hoff. *C++ Coding Standard: Documentation*. https://possibility.com/Cpp/CppCodingStandard.html. 2008.

[18] H. Hunter-Zinck, A. F. de Siqueira, V. N. Vásquez, R. Barnes, and C. C. Martinez. "Ten simple rules on writing clean and reliable open-source scientific software." In: *PLoS computational biology* 17.11 (2021). DOI: 10.1371/journal.pcbi.1009481.

[19] Jorge E. Ibarra-Esquer, Félix F. González-Navarro, Brenda L. Flores-Rios, Larysa Burtseva, and María A. Astorga-Vargas. "Tracking the Evolution of the Internet of Things Concept Across Different Application Domains." In: *Sensors* 17.6 (2017). DOI: 10.3390/s17061379.

[20] Attila Kővári, Jozsef Katona, and Cristina Pop. "Evaluation of Eye-Movement Metrics in a Software Debugging Task using GP3 Eye Tracker." In: *Acta Polytechnica Hungarica* 17 (Jan. 2020), pp. 57–76. DOI: 10.12700/APH.17.2.2020.2.4.

[21] Ivano Malavolta, Roberto Verdecchia, Bojan Filipovic, Magiel Bruntink, and Patricia Lago. "How Maintainability Issues of Android Apps Evolve." In: *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2018, pp. 334–344. DOI: 10.1109/ICSME.2018.00042.

[22] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Pearson. 2008.

[23] Steve McConnell. *Code Complete, 2nd Edition*. Cisco Press. 2004.

[24] Sun Microsystems. *Code Conventions for the Java Programming Language*. https://www.oracle.com/java/technologies/javase/codeconventions-contents.html. 1999.

[25] Sami Naim. *EyeTracking*. https://gitlab.cs.uni-saarland.de/boc/students/sami-naim-method-ordering. 2024.

[26] ORACLE. *The Java® Language Specification*. https://docs.oracle.com/javase/specs/jls/se7/html/jls-8.html. 2015.

[27] Unaizah Obaidellah, Mohammed Al Haek, and Peter C.-H. Cheng. "A Survey on the Usage of Eye-Tracking in Computer Programming." In: 51.1 (2018). ISSN: 0360-0300. DOI: 10.1145/3145904. URL: \url{https://doi.org/10.1145/3145904}.

[28] Delano Oliveira, Reydne Bruno, Fernanda Madeiral, and Fernando Castor. "Evaluating Code Readability and Legibility: An Examination of Human-centric Studies." In: *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2020, pp. 348–359. DOI: 10.1109/ICSME46990.2020.00041.

[29] Stack Overflow. *Most used programming languages among developers worldwide as of 2023*. https://www.statista.com/statistics/793628/worldwide-developer-survey-most-used-languages/. 2023.

[30] Python. *Python 3.12.2 documentation*. https://docs.python.org/3/tutorial/classes.html. 2024.

[31] Rnickle79. *Snake*. https://github.com/rnickle79/Snake/blob/master/src/Snake.java. 2022.

[32] Tobias Roehm and Walid Maalej. "Automatically detecting developer activities and problems in software development work." In: *2012 34th International Conference on Software Engineering (ICSE)*. 2012, pp. 1261–1264. DOI: 10.1109/ICSE.2012.6227104.

[33] H. Sackman, W. J. Erikson, and E. E. Grant. "Exploratory experimental studies comparing online and offline programming performance." In: *Commun. ACM* 11 (Jan. 1968), pp. 3–11. DOI: 10.1145/362851.362858.

[34] Yui Sasaki, Yoshiki Higo, and Shinji Kusumoto. "Reordering Program Statements for Improving Readability." In: *2013 17th European Conference on Software Maintenance and Reengineering*. 2013, pp. 361–364. DOI: 10.1109/CSMR.2013.50.

[35] Saxenaaviral11. *HactoberFest2023*. https://github.com/Saxenaaviral11/HactoberFest2023/blob/4227d8c76004cfd81f177ce209875bc4fd7f957e/tictactoejava. 2023.

[36] Todd Sedano. "Code Readability Testing, an Empirical Study." In: *2016 IEEE 29th International Conference on Software Engineering Education and Training (CSEET)*. 2016, pp. 111–117. DOI: 10.1109/CSEET.2016.36.

[37] S. S. Shapiro and M. B. Wilk. "An Analysis of Variance Test for Normality (Complete Samples)." In: *Biometrika* 52.3/4 (1965), pp. 591–611. ISSN: 00063444. URL: \url{http://www.jstor.org/stable/2333709} (visited on 03/15/2024).

[38] Zohreh Sharafi, Timothy Shaffer, Bonita Sharif, and Yann-Gaël Guéhéneuc. "Eye-Tracking Metrics in Software Engineering." In: *2015 Asia-Pacific Software Engineering Conference (APSEC)*. 2015, pp. 96–103. DOI: 10.1109/APSEC.2015.53.

[39] 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: *2012 20th IEEE International Conference on Program Comprehension (ICPC)*. 2012, pp. 27–36. DOI: 10.1109/ICPC.2012.6240505.

[40] Janet Siegmund. "Measuring program comprehension with fMRI." In: *Softwaretechnik-Trends Band 34, Heft 2* (2014). URL: \url{https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=36c351cffaa51bc1e4918c1df8147da9dedb00da}.

[41] Janet Siegmund. "Program Comprehension: Past, Present, and Future." In: *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. Vol. 5. 2016, pp. 13–20. DOI: 10.1109/SANER.2016.35.

[42] Lars Stahle and Svante Wold. "Analysis of variance (ANOVA)." In: *Chemometrics and Intelligent Laboratory Systems* 6.4 (1989), pp. 259–272. ISSN: 0169-7439. DOI: \url{https://doi.org/10.1016/0169-7439(89)80095-4}. URL: \url{https://www.sciencedirect.com/science/article/pii/0169743989800954}.

[43] Yahya Tashtoush, Zeinab Odat, Izzat Alsmadi, and Maryan Yatim. "Impact of Programming Features on Code Readability." In: *International Journal of Software Engineering and Its Applications* 7 (Nov. 2013), pp. 441–458. DOI: 10.14257/ijseia.2013.7.6.38.

[44] 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 2006 Symposium on Eye Tracking Research & Applications*. ETRA '06. San Diego, California: Association for Computing Machinery, 2006, pp. 133–140. ISBN: 1595933050. DOI: 10.1145/1117309.1117357.

[45] A. Von Mayrhauser and A.M. Vans. "Program comprehension during software maintenance and evolution." In: *Computer* 28 (1995), pp. 44–55. DOI: 10.1109/2.402076.

[46] Larry LeRoy Yourdon Edward; Constantine. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Yourdon Press. 1979.

[47] Yoyosource. *YAPI*. https://github.com/yoyosource/YAPI/blob/master/src/main/java/yapi/datastructures/IntegerBuffer.java. 2021.

[48] Shehnaaz Yusuf, Huzefa Kagdi, and Jonathan I. Maletic. "Assessing the Comprehension of UML Class Diagrams via Eye Tracking." In: *15th IEEE International Conference on Program Comprehension (ICPC '07)*. 2007, pp. 113–122. DOI: 10.1109/ICPC.2007.10.