Bachelor's Thesis

THE EFFECT OF COMMENTS ON PROGRAM COMPREHENSION: AN EYE-TRACKING STUDY

YOUSSEF ABDELSALAM

February 13, 2024

Advisors:

Dr. Norman Peitek	Chair of Software Engineering
Annabelle Bergum	Chair of Software Engineering

Examiners:

Prof. Dr. Sven Apel	Chair of Software Engineering
Prof. Dr. Vera Demberg	Chair of Computer Science and Computational Linguistics

Chair of Software Engineering
Saarland Informatics Campus
Saarland University

UNIVERSITÄT DES SAARLANDES

بِسْمِ ٱللَّهِ ٱلرَّحْمَٰنِ ٱلرَّحِيمِ

In the name of Allah, the Most Gracious, the Most Merciful

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

Software developers rely heavily on code documentation and comments to understand source code, with program comprehension tasks consuming a significant portion of maintenance time. Despite their importance, the impact of comments on program comprehension remains debated. Our study addresses this gap by investigating the influence of comments on program comprehension.

Employing a mixed-methods approach, we conducted an eye-tracking study involving 20 computer science students to explore the impact of code comments on program comprehension. By analyzing both quantitative and qualitative data, we aimed to comprehensively assess the influence of comments on various aspects of program comprehension. The quantitative data collected consisted of behavioral metrics assessing program comprehension in terms of correctness and response time, along with gaze data providing insights into visual attention, linearity of reading order, and gaze strategies. This was complemented by participants' subjective ratings on the perceived difficulty and contribution of comments. Additionally, participants' experiences were gathered through a post-questionnaire, enriching the analysis with qualitative insights into the effectiveness of comments, navigation strategies, and overall experiences with comments.

Findings revealed that the effect of comments on enhancing comprehension varied substantially across different code snippets, with effects ranging from a decrease of 30% to an increase of 34%. While comments were observed to significantly guide visual attention, accounting for up to 23% of all fixations, and promote a more linear reading approach, participants predominantly adhered to a "code-first" strategy, prioritizing code before considering comments. Moreover, comments were consistently rated positively for clarifying complex segments of code and contributing to program comprehension. However, this favorable perception did not consistently translate into improved performance or reduced perceived difficulty across snippets. This discrepancy between perceived and actual contribution highlights the necessity of prioritizing quantitative metrics over subjective viewpoints when considering strategic commenting practices.

We propose avenues for future research, including comparative studies on automated versus human-generated comments and the development of predictive models for assessing comment usefulness. Additionally, we highlight the potential for generative commenting systems within development environments, capable of generating tailored comments based on individual programmer needs, such as their task context, historical interaction patterns, and code complexity.

ACKNOWLEDGMENTS

First and foremost, I offer my sincere gratitude and praise to God Almighty, for His guidance and blessings throughout every stage of this journey, allowing me to successfully complete this thesis.

Furthermore, I would like to take this opportunity to extend my gratitude to all those who have played integral roles in the successful completion of this endeavor.

I would like to express my deepest appreciation to Dr. Norman Peitek for his constant support and mentorship throughout this research journey. His guidance, dedication and invaluable feedback greatly enhanced the quality of this work. I'd also like to thank Dr. Peitek for generously providing the initial code base that served as a fundamental starting point for my thesis.

My gratitude extends to Annabelle Bergum for sparking my interest and excitement for this research topic. Her initial encouragement and continued support were the very reason for starting this entire research.

I am immensely grateful to Prof. Dr. Sven Apel for his confidence in my potential and his support in guiding my academic journey. His expertise and guidance have not only inspired me but also provided me with great opportunities.

I would like to extend my sincere thanks to Prof. Dr. Vera Demberg for reviewing my thesis. I am honored and truly appreciate the time and effort she is dedicating to this task.

I am also thankful to everyone at the Software Engineering chair for always lending a helping hand. Their assistance and insights have been invaluable for conducting this research.

Special thanks to Friederike Repplinger for her help with administrative hurdles and regulations. Her understanding and reassurance were a constant source of relief.

I also wish to express my appreciation to everyone who participated in my study and made this research possible.

On a personal note, I am extremely grateful to my parents and sister, whose sacrifices have been the backbone of my achievements. Their support, patience, and confidence in me have fueled my determination at every turn and were a constant source of my motivation.

Finally, I would like to thank my friends, for their encouragement and support through all the ups and downs of this journey. Their companionship has made this experience truly memorable and enjoyable.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACRONYMS

AOI	Area of Interest	13
CM	Comments Missing	26
CP	Comments Present	26
EEG	Electroencephalography	13
FDR	False Discovery Rate	46
fMRI	Functional Magnetic Resonance Imaging	13
fNIRS	Functional Near-Infrared Spectroscopy	13
LME	Linear Mixed-Effects Model	27
N-W	Needleman-Wunsch	19
OGAMA	OpenGazeAndMouseAnalyzer	41
RTGCT	Radial Transition Graph Comparison Tool	41

1 INTRODUCTION

Software developers spend a considerable amount of time reading and comprehending source code. According to Dunsmore and Wood [32], program comprehension tasks have been reported to take up as much as 60% of the total maintenance time. During that process, developers frequently rely on code documentation and comments, which inform about the code by clarifying its purpose, functionality, and design reasons.

Comments can play a crucial role in maintaining code quality and readability, and are suggested to also serve as "beacons" (i. e., guiding signals highlighting and drawing attention to specific elements) thus aiding in comprehension and analysis of complex code. However, their impact on program comprehension, linearity and gaze strategy still remains a subject of debate. As suggested by several studies, comments can help programmers understand complex code more easily while decreasing their cognitive load [60]. On the other hand, comments can also be a source of distraction, especially if they are excessive, irrelevant, or obsolete, as previous studies have found that extensive commenting can negatively affect program comprehension, leading to longer reading times and potentially reducing the programmer's productivity [67].

Despite the importance of documentation in software development, limited empirical research has been conducted to establish how software developers utilise code documentation, especially comments, for program comprehension. Analysing how programmers commonly approach information exploration during software development and maintenance is needed, if such comprehension is to be facilitated by tool developments or adjustments to the software development process (e. g., guidelines on writing comments).

Our study contributes to the discourse on program comprehension by utilizing eye tracking to investigate the influence of comments on program comprehension, linearity and gaze strategy. Through analysis of participants' visual attention patterns, we aim to provide valuable insights into the role of comments in shaping cognitive processes.

The utilization of eye tracking enables us to collect data on participants' eye movements, fixations, and gaze patterns, providing objective measurements of visual attention. This data facilitates a quantitative assessment of how comments impact participants' visual attention and cognitive load during program comprehension. Our rigorous analysis seeks to improve understanding of whether comments facilitate or hinder program comprehension, as well as potential variations in reading order resulting from the presence or absence of comments. These findings have practical implications for software developers, educators, and researchers, informing code commenting practices and programming guidelines.

The thesis is structured as follows:

BACKGROUND. This section provides a theoretical foundation for our study. It explores relevant literature and theories related to program comprehension and discusses different types of comments, their structure, content, and their relationship with programming language and program size. Additionally, this section introduces the application of eye tracking in software engineering research, highlighting its potential for gathering insights into participants' cognitive processes. The foundations of eye tracking, including its principles and methodologies, are discussed, along with the metrics used to analyze visual attention and cognitive effort. By examining these topics, the background section establishes the necessary groundwork for the subsequent research analysis.

RELATED WORK. This section provides an overview of the existing research on the role of comments in program comprehension. Prior studies have demonstrated the significance of comments in enhancing program comprehension by offering explanations, clarifications, and insights into program functionality. The literature review highlights the gaps and limitations in the current understanding and identifies the need for further investigation. By exploring the previous research, we set the foundation for our study to examine the effects of comments on program comprehension, aiming to contribute to the existing body of knowledge and provide valuable insights for software development practices.

STUDY DESIGN. This section describes the study's design and execution, focusing on outlining the research questions, variables, code snippets, participant selection, use of eye-tracking technology, and ethical considerations.

DATA ANALYSIS METHODOLOGY. In this section, we outline the data analysis procedures for addressing each research question and explore the appropriate statistical tests we used to analyze the collected data.

DISCUSSION. This section interprets the findings of the study in light of the thematic analysis of the qualitative data, situating them within the broader context of software engineering research and previous literature. It discusses the implications of the results for both theory and practice, considering how comments can be more effectively used to support program comprehension.

THREATS TO VALIDITY. This chapter critically examines the potential limitations and biases inherent in the study's design and methodology, with a focus on construct, internal, and external validity.

CONCLUSION. The final section summarizes the key contributions of the thesis to the fields of software engineering and cognitive research. It reiterates the importance of understanding the role of comments in programming and how this knowledge can be applied to improve software development practices. The conclusion additionally outlines directions for future research and emphasizes the ongoing need for research that bridges the gap between theoretical insights and practical applications in software development.

2 BACKGROUND

The objective of the current chapter is to provide the fundamental contextual and background information required to thoroughly understand the significance of our proposed study on the effect of comments on program comprehension, linearity and gaze strategy. Following a discussion of program comprehension, the significance of comments in software development is examined. Furthermore, a review of the incorporation of eye-tracking technology in research related to software development is presented.

2.1 PROGRAM COMPREHENSION

Software development and maintenance rely largely on program comprehension, which includes activities such as software reuse, inspection, evolution, migration, reverse engineering, and reengineering of existing software systems [53]. Considerable research has been focused on investigating the cognitive processes involved in programmers' comprehension of software programs, leading up to the establishment of program comprehension as a distinct field of study within the domain of software engineering. It involves the understanding and interpretation of high-level programming code, which presents challenges due to its intricacies, abstractions, and interdependencies.

The concept of cognitive load, denoting the cognitive effort that is necessary for the processing and comprehension of information, is considered an important aspect of program comprehension. Within this particular context, it is necessary for programmers to identify and extract the relevant parts of the source code, thereby requiring an attentive screening procedure. The program comprehension process, as described by Peitek's work [74], describes the internal cognitive processes that occur when programmers are engaged in reading and comprehending source code.

Programmers use a variety of cognitive strategies to comprehend programs, depending on factors related to the programmer, the program, and the task [89]. The skills, knowledge, and familiarity of the programmer with the program and the domain serve as important maintainer factors. Program factors include the program's domain, size, complexity, quality, and documentation availability. Task factors include the type, size, complexity, time constraints, and environment of the comprehension task. Each of these aspects plays a role in deciding which comprehension method to choose, and can change from one context to the next.

However, there are challenges associated with program comprehension. Programmers frequently struggle with linking different conceptual areas, such as connecting the problem or application domain to the solution, navigating different levels of abstraction, reconciling the system's design or description with its actual implementation, and reconciling the associative nature of human cognition with the formal world of software [85]. Due to these difficulties, an increasing amount of research is being conducted to offer explanations and insights into the cognitive processes involved in program comprehension.

In the following sections, the literature review will explore some of the research topics associated with program comprehension, such as mental models, knowledge domains, and cognitive processes.

2.1.1 *Mental Model*

The notion of the mental model in program comprehension was initially put forward by Storey et al. [113]. They defined it as the maintainer's mental representation of the program. This representation is constructed through observation, inference, or interaction with the program [89]. The accuracy and comprehensiveness of a mental model's development can fluctuate based on variables such as the programmer's level of expertise and the complexity of the program. According to von Mayrhauser and Vans [62], a mental model is composed of static and dynamic elements. These components serve as a framework for organizing and categorizing program information within the model. Program comprehension is further aided by cognitive enablers, such as strategies and techniques that facilitate the formation and manipulation of mental models.

Static Elements

Although the structure of a mental model may differ among individual programmers, certain entities always appear in each mental model. These entities, referred to as static elements, include text structures, chunks, plans, and hypotheses. The following subsections describe these static elements in more detail.

TEXT STRUCTURES According to von Mayrhauser and Vans [62], the concept of "Text Structures" refers to both the textual representation and structural organization of a program. This involves the enumeration, sorting, and ordering of the instructions that make up the organization of the program [12].

CHUNKS The concept of "chunks" has been introduced by several researchers [25, 56]. These are textual structures that represent varying levels of abstraction. A hierarchical structure can be formed by lower-level chunks composing a larger chunk. Each chunk comprises a microstructure formed by statements and instructions that can be abstracted by a macrostructure represented by a label. For instance, a chunk labeled "sort" could represent a text structure associated with a sorting algorithm.

PLANS "Plans" are knowledge elements used to construct and validate expectations, interpretations, and inferences during program comprehension [62]. They can help programmers in inferring or identifying common scenarios within the program. Plans can be classified as either programming or domain plans. Plans for programming include generic objects and particular programming-related procedures or operations. On the contrary, domain plans are related to the problem or application domain of the program and encompass objects and operations that are specific to that domain.

HYPOTHESES Introduced by Letovsky [57], hypotheses are the programmer's educated guesses or reasonable deductions about how the code will behave. The hypotheses can be broadly categorized into three major types: "why" hypotheses, which offer conjectures regarding the purpose of program entities; "how" hypotheses, which suggest means by which objectives are achieved in the program; and "what" hypotheses, which relate to the categorization of program entities, such as variables and functions.

Dynamic Elements

The mental model's dynamic elements are linked to the strategies used by programmers to make sense of the code. Strategies refer to the systematic sequence of actions and procedures that are employed to reach the desired level of comprehension. Littman et al. [59] distinguished between two broad categories of strategies: systematic and as-needed. In systematic approaches, programmers strive to comprehend the entire program prior to performing maintenance tasks. As-needed strategies, on the other hand, focus on comprehending only the specific program parts that need maintenance. A second classification of strategies includes shallow and deep reasoning. In shallow reasoning, important lines of code are rapidly scanned and beacons are utilized to identify mental model plans. On the other hand, deep reasoning requires a comprehensive program synthesis and the development of causal links between functions and objects. This method is generally used when the programmer has little to no experience with the program. Chunking and cross-referencing, two of the mental model's dynamic components, are essential to the comprehension process.

CHUNKING "Chunking", which builds upon the concept of "chunks", is the process of grouping and aggregating lower-level chunks to form higher-level abstractions. By employing chunking, programmers can represent a program in a more abstract way, simplifying its complexity. Instead of dealing with the intricate details of individual statements and instructions, they can focus on the larger chunks that encapsulate multiple operations or functionalities allowing for a more efficient and intuitive understanding of the program's structure and behavior.

CROSS-REFERENCING The process of connecting multiple levels of abstraction within a program, such as linking distinct parts of the program to their respective functional descriptions, is known as "cross-referencing". This procedure also facilitates establishing relationships and gaining a comprehensive understanding of the program.

Cognitive Enablers

Elements that aid in comprehension are known as cognitive enablers. Beacons and rules of discourse are two types of cognitive enablers. Beacons [20, 125] are indicative markers embedded within a program that serve as cues for specific structures or operations. They might be useful for finding code patterns or specific functions that developers need. The rules of discourse, as defined by Soloway and Ehrlich [108], specify programming conventions such as standard algorithms, data structure implementations, and coding standards. These rules direct programmers in their interpretation and comprehension of the program.

In general, a programmer's comprehension of a program is represented by their mental model. It includes both static elements, such as text structures, chunks, plans, and hypotheses, and dynamic elements, such as chunking, cross-referencing, and strategies. Cognitive enablers, such as beacons and rules of discourse, play a significant role in enhancing the comprehension process by providing supplementary guidance and information.

2.1.2 *Knowledge Domains*

The comprehension of programs is significantly influenced by knowledge domains, which function as fundamental units for comprehending programs. A knowledge domain is a finite collection of primitive objects, the interactions between them, and the operators that can change those relationships or attributes [20]. Program comprehension requires bridging and mapping between different knowledge domains, especially between the problem domain (the real-world problem the program is intended to solve) and the program domain [124].

The problem domain, also known as the application domain [19, 91], refers to the specific part of the real world that a program aims to address and solve problems within [86]. It is characterised by a shared vocabulary, common assumptions, solution approaches, and an existing body of literature independent of the programs that tackle these problems.

The program domain [19], focuses on the knowledge domain associated with programming and the source code level. It encompasses both general programming concepts and language-specific concepts, reflecting the chosen programming paradigm (object-oriented, imperative, functional, etc.). Understanding the program domain is crucial for comprehending programs and their underlying code structures.

2.1.3 *Cognitive Models*

Cognitive models, like mental models, play a vital role in understanding program comprehension processes. While mental models specifically focus on the maintainer's representation of the program, cognitive models delve into the broader cognitive aspects involved in comprehending software. These models offer valuable insights into how programmers acquire, organize, and apply knowledge about programs, highlighting the underlying thought processes at work. By examining different cognitive frameworks within the discipline, we can gain a deeper understanding of the cognitive mechanisms and strategies employed during program comprehension. The following sections outline a number of notable cognitive frameworks within the discipline.

BROOKS MODEL: The Brooks model [20], suggests that program comprehension involves mapping between the problem domain (the real-world problem the program aims to solve) and the program domain (the program's code). The model emphasises the iterative construction and reconstruction of knowledge through mappings across various domains. The programmer's expertise in the problem domain is crucial in defining initial hypotheses, which are then refined using a top-down approach. The refinement process involves validating these hypotheses, often aided by beacons present in the source code.

SOLOWAY AND EHRLICH MODEL: The Soloway and Ehrlich model [108] focuses on the impact of rules of discourse and plans on program comprehension, considering both experienced and novice programmers. The model posits that programmers expect programs to consist of plans tailored to the problem domain. Comprehension is influenced by adherence to the rules of discourse. The study conducted by Soloway and Ehrlich demonstrated that expert programmers, with more experience in rules of discourse, exhibited faster comprehension. However, comprehension times were similar to novice programmers when dealing with programs that violated the expected discourse rules.

SHNEIDERMAN AND MAYER MODEL: The Shneiderman and Mayer model [101] views program comprehension as a bottom-up task. It suggests that programmers begin by reading individual code statements and gradually group them into higher-level structures. Comprehension is influenced by two types of programming knowledge: syntactic and semantic. Syntactic knowledge encompasses language-specific details such as keywords, syntax, and library routines, while semantic knowledge includes language-independent programming knowledge and understanding of the problem domain. By employing a bottom-up approach, programmers reconstruct the problem domain by leveraging both types of knowledge.

PENNINGTON MODEL: Pennington's model [77] also adopts a bottom-up approach to program comprehension. It involves the construction of two mental structures: the program model and the situation model. The program model is built first by extracting the sequence of operations and procedures from the program, creating an abstract control-flow representation. This process includes chunking microstructures and establishing cross-references to form higher-level programming plans and macrostructures. The situation model is then constructed bottom-up by mapping the program model to high-level domain plans that describe the program's goals within the problem domain.

LETOVSKY MODEL: Letovsky's model [57] considers program comprehension as a set of three elements: a knowledge base, a mental model, and an assimilation process. The knowledge base, unique to each programmer, contains information about the problem domain, program domain, rules of discourse, and plans. The mental model comprises three layers: the specification layer (problem domain and high-level goals), the implementation layer (program domain), and the annotation layer (links established by the programmer). Comprehension involves an assimilation process that can be either top-down or bottom-up, depending on the programmer's perceived knowledge source at a given moment.

SOLOWAY, ADELSON, AND EHRLICH MODEL: The Soloway, Adelson, and Ehrlich model [107] presents comprehension as a top-down task that involves external representations (documents, requirements, etc.), internal representations (mental model), rules of discourse, and plans. The model emphasises the role of plans in comprehension, distinguishing between three types: strategic, tactical, and implementation plans. Starting with an overall understanding of the program's goals, programmers consider the problem domain, external representations, and their expertise. The strategic plan defines the global strategy or algorithm, the tactical plan outlines the general steps of the chosen algorithm, and the implementation plan specifies the data structures, functions, and operations mapped to the source code. Comprehension involves mapping between these plan types, supported by rules of discourse and beacons.

VON MAYRHAUSER AND VANS MODEL: The von Mayrhauser and Vans model [63] incorporates elements from both the Pennington model (top-down) and the Letovsky model (knowledge base). It suggests that programmers switch between top-down and bottom-up approaches based on the program's familiarity. When the program and beacons are recognised, programmers tend to use a top-down approach. If the code remains unfamiliar, they invoke the program model to obtain a control-flow abstraction. As the program model is built, programmers map it into higher-level structures to form the situation model, completing the comprehension process.

These models offer diverse viewpoints on program comprehension, highlighting various aspects such as expertise, adherence to discourse rules, syntax, semantics, top-down or bottom-up strategies, and the interdependence between the problem domain and program domain. The various models presented in this study provide significant contributions towards comprehending the cognitive processes involved in program comprehension. These insights are of great benefit to both researchers and practitioners, as they facilitate a better understanding of this crucial aspect of programming and pave the way for its improvement.

2.1.4 *Summary*

Program Comprehension is a field of study in software engineering that focuses on how programmers understand programs. It is particularly important in the context of Software Maintenance, where efficient program comprehension is crucial. Programmers employ different cognitive strategies based on factors such as their own expertise, the program being understood, and the specific task at hand. The process of program comprehension involves building and updating a mental model, which is a mental representation of the program. This mental model is constructed using static and dynamic elements, as well as cognitive enablers.

Programmers also bring their own personal knowledge, which is represented in the form of a knowledge domain. The problem domain represents knowledge related to the problems the program aims to solve, while the program domain represents knowledge about programming in general and the programming language used in the program.

Cognitive models are proposed to explain the cognitive processes involved in program comprehension. These models can be classified into top-down, bottom-up, and hybrid models, each with its own characteristics and approaches.

Overall, program comprehension is a complex process that has been extensively studied in order to enhance the understanding of programs and improve software maintenance practices.

2.2 COMMENTS IN SOFTWARE DEVELOPMENT

Comments within source code are a crucial component for developers to effectively comprehend code, facilitating its modification and maintenance. The comprehension of source code holds significant importance for developers, as highlighted by Knuth [52], Standish [112], Tiarks [120], and Siegmund [102], despite its primary function of being executed by computers. The inclusion of comments in code has been shown to enhance the comprehension of said code, as shown by studies conducted by Elshoff and Marcotty [34] and Corazza et al. [24]. Nevertheless, in practical application, comments frequently prove to be inadequate. It has been observed that the practice of leaving new code without comments and neglecting to update existing comments in tandem with code modifications is prevalent [38, 48, 109]. In order to tackle this matter, scholars have put forth methodologies for identifying outdated comments, as seen by the works of Tan et al. [117], Sridhara [110], and Ratol [81]. However, the impact of comments on program comprehension remains uncertain and therefore requires empirical scrutiny. Some argue for self-explanatory code, emphasizing the creation of code that is easily understandable without relying heavily on comments, as advocated by Extreme Programming (XP).

Despite the importance of comments in understanding programs, their study and analysis in the literature have been relatively limited. In this chapter, we provide an extensive exploration of comment characteristics as fundamental elements of source code documentation, drawing from existing literature.

2.2.1 *Types and Structure of Comments*

Comments in source code can be classified into two types: inline comments and block comments. Inline comments consist of a single line, while block comments can span one or more lines. Each programming language may support one or both types of comments [122]. The structure of comments consists of three elements: comment extent, comment target, and comment category [100].

Comment Extent

The extent of a comment refers to a part of the source code that is considered as a single "chunk" of explanatory text. Although a single explanatory text can span multiple comment tags, the extent of a comment is defined as a sequence of consecutive comment tags that form a continuous text. It represents the entire comment as a whole rather than individual comment tags.

Comment Target

Comments are directed at specific subjects within the code. Comment targets can be specified relative to surrounding syntax elements. There are four types of comment targets identified [100]:

Table 2.1: Types of comment targets.

Target	Description
Left	The comment targets the syntax element that ends immediately before the comment. If there are overlapping elements ending at the same point, the one with the longest span is chosen.
Right	The comment targets the syntax element that starts immediately after the comment. If there are overlapping elements starting at the same point, the longest element is chosen.
Parent	The comment targets the parent element, which is the syntax element containing the comment. This type of target is commonly seen in if statements, where the comment targets the entire then-block.
In-Place	The comment does not describe any code and is considered to have the comment itself as its target, such as metadata of author or copyright notices.

Comment Category

The comment category represents the type of relationship between a comment and its target. Eleven comment categories were identified based on existing comments, providing a way of classifying the types of relationships between comments and their targets [100]:

Table 2.2: Types of comment categories.

Category	Description
Postcondition	Conditions that hold after execution explaining "what" the code does.
Precondition	Conditions that hold before execution, including statements that hold regardless of code execution, explaining "why" the code is needed.
Value Description	Phrases that can be equated with variables, constants, or expressions.
Instruction	Instructions for code maintainers, often denoted as `TODO` comments.
Guide	Guides for code users, distinct from instructions.
Interface	Descriptions of functions, types, classes, or interfaces.
Meta Information	Meta information such as author, date, or copyright.
Comment Out	Code that has been commented out, lacking a specific target.
Directive	Compiler directives not intended for human readers.
Visual Cue	Text inserted for ease of reading, such as indentation or section headers.
Uncategorized	All other comments that do not fit into the above categories.

2.2.2 *Content*

The content of comments has been a subject of debate and scrutiny within the software development community. Determining what constitutes a good or bad comment can be subjective, but there are certain characteristics that can help differentiate them. A bad comment is one that is inconsistent with the code it is commenting on, leading to confusion or misleading information for the reader [116]. Researchers have proposed taxonomies to classify comments based on their content [72]. Some of the relevant categories and subcategories identified in their taxonomy are:

Table 2.3: Types of comment contents.

Content	Description
Type	This category includes subcategories such as Unit and IntRange, which provide information about the type of functionality or range of values associated with the code.
Interface	This category includes subcategories like ErrorReturn, which describes how errors are handled in the code.
Code Relationship	This category includes subcategories like DataFlow and ControlFlow, which describe the relationship and flow of data or control within the code.
PastFuture	This category includes subcategories like `TODO` or `FIXME`, indicating tasks or issues that need attention or further work.
Meta	This category includes comments that provide meta-information, such as copyright notices, authors, dates, and other relevant details.
Explanation	This category includes comments that do not fit into the previous categories and provide additional explanations or clarifications about the code.

2.2.3 *Comments and Language*

Comments in source code use a sublanguage that consists of a limited vocabulary and specific syntactic and semantic constructions. The sublanguage of comments tends to be repetitive and has certain characteristics, such as the use of the present tense, indicative or imperative mood, and a limited set of verbs [35]. The terms used in comments have also been studied extensively. In a study by researchers [42], they examined the presence of problem domain terms in comments and identifiers of various programs. The study involved creating a list of problem domain terms relevant to the programs under investigation and measuring the percentage of terms appearing in comments versus identifiers. The results showed that approximately 23% of the problem domain terms appeared in comments alone, while only 11% appeared in identifiers.

2.2.4 *Comments and Program Size*

Maintaining and updating comments throughout the evolution of source code can be challenging. Studies have shown that comments and source code rarely co-evolve over time, and comment changes are primarily triggered by changes in the corresponding source code [38]. However, another study found that the percentage of commented functions remained consistent throughout the evolution [48].

2.2.5 *Comment Density and Practice*

The use and amount of comments in source code have been subjects of ongoing debate, with different software communities and organizations adopting varying practices. While some developers prefer extensive commenting to document their code comprehensively, others opt for self-explanatory code with minimal comments. One study [37] investigated comment density, which refers to the ratio of comment lines to total lines of code, across multiple software projects. The research found that comment density varied significantly among projects, ranging from less than 1% to over 50%.

Additionally, coding style guides and conventions often play a role in influencing commenting practices within development teams. These guidelines provide recommendations and standards for writing comments, aiming to improve code readability and maintainability. The Google Java Style Guide and the Oracle Java Code Conventions are examples of widely adopted coding style guides that provide recommendations on comment usage.

Overall, the content, language, size, and practices related to comments in software development play an important role in program comprehension and maintainability. While there are differing opinions on the ideal amount and style of commenting, it is generally agreed that well-written and informative comments can enhance the readability and comprehensibility of source code.

2.3 EYE TRACKING IN SOFTWARE ENGINEERING RESEARCH

Eye tracking is a method for gathering information about a participant's visual attention by observing eye gaze patterns [31, 82]. Visual attention is essential to the cognitive processes of understanding and problem-solving, and these same cognitive processes direct visual attention to particular regions. As a result, eye tracking is useful in analyzing participants' cognitive processes and effort during software engineering activities [31]. By understanding how eye tracking provides insights into cognitive processes, we can explore its applications in software engineering research and how it contributes to understanding various aspects of software development, such as source code reading, debugging, comprehension of software artifacts, and software traceability [94].

Eye trackers have undergone significant advancements, transforming from intrusive, costly, and challenging-to-use tools into versatile devices with widespread applications across diverse research domains [11]. These applications span various fields, including software development [94], driver-vehicle interfaces [131], airplane cockpits [30], and gaming [114].

By capturing and analyzing the visual information interactions of participants, eye trackers enable researchers to gather substantial and meaningful data, facilitating the examination of reading patterns [83], identification of visual indicators during search tasks [28], and exploration of interactions and engagement during spoken dialogue [8].

In the field of software engineering, eye tracking finds application in tasks such as source code reading, debugging, comprehension of software artifacts, and software traceability [94]. Recent studies have also combined eye tracking with other neuroimaging and biometric techniques, such as Electroencephalography (EEG), Functional Magnetic Resonance Imaging (fMRI), and Functional Near-Infrared Spectroscopy (fNIRS), to assess task difficulty and cognitive burden [36, 39, 55, 76]. However, the wide variety of eye-tracking devices, techniques, metrics, and analyses employed by researchers presents a major challenge, as it complicates the comparability and replication of study approaches, thereby impeding the progress of eye tracking and software engineering research.

To provide a coherent overview of eye tracking in software engineering research, this section will first introduce the foundations of eye tracking. Understanding the basics of eye tracking is crucial for comprehending how it can provide valuable insights into participants' cognitive processes and effort during software engineering activities [31]. We will then delve into the different types of data collected by eye trackers and examine how these data points are aggregated and utilized in research. This will enable us to gain a comprehensive understanding of the valuable information eye tracking can provide in the context of software engineering activities.

2.3.1 *Foundations of Eye Tracking*

A visual stimulus is any object, such as a fragment of source code, that is necessary for carrying out a task. This stimulus activates the cognitive processes of participants, leading to subsequent actions (e. g., altering a statement in the source code). Eye gaze data is analyzed in relation to Areas of Interest (AOIs), which are specific stimulus regions. An AOI's significance depends on the question and participant. In a code editor, for instance, the class annotation may be an irrelevant AOI, whereas the class name may be a relevant AOI.

The analysis of eye gaze data involves the utilization of an event detection algorithm to process the raw data. This processed data can then be classified according to various indicators of ocular behavior, as outlined by Rayner [82] and Duchowski [31].

- Fixation: In the context of visual perception, a fixation denotes a spatially-stable eye gaze that endures for a duration ranging from 100 to 300 milliseconds. During a fixation, the participant's visual attention becomes focused on a particular region of the stimulus, thereby initiating cognitive processes [50]. The duration of fixations can vary based on the given task and characteristics of the participant.
- Saccade: These are rapid, continuous eye movements that occur between fixations and last between 40 and 50 milliseconds. However, they only allow for a restricted visual perception.
- Pupil dilation and constriction: The fluctuation in pupil size, which is regulated by the iris muscle, can serve as an indicator of cognitive effort [79]. An augmented size of the pupil may serve as an indicator of elevated cognitive effort.
- Scan path: The eyes make a series of saccadic movements to fixate successive regions of a stimulus, creating a linear chronological sequence of fixations or visited AOIs.

According to psychological research, the acquisition and processing of information predominantly take place during fixations. It has been observed that participants can effectively comprehend a complex visual stimulus with only a limited number of fixations [80]. Context is crucial when attempting to make sense of fixations. A higher fixation rate on a specific AOI may suggest an increased degree of engagement with its content. Nevertheless, it is worth noting that an accumulation of fixations may also indicate increased effort or difficulty with comprehending the stimulus [79].

Eye Tracker Operation

There exists a wide range of commercially accessible eye trackers that cater to the needs of both the business and scientific communities [94]. The eye trackers under consideration vary in their physical forms and the techniques employed for tracking eye gaze [17]. An eye tracker typically comprises a set of hardware and software components, which collectively enable the measurement and analysis of ocular movements.

- One or more cameras, typically utilizing infrared technology
- One or more light sources, typically emitting infrared radiation
- Image-processing software that detects and locates the eyes and pupils and maps eye motion and stimulus
- Data collection software for eye gaze data collection and storage in real time
- Live visual feedback of where the observer's gaze is currently fixed

The prevailing eye-tracking systems currently in use predominantly employ the corneal-reflection / pupil-center methodology. In this methodology, an emitter releases invisible infrared radiation that is precisely directed towards the eyes, thereby penetrating the pupils. A substantial amount of the incident light is reflected back, resulting in the pupils exhibiting a luminous appearance. Glints may also be seen on the surface of the eyes due to the light that is reflected off them.

The reflections are detected and tracked by cameras, in addition to other notable attributes like the center of the pupil. The image-processing program then determines the eye gaze independent of head location and motion by using calibration, trigonometric calculations, and other modeling approaches [31, 45, 79].

Eye Tracking Assumptions

The correlation between eye gaze and cognitive processing depends on two fundamental assumptions originating from the theory of reading: the immediacy assumption and the eye-mind assumption [50]. The immediacy assumption asserts that as soon as participants encounter a stimulus, such as when a reader reads a word, interpretation of the stimulus starts immediately. The eye-mind assumption postulates that participants focus their attention solely on the stimulus element that is currently being processed.

The aforementioned assumptions serve as the foundation for the representation of participants' cognitive processes through eye gaze data. The analysis of eye gaze data provides valuable insights into participants' focus, cognitive effort, and temporal dynamics involved in processing a given stimulus. Furthermore, based on physiological investigations, psychologists hypothesize that individuals do not have conscious control over many features of their eye gaze, such as pupil size, except for the position of their focus.

Eye Tracking Limitations

Eye trackers possess inherent limitations. Here are a few of the most significant ones at the time of composing this paper, although many of these limitations may decrease or diminish as new technologies and algorithms are developed:

- Accuracy: The concept of accuracy in the context of gaze data analysis pertains to the extent of disparity between the actual gaze data and the recorded gaze data, typically quantified in terms of visual angle degrees [44]. Accuracy ratings for popular eye trackers range from 0.5 to 1 degree. For instance, if the distance between the participant and the stimulus is 50 cm, an eye tracker with an accuracy of 1 degree could locate the participant's eye gaze within a radius of approx. 1 cm of its actual position.

- Precision: Precision refers to an eye tracker's ability to reliably deliver the same data for several, consecutive eye gazes at the same spot. The precision values of commonly used eye trackers span from 0.01 degree to 1 degree.

- Drift: The phenomenon of drift refers to the gradual decline in the accuracy of eye-tracking measurements as they deviate from the actual positions of the fixation. The phenomenon arises as a result of the degradation of calibration caused by various factors, including fluctuations in moisture levels and other physiological characteristics of the eye [66].

- Extrafoveal Vision: Eye trackers generally record fixations at the fovea, which corresponds to the center of vision with the highest sharpness of vision. However, individuals have the ability to perceive and interpret visual stimuli in peripheral regions, despite their gaze not being directly focused on those areas. Nevertheless, extrafoveal vision, which accounts for around 98% of the human visual field, is not recorded by eye trackers.

2.3.2 *Metrics*

Analysis of eye-tracking data presents difficulties in numerous disciplines [45], including software engineering research, especially with regard to program comprehension [7]. In this section, we give definitions and metrics for analyzing eye-tracking data in software engineering research.

First Order Data

The term "first-order data" refers to the raw and unprocessed information obtained from eye-tracking devices [45]. Typically, the following metrics are used:

- X,Y position: The X and Y positions refer to the spatial coordinates of each gaze point, which provide insight into the participants' focal point of attention on the stimulus. Nevertheless, the participants' comprehension of the stimulus remains unknown.
- Pupil diameter: This metric represents the physical measurement of the pupil in millimeters. Pupil size variations are more important than absolute sizes since they change across individuals. Pupil size is correlated with cognitive load and task complexity [6].
- Eye blinks: One indicator of cognitive load is the quantity of blinks per interval of time, such as per minute. Lower blink rates are associated with greater focus [6, 79]. Nevertheless, the inclusion of blink rates in eye-tracking data is not consistently present. Certain eye-tracking devices, such as the Smart Eye trackers, offer the capability to gather blink data. However, in order to achieve real-time blink detection, supplementary techniques such as video-based eye tracking may be necessary [29].

Prior to analysis, it is imperative to perform a thorough cleaning of the X and Y positions, pupil diameters, and eye blinks data. This crucial step is essential due to the presence of various sources of interference, such as noise, outliers, and invalid entries, as highlighted by Soh et al. [104]. Researchers may either visually clean the data by reviewing fixations and saccades and erasing plainly wrong entries, or statistically clean the data by deleting outliers and abnormally extended fixations. Several variables can impact the accuracy and reliability of these metrics, including ambient light levels, the emotional and cognitive states of participants, the distance between the eye tracker and the individual, and the quality of the camera image.

Second Order Data

Fixations and saccades are examples of second-order data, which are derived from first-order data using physiological thresholds. Spatial and temporal criteria are used by eye trackers' event detection algorithms to distinguish between fixations (periods of steady gaze) and saccades (fast eye movements) [88]. However, it is important to recognize that the selection of event detection algorithms can affect the outcomes of data analysis.

Fixations may be either voluntary or involuntary. While involuntary fixations are caused by reflexes, such as the optokinetic reflex, which focuses attention on moving things, voluntary fixations are the deliberate concentration on certain elements. Within the field of software engineering, the primary area of interest for researchers lies in the examination of voluntary fixations. However, it is worth noting that involuntary fixations may also occur, particularly in instances where a window unexpectedly appears to notify the participant of something.

Third Order Data

By analyzing fixations and saccades captured by eye-tracking software, these third-order metrics are derived. Among them are:

- Fixation count: The number of fixations that occurred within a specific AOI or the entire stimulus.
- Fixation duration: Also referred to as fixation time, denotes the aggregate duration including all fixations made on a specific AOI or the stimulus under examination.
- Percentage of fixations or fixation rate: Refers to the proportion of total fixations on a AOI or stimulus in relation to another.
- Time to the first fixation in an AOI: The duration from the start of an experiment until the participant fixates on a given AOI.
- All fixations within a selected time: Includes all AOI and stimuli fixations occurring within a certain time window.

Prior research in the field of eye tracking has employed various metrics such as fixation count, fixation duration, and fixation rate to identify AOIs that cause more attention from individuals [27, 28, 121]. Additionally, these metrics have been utilized to evaluate the efficacy of participants' problem-solving approaches [105]. A lower fixation rate suggests decreased efficacy in search tasks, indicating that participants exert more effort to identify relevant regions [79]. On the contrary, elevated rates of fixation serve as an indication of higher effort necessary for completing different tasks, including finding bugs [8, 96], debugging [97], comprehending source code [14], recalling identifier names [98], or examining various stimulus layouts [41, 130].

In order to make appropriate comparisons between two AOIs or stimuli, it is essential to modify the values based on their proportions. In the context of textual analysis, it is necessary to normalize fixation counts by dividing it by the word count of each AOI. This normalization technique allows for fair and accurate comparisons between AOIs that may have varying word counts.

There is no correlation between fixation counts and durations, as demonstrated by [93]. Prior research has utilized a combination of fixation counts and durations in order to accurately evaluate the level of engagement shown by participants. These include the following metrics:

- Average Fixation Duration (AFD) or Mean Fixation Duration (MFD): the average duration of fixations within an AOI in comparison to fixation counts across all AOIs or the stimulus.
- Ratio of On-target to All-target Fixations (ROAF): the sum of fixation durations within an AOI divided by the total number of fixations across all AOIs or the stimulus. Higher ROAF values indicate increased efficacy and reduced effort.

In order to ensure suitable comparison between stimuli, it is necessary to consider the size of each stimulus. The Normalized Rate of Relevant Fixations (NRRF) was introduced by Jeanmart et al. [46] as a method to compare multiple stimuli. Higher NRRF levels suggest that the related stimulus required more effort to understand.

The third-order metrics provided by saccades are comparable to those provided by fixations and include:

- Saccade count: Total number of saccades within an AOI or in response to the stimulus.
- Saccade duration or saccade time: the duration of all saccades within an AOI or the stimulus.
- Regression rate: the proportion of backward or regressive saccades relative to the total number of saccades (e. g., leftward in left-to-right source code reading) [23, 79].

Research suggests that increased regression rates correspond to an increase in the difficulty of executing and finishing a task [40, 79]. Source code reading has been shown to have greater regression rates than natural language text [23]. Saccades were also used by Fritz et al. [39] to study how stimulus difficulty affected participants.

Fourth Order Data

Scan paths refer to sequences of fixations or AOIs. The analysis of scan paths provides valuable insights into the temporal and spatial characteristics of eye fixations, thereby serving as reliable indicators of search efficiency. Longer scan paths show that participants spend more time and effort studying a stimulus to find relevant AOIs, suggesting less efficient scanning and searching.

Longer experiment sessions result in longer scan paths, which are more difficult to evaluate and compare. Number, position, timing, and length of fixations are all factors that must be taken into account. There are several algorithmic methods that can be employed to analyze scan paths:

- Transition matrix: a tabular representation of the transition frequencies between AOIs. It is possible to compare two transition matrices by dividing the number of nonzero cells by the total number of cells to obtain the matrix density. Increased spatial density indicates a thorough search with ineffective scanning [93].
- Scan path recall, precision, and F-measure: measurements that indicate the relationship between AOIs and scan paths. Scan path recall is determined by dividing the number of fixated relevant AOIs by the total number of relevant AOIs. The precision of the scan path is calculated by dividing the number of fixated relevant AOIs by the total number of AOIs. The F-measure of a scan path is the weighted average of its precision and recall [78].
- Edit distance: the minimal editing cost necessary to turn one scan path into another using simple operations like insertion, deletion, and replacement (Levenshtein's algorithm [58]).
- Sequential Pattern Mining (SPAM): a depth-first algorithm that contrasts scan paths according to fixation locations and durations [4].
- ScanMatch: ScanMatch uses temporal binning to manipulate the length of two or more scan paths depending on fixation durations, and then generates a similarity score for the compared scan paths. It is based on the Needleman-Wunsch (N-W) method, which is used in bioinformatics to compare DNA sequences [26].

Several studies [22, 43, 92, 103] have used scan paths to detect and evaluate the visual processing approaches that participants used while exploring stimuli and completing tasks. They found that individuals with lower edit distance and SPAM levels were more likely to use similar reading approaches.

Other fourth-order data include the following:

- Attention switching: the overall count of attention shifts between AOIs per unit time.
- Fixation Spatial Density (SD): measures the uniformity with which participants' fixations are distributed over the stimuli [40]. When a stimulus is divided into a grid of equal cells, the number of cells that are visited (showing at least one fixation) is used to calculate SD. The less coverage there is, the lower the spatial density rating.

- Convex-hull area: the smallest convex collection of fixations that includes all participants' fixations [40]. A lower score suggests that fixations are concentrated in a narrow region and that participants made less effort to locate significant regions in a stimulus.
- Linearity: the concept of linearity is closely linked to the search strategies employed by participants [79]. In this context, linearity refers to the eye gaze patterns observed during reading, specifically the tendency to move from left to right and top to bottom, which is commonly observed among readers of Latin-based natural languages.

Researchers have used SD and convex-hull area to analyze the areas of interest inside participants' fixations [92, 95, 106], which provides insight into the effectiveness of participants' search techniques and reveals their favored aspects of visual stimuli.

Spatial distribution-based metrics are susceptible to invalid data. For instance, a little shift in the position of a single fixation may have a major impact on the size and contours of the resulting convex hull. As a result, when using fourth-order data, noise reduction and data cleaning are essential [93].

3 RELATED WORK

3.1 EXISTING STUDIES

The role of comments in program comprehension has been studied extensively, and various experiments have been conducted to explore their importance and impact [22, 67, 69, 100, 118, 119, 128]. One early experiment conducted by Woodfield et al. aimed to measure the effect of comments and modularization on program understanding [128]. The experiment involved dividing experienced programmers into groups and providing them with different versions of a Fortran program, some with comments and others without. The results showed that the groups given commented versions were more successful in answering a larger quantity of questions correctly. This indicated that comments alone provided significant help for program comprehension, independent of modularization.

Further experiments were conducted in 1985 by different authors to study the effect of comments and modularization on the readability of the Banker's Algorithm [119]. These experiments followed similar patterns as the previous one, and the results consistently showed that comments improved readability.

A recent experiment focused on understanding the differences between class and method comments [69]. Multiple versions of the same program were created, including versions with both types of comments, only one type of comment, and no comments. Participants had to answer a test quiz about the program. The results supported the findings of previous experiments, showing that the versions with comments were better understood than those without comments. Additionally, regarding the differences between class and method comment understanding, the experiment found that participants who had a method-commented version performed better on the quiz compared to those with a class-commented version [69]. Kernighan and Plauger [51] suggested that the best documentation for a computer program includes enlightening comments. Brooks [19] emphasized the importance of comments as a means of establishing a bridge between different knowledge domains, particularly between the program and problem domain. According to Brooks, programmers should be aware of this and incorporate information in comments that facilitate the establishment of this bridge between the two domains [19].

Several other works have investigated comments for different purposes and examined the effects of various factors on program comprehension. Wong et al. [127] and McBurney and Mcmillan [65] proposed approaches for automatically generating helpful comments to aid in understanding source code. McBurney and McMillan [64] also conducted an experiment to determine the characteristics of "good" comments and found that author-written comments often use keywords from the source code. Buse and Weimer [21] investigated the readability of code comments and developed a corresponding measure that correlates with other quality measures such as code changes and defect reports. Ying et al. [129] found that programmers use comments for internal communication, such as applying `TODO`-comments.

Ali et al. [2] investigated the impact of comments on requirements traceability and found significant effects. Ji et al. [47], Seiler and Paech [90], and Krüger et al. [54] utilized comment-like annotations to integrate feature traceability in the source code and emphasized their benefits. Antoniol et al. [3] described a technique for automatically recovering traceability links between code and documentation by analyzing identifier names. Sridhara et al. [111] proposed a technique for automatically generating summary comments for Java methods to provide up-to-date documentation in natural language.

A recent eye-tracking study [9] revealed interesting insights into the reading patterns of novices and experts when it comes to source code. Novices tend to read method signatures with less attention compared to code lines within the method body. They also show a tendency to transition between all lines of code without recognizing the relevance of each line. In contrast, experts demonstrate the ability to disregard irrelevant lines and focus on the current line being read. Furthermore, novices have a tendency to repeatedly read each line within a loop, while experts concentrate on lines with higher computational complexity in successive iterations.

A study by Adeli et al. [1] explored how providing the right information at the right time and place enhances program comprehension. Using a non-traditional IDE, annotations were implemented to facilitate access to relevant information. A user study with 22 novices showed that this approach improved accuracy and reduced cognitive load during program comprehension tasks without compromising tool usability.

In their study, Shinyama et al. [100] developed a method to identify explanatory code comments that enhance program comprehension. They proposed eleven categories of code comments and used a decision-tree-based classifier to achieve 60% precision and 80% recall. The researchers analyzed 2,000 GitHub projects and found two dominant comment types: preconditional and postconditional. Their findings also revealed consistent grammatical structures in English code comments across different projects.

In an exploratory study by Parkin [73], experienced C programmers performed maintenance tasks on a C program. Contrary to previous research, programmers implementing corrections utilized program documentation and header information more than those working on enhancements. Enhancers made specific use of task documentation to map out extensions and verify code modifications. This study sheds light on the program comprehension strategies employed during different maintenance tasks.

Blinman and Cockburn [16] focused on the effects of naming style and documentation on the comprehensibility of source code. It examines software development frameworks and their impact on developers' usage at the source code level. The research finds that using a descriptive interface naming style positively influences developers' comprehension. Additionally, documentation is shown to be important but increases the time spent studying the source code.

In their study on program comprehension, Busjahn et al. [22] investigated the linearity of reading source code compared to natural language text. They found that novices read source code less linearly than natural language text, and experts read code even less linearly than novices. These results highlight the specific differences in reading approaches between source code and natural language text, suggesting that non-linear reading skills increase with expertise.

Lastly, Nielebock et al. [67] investigated the effectiveness of comments in program comprehension for small programming tasks. They conducted an experiment involving 277 participants, mainly professional software developers, who performed programming tasks on differently commented code. The study aimed to replicate previous findings, examine the performance of participants with varying levels of experience, and explore developers' opinions on comments. The results indicate that comments were considered less important for small programming tasks compared to other mechanisms such as proper identifiers. However, participants acknowledged the necessity of comments in specific situations. The study adds to the existing body of research on the uncertain impact of comments on software development.

3.2 RATIONALE FOR OUR STUDY

The existing experiments and discussions in the literature support the notion that comments play a crucial role in program comprehension [16, 67, 69, 118, 119, 128]. They aid programmers in understanding code by providing additional explanations, clarifications, and insights into the program's functionality, which can significantly improve readability and comprehension. However, despite the existing body of research, there are several reasons that justify the need for this study.

3.2.1 *Addressing Research Gaps*

To address existing research gaps, our study explores three key aspects. First, we employ eye tracking, which allows us to collect precise data on how programmers visually process and comprehend code. This approach offers valuable insights into the cognitive processes involved in program comprehension and the role of comments in guiding attention and facilitating comprehension. Second, we examine the linearity of code reading order and investigate how the presence of comments affects the sequential and orderly reading of code elements. Lastly, we also investigate the employed gaze strategies during code comprehension, specifically focusing on how participants transition their visual attention between code and comments. By focusing on these issues, our study aims to provide a more thorough analysis of the effects of comments on program comprehension, linearity, and gaze strategies. This will contribute to the existing body of knowledge and offer valuable insights for software development practices.

Another critical aspect our study addresses is the isolation of the effect of comments. Previous studies have often considered additional aspects of programs, such as modularity [128] or identifier names [87], which makes isolating the effect of comments challenging. This poses a threat to the internal validity of those studies. Moreover, the measurements used in prior research are often subjective, resulting in limited quantitative results. The potential presence of confounding factors, which may affect program comprehension, linearity, and gaze strategies, makes it difficult to analyze the isolated effect of comments. Therefore, our study fills this gap by conducting research that controls for potential confounding factors, providing a clearer understanding of their influence.

3.2.2 *Enhancing External Validity*

Previous studies were conducted with older programming languages and paradigms [33, 68, 99, 115, 118, 119, 128], which may compromise the applicability of their results in modern software development practices. To enhance the external validity and relevance of the findings, our study investigates the effects of comments on program comprehension and reading order using a contemporary programming language and environment. Consequently, we aim to provide insights directly applicable to present-day software development, improving the generalizability of the findings.

Lastly, replication studies in empirical software engineering are crucial for validating and consolidating existing knowledge [5, 13, 49]. Therefore, we emphasize the importance of conducting further replications to gain a deeper understanding of the effects of comments on program comprehension, linearity, and gaze strategies. By replicating and consolidating previous findings, we can establish a more robust foundation of knowledge regarding the impact of comments, addressing varying and sometimes contradictory results from prior studies. Such replication efforts contribute significantly to the advancement of the field and help establish more reliable and generalizable conclusions about the role of comments in program comprehension and reading strategies.

4 STUDY DESIGN

4.1 AIM

Our study aims to explore the impact of comments on program comprehension, visual attention, linearity of reading order, gaze strategies, and participants' perceptions of the role of comments in program comprehension using a combination of quantitative and qualitative methods. The following sections provide an overview of the research questions, study design, variables and measures, code snippets, participant details, materials, procedures, and ethical considerations. By examining these aspects in detail, this research intends to enhance our understanding of how comments influence program comprehension, reading strategies, and the sequential and orderly reading of code elements.

4.2 RESEARCH QUESTIONS

1. *How do comments affect program comprehension?*
   This research question aims to investigate the impact of comments on program comprehension. It examines how the presence or absence of comments influences participants' understanding and interpretation of the code snippets presented in the study. The effects of comments on program comprehension were assessed through a combination of quantitative and qualitative analyses.
2. *How do comments affect visual attention during program comprehension?*
   This research question aims to examine the influence of comments on the allocation and distribution of visual attention during program comprehension. It investigates whether comments attract or redirect participants' gaze within the code snippets and how they impact participants' eye gaze patterns and fixations. Visual attention, in the context of our study, refers to participants' focus and allocation of attention while observing the code snippets. The effects of comments on visual attention were analyzed using eye-tracking data, providing insights into the role of comments in shaping participants' visual attention during program comprehension.
3. *How do comments affect the linearity of reading order?*
   This research question aims to explore the impact of comments on the linearity of reading order. Linearity refers to the sequential and orderly reading of code elements and can play a significant role in program comprehension [75]. The question seeks to investigate how the presence of comments influences the flow and organization of reading code snippets. By examining participants' reading patterns and comparing the linearity of reading order for snippets with and without comments, this research aims to gain insights into the effects of comments on the structured and sequential understanding of code.

4. *How do comments affect participants' gaze strategies during program comprehension?* Gaze strategies refer to the intentional patterns and decisions participants make when directing their visual attention while observing code elements, including code and comments. In this study, we define and categorize gaze strategies into "code-first," and "comment-first" strategies based on participants' initial area of interest and their subsequent gaze transitions. The research question aims to explore how the presence of comments influences participants' gaze strategies, potentially affecting the prioritization and skipping of certain code elements. By comparing gaze strategies with and without comments, we seek to understand how comments guide attention, impact gaze transitions, and influence the cognitive processes involved in program comprehension.

5. *How do participants perceive the role of comments in facilitating program comprehension?* Understanding participants' subjective perceptions of code snippet difficulty and the role of comments in program comprehension is crucial to gaining a comprehensive understanding of the impact of comments. This research question aims to investigate participants' subjective perspectives on the difficulty of code snippets and the contribution of comments to program comprehension.

These research questions will guide our study methodology and serve as the basis for analyzing the collected eye-tracking data. Further elaboration on the operationalization of these variables are provided in the variables section of this study (4.4), where specific measures and methods for assessing program comprehension, visual attention, reading order, and participants' perceptions are detailed. The research questions provide a clear framework for investigation, allowing for a systematic examination of the effects of comments on program comprehension. However, it is important to acknowledge that limitations or challenges may arise during our study, such as a small sample size or limited generalizability of the results. These potential limitations are discussed in Chapter 8.

4.3 EXPERIMENTAL DESIGN

The study employs a mixed-methods design, integrating quantitative and qualitative approaches to comprehensively investigate the impact of comments on program comprehension, linearity, and gaze strategies. The quantitative aspect involves presenting participants with pre-randomized Java code snippets in two conditions: Comments Missing (CM) and Comments Present (CP). Eye movements were tracked using an eye-tracking device to capture participants' visual attention and gaze data during comprehension, along with recording completion time and correctness. The qualitative component includes a post-questionnaire to rate snippet difficulty and provide subjective views on the role of comments in comprehension.

The study adopts a within-subjects design, which allows for a comprehensive exploration of the effects of comments on each of the dependent variables. Each participant was presented with a set of Java code snippets, with each snippet representing a Java class that compiles and writes to the console. The code snippets were carefully selected to ensure a similar level of code complexity and comment relevance. Further details regarding the code snippet selection process and any necessary modifications are provided in Section 4.5.

The snippet order was randomly assigned to each participant to minimize order effects. A script was used to create a predefined assignment of code snippets, ensuring that each participant encounters a balanced mix of snippets with and without comments. This script also guarantees that each code snippet appears an equal number of times throughout the study (cf. Table A.1).

In a fully crossover design, each participant would experience all conditions with the same set of code snippets, providing a direct comparison of the conditions on the same data points. However, in this partially crossover design, participants may encounter different code snippets under each condition. For example, Participant 1 may encounter "Snippet 3 CP" (Comments Present) but not "Snippet 3 CM" (Comments Missing).

This design choice was made to mitigate potential carryover effects or learning biases that could occur if participants encounter the same code snippets in both conditions. However, it introduces additional challenges in the analysis, as the data points are not fully matched between conditions for each participant. The use of a Linear Mixed-Effects Model (LME) was appropriate for handling the within-subject variability in this partially crossover design. This model allows for the inclusion of both fixed effects (such as the presence or absence of comments) and random effects (to account for individual differences between participants) in the analysis (see Section 5.3).

The experimental layout allows for controlled manipulation of the independent variable (presence or absence of comments) and facilitates the assessment of dependent variables (program comprehension, visual attention, linearity of reading order, and gaze strategies). However, certain limitations apply to the experimental design. As our study was conducted in a controlled setting, generalizing the results to other contexts may be restricted. The design does not account for all factors that can influence program comprehension, such as software design techniques, domain-specific languages, visual code highlighting, static typing, code repetition, identifier names, or developers' memory. However, the selected tasks offer valuable insights. Maintenance tasks often involve understanding and modifying small code sections, and bug fixes typically require minor code modifications. By focusing on simple programs, this investigation provides preliminary information on whether and which comments aid comprehension of such basic code components, specifically single methods with approximately 20 lines of code. This approach serves as an appropriate foundation for determining the role of comments in understanding and working with these code snippets.

4.4 VARIABLES

This study was structured around a single independent variable, namely the presence of comments within code snippets, which was manipulated across two distinct levels: Comments Missing (CM) and Comments Present (CP). This manipulation aimed to explore how the inclusion or exclusion of comments affects program comprehension and navigation among participants. The impact of the independent variable on participants' program comprehension and navigation was assessed through four latent dependent variables, which are outlined as follows:

1. *Program comprehension*:
   This variable was operationalized by measuring participants' completion time and error rates for each code snippet. After reading a code snippet, participants were asked to write the output it produces. Their responses were scored based on the completion time and correctness of their answers, providing an indication of their level of program comprehension.

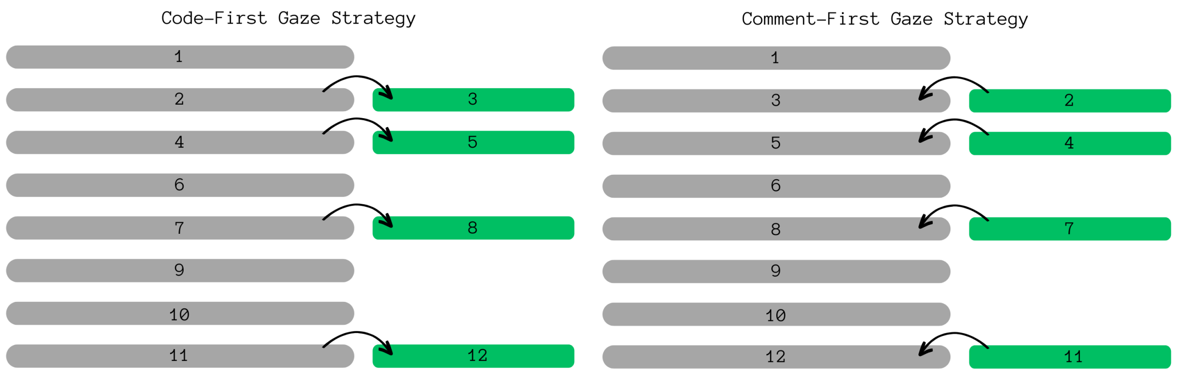
2. *Visual attention*:
   Visual attention refers to participants' focus and allocation of attention during program comprehension. The eye-tracking data provided insights into participants' fixation duration and frequency, indicating their visual attention on different code elements. The code snippets were divided into code AOIs and comment AOIs, allowing for an analysis of the number and duration of fixations on specific elements, such as code lines and comments.

3. *Linearity*:
   Linearity refers to the sequential and orderly reading of code elements. The foundational work by Busjahn et al. [22] as well as subsequent research by Peitek et al. [75] provide many eye-gaze metrics to assess the linearity of reading order. These include local measures, such as vertical next text, vertical later text, horizontal later text, regression rate, line regression rate, and saccade length, as well as global measures such as the N-W Scores of the line reading order compared to the story and execution order. These measures were used to assess the linearity of the participants' reading order and the extent to which their gaze patterns align with the linear text reading order and the source code's execution order.

4. *Gaze Strategies*:
   Gaze strategies refer to the different approaches participants take when directing their gaze during the study. Two variations of gaze strategies were considered based on participants' initial area of interest and their subsequent gaze transitions: *code-first*, and *comment-first*.

   In the code-first strategy (Figure 4.1a), participants predominantly focus their gaze on the code elements of the given snippet, with the initial fixation often landing on a specific code line. Their visual attention is directed towards lines of code, variable names, and control structures, seeking to understand the logic and functionality of the program. Comments, if present, may receive occasional glances, but the primary emphasis is on the code itself.

Contrary to the code-first strategy, the comment-first gaze strategy (Figure 4.1b) places a higher priority on comments rather than code lines. Participants' initial fixation often lands on a specific comment, seeking to gain context, explanations, and insights about the code's purpose and functionality. Code elements are still observed but may receive less attention compared to the comments.



(a) Code-First Gaze Strategy (b) Comment-First Gaze Strategy

Figure 4.1: Illustration of code-first and comment-first gaze strategies.

To compare the measured reading order with the proposed reading orders, two types of measures were used: Locally, code-to-comment and comment-to-code saccades were compared to examine the order in which participants fixate on code elements and comments. Globally, we defined AOI sequences for each of the reading strategies and compared them to the actual reading order using the same N-W algorithm to determine the closest match.

5. *Participant's Perception*:
   The subjective perception of each participant on the snippets and the role of comments in facilitating program comprehension was captured through their likert-scale ratings of the snippet difficulty and comment contribution as well as through evaluating their responses from the post-questionnaire.

The measures used in our study have been previously validated in similar research studies and have demonstrated good reliability and validity. Tasks assessing program comprehension have been widely employed in previous studies on program understanding and have been found to be reliable indicators of participants' comprehension levels. The eye-tracking measures, including fixation duration and frequency, have been extensively used in research on visual attention and reading patterns.

4.5 CODE SNIPPPETS

The code snippets employed in our study were carefully selected to meet specific criteria concerning complexity and suitability for investigating the effect of comments on program comprehension, visual attention, linearity and gaze strategy. Instead of relying solely on one study, we curated appropriate code snippets from various studies, creating a diverse set of 12 code snippets.

Our selection process involved evaluating the snippets based on their potential to accommodate meaningful comments. We considered both the presence of comments and their quality. Additionally, the code snippets should encompass more complex programming concepts, such as recursion and pointers, to further add complexity to the comprehension task.

To align the code snippets with the objectives of our study, several adaptations were made. These adaptations included standardizing the task type, removing code documentation (JavaDocs), obfuscating obvious function and variable names, and adopting conventions and writing styles from the most recent Java version (at the time Java 18). These modifications ensured consistency and optimized the suitability of the code snippets for our study.

The final selection of code snippets is presented in Table 4.1. For the actual snippets, along with any modifications made, and the randomized order assigned to participants refer to the Appendix A.

Table 4.1: Final snippet selection with descriptions and lines of code.

Snippet	Description	LOC	(+ Comments)
1	Identifies a peak element in an array.	18	(+7)
2	Finds two elements that sum to a target.	19	(+8)
3	Computes maximum zero-sum subarray length.	19	(+10)
4	Longest consecutive sequence in array.	24	(+9)
5	Longest common subsequence between strings.	21	(+10)
6	Longest increasing subsequence length.	22	(+10)
7	Efficient power calculation.	17	(+7)
8	Fibonacci number at given position.	18	(+8)
9	Lists primes up to a number (Sieve of Eratosthenes).	18	(+6)
10	Binary search in sorted array.	21	(+7)
11	Counts palindromic substrings.	19	(+10)
12	Anagram check for two strings.	16	(+5)

4.6 post-questionnaire

Aligned with the research questions of our study, the post-questionnaire was designed to gather in-depth insights into the participants' experiences and perspectives on the influence of comments on program comprehension. This alignment ensured that the questions directly correlated with the core themes of our study, namely understanding the impact of comments on program comprehension, visual attention, linearity of reading order, gaze strategies, and overall comment contribution in programming tasks.

To provide a structured and consistent framework for participant responses, the questionnaire was divided into sections, each targeting specific aspects of the study's focus. This organization helped in correlating the responses with the quantitative data gathered during the experiment's first phase, thereby enriching the thematic analysis with interesting, qualitative insights. The detailed list of questions asked in the post-questionnaire is presented in Table 4.2.

Table 4.2: Post-questionnaire questions

Research Question	**Questions**
RQ_1: Program Comprehension	How did the presence of comments impact your understanding of the code? Please describe your experience with the code snippets that contained comments.
RQ_2: Visual Attention	Were there any instances where your visual attention was drawn to comments within the code snippets? How did the presence of comments influence your gaze patterns while comprehending the code?
RQ_3: Linearity of Reading Order	Did the presence of comments affect the order in which you read and interpret the code elements? Please explain how comments may have influenced the flow of your reading.
RQ_4: Gaze Strategies	Did you adopt a specific gaze strategy while navigating through code snippets containing comments? How did the presence of comments influence your choices in directing visual attention between code and comments?
RQ_5: Role of Comments	From your perspective, how do comments contribute to your overall understanding of the code snippets? Do you find comments helpful in clarifying complex code segments or guiding your comprehension?
General Impressions	Please share any additional observations, insights, or thoughts you have regarding the role of comments in program comprehension.

4.7 PARTICIPANTS AND SAMPLING STRATEGY

Our study aimed to recruit computer science students with a solid foundation in programming, encompassing both Java syntax and fundamental programming concepts, such as recursion, data structures, and algorithms. To ensure homogeneity in participants' experience level, we targeted students enrolled in the computer science program at our university. This setting offers an ideal opportunity to approach students directly, inviting them to participate in our research.

We employed a convenience sampling method to enlist 20 participants who meet the following inclusion criteria: (1) Currently enrolled in a computer science or similar program. (2) Successfully completed the Programming 2 course. [1] (3) Possess a solid understanding of Java and general programming concepts.

Participants were asked to complete a questionnaire at the beginning of the study to collect information on their personal information experience and proficiency with programming. The purpose of this questionnaire is to ensure that all participants have a sufficient level of programming experience.

The group comprised 18 males and 2 females, reflecting a gender distribution that aligns with typical patterns in the field of computer science. The mean age of the participants was 25.8 $\pm$ 4.20 years, suggesting a predominantly young cohort. Their educational background varied, with the majority (11 participants) holding a High School Diploma (Abitur). This was followed by 7 participants with a Bachelor's degree, and 2 with a Master's degree, reflecting a range of educational experiences. Regarding Java programming experience, the participants displayed a spectrum of expertise. A significant portion (12 participants) had practical project experience in Java, while 5 had basic knowledge of the language. Additionally, 2 participants regularly used Java for coding tasks, and 1 participant categorized himself as an expert, showcasing a range of proficiency levels within the group. Figure 4.2 depicts the age distribution, Java programming experience, and overall programming experience in years among the participants.

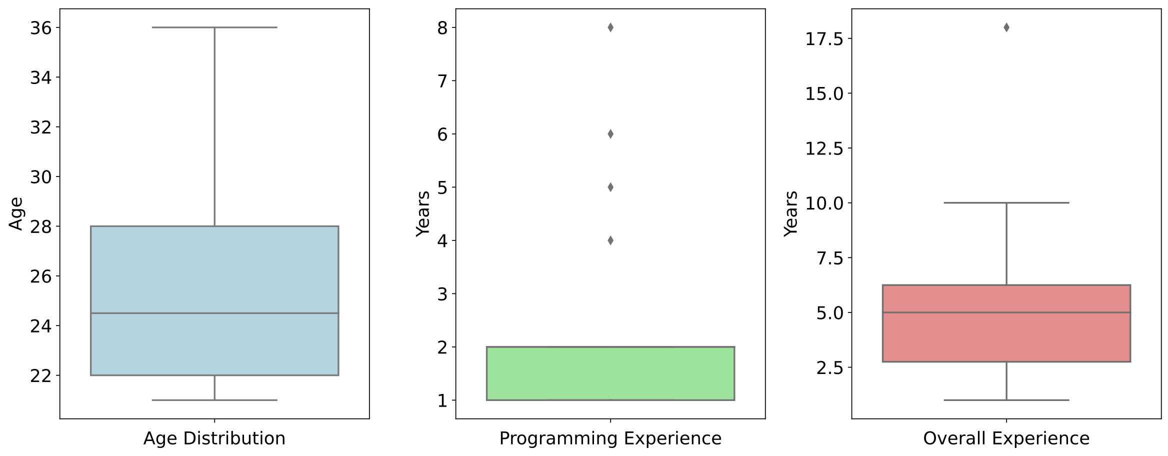


Figure 4.2: Participants' age, Java programming experience, and overall programming experience in years.

1 This lecture deals with the basics of imperative/object-oriented programming and primarily uses Java.

4.8 eye-tracking equipment and data collection

In our study, the Tobii EyeX [2] eye tracker was used to collect gaze data. The Tobii EyeX is a portable eye tracker that utilizes near-infrared light to track the position of the eyes. It has a compact size, measuring approximately 20 x 15 x 318 mm and weighing 91 grams. With a frequency of 70Hz and backlight-assisted near-infrared (NIR) illuminators operating at 850nm, along with red light at 650nm, the Tobii EyeX offers high accuracy and reliability in capturing eye movements. It has a tracking population of 95%, ensuring precise measurements.

The Tobii EyeX is compatible with screens up to 27 inches and has an operating distance range of 50 - 90 cm. The track box dimensions, representing the area where eye movements can be accurately captured, are approximately 40 x 30 cm at a distance of 75 cm.

To ensure accurate eye-tracking measurements, participants were instructed to position themselves at the right distance and position to the Tobii EyeX according to the manufacturer's instructions. Prior to the study, participants completed a calibration procedure, during which they looked at a series of points displayed on the screen. This calibration process allowed the eye tracker to accurately track their gaze data during the study.

A custom C# program, adapted from the study by Peitek et al. [75], was used to operate the Tobii EyeX eye tracker. The program guided participants through the study, including the calibration process and the presentation of code snippets. Throughout the study, the program collected participants' responses, completion times, and gaze data, which were stored for later analysis. These measures provided valuable insights into participants' reading behavior, gaze strategies, and linearity of reading order.

4.9 ethical considerations and data handling

Ethical considerations are an important aspect of any study involving human subjects. In our study, informed consent was obtained from all participants prior to their participation. They were provided with detailed information about the nature of our study, the procedures involved, and their right to withdraw from the study at any time without any negative consequences.

To ensure participant anonymity and protect their privacy, each participant was assigned a unique identifier. This was used to link their data to their responses and ensure confidentiality. All personal identifying information is kept confidential and accessible only to the research team. Data collected during the study is securely stored and reported in aggregate form, ensuring that no individual participant can be identified.

These measures are in place to uphold the ethical standards of participant privacy and confidentiality, and to ensure that participants' rights and well-being are protected throughout the study.

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

5 DATA ANALYSIS METHODOLOGY

This section outlines the methodologies and procedures employed in managing and refining the data gathered from study participants. It is dedicated to describing the process of handling both behavioral and eye-tracking data, from the collection of raw data to its preparation for analysis.

5.1 BEHAVIORAL DATA ANALYSIS

For the behavioral data analysis, each response was manually evaluated to determine its semantic correctness. Minor formatting inaccuracies, such as variations in decimal places, were considered semantically correct. This evaluation ensured the accuracy of the behavioral data for further analysis.

To quantify the combined influence of correctness and time, we devised a scoring system for each snippet. The score was calculated using a formula that equally weighs the accuracy of the participants' responses and the normalized completion time. Specifically, the score was determined by the following expression.

$$\text{Score} = 0.5 \times \text{correctness} + 0.5 \times \left(1 - \frac{\text{time} - \text{min\_time}}{\text{max\_time} - \text{min\_time}}\right) \times 100$$

In this formula, 'correctness' represents the correctness of the snippet responses, and 'time' is normalized against the common minimum and maximum completion times for each snippet, considering both CM and CP conditions. This normalization ensures that the time scores for CM and CP conditions of the same snippet are comparable. Lower time results in a higher score. This approach allowed to create a balanced metric that encapsulates both the accuracy of responses and the efficiency of snippet completion under each condition.

5.2 EYE-TRACKING DATA ANALYSIS

The eye-tracking data underwent several preprocessing steps to ensure data quality and reliability. The data analysis pipeline employed in our study is based on the scripts used by Peitek et al. in their recent eye-tracking study [75]. The pipeline encompasses various stages, including preprocessing, analysis of general metrics, analysis of AOI metrics, statistical analysis, data export, and data visualization. Each step in the pipeline is designed to extract valuable information from the collected eye-tracking data and facilitate its interpretation. The subsequent subsections detail each stage of the pipeline, providing a comprehensive overview of the data analysis process and its significance for our study.

5.2.1 *Data Processing*

The first stage of the pipeline involved preprocessing the raw data, which is essential for ensuring data quality, reducing noise, and preparing the data for subsequent analyses. In this stage, the raw eye-tracking data was processed and refined using various techniques. The preprocessing pipeline class encapsulates the functionality for preprocessing the raw data. It consists of several methods that handle different aspects of the preprocessing pipeline. Next, we delve into each step of the data preparation process:

Step I: Reading and Cleaning Data
First, the raw eye-tracking data is read and essential data cleaning operations are performed. The gaze data frames are collected and loaded from the specified directory, and subsequent cleaning steps are executed to ensure data quality and reliability. These cleaning operations include removing unnecessary columns, dropping rows related to fixation cross conditions, and renaming columns with more meaningful labels. Moreover, timestamps, gaze positions, and experiment time values are adjusted and rounded to facilitate further analysis. Additionally, the pipeline reads the general information data, which contains participant-specific information such as the actual screen height and eye tracker resolution. This information is crucial for scaling the eye tracker data to match the actual screen resolution if necessary. The cleaned and scaled data frames are then ready for subsequent stages.

Step II: Preprocessing Data
In this step, a series of operations to enhance data quality and extract additional features are performed. These operations include dropping duplicate timestamps, scaling the gaze data based on screen height, smoothing the gaze positions using a Savitzky-Golay filter (window length of 5, polynomial order of 3) [70], calculating the velocity of eye movements, and detecting gaze events (fixations and saccades) based on a specified velocity threshold.

Step III: Classifying Data
Once the gaze data frames are preprocessed, the data frames can be classified into fixations and saccades based on the detected gaze events from the previous step. We classified fixations and saccades using a velocity-based algorithm with a velocity threshold of 150 pixels per 100 milliseconds. If the velocity was below the threshold, it was interpreted as a fixation; if it exceeded the threshold, it was interpreted as a saccade. Furthermore, relevant statistics for each fixation and saccade are computed, including average position, time duration, frame count, average velocity, and distance traveled. The resulting fixations and saccades are stored for further analysis.

Step IV: Reducing Gaze Dataframes
The final step in the preprocessing stage involves reducing the size of the gaze data frames to include only the necessary columns for subsequent analysis. The reduced gaze data frames are rounded to an appropriate decimal precision and stored for further analysis.

5.2.2 *Analysis of General Metrics*

Following the preprocessing stage, the pipeline proceeds to analyze general metrics obtained from the eye-tracking data. The analysis of general metrics involved computing various metrics that provide insights into participants' eye gaze behavior and eye movement patterns. The key steps involved in the analysis of these general metrics are outlined below:

Step I: Fixation and Saccade Analysis
The first step focuses on analyzing fixations and saccades. Various statistics related to fixations and saccades are computed, including the number of fixations and saccades, fixation and saccade rates per second, and average fixation length and saccade distance. These statistics provide valuable insights into the temporal aspects and spatial characteristics of participants' eye movements.

Step II: Snippet-Level Metrics
The analysis pipeline also considers snippet-level metrics by examining the eye gaze behavior within specific snippets. Fixation and saccade data for each snippet are collected, and metrics such as fixations per second, saccades per second, fixation length, and saccade distance are computed. This analysis allows for a more detailed understanding of participants' eye movements during different snippets.

Step III: Results Collection
Throughout the analysis pipeline, the computed metrics are stored in a results dictionary. The dictionary organizes the metrics based on their categories, including fixation metrics (e. g., fixations per second, fixation length) and saccade metrics (e. g., saccades per second, saccade distance). For snippet-level metrics, the results are further grouped by snippet, allowing for easy comparison and exploration of participants' eye gaze behavior across different snippets.

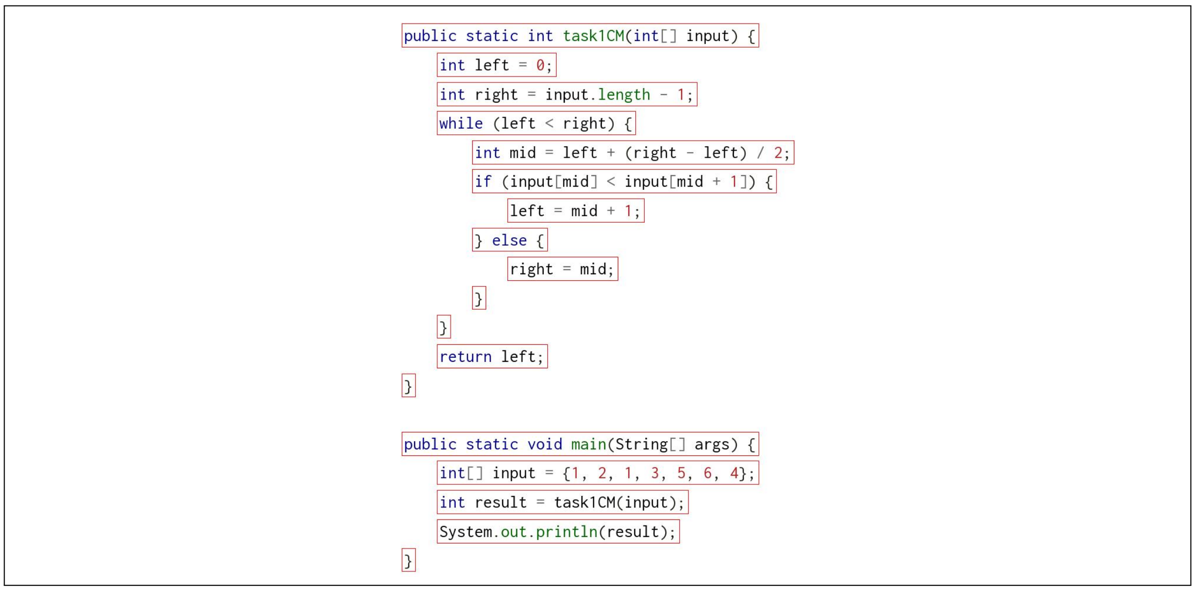
At the end of the analysis pipeline, the participant data frames and the collected results are returned. These results provide a comprehensive overview of the general metrics for each participant, including aggregated metrics across the entire experiment and snippet-specific metrics.

5.2.3 *Analysis of AOI Metrics*

In addition to general metrics, the pipeline also includes the analysis of AOI metrics. By evaluating participants' gaze behavior within these AOIs, insights into the specific areas or elements that attract visual attention can be gained. The analysis of AOI metrics involved computing various metrics within the eye gaze data. The following details the steps involved in this process:

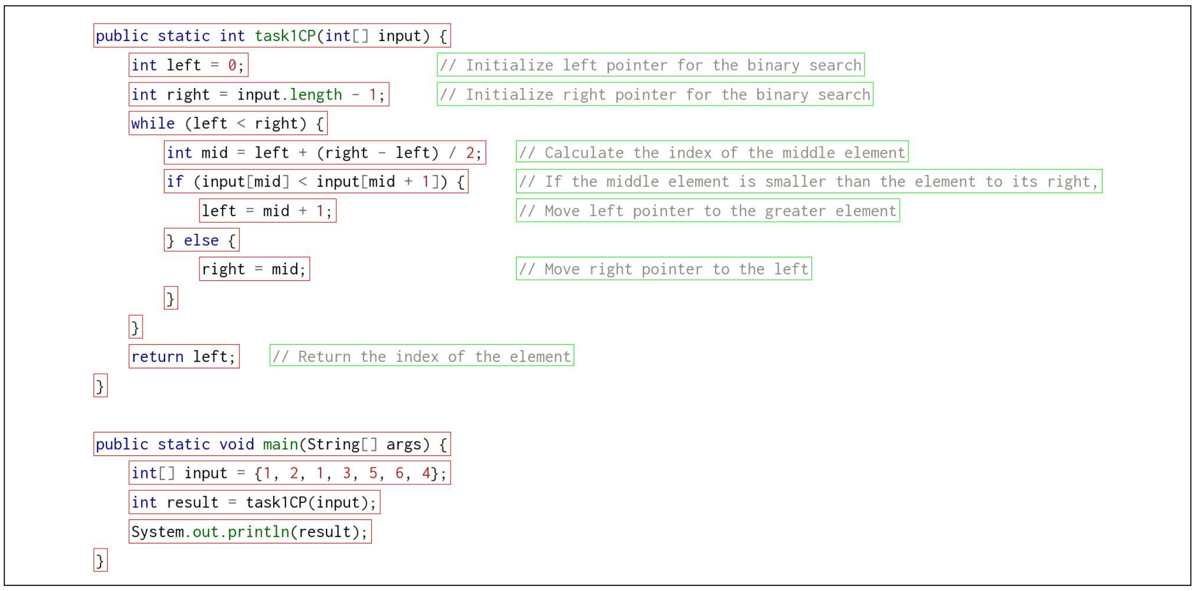
Step I: Fixation and Saccade Data Extraction
The analysis begins by extracting fixation and saccade data for each participant and snippet. Fixations and saccades associated with the specified snippet are retrieved and stored for further analysis.

Step II: Creating Areas of Interest (AOIs)
For addressing RQ.3, each line within the snippet is considered as an individual AOI. This approach enables the computation of metrics that characterize the linearity of reading order, following the methodology proposed by Busjahn et al. [22]. Additionally, by distinguishing between code and comment AOIs, the study aims to examine the reading flow between these elements and identify the employed gaze strategies. For a better visualization of the AOIs, Figure 5.1 shows an example snippet with AOI overlays, where each line is represented as an individual AOI. Additionally, Figure 5.2 illustrates AOI overlays with a distinction between code and comment elements.



```java
public static int task1CM(int[] input) {
    int left = 0;
    int right = input.length - 1;
    while (left < right) {
        int mid = left + (right - left) / 2;
        if (input[mid] < input[mid + 1]) {
            left = mid + 1;
        } else {
            right = mid;
        }
    }
    return left;
}

public static void main(String[] args) {
    int[] input = {1, 2, 1, 3, 5, 6, 4};
    int result = task1CM(input);
    System.out.println(result);
}
```

Figure 5.1: AOI Overlays with each code line as AOI for analyzing the linearity of reading order.



```java
public static int task1CP(int[] input) {
    int left = 0;                           // Initialize left pointer for the binary search
    int right = input.length - 1;           // Initialize right pointer for the binary search
    while (left < right) {
        int mid = left + (right - left) / 2;    // Calculate the index of the middle element
        if (input[mid] < input[mid + 1]) {      // If the middle element is smaller than the element to its right,
            left = mid + 1;                     // Move left pointer to the greater element
        } else {
            right = mid;                        // Move right pointer to the left
        }
    }
    return left;    // Return the index of the element
}

public static void main(String[] args) {
    int[] input = {1, 2, 1, 3, 5, 6, 4};
    int result = task1CP(input);
    System.out.println(result);
}
```

Figure 5.2: AOI Overlays with distinction between code and comment AOIs.

Step III: Matching Fixations to AOIs
The next step involves matching the extracted fixations to the appropriate AOIs. Fixations falling within the boundaries of an AOI are considered hits, and relevant information such as the AOI name, line number, and fixation duration are recorded. For all subsequent analyses based on AOIs, we filtered out all fixations outside of defined AOIs. Following Busjahn et al. [22], we included fixations with a maximum of a 100 pixel horizontal deviation (ca. 7–8 characters), as small AOIs can otherwise be easily missed (e. g., a closing bracket) and may distort the results.

Step IV: Computing AOI Metrics
Once the fixations are matched to the AOIs, various AOI metrics are calculated based on the matched fixations. These metrics provide insights into participants' eye gaze behavior within specific AOIs.

a) **Visual Attention Metrics:**
Through analyzing and filtering the fixation hits for the different AOIs - Code and Comments - we aggregated detailed data on where and how long participants focused their gaze while comprehending code snippets. This approach allows us to gain invaluable insights into the cognitive processes involved in code comprehension, particularly in terms of how comments either guide or distract visual attention. The data is segmented into various categories: All Fixations, AOI Fixations, Code Fixations, and Comment Fixations, each quantified in terms of count and duration. A breakdown of these metrics is presented in Table 5.1.

Table 5.1: Description of visual attention metrics.

Metric	**Description**
`All Fixations`	Total number of fixations within the entire snippet.
`AOI Fixations`	Total number of fixations within AOIs.
`Code Fixations`	Total number of fixations on code AOIs.
`Comment Fixations`	Total number of fixations on comment AOIs.

b) **Linearity Metrics:**
This builds directly on the insights gained from the work of Busjahn et al. [22] and Peitek et al. [75]. The papers highlight the distinction between natural language text reading and source code reading, emphasizing that while natural language is typically read linearly, source code reading often deviates from this linear pattern. To measure this, both local and global gaze-based measures are used to assess the degree of linearity in reading behavior. Our study replicates and extends this approach by employing five fixation-based local metrics: Vertical Next Fixations, Vertical Later Fixations, Regression Fixations, Horizontal Later Fixations, and Line Regression Fixations. These metrics reflect the immediate gaze patterns of participants during reading snippets. Additionally, we incorporate the global metrics established by the referenced studies. It utilizes the N-W algorithm for analyzing reading patterns in programming. This algorithm is employed in two forms: a naïve and a dynamic calculation for both story and execution order of the code. The naïve calculation compares the participant's reading order directly with the expected linear sequence of the code, while the dynamic calculation allows the sequence to be repeated multiple times accounting for the iterative nature of reading complex code. Details of these metrics are catalogued in Table 5.2.

Table 5.2: Description of linearity metrics.

	Metric	Description
Local	`Vertical Next Text`	% of forward saccades that either stay on the same line or move one line down.
	`Vertical Later Text`	% of forward saccades that either stay on the same line or move down any number of lines.
	`Horizontal Later Text`	% of forward saccades within a line.
	`Regression Rate`	% of backward saccades of any length.
	`Line Regression Rate`	% of backward saccades within a line.
	`SaccadeLength`	Average Euclidean distance between every successive pair of fixations.
Global	`Story Order`	N-W alignment score of fixation order with linear text reading order.
	`Execution Order`	N-W alignment score of fixation order with the program's control flow order.

c) **Gaze Strategy Metrics:**
Our study extends this framework by adopting a similar approach for investigating Gaze Strategies within CP snippets, where we aim to examine how participants shift their focus between code and comments during comprehension tasks. Local metrics, such as code-to-comment and comment-to-code ratios, provide insights into immediate gaze patterns and reveal how participants navigate between code and accompanying comments. For a broader perspective, we applied the N-W algorithm once more. This time, it was used to assess the alignment of participants' gaze patterns with pre-defined AOI sequences for Code-First and Comment-First reading orders. This global measure will help us understand the overarching gaze strategies adopted by participants in the context of different reading approaches. Details of these metrics are catalogued in Table 5.3.

Table 5.3: Description of gaze strategy metrics.

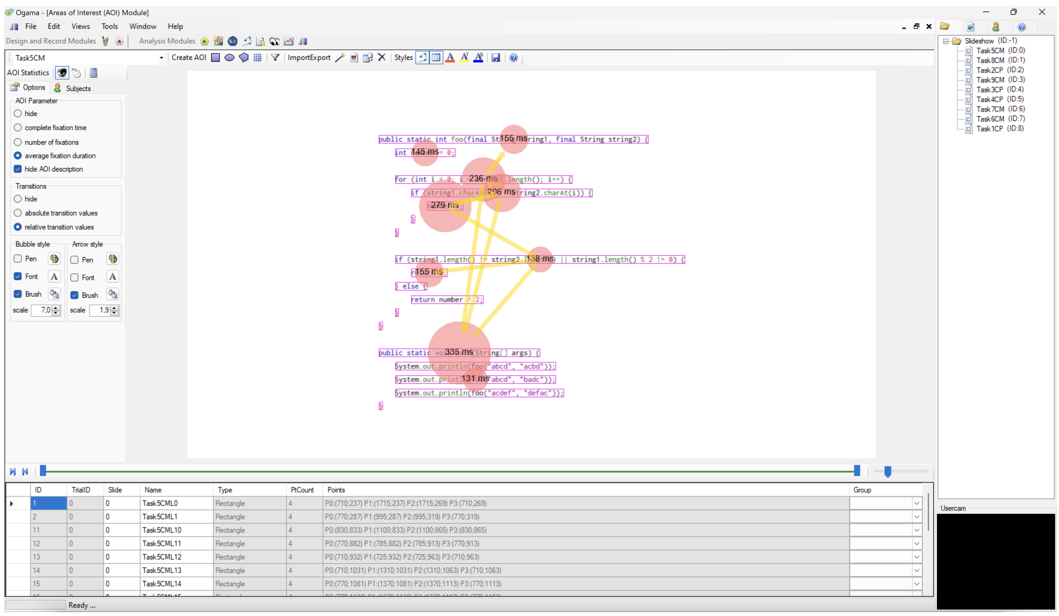
	Metric	**Description**
Local	`CodeToComment`	% of saccades that move from code to comment.
	`CommentToCode`	% of saccades that move from comment to code.
Global	`Code-First`	N-W alignment score of fixation order with code-first reading order.
	`Comment-First`	N-W alignment score of fixation order with comment-first reading order.

Step V: Results Collection
The computed AOI metrics are collected into a results dataframe. Each row of the dataframe represents a participant's metrics for a specific snippet, including the participant ID, snippet name, CP flag, and all the computed metrics mentioned above. The results dataframe allows for easy comparison and statistical analysis of AOI metrics across participants and snippets.

5.2.4 *Data Exports*

To facilitate further analysis, the pipeline also includes data export functionalities. Specifically, two types of data exports are performed: OpenGazeAndMouseAnalyzer (`OGAMA`) [123] and Radial Transition Graph Comparison Tool (`RTGCT`) [15] data exports.

1. `OGAMA`
   The eye-tracking data is exported in a format suitable for analysis and visualization in the `OGAMA` software tool. `OGAMA` is an open-source software designed to analyze eye and mouse movements in slideshow study designs. It provides features such as creation of attention maps, definition of areas of interest, and calculation of saliency. The exported data includes gaze positions, timestamps, trial sequence, and trial images, which are saved as individual CSV files for each participant.



Ogama - [Areas of Interest (AOI) Module]
File Edit Views Tools Window Help
Design and Record Modules
Analysis Modules
Task5CM
Create AOI
ImportExport
Styles
AOI Statistics
Options
Subjects
Usercam

AOI Parameter
- hide
- complete fixation time
- number of fixations
- [✓ average fixation duration]
- [✓ hide AOI description]

Transitions
- hide
- absolute transition values
- [✓ relative transition values]

Bubble style		Arrow style	
Pen		Pen	
[✓ Font]		Font	
[✓ Brush]		[✓ Brush]	
scale	7.0	scale	1.9

```
public static int foo(final St155 msring1, final String string2) {
    int 145 ms= 0;

    for (int i = 0; i 236 ms1.length(); i++) {
        if (string1.charAt206 msstring2.charAt(i)) {
            279 ms;
        }
    }

    if (string1.length() != string2.138 ms) || string1.length() % 2 != 0) {
        r155 ms;
    } else {
        return number / 2;
    }
}

public static vo335 msString[] args) {
    System.out.println(foo("abcd", "acbd"));
    System.out.print131 ms"abcd", "badc"));
    System.out.println(foo("acdef", "defac"));
}
```

ID	TrialID	Slide	Name	Type	PtCount	Points	Group
1	0	0	Task5CML0	Rectangle	4	P0:(710;237) P1:(1715;237) P2:(1715;269) P3:(710;269)	
2	0	0	Task5CML1	Rectangle	4	P0:(770;287) P1:(995;287) P2:(995;319) P3:(770;319)	
11	0	0	Task5CML10	Rectangle	4	P0:(830;833) P1:(1100;833) P2:(1100;865) P3:(830;865)	
12	0	0	Task5CML11	Rectangle	4	P0:(770;882) P1:(785;882) P2:(785;913) P3:(770;913)	
13	0	0	Task5CML12	Rectangle	4	P0:(710;932) P1:(725;932) P2:(725;963) P3:(710;963)	
14	0	0	Task5CML13	Rectangle	4	P0:(710;1031) P1:(1310;1031) P2:(1310;1063) P3:(710;1063)	
15	0	0	Task5CML14	Rectangle	4	P0:(770;1081) P1:(1370;1081) P2:(1370;1113) P3:(770;1113)	

Ready ...

- Slideshow (ID:-1)
  - Task5CM (ID:0)
  - Task8CM (ID:1)
  - Task2CP (ID:2)
  - Task9CM (ID:3)
  - Task3CP (ID:4)
  - Task4CP (ID:5)
  - Task7CM (ID:6)
  - Task6CM (ID:7)
  - Task1CP (ID:8)

Figure 5.3: OGAMA [123] interface displaying AOIs for eye movement analysis.

2. RTGCT
   Eye-tracking data is exported in a format that works with the RTGCT tool [1], enabling the comparison of participant eye movements through graphs. This data shows which AOIs participants focused on and for how long. Each AOI has a unique color, and transitions between AOIs are represented by arcs – the thicker the arc, the more frequent the transition. Small circles at the start and end of each arc indicate the direction of movement: black circles for outgoing transitions and white for incoming. For analysis, the data is split into individual CSV files for each participant and code snippet, along with a combined file for overarching analysis. This setup simplifies understanding how participants navigate and interact with different parts of the content visually.

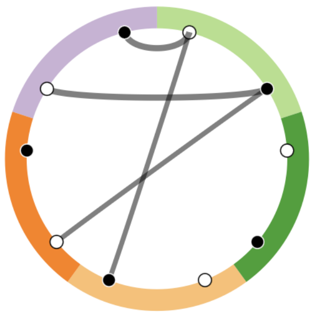


Figure 5.4: Example of radial transition graph generated by the RTGCT [15].

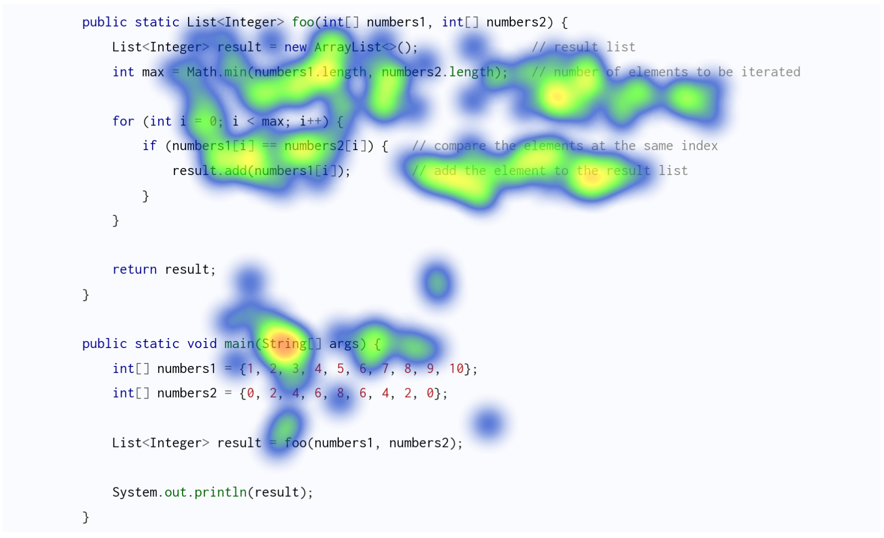
1 http://www.rtgct.fbeck.com

5.2.5 *Data Visualization*

Finally, the pipeline incorporates various data visualization techniques to enhance the interpretation and communication of the findings. Data visualizations also provide intuitive and illustrative insights into participants' eye movements, aiding the exploration and presentation of the study's findings. The visualization process involves iterating through the preprocessed eye-tracking data for each participant. Relevant data, such as fixations and snippet information, are extracted. The necessary visualizations are then generated based on the study configuration settings. The available visualizations include:

1. XY Velocity Plots:
   These plots visualize eye movement velocities in the XY plane, providing information about speed and direction.

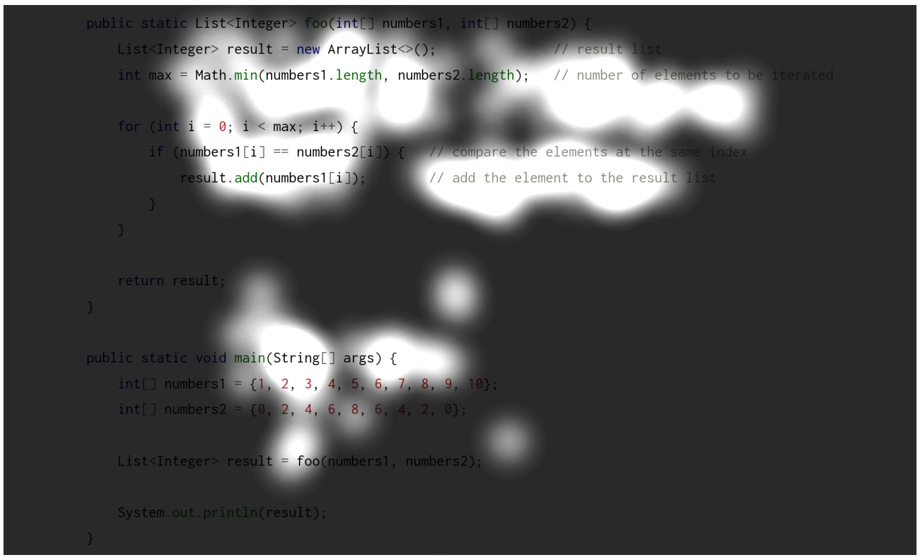
2. Heatmaps:
   Heatmaps display the fixation density on a snippet, providing a visual representation of where participants focused their gaze. The color intensity indicates the frequency and duration of fixations on different areas of the code snippet.



```
public static List<Integer> foo(int[] numbers1, int[] numbers2) {
    List<Integer> result = new ArrayList<>();             // result list
    int max = Math.min(numbers1.length, numbers2.length);   // number of elements to be iterated

    for (int i = 0; i < max; i++) {
        if (numbers1[i] == numbers2[i]) {   // compare the elements at the same index
            result.add(numbers1[i]);        // add the element to the result list
        }
    }

    return result;
}

public static void main(String[] args) {
    int[] numbers1 = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int[] numbers2 = {0, 2, 4, 6, 8, 6, 4, 2, 0};

    List<Integer> result = foo(numbers1, numbers2);

    System.out.println(result);
}
```

Figure 5.5: Example heatmap of eye fixations on a code snippet. The color intensity represents the fixation density, with warmer colors indicating higher fixation frequency.

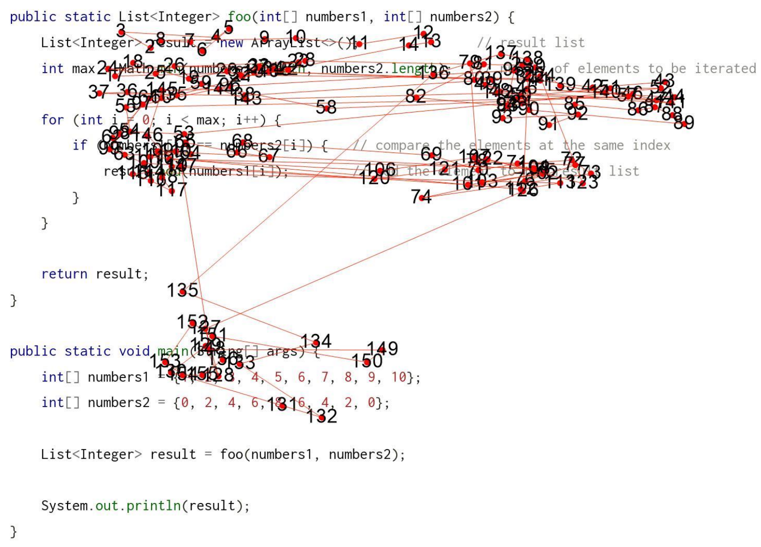
3. Reveal Images:
   Reveal Images highlight areas with high fixation density, emphasizing the most visually attended regions on the code snippet. By revealing these high-density areas, Reveal Images offer a more focused view of participants' gaze behaviors.



```
public static List<Integer> foo(int[] numbers1, int[] numbers2) {
    List<Integer> result = new ArrayList<>();              // result list
    int max = Math.min(numbers1.length, numbers2.length);   // number of elements to be iterated

    for (int i = 0; i < max; i++) {
        if (numbers1[i] == numbers2[i]) {   // compare the elements at the same index
            result.add(numbers1[i]);        // add the element to the result list
        }
    }

    return result;
}

public static void main(String[] args) {
    int[] numbers1 = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int[] numbers2 = {0, 2, 4, 6, 8, 6, 4, 2, 0};

    List<Integer> result = foo(numbers1, numbers2);

    System.out.println(result);
}
```

Figure 5.6: Example reveal image of eye fixations on a code snippet. High-density areas are highlighted to emphasize the most visually attended regions.

4. Scan Paths:
   Scan Paths visualize the sequential eye movements made by participants while examining the code snippet. The path shows the order and duration of fixations, revealing the gaze flow and exploration patterns during program comprehension.



```
public static List<Integer> foo(int[] numbers1, int[] numbers2) {
    List<Integer> result = new ArrayList<>();              // result list
    int max = Math.min(numbers1.length, numbers2.length);   // number of elements to be iterated

    for (int i = 0; i < max; i++) {
        if (numbers1[i] == numbers2[i]) {   // compare the elements at the same index
            result.add(numbers1[i]);        // add the element to the result list
        }
    }

    return result;
}

public static void main(String[] args) {
    int[] numbers1 = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int[] numbers2 = {0, 2, 4, 6, 8, 6, 4, 2, 0};

    List<Integer> result = foo(numbers1, numbers2);

    System.out.println(result);
}
```

Figure 5.7: Example scan path of eye movements on a code snippet. The path shows the sequential eye movements made by a participant during program comprehension.

5.3 STATISTICAL ANALYSIS

Our study employs a partially crossover within-subject design, exposing participants to both Comments Missing (CM) and Comments Present (CP) conditions across varied code snippets. This approach introduces challenges related to data imbalance and the necessity to manage within-subject variability. In our analysis, we take a holistic statistical approach to assess the impact of CM and CP conditions on code snippet performance and behavior, capturing both overall trends and individual variations.

5.3.1 *Wilcoxon Signed-Rank Test*

Initially, the Wilcoxon Signed-Rank Test [126] was employed for the analysis of aggregated data, focusing on the median differences between CM and CP conditions without the need to account for individual participant variations. This non-parametric test was appropriate for datasets not conforming to the normal distribution assumptions required by parametric tests. It allowed for a direct comparison of paired observations, such as the performance or behavior under CM and CP conditions, to discern statistically significant median differences across these conditions. This method was instrumental in identifying general trends and overarching effects of comments on code comprehension across the entire participant pool. The analysis typically adopted a two-sided approach to assess median differences between snippet conditions, ensuring an unbiased investigation. However, when a specific effect direction was hypothesized, a one-sided test was applied, tailoring the analysis to align with anticipated outcomes and theoretical predictions.

5.3.2 *Linear Mixed-Effects Models (LMEs)*

Following the broader insights gained from the Wilcoxon Signed-Rank Test, separate LMEs were fitted in cases where a more in-depth analysis was needed on snippet-level, allowing for a more detailed analysis of various performance and eye-tracking metrics across the snippets. In these models, the snippet type was treated as a fixed effect, directly evaluating its influence on the measured outcomes. Importantly, individual differences among participants were modeled as random effects. This approach acknowledged and controlled for the individual differences in baseline performance levels across snippets. The models were defined as:

$$\text{Prediction}_{ij} = \beta_0 + \beta_1 \times \text{isCP}_{ij} + u_j + \epsilon_{ij}$$

where Prediction_{ij} represents the metric entry for the i-th observation from the j-th participant, β_0 represents the intercept, β_1 is the coefficient assessing the effect of the CP condition (relative to the CM condition, which serves as the reference category), u_j denotes the random effect to capture individual participant differences, and ϵ_{ij} is the residual error. For instance, regarding correctness, the models predicted the binary outcome (correct or incorrect) based on the snippet type, and for time, they predicted the duration taken to complete each snippet. The significance of the CP condition's effect was then evaluated using p-values for the coefficient β_1, with lower values indicating a more significant impact of the condition on the respective response variable.

To further refine the analysis, False Discovery Rate (FDR) [84] correction was applied whenever several metrics were tested on the same dataset. The Benjamini-Hochberg [10] procedure was used for this purpose, adjusting p-values to reflect a more accurate significance level of the observed effects. This correction aimed to control for the increased risk of Type I errors associated with multiple comparisons, thereby ensuring the statistical integrity of the findings.

This combination of LMEs and the Wilcoxon Signed-Rank Test provided a comprehensive methodological framework for the study. By employing the Wilcoxon test, the study could ascertain broad, overarching trends, establishing a baseline understanding of the effects of snippet conditions. Subsequently, LMEs offered a deeper dive into these effects across the snippets while accommodating individual participant strategies.

5.4 QUALITATIVE ANALYSIS OF RATINGS

Following the quantitative phase of our study, we conducted a lightweight thematic analysis of the responses collected from the post-questionnaire (cf. Section 4.6) to discern the broader implications of our findings. This analysis was guided by Braun and Clarke's six-phase framework, providing a systematic approach to understanding the nuanced insights of how comments influence program comprehension from a programmer's perspective [61]. The steps included:

1. *Familiarization with the Data:*
   Engaging with the data through repeated reading and note-taking to grasp the depth of responses.

2. *Generating Initial Codes:*
   Systematically coding the data to identify significant phrases, patterns, or concepts emerging directly from the responses.

3. *Searching for Themes:*
   Organizing the codes into potential themes that encapsulate the core insights from the data.

4. *Reviewing Themes:*
   Refining these themes to ensure they accurately reflect the coded data and the entire dataset.

5. *Defining and Naming Themes:*
   Clarifying the essence of each theme and how they interrelate, offering a coherent narrative of the data.

6. *Integration and Reporting:*
   Relating the thematic findings back to the study's research questions and the broader context, illustrating how these qualitative insights enrich our quantitative analysis.

This structured thematic analysis enabled a comprehensive exploration of the qualitative data, enhancing our understanding of the implications of comments on program comprehension.

6 RESULTS

In this section, we offer a detailed presentation of our study's findings. Initially, we delve into the behavioral data to explore the impact of comments on program comprehension. Subsequently, our analysis shifts to eye-tracking data, focusing on aspects such as visual attention, the linearity of reading order, and gaze strategies. By combining eye tracking with behavioral data, we explore how comments affect the understanding, cognitive load, and eye movement patterns of computer science students during program comprehension. The structure aligns with the five research questions outlined in Section 4.2, with each addressing a specific aspect of the study:

RQ_1: Program Comprehension: Analysis of correctness rates, and completion times to understand the effect of comments on program comprehension.

RQ_2: Visual Attention: Assessment of eye-tracking data, including fixation counts and duration, providing insights into the allocation of visual attention in the presence and absence of comments.

RQ_3: Linearity of Reading Order: Investigation of the reading patterns and their linearity, as influenced by the presence of comments, using eye-tracking measures and an adapted version of the N-W algorithm.

RQ_4: Gaze Strategies: Exploration of gaze strategies, focusing on how programmers navigate between code and comments. The study uses both local and global metrics to assess these strategies.

RQ_5: Participants' Perceptions: Analysis of participants' subjective views on the difficulty of code snippets and the perceived contribution of comments to program comprehension.

This section focuses on presenting key results that are most relevant to the overarching objectives of our study. A thorough discussion of these results, including their implications and potential interpretations, are presented in Chapter 7.

6.1 EFFECT OF COMMENTS ON PROGRAM COMPREHENSION

In our study, we conducted a comprehensive evaluation of the effect of comments on program comprehension. This was achieved by analyzing the correctness and completion time of participants' responses to the 12 code snippets under two distinct conditions: Comments Missing (CM) and Comments Present (CP).

Our study highlights a complex and varied impact of comments on the comprehension of different code snippets, as evidenced by changes in correctness rates, completion times, and overall scores. To illustrate this point, we provide four distinct examples, each representing a unique combination of outcomes:

- Snippet 1, which involved identifying a peak element in an array, showed a decrease in performance with comments present (CP). The correctness rate dropped from 70% to 30%, and the completion time increased from 190.9 to 271.7 seconds. Correspondingly, the overall score decreased by 30% in the CP condition, indicating a negative impact of comments.

- Conversely, Snippet 9, focused on listing primes up to a given number, exhibited a substantial improvement with comments. The correctness rate rose from 60% under CM to 100% under CP, and the completion time decreased from 339.2 to 189.4 seconds. This positive effect was further reflected in the overall score, which increased by 34.1% under CP.

- Snippet 5, dealing with finding the longest common subsequence between strings, showed an increase in correctness from 50% in CM to 70% in CP. However, this was offset by a longer completion time. Despite this, the overall score increased by 7% in the CP condition, suggesting a net positive effect of comments.

- Snippet 11, involving the count of palindromic substrings, showed no variation in correctness (10%) between the two conditions. Nonetheless, there was a notable decrease in completion time under CP, and the overall score increased by 8.2%, highlighting the efficiency gains due to comments.

The detailed results are presented in Table 6.1, contrasting the effects of Comments Missing (CM) and Comments Present (CP) snippets on correctness rates and completion times. Additionally, Figure 6.1 visually complements these findings by graphically illustrating the effects captured in the table. By examining the distance of each snippet from the neutral benchmark line in the plots, we can visually deduce the overall effect of comments. This visual analysis parallels the numerical scores from the table, providing an intuitive understanding of how comments influence correctness and time efficiency.

Table 6.1: Comparative analysis of correctness and completion time of CM and CP snippets. The overall effect was determined by equally weighting correctness and normalized time (cf. Section 5.1).

Snippet	Description	LOC (+ Comments)	Type	Correctness	Time (in sec.)	Overall Effect
1	Identifies a peak element in an array.	18 (+7)	CM	70.0%	190.9 ± 87.6	-30.0%
			CP	30.0%	271.7 ± 74.1	
2	Finds two elements that sum to a target.	19 (+8)	CM	60.0%	206.4 ± 100.4	-10.2%
			CP	50.0%	241.4 ± 81.3	
3	Computes maximum zero-sum subarray length.	19 (+10)	CM	50.0%	238.0 ± 71.1	-18.7%
			CP	30.0%	291.0 ± 86.4	
4	Longest consecutive sequence in array.	24 (+9)	CM	50.0%	203.0 ± 90.3	-11.4%
			CP	40.0%	244.3 ± 90.2	
5	Longest common subsequence between strings.	21 (+10)	CM	50.0%	295.5 ± 156.8	7.0%
			CP	70.0%	331.3 ± 151.5	
6	Longest increasing subsequence length.	22 (+10)	CM	40.0%	333.4 ± 240.1	-2.3%
			CP	30.0%	285.3 ± 93.0	
7	Efficient power calculation.	17 (+7)	CM	40.0%	229.1 ± 85.2	-1.1%
			CP	40.0%	238.9 ± 121.9	
8	Fibonacci number at given position.	18 (+8)	CM	60.0%	230.1 ± 71.8	11.3%
			CP	60.0%	167.3 ± 80.8	
9	Lists primes up to a number (Sieve of Eratosthenes).	18 (+6)	CM	60.0%	339.2 ± 107.9	34.1%
			CP	100.0%	189.4 ± 97.1	
10	Binary search in sorted array.	21 (+7)	CM	80.0%	169.5 ± 65.0	9.4%
			CP	90.0%	144.6 ± 92.4	
11	Counts palindromic substrings.	19 (+10)	CM	10.0%	276.5 ± 101.6	8.2%
			CP	10.0%	198.8 ± 103.4	
12	Anagram check for two strings.	16 (+5)	CM	90.0%	62.9 ± 20.8	2.4%
			CP	100.0%	69.7 ± 44.7	

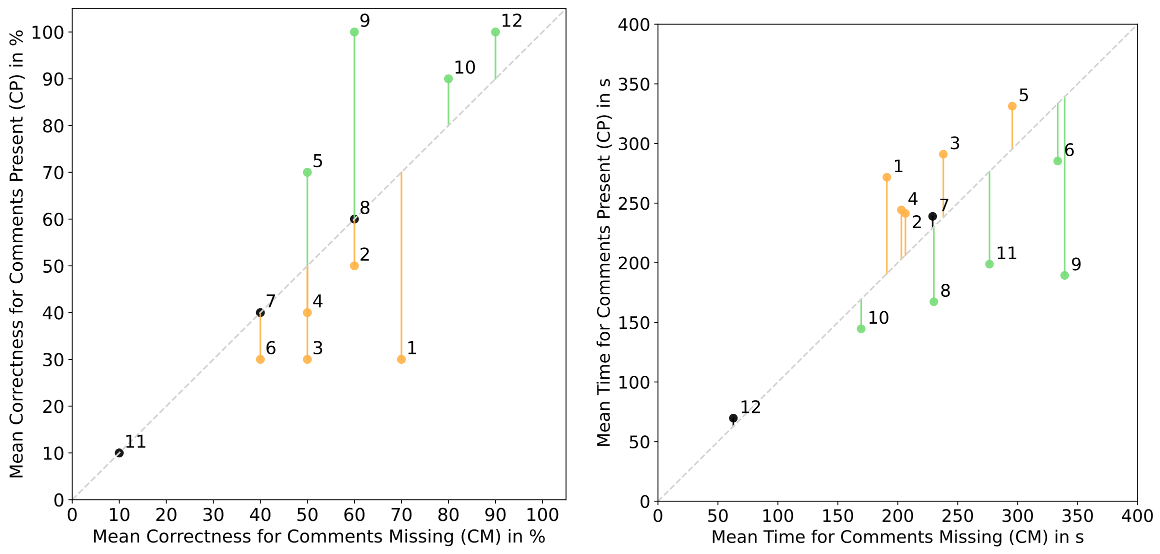


Figure 6.1: Effect of comments on mean correctness and time across snippets. Green signifies positive impacts, orange denotes negative impacts, and black indicates a neutral effect.

When considering the snippets collectively, a nuanced picture emerges:

- **Positive Impact of Comments:** Snippets 5, 8, 9, 10, and 11 showed a beneficial effect of comments, as evidenced by their net positive scores and distances to the neutral benchmark in both plots. These snippets showed notable improvements in correctness and/or efficiency, suggesting that comments can significantly enhance understanding and performance in specific contexts.

- **Negative Impact of Comments:** Snippets 1, 2, 3, and 4 demonstrate a negative impact of comments, characterized by lower correctness and longer completion times, as seen through their negative overall scores and relative positions to the neutral line.

- **Neutral/Mixed Effect of Comments:** Snippets 6, 7, and 12 present a more ambiguous effect, with slight variations in performance that do not strongly trend towards positive or negative outcomes.

Furthermore, we conducted a statistical analysis to examine the effects of comments on the correctness and time taken to complete snippets. To account for both within-subject and between-subject variations, we employed a Linear Mixed-Effects Model for each snippet individually (cf. Section 5.3.2). This approach allowed us to consider the fixed effects of snippet type (CM and CP) while incorporating random effects to accommodate inter-participant variability.

The analysis revealed varying effects of comments on correctness, and time across different snippets. In some snippets, the CP condition showed a significant difference in performance metrics compared to the CM condition, whereas, in others, the differences were not statistically significant. These results further indicate that the influence of snippet type on performance is snippet-dependent. The results from these models are presented in Table 6.2.

RQ$_1$ The collective insights reveal that the presence of comments can exert diverse effects on programming task outcomes. While some snippets like Snippet 9 benefit extremely from comments in terms of both time and correctness, others such as Snippet 1 suffer from longer completion times without a corresponding rise in correctness. These variations highlight a complex and snippet-specific interplay between comments and task execution, suggesting that the efficacy of comments is highly contextual, enhancing comprehension and efficiency in some scenarios while potentially hindering them in others.

Table 6.2: LME results for correctness and time of each snippet. The cell shading highlights a significant effect of comments (**positive** or **negative**).

	Correctness			Time		
Snippet	Intercept	Effect of CP	p-Value	Intercept	Effect of CP	p-Value
1	0.7	-0.4	0.059	190.87	80.87	**0.002**
2	0.6	-0.1	0.544	206.43	34.98	0.173
3	0.5	-0.2	0.211	238.03	52.96	**0.008**
4	0.5	-0.1	0.544	203.01	41.26	0.148
5	0.5	0.2	0.211	295.52	35.80	**0.017**
6	0.4	-0.1	0.651	333.44	-48.15	0.439
7	0.4	0.0	1.000	229.06	9.79	0.662
8	0.6	0.0	1.000	230.08	-62.76	**<0.001**
9	0.6	0.4	**<0.001**	339.24	-149.85	**<0.001**
10	0.8	0.1	0.514	169.47	-24.88	0.204
11	0.1	0.0	1.000	276.46	-77.70	**0.002**
12	0.9	0.1	0.157	62.92	6.75	0.637

6.2 effect of comments on visual attention

Building on the behavioral results observed in the previous section, we now turn our focus to a more detailed investigation of how these behaviors manifest during program comprehension. Specifically, this section delves into the effect of comments on visual attention, a pivotal aspect of understanding how programmers interact with commented code.

Our study reveals a significant shift in visual attention when comments are present. In snippets where comments are absent, all AOI fixations are focused on the code. However, with the introduction of comments, about 23% of visual attention is captured by these comments, reducing the focus on code to around 77%. This significant diversion of attention highlights that comments play a critical role in shaping programmers' visual engagement with code. In snippets where comments are present, participants are dedicating a substantial portion of their attention to them.

To illustrate these findings more concretely, Table 6.3 encapsulates the aggregated results of our study. It compares the metrics of visual attention (cf. *Step IV:*a in Section 5.2.3) in snippets with Comments Missing (CM) against those with Comments Present (CP).

In addition to the above analysis on the aggregated data, we quantitatively analyzed the fixation counts and durations for the different categories across the different snippets to capture possible snippet-specific effects. LMEs were again employed for each snippet to rigorously evaluate these differences. The models were constructed similar to the previous section to assess the fixed effects of the snippet type (CM or CP) on visual attention, while controlling for random effects due to individual participant variations (cf. Section 5.3). This approach allowed us to draw statistically robust conclusions about the role of comments in directing visual attention during program comprehension on a snippet-specific level.

Table 6.3: Comparative analysis of visual attention metrics of CM and CP snippets.

		Type		Statistical Tests (Wilcoxon Signed-Rank Test)			
		CM	CP	Effect Direction	Statistic	p -Value	Corr. p -Value
All Fixations	Count	525	579	two-sided	2867.0	**0.046**	0.067
	Duration	177.0	171.4	two-sided	3620.0	0.979	0.979
AOI Fixations	Count	358	397	two-sided	2881.5	**0.050**	0.067
	Duration	123.9	121.2	two-sided	3582.5	0.901	0.979
Code Fixations	Count	358	305	less	2732.5	**0.018**	**0.037**
	Duration	123.9	100.3	less	2468.0	**0.001**	**0.003**
Comment Fixations	Count	-	92	greater	7260.0	$<$ **0.001**	$<$ **0.001**
	Duration	-	20.9	greater	7260.0	$<$ **0.001**	$<$ **0.001**

Tables 6.4, 6.5, 6.6, and 6.7 present the results of these analyses. Each table corresponds to one of the different fixation categories - All Fixations, AOI Fixations, Code Fixations, and Comment Fixations, respectively.

Table 6.4 demonstrates notable findings. For Snippets 1 – 4, there are significant increases in both fixation counts and durations when comments are present. This result aligns with expectations, considering that the inclusion of comments adds to the total text visible on the screen. However, the pattern observed in these snippets is not universally consistent. For the subsequent snippets, the impact of comments varies. In some cases, there is no significant difference in fixation counts between the two conditions, while in others, particularly Snippets 8, 9, and 10, an interesting trend emerges. In these snippets, not only does the fixation count decrease with the presence of comments, but there is also a notable reduction in the duration of fixations per participant. This suggests that comments, in certain contexts, may actually streamline the process of visual engagement, possibly making the code easier to comprehend or navigate.

Table 6.5 presents data on fixations specifically targeting the designated AOIs for code and comments. We observe a similar trend of variability as before, although not as pronounced. The data indicate that the presence of comments influences where participants focus their attention, but the effect varies across different snippets.

Table 6.4: LME results for all fixation counts and durations of each snippet. Lower numbers are generally preferable and suggest less cognitive load. The cell shading highlights a significant effect of comments (**positive** or **negative**). Values marked with an asterisk (*) indicate the use of alternate fitting methods ('powell', 'lbfgs') due to convergence issues, where the determinant of matrices approached zero, leading to linear algebra errors.

	All Fixations							
	Count				Duration			
Snippet	Intercept	Effect of CP	p-Value	Corr. p-Value	Intercept	Effect of CP	p-Value	Corr. p-Value
1	469.8	191.7	0.003	**0.004**	149.362	56.270	< 0.001	**< 0.001**
2	480.4	217.9	0.006	**0.013**	154.885	28.577	0.108	0.123
3	557.2	285.9	0.001	**0.002**	177.034	44.258	< 0.001	**< 0.001**
4	497.0	239.1	0.002	**0.003**	160.651	30.799	< 0.001	**< 0.001**
5	722.6	64.8	< 0.001	**< 0.001**	219.540	36.580	0.292	0.467
6	642.1	68.6	0.617	0.729	245.289	-18.819	0.653	0.729
7	456.3	25.7	0.650	1.000	183.148	-0.723	0.930	1.000
8	441.4	-30.2	0.768	0.768	163.040	-39.757	< 0.001	**< 0.001**
9	833.4	-351.2	< 0.001	**< 0.001**	269.014	-125.487	< 0.001	**< 0.001**
10	418.4	-35.8	0.689	0.689	138.568	-25.288	0.031	**0.049**
11	*610.0	*-56.0	*0.685	*0.685	214.397	-59.307	0.122	0.195
12	172.7	27.4	0.504	0.806	49.439	5.230	0.450	0.806

Table 6.5: LME results for AOI fixation counts and durations of each snippet. Lower numbers are generally preferable and suggest less cognitive load. The cell shading highlights a significant effect of comments (**positive** or **negative**).

	AOI Fixations							
	Count				Duration			
Snippet	Intercept	Effect of CP	p-Value	Corr. p-Value	Intercept	Effect of CP	p-Value	Corr. p-Value
1	314.7	133.0	< 0.001	**< 0.001**	97.9	40.9	0.066	0.075
2	333.4	178.8	< 0.001	**< 0.001**	108.4	25.3	0.060	0.081
3	324.8	228.4	< 0.001	**< 0.001**	102.7	43.1	< 0.001	**< 0.001**
4	321.8	127.8	0.054	0.087	104.5	18.9	0.417	0.477
5	534.2	75.6	0.477	0.635	174.8	35.5	0.223	0.447
6	377.4	78.2	0.280	0.559	141.8	9.9	< 0.001	**< 0.001**
7	334.2	8.2	0.876	1.000	145.5	-7.0	0.580	1.000
8	339.0	-41.8	0.124	0.142	133.5	-40.1	0.007	**0.011**
9	564.7	-250.2	< 0.001	**< 0.001**	191.9	-92.5	< 0.001	**< 0.001**
10	298.5	-72.7	0.229	0.262	100.9	-31.4	0.064	0.086
11	445.5	-10.8	0.640	0.853	156.4	-34.8	0.289	0.463
12	112.1	4.9	0.830	0.949	28.7	-0.1	0.994	0.994

Table 6.6: LME results for code fixation counts and durations of each snippet. Lower numbers are generally preferable and suggest less cognitive load. The cell shading highlights a significant effect of comments (**positive** or **negative**).

	Code Fixations							
	Count				Duration			
Snippet	Intercept	Effect of CP	p-Value	Corr. p-Value	Intercept	Effect of CP	p-Value	Corr. p-Value
1	314.7	74.9	< 0.001	**< 0.001**	97.9	30.4	0.270	0.270
2	333.4	105.4	0.045	0.072	108.4	10.2	0.499	0.499
3	324.8	20.4	< 0.001	**< 0.001**	102.7	-9.5	0.509	0.582
4	321.8	59.6	0.324	0.432	104.5	5.4	0.814	0.814
5	534.2	-63.0	0.557	0.637	174.8	4.4	0.890	0.890
6	377.4	-24.5	0.729	0.729	141.8	-12.4	0.535	0.729
7	334.2	-68.3	0.142	0.567	145.5	-27.2	0.219	0.583
8	339.0	-118.2	0.029	**0.038**	133.5	-54.5	< 0.001	**< 0.001**
9	564.7	-326.1	< 0.001	**< 0.001**	191.9	-109.7	< 0.001	**< 0.001**
10	298.5	-105.9	< 0.001	**< 0.001**	100.9	-37.4	0.008	**0.017**
11	445.5	-189.8	< 0.001	**< 0.001**	156.4	-80.0	0.001	**0.002**
12	112.1	-10.7	0.698	0.931	28.7	-2.5	< 0.001	**< 0.001**

Table 6.7: LME results for comment fixation counts and durations of each snippet. Lower numbers are generally preferable and suggest less cognitive load. The cell shading highlights a significant effect of comments (**positive** or **negative**). Values marked with an asterisk (*) indicate the use of alternate fitting methods ('powell', 'lbfgs') due to convergence issues, where the determinant of matrices approached zero, leading to linear algebra errors.

	Comment Fixations							
	Count				Duration			
Snippet	Intercept	Effect of CP	p-Value	Corr. p-Value	Intercept	Effect of CP	p-Value	Corr. p-Value
1	0.0	58.1	< 0.001	**< 0.001**	0.0	10.5	< 0.001	**< 0.001**
2	0.0	73.4	< 0.001	**< 0.001**	0.0	15.1	< 0.001	**< 0.001**
3	*0.0	*208.0	*< 0.001	***< 0.001**	0.0	52.6	< 0.001	**< 0.001**
4	0.0	68.2	< 0.001	**< 0.001**	0.0	13.5	< 0.001	**< 0.001**
5	0.0	138.6	< 0.001	**< 0.001**	0.0	31.0	< 0.001	**< 0.001**
6	0.0	102.7	< 0.001	**< 0.001**	0.0	22.3	< 0.001	**< 0.001**
7	*0.0	*76.5	*< 0.001	***< 0.001**	0.0	20.2	< 0.001	**< 0.001**
8	0.0	76.4	< 0.001	**< 0.001**	0.0	14.4	< 0.001	**< 0.001**
9	0.0	75.9	< 0.001	**< 0.001**	0.0	17.2	< 0.001	**< 0.001**
10	0.0	33.2	< 0.001	**< 0.001**	0.0	6.0	< 0.001	**< 0.001**
11	*0.0	*179.0	*< 0.001	***< 0.001**	0.0	45.2	< 0.001	**< 0.001**
12	0.0	15.6	< 0.001	**< 0.001**	0.0	2.4	< 0.001	**< 0.001**

However, the impact of comments becomes more distinct when analyzing Tables 6.6 and 6.7, which categorize fixations based on their focus on code versus comments. An interesting observation from these tables is the shift in visual attention from code to comments for Snippets 5 – 11. In these snippets, we find that the inclusion of comments corresponds to a decrease in both the count and duration of fixations on code elements, with some snippets showing significant reductions. Simultaneously, there is an increase in fixation counts and durations on the comments. This pattern suggests that comments, in these instances, may be redirecting attention away from the code to themselves, possibly providing clarifications or additional information that makes understanding the code easier. This effect underscores the potential of comments in guiding visual attention and influencing the comprehension process in programming tasks.

RQ2 Our study's findings illuminate the significant impact of comments on visual attention during program comprehension. We discovered that the presence of comments redirects approximately 23% of visual attention away from the code, suggesting a substantial shift in focus towards the comments. This reallocation of attention from code to comments underlines the influential role comments play in the cognitive process of understanding code. These insights, derived from detailed analysis of fixation counts and durations across Code and Comment AOIs, reveal a nuanced dynamic in how programmers interact with and process information in commented code.

6.3 effect of comments on linearity of reading order

Having established the significant role of visual attention in program comprehension, as evidenced by the substantial focus programmers dedicate to comments, we now extend our exploration to another critical dimension: the linearity of reading order in programming tasks as described in *Step IV:*b of Section 5.2.3.

6.3.1 *Local Metrics*

In examining the impact of comments on the local linearity of reading order in program comprehension, our analysis unveils nuanced effects. The presence of comments significantly reduces the Vertical Later Fixations, suggesting a more linear, top-to-bottom reading pattern with comments, with participants less likely to skip over multiple lines. Conversely, we observe an increase in Horizontal Later Fixations, suggesting that comments encourage more lateral scanning within a line. Interestingly, the Regression Rate diminishes in the presence of comments, pointing to a reduction in backward saccades, thereby streamlining the reading process. This trend is combined with an increased Line Regression Rate, hinting at more frequent revisits within a line, potentially to reconcile code understanding with accompanying comments. Notably, the Saccade Length remains consistent across both scenarios, underscoring a uniformity in the distance of eye movements regardless of comment inclusion.

Table 6.8 presents our findings, offering a statistical analysis of various metrics such as Vertical Next and Later Fixations, Regression Fixations, Horizontal Later Fixations, Line Regression Fixations, and Saccade Length.

Table 6.8: Comparative analysis of local linearity metrics of CM and CP snippets.

	Type		Statistical Tests (Wilcoxon Signed-Rank Test)			
Metric	CM	CP	Effect Direction	Statistic	p -Value	Corr. p -Value
Vertical Next Fixations	14.4%	14.3%	two-sided	14.0	1.000	1.000
Vertical Later Fixations	14.3%	12.1%	two-sided	1.0	**0.011**	**0.018**
Regression Fixations	28.7%	25.7%	two-sided	2.0	**0.001**	**0.004**
Horizontal Later Fixations	16.6%	22.8%	two-sided	0.0	$<$ **0.001**	**0.003**
Line Regression Fixations	14.8%	16.1%	two-sided	3.0	**0.012**	**0.018**
Saccade Length (in pixels)	43.1	41.6	two-sided	29.0	0.470	0.564

6.3.2 *Global Metrics*

Next, we present the results of our study on the impact of comments on the global metrics of code reading. The N-W scores obtained from this analysis quantified how participants' reading patterns aligned with the expected reading sequences of the code. Higher scores translate to better alignment while lower scores indicate poor alignment. The `RTGCT` offers a visual representation of the varied line reading orders we used for comparison. Figure 6.2 depicts the different line reading orders for Snippet 1. The radial transition graphs showing the global line orders for all snippets can be found in the appendix (Figures A.14 and A.15).

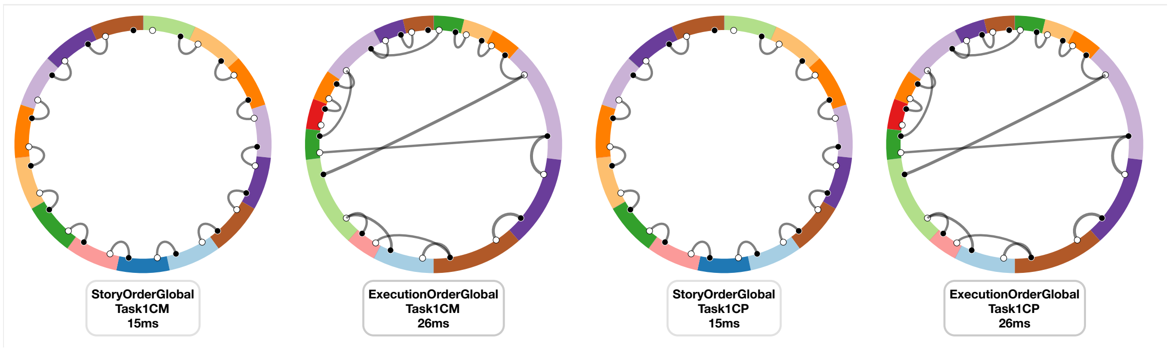


Figure 6.2: `RTGCT` visualization of different line reading orders for Snippet 1.

Our aggregated findings show that the dynamic N-W scores (Story and Execution Order) consistently surpass the naive scores across the snippets. This indicates that the algorithm's dynamic adaptation better reflects participants' reading order, which usually includes several iterations of reading the same lines. Furthermore, the reading order mostly resembled the execution order of the code rather than the story order. Comments seem to have little to no impact on the N-W scores of the different metrics.

This suggests that the presence of comments does not uniformly influence the linearity of reading patterns as initially hypothesized. The results are visually summarized in Figure 6.3, which illustrates the negligible effect of comments on global metrics. Furthermore, Table 6.9 corroborates these insights, demonstrating that the inclusion of comments does not significantly affect the evaluated metrics when aggregated.

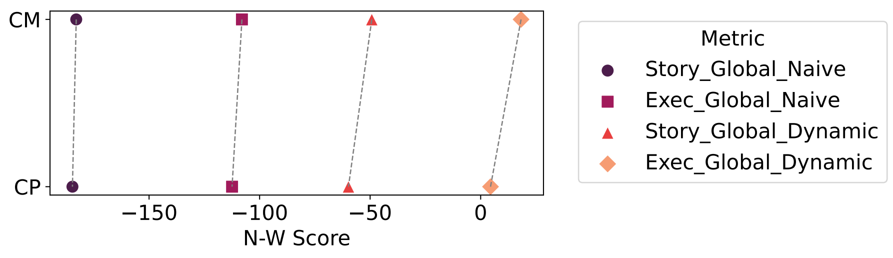


Figure 6.3: Aggregated linearity N-W scores for CM and CP snippets. The figure illustrates the aggregate scores for both story and execution reading sequences using naive and dynamic N-W calculations.

Table 6.9: Comparative analysis of global linearity N-W scores of CM and CP snippets.

	N-W Score		Statistical Tests (Wilcoxon Signed-Rank Test)			
Metric	CM	CP	Effect Direction	Statistic	p-Value	Corr. p-Value
Story Order Naive	-182.95	-184.67	two-sided	3451.0	0.639	0.639
Exec Order Naive	-107.99	-112.46	two-sided	3226.5	0.362	0.483
Story Order Dynamic	-49.25	-59.80	two-sided	2935.0	0.069	0.137
Exec Order Dynamic	18.27	4.46	two-sided	2751.5	0.057	0.137

However, deeper examination of the individual snippets presents a more detailed perspective. For Snippets 1 – 6, we observe a predominantly negative effect of comments on both naive story and execution orders, with some effects reaching statistical significance. Conversely, for Snippets 7 – 12, we see a tendency for comments to be correlated with a more linear reading order. This pattern is consistent in the dynamic story order, where comments appear to decrease linearity in Snippets 1 – 6 and increase it in Snippets 7 – 12. Tables 6.10 and 6.11 further emphasize these findings.

As illustrated in Figure 6.4, snippets where comments had a positive impact (see 6.1, Snippets 7 – 12), comments enhanced the story metric but adversely influenced execution order alignment. Essentially, when comments were actually beneficial, they made the code reading order more linear, whereas in snippets 1 – 6, where comments had no or negative effect, the alignment with the story order decreased.

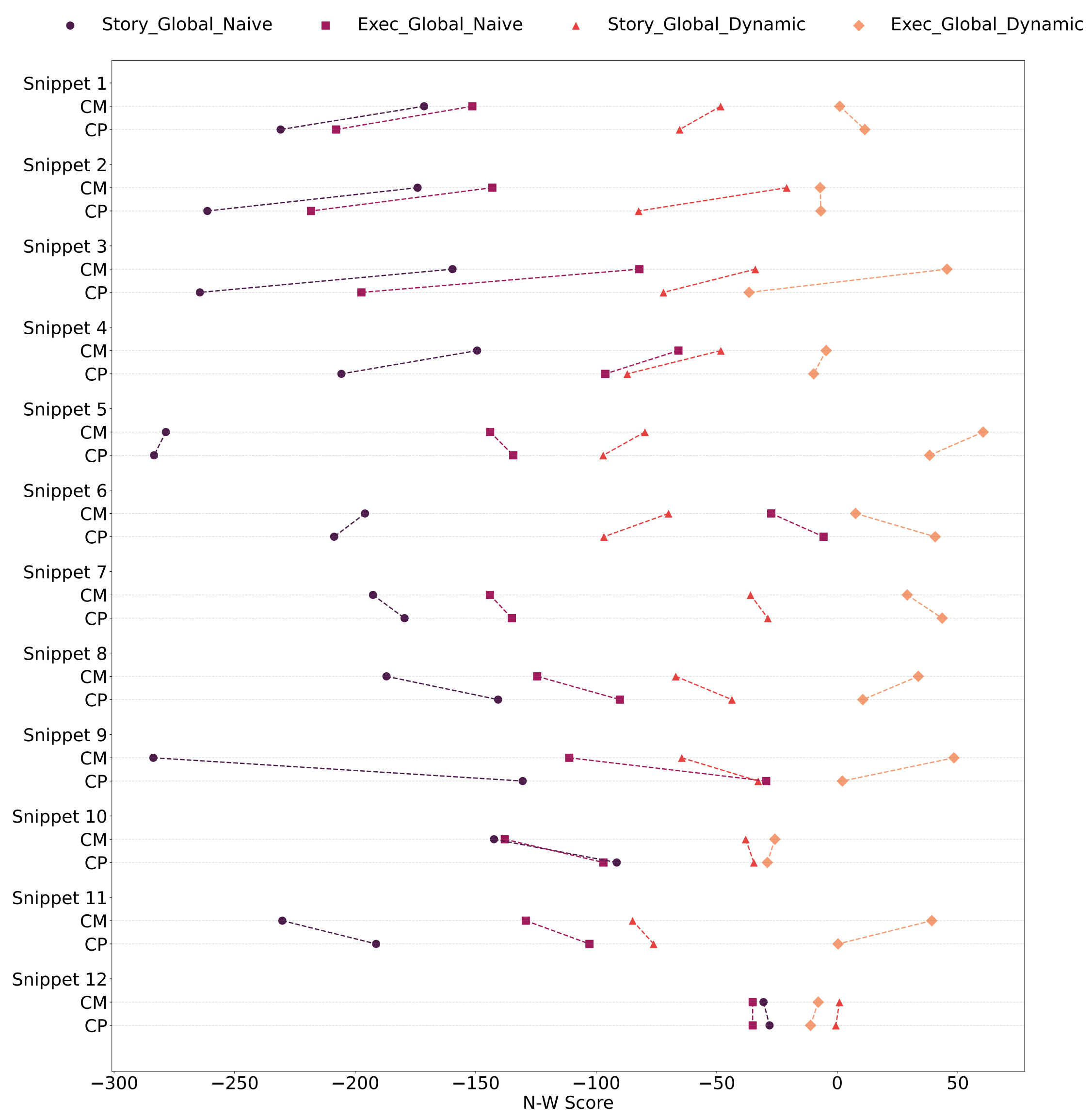


Figure 6.4: Global linearity N-W scores for CM and CP snippets. Higher scores translate to better alignment while lower scores indicate poor alignment with the expected reading sequences.

Table 6.10: LME results for naive linearity metrics of each snippet. Higher numbers are preferable and suggest a better alignment with the compared reading order. The cell shading highlights a significant effect of comments (**positive** or **negative**).

	Story Global Naive				Exec Global Naive			
Snippet	Intercept	Effect of CP	p-Value	Corr. p-Value	Intercept	Effect of CP	p-Value	Corr. p-Value
1	-171.4	-59.5	0.093	0.192	-151.4	-56.5	0.096	0.192
2	-174.1	-87.2	0.001	**< 0.001**	-143.1	-75.2	0.001	**< 0.001**
3	-159.6	-104.8	0.003	**0.003**	-82.1	-115.3	0.001	**< 0.001**
4	-149.4	-56.3	0.001	**< 0.001**	-65.9	-30.3	0.290	0.387
5	-278.5	-4.9	0.911	0.911	-144.0	9.6	0.775	0.911
6	-195.9	-12.8	0.762	0.762	-27.4	21.7	0.001	**< 0.001**
7	-192.6	13.1	0.593	0.683	-144.1	9.1	0.683	0.683
8	-187.0	46.3	0.237	0.272	-124.5	34.3	0.255	0.272
9	-283.7	153.2	0.001	**< 0.001**	-111.2	81.7	0.001	**< 0.001**
10	-142.4	50.9	0.210	0.571	-137.9	40.9	0.286	0.571
11	-230.2	38.9	0.585	0.799	-129.2	26.4	0.619	0.799
12	-30.6	2.5	0.800	1.000	-35.1	0.000	1.0	1.0

Table 6.11: LME results for dynamic linearity metrics of each snippet. Higher numbers are preferable and suggest a better alignment with the compared reading order. The cell shading highlights a significant effect of comments (**positive** or **negative**).

	Story Global Dynamic				Exec Global Dynamic			
Snippet	Intercept	Effect of CP	p-Value	Corr. p-Value	Intercept	Effect of CP	p-Value	Corr. p-Value
1	-48.4	-17.0	0.402	0.402	1.0	10.4	0.181	0.242
2	-21.0	-61.4	< 0.001	**< 0.001**	-7.1	0.3	0.976	0.976
3	-34.0	-38.1	< 0.001	**0.001**	45.5	-82.1	< 0.001	**< 0.001**
4	-48.3	-38.8	< 0.001	**< 0.001**	-4.6	-5.2	0.538	0.538
5	-79.8	-17.3	0.518	0.911	60.5	-22.2	0.181	0.723
6	-70.0	-26.8	0.033	0.065	7.6	33.0	0.346	0.461
7	-36.0	7.2	0.506	0.683	29.0	14.5	0.193	0.683
8	-67.0	23.3	0.211	0.272	33.6	-23.0	0.272	0.272
9	-64.5	31.7	< 0.001	**< 0.001**	48.4	-46.3	0.138	0.138
10	-38.0	3.4	0.829	0.829	-25.9	-3.1	0.675	0.829
11	-84.9	8.7	0.799	0.799	39.2	-38.9	< 0.001	**< 0.001**
12	0.9	-1.5	0.816	1.000	-7.9	-3.2	0.166	0.663

RQ_3 Our investigation into the influence of comments on the linearity of reading order reveals nuanced interactions. The analysis demonstrates that comments significantly alter local reading patterns, leading to a more linear progression through the code. In addition to that, an increase in Horizontal Later Fixations points to more lateral focus within lines, possibly reflecting a deeper engagement with comments. The decrease in Regression Fixations, combined with an uptick in Line Regression Fixations, underscores a more focused yet revisitory reading behavior. Despite these local effects, comments showed limited impact on global reading linearity, when aggregated and measured using the N-W algorithm, challenging our initial hypothesis of a uniform influence. However, the more detailed analysis of the individual snippets reveal interesting patterns that suggest a more complex relationship than previously anticipated.

6.4 effect of comments on gaze strategy

Expanding on our prior analysis of reading order linearity, we now explore gaze strategies in programming tasks. For a detailed analysis, we employed local and global measures (cf. *Step IV:*c of Section 5.2.3) to provide insights into immediate gaze patterns as well as reveal how participants navigate between code and accompanying comments on a broader perspective.

6.4.1 *Local Metrics*

Regarding the local gaze strategy metrics, our findings show that the saccade counts are relatively balanced between the two transition types for most snippets. However, there seems to be a general tendency for more Code-to-Comment transitions, with certain snippets exhibiting statistically significant differences. The overall saccade counts across all snippets show a significant difference (p-value = 0.007), indicating a general tendency for programmers to transition more from code to comments than vice versa. Table 6.12 encapsulates these findings for the local gaze strategy metrics.

Table 6.12: Comparative analysis of saccade counts between code-to-comment and comment-to-code gaze strategies.

	Gaze Strategy (Saccade Counts)		Statistical Tests (Wilcoxon Signed-Rank Test)		
Snippet	Code-to-Comment	Comment-to-Code	Effect Direction	Statistic	p -Value
1	4.5	4.7	two-sided	14.5	0.608
2	5.2	4.8	two-sided	23.5	0.770
3	11.7	8.6	two-sided	4.0	**0.050**
4	9.5	7.6	two-sided	4.5	**0.031**
5	13.5	10.3	two-sided	5.0	**0.020**
6	6.8	6.7	two-sided	21.0	0.855
7	7.0	7.5	two-sided	7.5	0.527
8	6.2	5.4	two-sided	12.0	0.398
9	7.1	5.6	two-sided	6.5	0.105
10	2.3	1.4	two-sided	11.0	0.168
11	15.6	12.0	two-sided	7.0	0.066
12	3.6	2.7	two-sided	5.0	0.236
Overall	7.8	6.4	two-sided	6.0	**0.007**

6.4.2 *Global Metrics*

We now shift our focus to the global metrics. Figure 6.5 visually represents the different AOI reading orders employed for Snippet 1 using radial transition graphs. For the complete set of radial transition graphs refer to Figures A.16 and A.17 in the appendix.

Aligning the participants AOI reading order to these predefined sequences and comparing 'Naive' and 'Dynamic' scores within both 'Code-First' and 'Comment-First' approaches showed again a consistent pattern of improved performance in the 'Dynamic' calculation further supporting the iterative reading nature of source code. It is however evident that both 'Code-First' and 'Comment-First' approaches result in similar performance scores, with no significant statistical difference between them. This is demonstrated across different metrics, including Story Order and Execution Order, in both Naive and Dynamic algorithms. The negative N-W Scores across all categories highlight a below-baseline performance, suggesting challenges in aligning gaze patterns even with the best aligned reading sequence. The detailed N-W scores of the different reading orders across the snippets are illustrated in Figure 6.6. Furthermore, a detailed comparative analysis of the scores between the two gaze strategies can be found in Table 6.13.

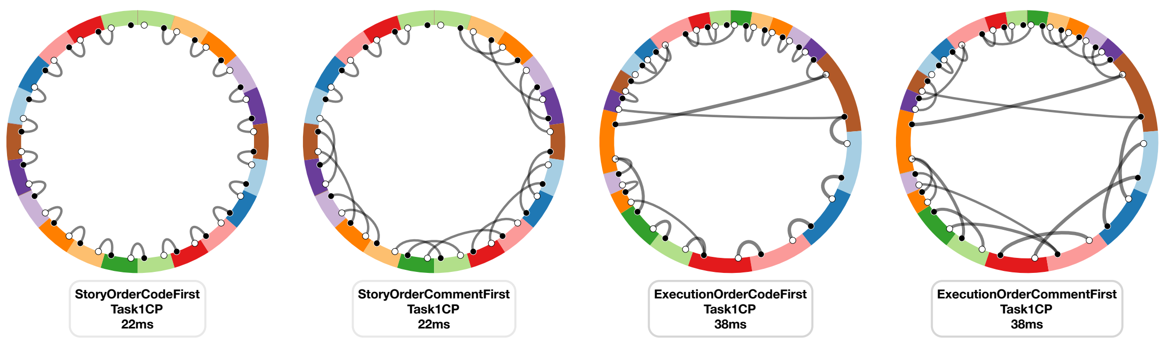


Figure 6.5: RTGCT visualization of AOI order for Snippet 1 CP.

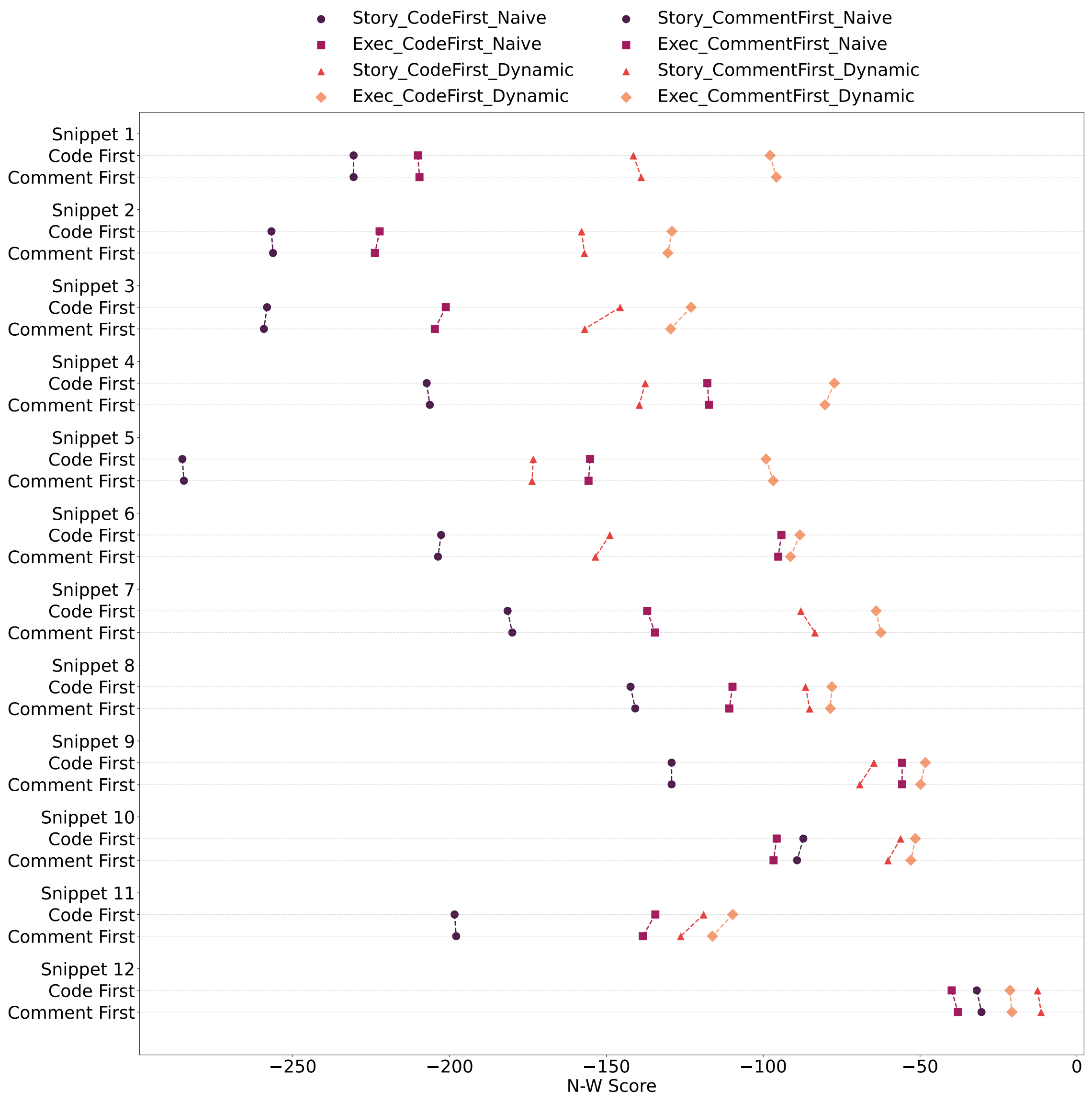


Figure 6.6: Gaze strategy N-W scores for CP snippets. Higher scores indicate better alignment while lower scores indicate poor alignment with the expected gaze strategy sequences.

Table 6.13: Comparative analysis of N-W Scores between code-first and comment-first gaze strategies.

	Global Gaze Strategy (N-W Score)		Statistical Tests (Wilcoxon Signed-Rank Test)			
Metric	Code-First	Comment-First	Effect Direction	Statistic	p-Value	Corr. p-Value
Story Order Naive	-92.1	-92.0	two-sided	20.0	0.441	0.441
Execution Order Naive	-65.6	-65.8	two-sided	21.0	0.284	0.379
Story Order Dynamic	-55.5	-56.5	two-sided	23.5	0.224	0.379
Execution Order Dynamic	-41.1	-41.9	two-sided	22.5	0.195	0.379

RQ$_4$ Our study into gaze strategies highlights a general tendency for programmers to move their gaze from code to comments, indicating a directional strategy that seeks additional context or clarification within comments after having read the code first. Despite this, the assessment of global gaze strategies through the N-W algorithm reveals no significant preference between 'Code-First' and 'Comment-First' approaches. This suggests that while local gaze transitions favor a move towards comments, the overarching strategy of navigating code and comments does not significantly lean towards starting with one over the other on a global scale.

6.5 effect of comments on participants' perception

In this final section, we examine participants' difficulty and comment contribution ratings, offering insights into their subjective experiences with program comprehension. This examination is crucial for understanding the perceived impact of comments on navigating code complexity and enhancing the comprehension process.

The examination of difficulty ratings across CM and CP snippets uncovers intriguing patterns. For example, Snippet 1 exhibits a slight decrease in perceived difficulty with comments, moving from a mean rating of 2.7 in the CM condition to 2.5 in the CP condition. This suggests a marginal but positive impact of comments. Conversely, Snippet 2 sees an increase in difficulty from 2.4 to 3.1 with the introduction of comments, hinting at the potential for comments to sometimes introduce additional complexity or distraction. Among the different snippets, Snippet 9 exhibited a substantial 34% reduction in perceived difficulty with comments ($p<0.001$), underscoring comments' potential to significantly clarify and simplify the comprehension process.

Interestingly, the effect of comments on perceived difficulty does not uniformly correlate with their contributory ratings to comprehension. While Snippet 9's significant ease in difficulty is paralleled by a high comment contribution rating (3.9), other snippets with negative or minimal changes in difficulty still report substantial comment contribution scores, such as Snippet 4 (3.7) and Snippet 10 (3.8). This suggests that while comments can directly influence the perceived difficulty of snippets, their value in enhancing comprehension may be perceived independently of this difficulty impact.

Table 6.14 details the comprehensive results and specific distributions of difficulty and comment contribution ratings across snippets, providing a granular view of how participants' perceptions vary by snippet. Table 6.15 then synthesizes these individual observations into a broader context, presenting the mean ratings across snippets to summarize the overarching trends in perceived difficulty and the value of comments in program comprehension. Additionally, Figure 6.7 visualizes these summarized findings and illustrates the relationship between the effect on difficulty and the perceived comment contribution.

Even when comments do not significantly alter the difficulty level, their presence is appreciated for providing context, clarifying intent, or offering insights that aid in understanding complex code segments.

Table 6.14: Participants' Likert scale ratings on snippet difficulty and comment contribution.

			Ratings (Likert Scale 1 → 5)	
			Difficulty	Comment Contribution
Snippet	Description	Type	(Very Easy → Very Difficult)	(Not Helpful → Very Helpful)
1	Identifies a peak element in an array.	CM	2 2 3 3	3 7 2 5 3
		CP	2 4 2 1 1	
2	Finds two elements that sum to a target.	CM	2 4 3 1	5 7 3 5
		CP	4 3 1 2	
3	Computes maximum zero-sum subarray length.	CM	1 2 3 1 3	1 5 4 6 4
		CP	1 1 5 3	
4	Longest consecutive sequence in array.	CM	3 2 5	1 1 6 8 4
		CP	1 4 5	
5	Longest common subsequence between strings.	CM	2 1 1 6	6 5 5 4
		CP	3 2 5	
6	Longest increasing subsequence length.	CM	1 7 2	1 5 1 9 4
		CP	1 4 4 1	
7	Efficient power calculation.	CM	1 5 3 1	6 3 3 4 4
		CP	2 4 2 1 1	
8	Fibonacci number at given position.	CM	4 4 2	4 4 6 5 1
		CP	4 4 2	
9	Lists primes up to a number (Sieve of Eratosthenes).	CM	3 4 2 1	7 8 5
		CP	7 2 1	
10	Binary search in sorted array.	CM	4 4 2	1 3 3 6 7
		CP	8 1 1	
11	Counts palindromic substrings.	CM	3 2 3 2	3 4 2 4 7
		CP	2 1 1 3 3	
12	Anagram check for two strings.	CM	8 1 1	7 4 1 2 6
		CP	10	

Table 6.15: Summary of mean difficulty and comment contribution ratings. The overall effect represents the percentage change in difficulty rating between CM and CP snippets. The cell shading highlights a significant effect of comments (**positive** or **negative**).

	Difficulty (Likert Scale 1 → 5)				Comment
Snippet	CM	CP	Overall Effect	p-Value	Contribution
1	2.7	2.5	-4%	0.603	2.9
2	2.4	3.1	+14%	0.062	3.4
3	3.3	3.0	-6%	0.431	3.4
4	3.2	3.4	+4%	0.564	3.7
5	3.9	4.2	+6%	**< 0.001**	3.4
6	4.1	3.5	-12%	0.062	3.5
7	2.4	2.5	+2%	0.769	2.9
8	1.8	1.8	0%	1.000	2.8
9	3.1	1.4	-34%	**< 0.001**	3.9
10	1.8	1.3	-10%	0.128	3.8
11	3.4	3.4	0%	1.000	3.4
12	1.4	1.0	-8%	0.168	2.8

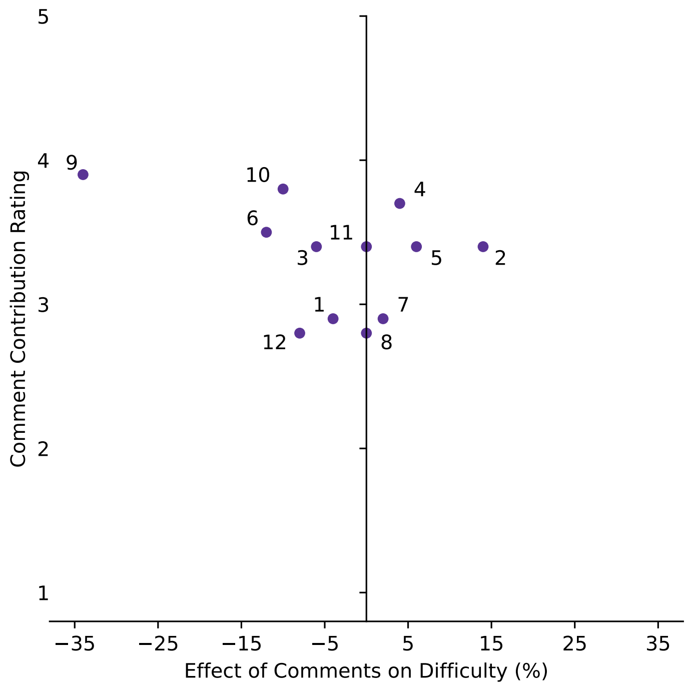


Figure 6.7: Relationship between the effect of comments on difficulty and their contribution ratings.

RQ_5 Our analysis of participants' difficulty and comment contribution ratings shows a snippet-dependent influence of comments on perceived difficulty. Notably, Snippet 9 stands out with a substantial 34% reduction in perceived difficulty when comments are present, highlighting the potential of good comments to significantly clarify and simplify the comprehension process. Conversely, snippets such as Snippet 2 indicate that comments can sometimes introduce additional complexity, suggesting the importance of comment quality and relevance. Despite these variations, consistently high comment contribution ratings across snippets indicate a generally positive perception of comments.

7 DISCUSSION

This study aimed to explore the impact of comments on program comprehension, focusing on how comments affect program understanding, visual attention, linearity of reading order, gaze strategies, and participants' perceptions. By conducting a detailed analysis involving eye-tracking data and participant feedback, we have gained insights that contribute to the existing body of knowledge on the role of comments in software development.

Central to our investigation was not only the collection of eye-tracking data but also the gathering of qualitative feedback through the post-questionnaire (cf. Section 4.6) designed to capture participants' perceptions and experiences with comments in code. The incorporation of participant feedback provides a rich, qualitative layer to our analysis, allowing us to delve deeper into the subjective experiences of programmers as they navigate code with and without comments.

The discussion that follows is structured that way to weave together the insights from our quantitative analysis with the qualitative themes identified through the thematic analysis (cf. Section 5.4). By integrating these findings, we aim to offer a comprehensive overview of the impact of comments on program comprehension, highlighting the significant role comments play in aiding understanding, guiding attention, and influencing reading and gaze strategies.

7.1 PROGRAM COMPREHENSION

Our findings indicate that the impact of comments on program comprehension is complex and varies according to specific contexts, suggesting a more nuanced effect than the uniformly positive improvements reported by Dunsmore [33] and Tenny [118, 119]. For instance, Snippet 9, which listed primes up to a given number, demonstrated a significant improvement in comprehension when comments were included (+34.1%). On the other hand, Snippet 1, which identified a peak element in an array, showed a significantly decreased performance as a result of the comments added to it (-30.0%). This variability in the effectiveness of comments seems to depend on various factors such as the complexity of the code, the quality of the comments, and the background knowledge of the programmer. These findings align with the conditional improvements noted by Woodfield et al. [128] and the varied outcomes for different user groups reported by Salviulo and Scanniello [87]. It suggests that the effectiveness of comments is not universally applicable but contingent on the interplay of various factors, which is also supported by Nurvitadhi et al. [69], who underscore the importance of considering the specific application and content of comments in programming tasks.

Our study, therefore, contributes to the discourse on comments in programming by offering evidence of their context-dependent utility, subtly diverging from the findings of researchers like Sheppard et al. [99], Börstler and Paech [18], and Nielebock et al. [67], who observed minimal or no influence of comments on comprehension for small tasks.

From the responses regarding the impact of comments on program comprehension, several key phrases and concepts emerged, providing some perspectives on how comments influence understanding of code. Here are a some of the identified themes:

- **Clarification and Summary:** Comments are valued for providing structure and summarizing complex code segments, making them easier to understand. For instance, comments that "give a structure to understanding", or "can summarize complicated code sections simply" are highlighted as particularly helpful.
- **Providing Context and Facilitating Comprehension:** For code snippets that were not immediately clear, comments not only provided necessary context, which was particularly valued in instances of uncertainty, but many responses also indicated that comments facilitated the comprehension process, as seen in phrases like "Comments have generally simplified and sped up my understanding."
- **Intention Behind the Code:** Comments adding new information about the intention behind the code or explaining complex lines are seen as beneficial. However, comments that "merely restate easy to understand lines were not helpful."
- **Relevance and Quality of Comments:** The effectiveness of comments is tied to their relevance and quality. Short, descriptive comments are appreciated for aiding comprehension, whereas "misleading comments" are criticized.
- **Selective Reading Based on Understanding:** Some participants indicated they would not read comments if they understood the code, suggesting a selective approach to using comments based on immediate comprehension needs.

The analysis reveals that comments play a crucial role in facilitating program comprehension by providing clarification, summarizing complex parts, and explaining the intention behind code. Their utility, however, is highly dependent on their quality and relevance, with effective comments being those that are concise, informative, and directly related to the complexities or uncertainties of the code. Participants' engagement with comments appears to be selective, influenced by their initial understanding of the code and the perceived value of the comments in enhancing that understanding.

In comparing these findings with Nielebock et al. [67], we note some parallels and differences. Like our study, Nielebock et al. found that participants generally viewed comments as potentially helpful in reducing the time for comprehension, which aligns with our observations on speeding up comprehension and providing context. However, they also noted no significant differences in the perceived effectiveness of different types of comments, which contrasts with our findings that suggest that the relevance and quality of comments plays a role in their effectiveness.

In summary, our research adds an additional layer to the understanding of how comments affect program comprehension, drawing attention to the subtleties and variations in their impact. Moving forward, these insights pave the way for further research into optimizing the use of comments in programming, focusing on how they can be tailored to enhance comprehension and efficiency in diverse coding scenarios.

7.2 VISUAL ATTENTION

Our quantitative analysis, enriched by thematic insights from participants, highlights the significant impact comments have on visual attention during program comprehension. The eye-tracking data demonstrated that a notable portion of visual attention – approx. 23% – was directed towards comments when present, underscoring their vital role in engaging programmers. This finding is complemented by the themes identified through participants' feedback, which further elucidate the nuances of how comments attract visual attention:

- **Variable Attention Allocation:** Participants exhibited a wide range of attention distribution between comments and code. While some participants were drawn more to comments, particularly noting variable declarations and output lines, others remained primarily focused on the code, occasionally overlooking comments.
- **Dependency on Code Clarity:** The emphasis on comments was markedly reduced when participants encountered clear code. This suggests that the presence of comments is particularly valued in instances of complexity or when the code lacks intuitiveness, serving as a beacon for clarification.
- **Proportion of Focus:** A striking observation from some participants highlighted a focus distribution where comments commanded a majority of the attention, with ratios self-reported as 65% on comments to 35% on code. This indicates that, under certain circumstances, comments can dominate the visual exploration of code, potentially guiding comprehension and focus.

Furthermore, the thematic analysis uncovered an increased reliance on comments among participants with varying levels of experience, indicating that comments are not just complementary but essential components of code that significantly influence comprehension strategies. Participants frequently shifted their primary focus to comments, particularly in search of clarifications or additional explanations not readily apparent in the code itself. This shift often occurred when comments were perceived as more informative or when the code alone was deemed insufficient for full comprehension.

The relation between comments and visual attention in program comprehension has profound implications for software development practices. Firstly, the findings emphasize the importance of clear and informative comments, especially in complex or unclear code segments, to aid programmers in navigating and understanding codebases efficiently. Secondly, the variability in attention allocation between comments and code highlights the need for tailored commenting strategies that consider the diverse preferences and experiences of programmers. Lastly, the significant reliance on comments for understanding underscores the value of comments and the need for further investigation on commenting practices.

7.3 LINEARITY OF READING ORDER

Following our exploration of how comments shape visual attention, we delved into their influence on the linearity of reading order. Quantitatively, we observed that comments significantly altered local reading patterns, encouraging a more linear, top-to-bottom progression through the code. This was coupled with an increase in Horizontal Later Fixations, suggesting a lateral focus within lines that likely reflects deeper engagement with comments. Concurrently, a decrease in Regression Fixations, paired with an increase in Line Regression Fixations, highlighted a more focused yet revisitory reading behavior. These shifts suggest that comments can both guide a systematic approach to reading code and encourage programmers to seek out specific, detailed understanding within the codebase. However, the impact of comments on the global linearity of reading, as measured using the N-W algorithm, was limited when aggregated. This finding challenges the initial hypothesis of a uniform influence of comments on reading order, suggesting a more complex relationship that varies across individual snippets and programmer experiences.

The thematic analysis of participant feedback, sheds light on the different ways comments impact reading strategies:

- **Top-Down vs. Selective Reading:** The presence of comments appears to modify reading strategies from a purely top-down approach to a more selective one. Participants often shifted focus to specific lines of interest, particularly output lines or those directly clarified by comments, indicating a departure from linear navigation to a more targeted engagement with the code.

- **Impact on Code Navigation:** Responses suggests that without comments, programmers might find themselves more inclined to jump around the code. This implies that comments serve not just as explanatory aids but also as navigational beacons, offering a structured pathway through the complexities of code, thereby enhancing the comprehensibility and accessibility of the codebase.

- **Mixed Effects on Linearity:** The effects of comments on the linearity of reading patterns were mixed. While some participants observed no change in their reading order, others noted that comments influenced them to prioritize specific lines. However, a generally top-down reading pattern was maintained, albeit with deviations to accommodate the insights offered by comments.

These insights highlight the significant role comments play in aiding program comprehension and navigation, especially in guiding developers through complex or ambiguous sections. Nonetheless, the research reveals a critical gap in the existing knowledge regarding the optimal placement and formulation of comments within code to maximize their utility. This gap points to the issue that the effectiveness of comments is not universal but may vary significantly among developers, suggesting that what constitutes a "good" comment is subjective and potentially dependent on individual preferences and coding styles.

This variability in the perceived value of comments underscores the need for further exploration into commenting practices and developer education. It indicates that a universal approach to commenting may not be sufficient. Instead, there is a need for further research to define more effective guidelines that can accommodate the diverse perspectives and preferences of developers. Specifically, future studies could benefit from exploring the impact of different types of comments on program comprehension by comparing identical code snippets annotated with varied commenting styles.

An intriguing extension of this research could also explore the integration of Large Language Models (LLMs) into Integrated Development Environments (IDEs) to provide personalized comment generation, similar to the work of Wong et al. [127] and McBurney and Mcmillan [65]. The extension would analyze the code being read and automatically generate comments that are tailored to the developer's specific reading and comprehension strategies. By doing so, it could significantly reduce the time developers spend on writing comments and improve code understanding efficiency. Such an approach would not only personalize the development experience but also potentially transform commenting practices by making them more dynamic and adaptive to individual needs.

7.4 GAZE STRATEGY

Building on our insights into how comments influence visual attention and the linearity of reading order, we further explored their impact on gaze strategies during program comprehension. Our investigation aimed to understand how programmers navigate between code and comments.

The findings from our study reveal a nuanced picture of how programmers navigate between code and comments, with local metrics indicating a statistically significant preference for a Code-First approach. This preference suggests that, at least in the context of shorter or more focused interactions, programmers tend to engage with the code before seeking out comments for additional context or clarification. In contrast, the global metrics did not reveal a definitive preference for starting with code versus comments, highlighting a potential limitation in our methodology. The application of the N-W algorithm, while effective in certain analytical contexts, may not have been ideally suited to capture the slight differences in AOI orders between the sequences, potentially obscuring subtle but significant gaze strategy patterns.

Complementing our quantitative analysis, the thematic analysis of participant feedback offers additional insights into the gaze strategies among programmers. This qualitative approach revealed a clear division in preferences and behaviors:

- **Comment-First vs. Code-First Strategies:** Participants exhibited distinct preferences, with some focusing on comments before delving into the code, while others prioritized understanding the code directly, resorting to comments only when further clarification was needed. A few participants even adopted a "code-only" approach, bypassing comments unless they deemed it absolutely necessary for comprehension.

- **Adaptive Strategies:** A significant insight from our study is the adaptability of programmers' gaze strategies. Participants reported adjusting their focus based on the perceived utility of comments, indicating a highly contextual and responsive approach to navigating code. This adaptability suggests that programmers are not rigid in their strategies but are instead capable of dynamically altering their focus to maximize comprehension.

Given these observations, future studies should consider employing a mix of local and global metrics, potentially integrating alternative measures that are better suited for capturing the intricacies of programmer behavior. A more appropriate design might involve focusing on shorter code snippets which could offer a more controlled environment to compare code-first versus comment-first reading strategies. Shorter snippets are likely to result in shorter AOI sequences, facilitating a more accurate alignment score with fewer gaps. This methodological adjustment promises a clearer understanding of programmers' navigational preferences by minimizing the confounding factors present in longer code segments.

Overall, the exploration of gaze strategies reveals the critical role comments play as dynamic elements that programmers interact with in a strategic manner. This interaction is not only a matter of preference but is also deeply influenced by the content and perceived utility of comments, suggesting that effective commenting practices are crucial for enhancing program comprehension.

7.5 PARTICIPANTS' PERCEPTION

Following our examination of gaze strategies, we turn our focus towards understanding how participants perceive the role of comments in program comprehension. This perspective offers a direct insight into the subjective experience of programmers as they navigate through code, complemented by comments.

Our analysis reveals nuanced patterns that underscore the multifaceted impact of comments. Significantly, Snippet 9 exhibited a notable reduction in difficulty by 34% with comments, highlighting how substantial the effect of well-written comments could be.

Interestingly, the impact of comments on snippet difficulty did not always align with their perceived helpfulness, mirroring the discrepancy observed by Nielebock et al. [67] between expectations of comments' utility and actual performance outcomes. Despite participants consistently perceiving and rating comments as helpful, this did not uniformly translate to a reduction in perceived snippet difficulty.

The thematic analysis of participant responses further deepens our understanding, unveiling several key themes:

- **Essential for Understanding Complex Code:** Participants overwhelmingly recognized comments as indispensable, particularly for navigating through challenging code blocks. This highlights comments as "lifelines" in certain situations and indispensable in reducing code complexity.

- **Varied Contribution:** While some participants viewed comments as potentially time-consuming for straightforward code snippets, others praised their role in making conditions, loops, and complex structures more clear.
- **Perception of Code Complexity:** Interestingly, the presence of comments sometimes contributed to an initial perception of increased code complexity. This indicates that the mere presence of comments can influence programmers' first impressions, potentially framing their approach to understanding the code.

Combining these insights, our study illustrates that while comments can directly influence the perceived difficulty of programming tasks, they are consistently valued for their broader contribution to comprehension. Even in scenarios where comments appear to increase complexity, their overall contribution to clarifying code logic and intent is highly regarded.

This refined understanding of comments – captured through both quantitative ratings and qualitative themes – highlights their critical role in software development. Comments serve not just as aids for reducing immediate snippet difficulty but as essential tools for enhancing the deeper comprehension of code. They provide context, clarify intent, and offer insights crucial for understanding complex code segments.

Our findings suggest that the effectiveness of comments is contingent upon their quality, relevance, and the context in which they are used. Future research should further explore how to optimize comment use to maximize their benefits in program comprehension, considering factors such as code complexity, programmer experience, and the specific needs of different programming tasks and comprehension strategies.

8 THREATS TO VALIDITY

8.1 CONSTRUCT VALIDITY

Construct validity was a critical consideration in our eye-tracking study to ensure that the measured variables effectively represent the intended constructs. Our study employed various eye movement measures to assess participants' visual behavior during program comprehension and the interaction between code and comments.

One potential threat was the possibility of participants using peripheral vision or not focusing directly on a source code line during eye tracking sessions [71]. To address this concern, we employed eye-movement measures that are based on matching fixations to specific AOIs. This approach, similar to the work of Busjahn et al. [22], allowed us to capture participants' actual fixation patterns and align them accurately with the corresponding code elements. By considering horizontal proximity, we accounted for peripheral vision while ensuring a reliable assessment of participants' reading behavior.

For detailed descriptions of the operationalizations of program comprehension, visual attention, linearity, gaze strategies, and participants' perceptions refer to Section 4.4. By diligently considering construct validity and aligning our operationalizations with established measures and methods, we ensure the accuracy and reliability of our study's findings regarding the effects of comments on program comprehension and gaze strategies during code and comment interaction.

8.2 INTERNAL VALIDITY

Internal validity is crucial to ensure the accurate measurement of the cause-and-effect relationships between the independent and dependent variables in our study. To enhance internal validity, we have employed a carefully designed within-subject experimental design. Each participant was exposed to both conditions (Comments Present and Comments Missing) in a counterbalanced order to control for order effects. The random assignment of code snippets to participants further minimized potential biases or confounding factors.

The use of eye-tracking data allowed for accurate and objective measurements of participants' visual attention during program comprehension. To ensure the reliability of eye-tracking data, the eye tracker was carefully positioned according to the manufacturer's instructions, and participants underwent a calibration process before the study.

To minimize potential biases, the study was conducted in a controlled environment. Each snippet was presented only once in a random condition to avoid any potential learning effects. Participants received clear written instructions and a warm-up snippet to ensure they are familiar with the study procedures, reducing the risk of misunderstanding or confusion.

Furthermore, the selected code snippets have undergone careful adaptations to standardize tasks, remove code documentation, and obfuscate obvious function and variable names. These modifications aimed to minimize potential biases arising from specific coding styles or identifier names and allowed us to focus on the influence of comments on comprehension.

By employing established measures and methodologies, our study ensured accurate representations of program comprehension, visual attention, linearity, and gaze strategies during code and comment interaction. The rigorous experimental design and the control of potential confounding factors enhance the internal validity of our findings.

8.3 EXTERNAL VALIDITY

External validity concerns how well our study's findings can be applied to different populations, settings, and contexts beyond our specific sample and conditions. While our study provides valuable insights into the effect of comments, it is essential to consider certain limitations that may affect the generalizability of the results.

The sample for our study consists of computer science students with some programming experience, which may not fully represent professional software developers or individuals from different academic backgrounds. Therefore, the generalizability of our findings to a broader population of developers should be interpreted with caution.

Additionally, the study focused on relatively simple code snippets with approximately 20 lines of code. While this design choice allows us to investigate the role of comments in understanding basic code components, the findings may not fully capture the effects of comments in more complex codebases or domains. The use of a controlled environment and code snippets also differs from the dynamics of real-world software development, potentially affecting the external validity.

Future research could consider expanding the participant pool to include developers with varying levels of experience and from different programming languages or domains. Furthermore, studying larger and more complex codebases could provide insights into how comments influence program comprehension in real-world scenarios.

9 CONCLUSION

This thesis set out to explore the impact of comments on program comprehension among computer science students, addressing five critical research questions through a combination of behavioral data, eye-tracking analysis, and participants' subjective perceptions. By providing a quantifiable analysis of how comments impact various aspects of program comprehension, this research marks a notable contribution to the field, offering empirical evidence to deepen our understanding the role of comments in software development.

Our findings revealed a complex relation between comments and the various dimensions of program comprehension investigated, offering nuanced insights as outlined below:

RQ$_1$: Program Comprehension: The study revealed that comments have a variable impact on program comprehension. Some code snippets showed improved performance with comments, indicating their potential to enhance understanding, while others displayed a decrease in performance, highlighting the context-dependent effectiveness of comments. Participants' feedback supported these findings, noting that the value of comments lies in their ability to clarify and summarize complex code segments, offer context, and elucidate code intentions. This suggests that the efficacy of comments in programming is highly reliant on their relevance, quality, and the specific context of the code.

RQ$_2$: Visual Attention: The study found that comments notably shift visual attention during code comprehension, with approximately 23% of focus redirected from code to comments. This demonstrates comments' critical role in influencing how programmers engage with and understand code. Participant feedback supported these findings, indicating that comments are particularly valuable in navigating complex or unclear code segments.

RQ$_3$: Linearity of Reading Order: The study found that comments influence the linearity of reading order, encouraging a more linear, top-to-bottom reading pattern locally. Globally, the impact of comments on reading linearity was snippet-dependent, suggesting a complex interplay between comments and reading strategies. These findings indicate that comments serve as navigational aids in code comprehension, particularly in guiding through complex sections, and underscore the importance of effective commenting practices to enhance program understanding.

RQ$_4$: Gaze Strategies: The investigation into gaze strategies revealed a significant tendency for programmers to shift their gaze from code to comments, suggesting a directional strategy that integrates additional context or clarifications found within comments. This dynamic behavior indicates the significant yet variable role of comments in programming, pointing to the need for flexible commenting practices that accommodate diverse programming tasks and individual preferences.

RQ$_5$: Participants' Perceptions: The study on participants' perceptions highlighted that comments can both simplify and complicate program comprehension, with significant variability across tasks. Notably, comments significantly reduced perceived difficulty in certain tasks, underscoring their potential to enhance comprehension significantly. Despite varied impacts on perceived difficulty, comments were universally valued for their contribution to understanding, suggesting that their role extends beyond simplifying tasks to enriching overall comprehension.

These findings contribute to a deeper understanding of the role of comments in program comprehension, suggesting that while comments can be a powerful tool for enhancing program comprehension, their impact is not universally positive nor negative and depends on various factors including the complexity of the code, the quality and relevance of the comments, and the individual programmer's strategy for navigating code and comments.

The implications of these insights extend beyond academic inquiry into potential practical applications for integrating adaptive commenting systems within development environments. Such systems could leverage insights from our study to dynamically adjust the presentation or emphasis of comments based on the programmer's current task, their reading patterns, and possibly even their historical interaction with similar code structures. This personalized approach could optimize program comprehension by ensuring that comments serve as effective guides through the code, enhancing understanding without overwhelming the programmer with unnecessary information.

In conclusion, this thesis not only contributes to the theoretical understanding of the role of comments in program comprehension but also opens avenues for practical applications and future research. By shedding light on how comments influence various aspects of program comprehension, we offer a foundation for developing more effective commenting practices and tools that can adapt to the needs of individual programmers. This work points towards the need for further research into the cognitive processes underlying programming. Understanding the interplay between comments and program comprehension, visual attention, and comprehension strategies could provide a foundation for designing more intuitive and supportive development tools and environments.

Future work in this area could focus on developing algorithms or models that predict the usefulness of comments based on the code context, programmer expertise, and task complexity. Additionally, exploring the role of automated comment generation and its effectiveness compared to human-written comments could offer insights into how best to support programmers in their work. Ultimately, this thesis lays the groundwork for a more nuanced understanding of comments in programming, advocating for their strategic use to improve program comprehension and the software development process at large.

A APPENDIX

```java
public static boolean taskWarmUp(int[] input) {
    boolean understood = false;
    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, 5};
    boolean result = taskWarmUp(input);
    System.out.println(result);
}
```

Figure A.1: Warm-Up Snippet. **Solution:** true

```java
public static int task1CM(int[] input) {
    int left = 0;
    int right = input.length - 1;
    while (left < right) {
        int mid = left + (right - left) / 2;
        if (input[mid] < input[mid + 1]) {
            left = mid + 1;
        } else {
            right = mid;
        }
    }
    return left;
}

public static void main(String[] args) {
    int[] input = {1, 2, 1, 3, 5, 6, 4};
    int result = task1CM(input);
    System.out.println(result);
}
```

(a) Snippet 1 CM

```java
public static int task1CP(int[] input) {
    int left = 0;                      // Initialize left pointer for the binary search
    int right = input.length - 1;      // Initialize right pointer for the binary search
    while (left < right) {
        int mid = left + (right - left) / 2;    // Calculate the index of the middle element
        if (input[mid] < input[mid + 1]) {      // If the middle element is smaller than the element to its right,
            left = mid + 1;                     // Move left pointer to the greater element
        } else {
            right = mid;                        // Move right pointer to the left
        }
    }
    return left;    // Return the index of the element
}

public static void main(String[] args) {
    int[] input = {1, 2, 1, 3, 5, 6, 4};
    int result = task1CP(input);
    System.out.println(result);
}
```

(b) Snippet 1 CP

Figure A.2: Snippet 1. Identifies a peak element of an array. **Solution:** 5

```java
public static List<Integer> task2CM(int[] input, int target) {
    HashSet<Integer> set = new HashSet<>();
    List<Integer> result = new ArrayList<>();
    for (int num : input) {
        if (set.contains(num)) {
            result.add(target - num);
            result.add(num);
            return result;
        }
        set.add(target - num);
    }
    return result;
}

public static void main(String[] args) {
    int[] input = {2, 4, 9, 5, 3, 1, 7};
    int target = 10;
    List<Integer> result = task2CM(input, target);
    System.out.println(result);
}
```

(a) Snippet 2 CM

```java
public static List<Integer> task2CP(int[] input, int target) {
    HashSet<Integer> set = new HashSet<>();      // Store complements encountered
    List<Integer> result = new ArrayList<>();    // Store the result pair
    for (int num : input) {                      // Iterate through the array
        if (set.contains(num)) {                 // If the current number is a complement of a previous number
            result.add(target - num);            // Add the first number of the result pair
            result.add(num);                     // Add the second number of the result pair
            return result;
        }
        set.add(target - num);                   // Add the complement of the current number to the set
    }
    return result;                               // Return an empty list if no result is found
}

public static void main(String[] args) {
    int[] input = {2, 4, 9, 5, 3, 1, 7};
    int target = 10;
    List<Integer> result = task2CP(input, target);
    System.out.println(result);
}
```

(b) Snippet 2 CP

Figure A.3: Snippet 2. Finds two elements that sum to a target. **Solution:** [9, 1]

```java
public static int task3CM(int[] input) {
    int maxLength = 0;
    int sum = 0;
    HashMap<Integer, Integer> sumMap = new HashMap<>();
    for (int i = 0; i < input.length; i++) {
        sum += input[i];
        if (sumMap.containsKey(sum)) {
            maxLength = Math.max(maxLength, i - sumMap.get(sum));
        } else {
            sumMap.put(sum, i);
        }
    }
    return maxLength;
}

public static void main(String[] args) {
    int[] input = {15, -2, 2, -8, 1, 7, 10, 23};
    int result = task3CM(input);
    System.out.println(result);
}
```

(a) Snippet 3 CM

```java
public static int task3CP(int[] input) {
    int maxLength = 0;               // Variable to store the length of the longest subarray
    int sum = 0;                     // Variable to keep track of the cumulative sum of elements
    HashMap<Integer, Integer> sumMap = new HashMap<>();  // Key: cumulative sum, Value: index of first occurrence
    for (int i = 0; i < input.length; i++) {  // Iterate through the input array
        sum += input[i];                      // if the same cumulative sum is encountered again, it means that
                                              // the elements between the two occurrences have a sum of 0.
        if (sumMap.containsKey(sum)) {        // There is a subarray with the desired sum.
            maxLength = Math.max(maxLength, i - sumMap.get(sum));  // Update the maximum length if necessary
        } else {
            sumMap.put(sum, i);               // Otherwise, add the cumulative sum and its index to the map
        }
    }
    return maxLength;   // Return the length of the longest subarray
}

public static void main(String[] args) {
    int[] input = {15, -2, 2, -8, 1, 7, 10, 23};
    int result = task3CP(input);
    System.out.println(result);
}
```

(b) Snippet 3 CP

Figure A.4: Snippet 3. Computes maximum zero-sum subarray length. **Solution:** 5

```java
public static int task4CM(int[] input) {
    HashSet<Integer> numSet = new HashSet<>();
    int maxLength = 0;
    for (int num : input) {
        numSet.add(num);
    }
    for (int num : input) {
        if (!numSet.contains(num - 1)) {
            int currentNum = num;
            int currentLength = 1;
            while (numSet.contains(currentNum + 1)) {
                currentNum++;
                currentLength++;
            }
            maxLength = Math.max(maxLength, currentLength);
        }
    }
    return maxLength;
}

public static void main(String[] args) {
    int[] input = {100, 4, 200, 1, 3, 2, 5};
    int result = task4CM(input);
    System.out.println(result);
}
```

(a) Snippet 4 CM

```java
public static int task4CP(int[] input) {
    HashSet<Integer> numSet = new HashSet<>();   // Set to store unique elements of the array
    int maxLength = 0;                           // Variable to store the length of the  longest consecutive sequence
    for (int num : input) {
        numSet.add(num);                         // Add all elements to the set to remove duplicates and order
    }
    for (int num : input) {
        if (!numSet.contains(num - 1)) {         // Check if the current element is the start of a sequence
            int currentNum = num;                // Initialize the current number
            int currentLength = 1;               // Initialize the current sequence length
            while (numSet.contains(currentNum + 1)) {        // Continue counting the consecutive sequence
                currentNum++;
                currentLength++;
            }
            maxLength = Math.max(maxLength, currentLength);  // Update the maximum length if necessary
        }
    }
    return maxLength;                            // Return the max length
}

public static void main(String[] args) {
    int[] input = {100, 4, 200, 1, 3, 2, 5};
    int result = task4CP(input);
    System.out.println(result);
}
```

(b) Snippet 4 CP

Figure A.5: Snippet 4. Longest consecutive sequence in array. **Solution:** 5

```java
public static int task5CM(String input1, String input2) {
    int m = input1.length();
    int n = input2.length();
    int[][] dp = new int[m + 1][n + 1];
    for (int i = 1; i <= m; i++) {
        for (int j = 1; j <= n; j++) {
            if (input1.charAt(i - 1) == input2.charAt(j - 1)) {
                dp[i][j] = dp[i - 1][j - 1] + 1;
            } else {
                dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]);
            }
        }
    }
    return dp[m][n];
}

public static void main(String[] args) {
    String input1 = "abcde";
    String input2 = "ace";
    int result = task5CM(input1, input2);
    System.out.println(result);
}
```

(a) Snippet 5 CM

```java
public static int task5CP(String input1, String input2) {
    int m = input1.length();                    // Length of the first string
    int n = input2.length();                    // Length of the second string
    int[][] dp = new int[m + 1][n + 1];         // 2D array to keep track of the lengths of common subsequences
    for (int i = 1; i <= m; i++) {              // Loop through each character of the first string
        for (int j = 1; j <= n; j++) {          // Loop through each character of the second string
            if (input1.charAt(i - 1) == input2.charAt(j - 1)) {   // If the characters match
                dp[i][j] = dp[i - 1][j - 1] + 1;                  // add 1 to the length of the common subsequence
            } else {
                dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]);  // Otherwise, take the maximum of the previous
            }                                                     // lengths for the smaller parts of the strings
        }
    }
    return dp[m][n];                           // Return the maximum length
}

public static void main(String[] args) {
    String input1 = "abcde";
    String input2 = "ace";
    int result = task5CP(input1, input2);
    System.out.println(result);
}
```

(b) Snippet 5 CP

Figure A.6: Snippet 5. Longest common subsequence between strings. **Solution:** 3

```java
public static int task6CM(int[] input) {
    int n = input.length;
    int maxLength = 0;
    int[] dp = new int[n];
    for (int i = 0; i < n; i++) {
        dp[i] = 1;
        for (int j = 0; j < i; j++) {
            if (input[i] > input[j]) {
                dp[i] = Math.max(dp[i], dp[j] + 1);
            }
        }
    }
    for (int length : dp) {
        maxLength = Math.max(maxLength, length);
    }
    return maxLength;
}

public static void main(String[] args) {
    int[] input = {10, 9, 2, 5, 3, 7, 101, 18};
    int result = task6CM(input);
    System.out.println(result);
}
```

(a) Snippet 6 CM

```java
public static int task6CP(int[] input) {
    int n = input.length;                // Length of the input array
    int maxLength = 0;                   // Variable to store the max length
    int[] dp = new int[n];               // Array to store lengths of subsequences that end at each index.
    for (int i = 0; i < n; i++) {        // Loop through each element of the input array
        dp[i] = 1;                       // Initialize the length of the subsequence to 1
        for (int j = 0; j < i; j++) {    // Loop through each element before the current element
            if (input[i] > input[j]) {   // If the current element can be included in the increasing subsequence
                dp[i] = Math.max(dp[i], dp[j] + 1);  // Update the length of the subsequence if necessary
            }
        }
    }
    for (int length : dp) {
        maxLength = Math.max(maxLength, length);  // Find the maximum length from the array
    }
    return maxLength;                             // Return the maximum length
}

public static void main(String[] args) {
    int[] input = {10, 9, 2, 5, 3, 7, 101, 18};
    int result = task6CP(input);
    System.out.println(result);
}
```

(b) Snippet 6 CP

Figure A.7: Snippet 6. Longest increasing subsequence length. **Solution:** 4

```java
public static long task7CM(long input1, long input2) {
    long result = 1;
    while (input2 > 0) {
        if (input2 % 2 == 1) {
            result *= input1;
        }
        input1 *= input1;
        input2 /= 2;
    }
    return result;
}

public static void main(String[] args) {
    long input1 = 2;
    long input2 = 10;
    long result = task7CM(input1, input2);
    System.out.println(result);
}
```

(a) Snippet 7 CM

```java
public static long task7CP(long input1, long input2) {
    long result = 1;                 // Initialize the result to 1
    while (input2 > 0) {             // Loop until input2 is 0
        if (input2 % 2 == 1) {       // If input2 is odd (last binary digit is 1)
            result *= input1;        // multiply the result by input1
        }
        input1 *= input1;            // Square input1 to increase the power of the result by 2
        input2 /= 2;                 // Divide input2 by 2 to remove the last binary digit
    }
    return result;                   // Return the final result
}

public static void main(String[] args) {
    long input1 = 2;
    long input2 = 10;
    long result = task7CP(input1, input2);
    System.out.println(result);
}
```

(b) Snippet 7 CP

Figure A.8: Snippet 7. Efficient power calculation. **Solution:** 1024

```java
public static int task8CM(int input) {
    if (input <= 1) {
        return input;
    }
    int prev = 0;
    int current = 1;
    for (int i = 2; i <= input; i++) {
        int next = prev + current;
        prev = current;
        current = next;
    }
    return current;
}

public static void main(String[] args) {
    int input = 7;
    int result = task8CM(input);
    System.out.println(result);
}
```

(a) Snippet 8 CM

```java
public static int task8CP(int input) {
    if (input <= 1) {
        return input;               // Return n if n is less than or equal to 1
    }
    int prev = 0;                   // Initialize the first number in the sequence
    int current = 1;                // Initialize the second number in the sequence
    for (int i = 2; i <= input; i++) {      // Iterate to find the nth number in the sequence
        int next = prev + current;          // Calculate the next number in the sequence
        prev = current;                     // Update the first number with the second number
        current = next;                     // Update the second number with the next number
    }
    return current;                 // Return the nth number in the sequence
}

public static void main(String[] args) {
    int input = 7;
    int result = task8CP(input);
    System.out.println(result);
}
```

(b) Snippet 8 CP

Figure A.9: Snippet 8. Fibonacci number at given position. **Solution:** 13

```java
public static List<Integer> task9CM(int input) {
    boolean[] marks = new boolean[input + 1];
    List<Integer> numbers = new ArrayList<>();
    for (int num = 2; num <= input; num++) {
        if (!marks[num]) {
            numbers.add(num);
            for (int multiple = num * num; multiple <= input; multiple += num) {
                marks[multiple] = true;
            }
        }
    }
    return numbers;
}

public static void main(String[] args) {
    int input = 18;
    List<Integer> result = task9CM(input);
    System.out.println(result);
}
```

(a) Snippet 9 CM

```java
public static List<Integer> task9CP(int input) {
    boolean[] marks = new boolean[input + 1];     // Array to store marks for each number that is not prime
    List<Integer> numbers = new ArrayList<>();    // List to store the numbers
    for (int num = 2; num <= input; num++) {
        if (!marks[num]) {                        // If the current number is not marked as true
            numbers.add(num);                     // Add the number to the list
            for (int multiple = num * num; multiple <= input; multiple += num) {
                marks[multiple] = true;           // Mark all multiples of the number as true
            }
        }
    }
    return numbers;                               // Return the list of numbers
}

public static void main(String[] args) {
    int input = 18;
    List<Integer> result = task9CP(input);
    System.out.println(result);
}
```

(b) Snippet 9 CP

Figure A.10: Snippet 9. Lists primes up to a number (Sieve of Eratosthenes).
Solution: [2, 3, 5, 7, 11, 13, 17]

```java
public static int task10CM(int[] input, int target) {
    int left = 0;
    int right = input.length - 1;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (input[mid] == target) {
            return mid;
        } else if (input[mid] < target) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }
    return -1;
}

public static void main(String[] args) {
    int[] input = {2, 5, 8, 12, 16, 23, 38, 45, 56, 72};
    int target = 45;
    int result = task10CM(input, target);
    System.out.println(result);
}
```

(a) Snippet 10 CM

```java
public static int task10CP(int[] input, int target) {
    int left = 0;                              // Initialize the left pointer
    int right = input.length - 1;              // Initialize the right pointer
    while (left <= right) {
        int mid = left + (right - left) / 2;   // Calculate the mid index of the array (left + right) / 2
        if (input[mid] == target) {
            return mid;                        // If the target element is found, return the target index
        } else if (input[mid] < target) {
            left = mid + 1;                    // If the target is greater, the target is in the right half
        } else {
            right = mid - 1;                   // If the target is smaller, the target is in the left half
        }
    }
    return -1;                                 // If the target element is not found, return -1
}

public static void main(String[] args) {
    int[] input = {2, 5, 8, 12, 16, 23, 38, 45, 56, 72};
    int target = 45;
    int result = task10CP(input, target);
    System.out.println(result);
}
```

(b) Snippet 10 CP

Figure A.11: Snippet 10. Binary search in sorted array. **Solution:** 7

```java
public static int task11CM(String str) {
    int count = 0;
    int n = str.length();
    for (int center = 0; center < 2 * n - 1; center++) {
        int left = center / 2;
        int right = left + center % 2;
        while (left >= 0 && right < n && str.charAt(left) == str.charAt(right)) {
            count++;
            left--;
            right++;
        }
    }
    return count;
}

public static void main(String[] args) {
    String input = "abba";
    int result = task11CM(input);
    System.out.println(result);
}
```

(a) Snippet 11 CM

```java
public static int task11CP(String str) {
    int count = 0;          // Initialize the count of symmetrical substrings to zero
    int n = str.length();   // Get the length of the input string
    for (int center = 0; center < 2 * n - 1; center++) {    // Loop through each possible center (2n - 1)
        int left = center / 2;          // if the center is a character, left and right pointers are the same
        int right = left + center % 2;  // if the center is between two characters, left and right pointers are different
        while (left >= 0 && right < n && str.charAt(left) == str.charAt(right)) {   // Loop as long as the substring is symmetrical
            count++;    // Increment the count for each symmetrical substring found
            left--;     // Move the left pointer to the left to check the next character
            right++;    // Move the right pointer to the right to check the next character
        }
    }
    return count;       // Return the total count of symmetrical substrings
}

public static void main(String[] args) {
    String input = "abba";
    int result = task11CP(input);
    System.out.println(result);
}
```

(b) Snippet 11 CP

Figure A.12: Snippet 11. Counts palindromic substrings. **Solution:** 6

```java
public static boolean task12CM(String input1, String input2) {
    if (input1.length() != input2.length()) {
        return false;
    }
    char[] charArray1 = input1.toCharArray();
    char[] charArray2 = input2.toCharArray();
    Arrays.sort(charArray1);
    Arrays.sort(charArray2);
    return Arrays.equals(charArray1, charArray2);
}

public static void main(String[] args) {
    String input1 = "listen";
    String input2 = "silent";
    boolean result = task12CM(input1, input2);
    System.out.println(result);
}
```

(a) Snippet 12 CM

```java
public static boolean task12CP(String input1, String input2) {
    if (input1.length() != input2.length()) {   // If the lengths are different, return false
        return false;
    }
    char[] charArray1 = input1.toCharArray();  // Convert the first string to a character array
    char[] charArray2 = input2.toCharArray();  // Convert the second string to a character array
    Arrays.sort(charArray1);  // Sort the character arrays
    Arrays.sort(charArray2);
    return Arrays.equals(charArray1, charArray2);  // Check if the sorted arrays are equal
}

public static void main(String[] args) {
    String input1 = "listen";
    String input2 = "silent";
    boolean result = task12CP(input1, input2);
    System.out.println(result);
}
```

(b) Snippet 12 CP

Figure A.13: Snippet 12. Anagram check for two strings. **Solution:** true

Table A.1: Order of snippets shown to participants.

	Snippet Order											
Participant	1	2	3	4	5	6	7	8	9	10	11	12
1	12 CM	2 CP	10 CM	6 CP	3 CP	1 CP	11 CM	4 CP	7 CM	5 CP	8 CM	9 CM
2	5 CP	12 CM	6 CP	1 CP	9 CM	11 CM	3 CP	10 CM	7 CM	4 CP	2 CP	8 CM
3	10 CM	1 CP	4 CP	3 CP	12 CM	8 CM	9 CM	2 CP	6 CP	5 CP	11 CM	7 CM
4	9 CM	11 CM	5 CP	12 CM	7 CM	3 CP	8 CM	2 CP	4 CP	6 CP	1 CP	10 CM
5	10 CM	11 CM	5 CP	1 CP	9 CM	7 CM	4 CP	2 CP	12 CM	8 CM	6 CP	3 CP
6	5 CP	11 CM	10 CM	8 CM	4 CP	2 CP	1 CP	6 CP	3 CP	7 CM	12 CM	9 CM
7	9 CM	6 CP	11 CM	8 CM	3 CP	7 CM	5 CP	12 CM	10 CM	1 CP	4 CP	2 CP
8	1 CP	11 CM	7 CM	8 CM	9 CM	4 CP	10 CM	12 CM	5 CP	2 CP	3 CP	6 CP
9	1 CP	4 CP	5 CP	9 CM	2 CP	12 CM	7 CM	6 CP	11 CM	3 CP	10 CM	8 CM
10	7 CM	9 CM	3 CP	4 CP	5 CP	6 CP	10 CM	1 CP	12 CM	11 CM	2 CP	8 CM
11	3 CM	11 CP	1 CM	12 CP	6 CM	2 CM	9 CP	5 CM	4 CM	10 CP	7 CP	8 CP
12	7 CP	10 CP	9 CP	5 CM	2 CM	6 CM	4 CM	8 CP	11 CP	1 CM	12 CP	3 CM
13	6 CM	10 CP	2 CM	12 CP	8 CP	1 CM	11 CP	3 CM	9 CP	4 CM	5 CM	7 CP
14	5 CM	7 CP	12 CP	9 CP	6 CM	8 CP	2 CM	10 CP	3 CM	11 CP	4 CM	1 CM
15	8 CP	10 CP	2 CM	1 CM	5 CM	3 CM	12 CP	9 CP	11 CP	7 CP	6 CM	4 CM
16	2 CM	6 CM	7 CP	12 CP	9 CP	10 CP	4 CM	3 CM	5 CM	8 CP	1 CM	11 CP
17	12 CP	8 CP	7 CP	4 CM	10 CP	6 CM	3 CM	11 CP	5 CM	2 CM	1 CM	9 CP
18	11 CP	1 CM	8 CP	4 CM	12 CP	7 CP	10 CP	3 CM	6 CM	2 CM	9 CP	5 CM
19	6 CM	7 CP	9 CP	1 CM	4 CM	3 CM	10 CP	12 CP	2 CM	5 CM	11 CP	8 CP
20	3 CM	9 CP	10 CP	6 CM	7 CP	1 CM	2 CM	4 CM	5 CM	8 CP	11 CP	12 CP

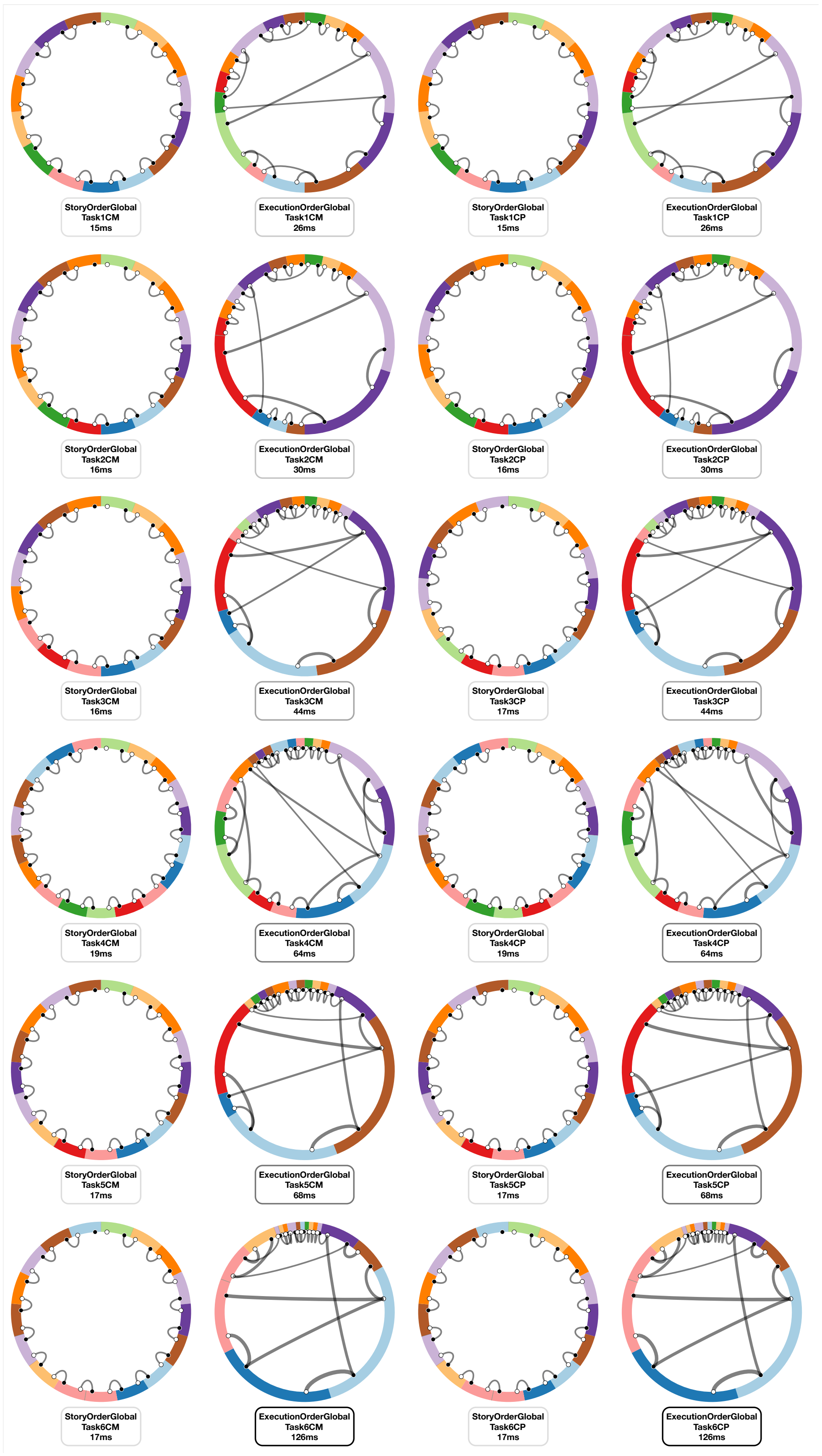


Figure A.14: RTGCT Visualization of Line Order for Snippets 1-6

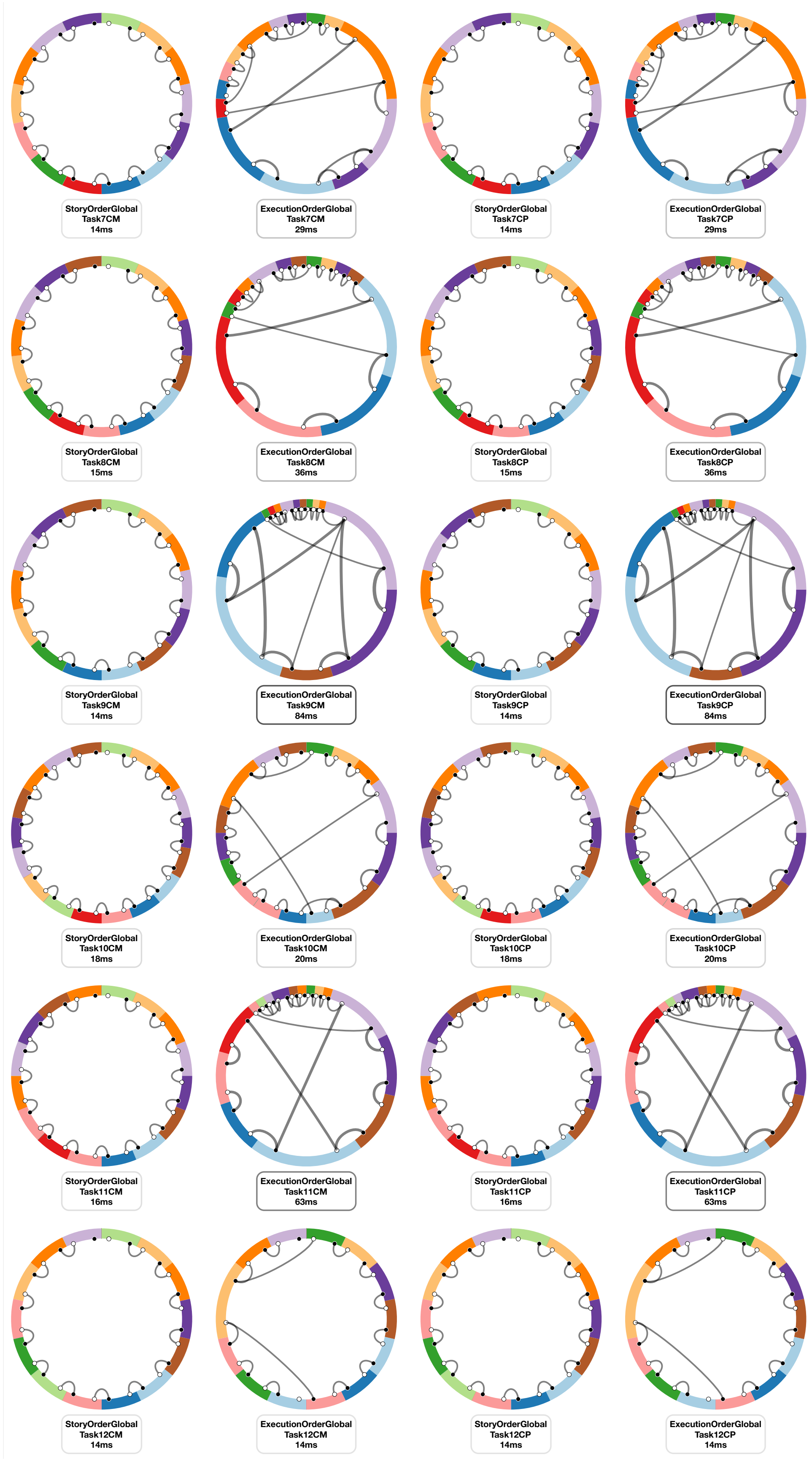


Figure A.15: RTGCT Visualization of Line Order for Snippets 7-12

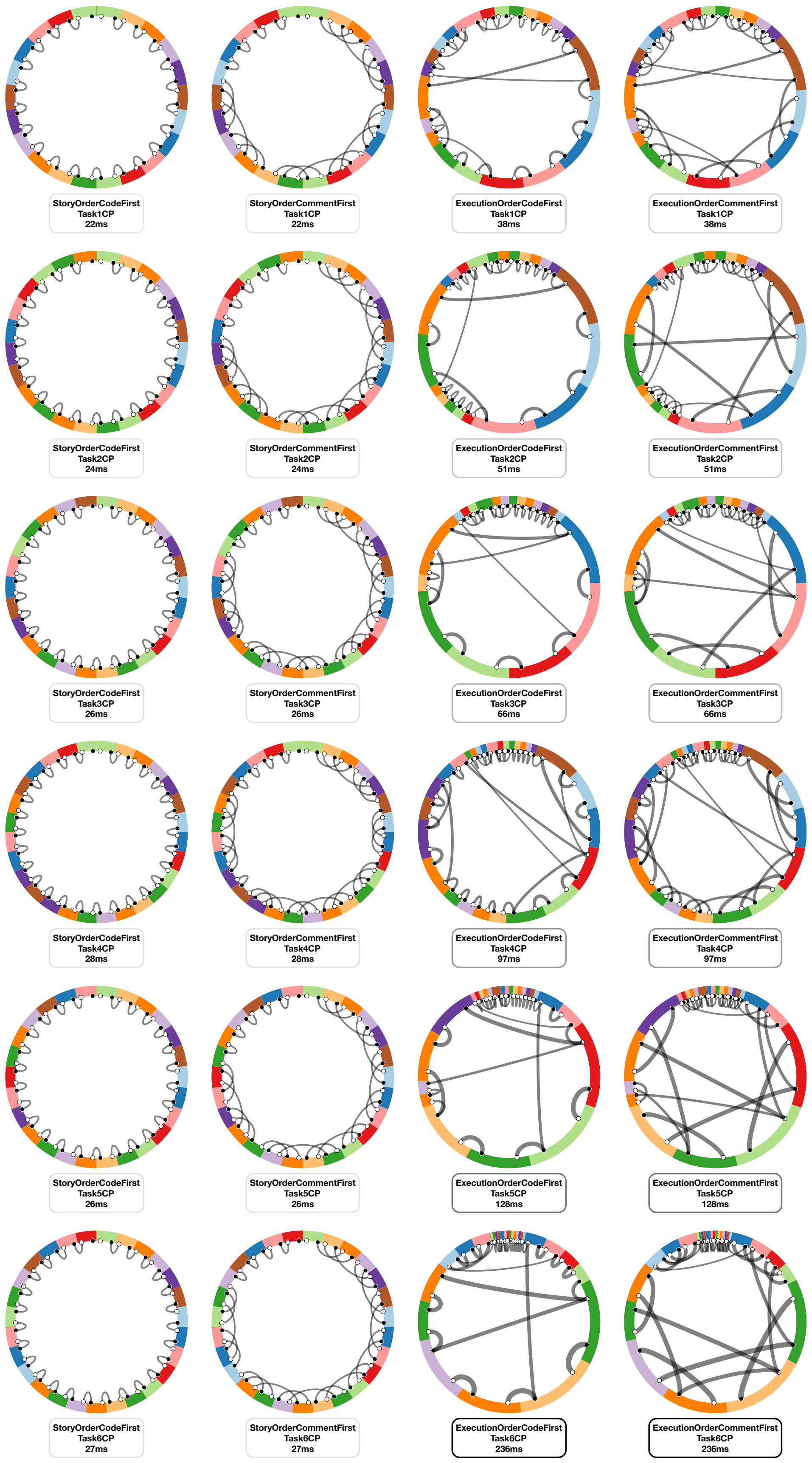


Figure A.16: RTGCT Visualization of AOI Order for Snippets 1-6

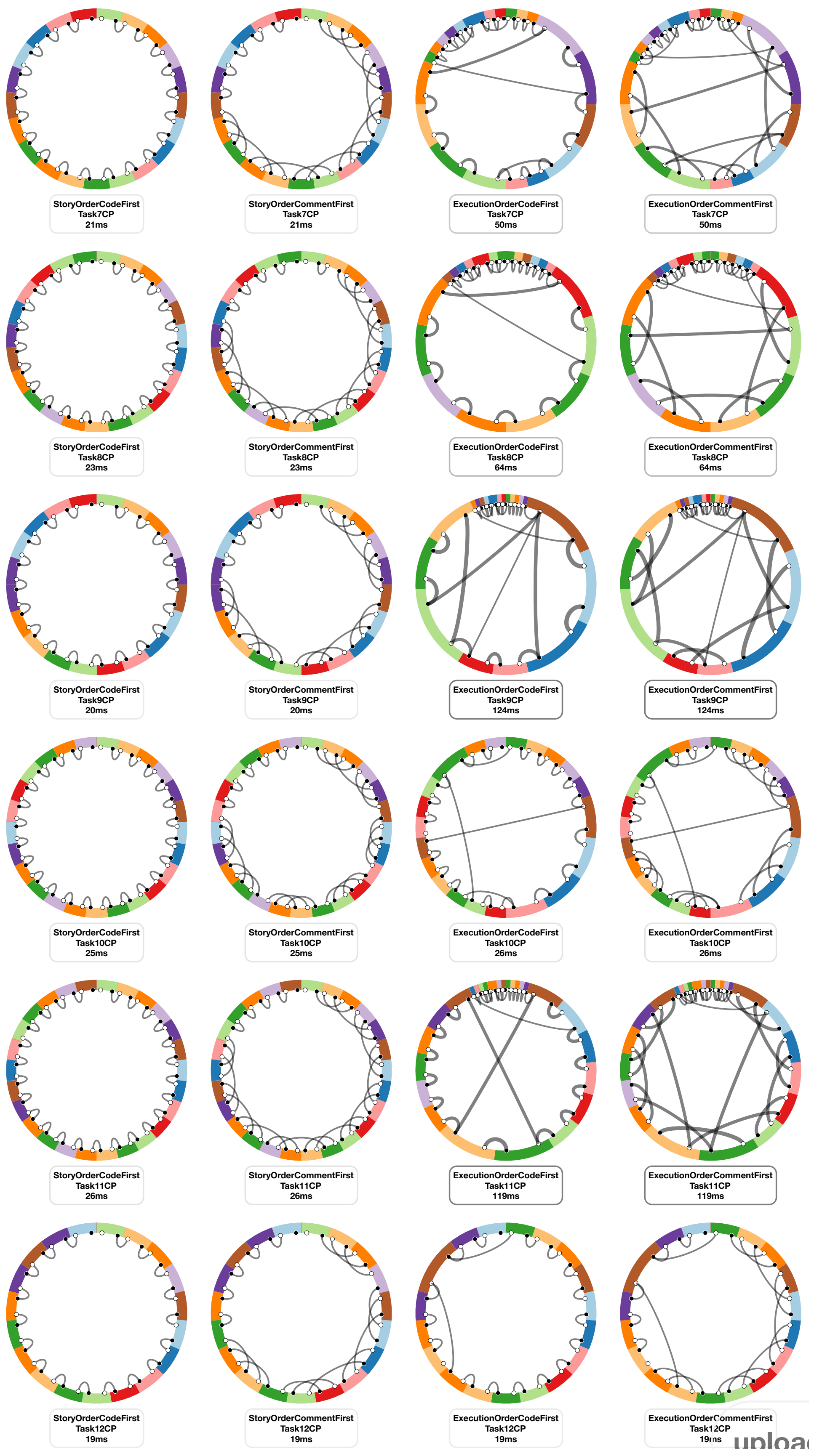

StoryOrderCodeFirst Task7CP 21ms
StoryOrderCommentFirst Task7CP 21ms
ExecutionOrderCodeFirst Task7CP 50ms
ExecutionOrderCommentFirst Task7CP 50ms
StoryOrderCodeFirst Task8CP 23ms
StoryOrderCommentFirst Task8CP 23ms
ExecutionOrderCodeFirst Task8CP 64ms
ExecutionOrderCommentFirst Task8CP 64ms
StoryOrderCodeFirst Task9CP 20ms
StoryOrderCommentFirst Task9CP 20ms
ExecutionOrderCodeFirst Task9CP 124ms
ExecutionOrderCommentFirst Task9CP 124ms
StoryOrderCodeFirst Task10CP 25ms
StoryOrderCommentFirst Task10CP 25ms
ExecutionOrderCodeFirst Task10CP 26ms
ExecutionOrderCommentFirst Task10CP 26ms
StoryOrderCodeFirst Task11CP 26ms
StoryOrderCommentFirst Task11CP 26ms
ExecutionOrderCodeFirst Task11CP 119ms
ExecutionOrderCommentFirst Task11CP 119ms
StoryOrderCodeFirst Task12CP 19ms
StoryOrderCommentFirst Task12CP 19ms
ExecutionOrderCodeFirst Task12CP 19ms
ExecutionOrderCommentFirst Task12CP 19ms

Figure A.17: RTGCT Visualization of AOI Order for Snippets 7-12

BIBLIOGRAPHY

[1] Marjan Adeli, Nicholas Nelson, Souti Chattopadhyay, Hayden Coffey, Austin Henley, and Anita Sarma. “Supporting Code Comprehension via Annotations: Right Information at the Right Time and Place.” In: *2020 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, Aug. 2020. DOI: 10.1109/vl/hcc50065.2020.9127264.

[2] Nasir Ali, Zohreh Sharafi, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. “An Empirical Study on the Importance of Source Code Entities for Requirements Traceability.” In: *Empirical Software Engineering* 20.2 (July 2014), pp. 442–478. DOI: 10.1007/s10664-014-9315-y.

[3] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo. “Recovering Traceability Links between Code and Documentation.” In: *IEEE Transactions on Software Engineering* 28.10 (Oct. 2002), pp. 970–983. DOI: 10.1109/tse.2002.1041053.

[4] Jay Ayres, Jason Flannick, Johannes Gehrke, and Tomi Yiu. “Sequential PAttern Mining Using a Bitmap Representation.” In: *Proceedings of the 8th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, July 2002. DOI: 10.1145/775047.775109.

[5] V. R. Basili, F. Shull, and F. Lanubile. “Building Knowledge through Families of Experiments.” In: *IEEE Transactions on Software Engineering* 25.4 (1999), pp. 456–473. DOI: 10.1109/32.799939.

[6] Jackson Beatty. “Task-Evoked Pupillary Responses, Processing Load, and the Structure of Processing Resources.” In: *Psychological Bulletin* 91.2 (1982), pp. 276–292. DOI: 10.1037/0033-2909.91.2.276.

[7] Roman Bednarik. *Methods to Analyze Visual Attention Strategies: Applications in the Studies of Programming*. University of Joensuu, 2007.

[8] Roman Bednarik, Shahram Eivazi, and Michal Hradis. “Gaze and Conversational Engagement in Multiparty Video Conversation.” In: *Proceedings of the 4th Workshop on Eye Gaze in Intelligent Human Machine Interaction*. ACM, Oct. 2012. DOI: 10.1145/2401836.2401846.

[9] Tanya Beelders. “Eye-Tracking Analysis of Source Code Reading on a Line-By-Line Basis.” In: *Proceedings of the 10th International Workshop on Eye Movements in Programming*. ACM, May 2022. DOI: 10.1145/3524488.3527364.

[10] Yoav Benjamini and Yosef Hochberg. “Controlling The False Discovery Rate - A Practical And Powerful Approach To Multiple Testing.” In: *J. Royal Statist. Soc., Series B* 57 (Nov. 1995), pp. 289–300. DOI: 10.2307/2346101.

[11] Jennifer Romano Bergstrom and Andrew Jonathan Schall. *Eye Tracking in User Experience Design*. 2014, p. 374. ISBN: 9780124081383.

[12] Mario Berón, Pedro Rangel Henriques, Maria Pereira, and Roberto Uzal. "Program Inspection to Interconnect Behavioral and Operational View for Program Comprehension." In: (Jan. 2007).

[13] Roberta M. M. Bezerra, Fabio Q. B. da Silva, Anderson M. Santana, Cleyton V. C. Magalhaes, and Ronnie E. S. Santos. "Replication of Empirical Studies in Software Engineering: An Update of a Systematic Mapping Study." In: *2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, Oct. 2015. DOI: `10.1109/esem.2015.7321213`.

[14] Dave Binkley, Marcia Davis, Dawn Lawrie, Jonathan I. Maletic, Christopher Morrell, and Bonita Sharif. "The Impact of Identifier Style on Effort and Comprehension." In: *Empirical Software Engineering* 18.2 (May 2012), pp. 219–276. DOI: `10.1007/s10664-012-9201-4`.

[15] Tanja Blascheck, Markus Schweizer, Fabian Beck, and Thomas Ertl. "Visual Comparison of Eye Movement Patterns." In: *Computer Graphics Forum* 36.3 (June 2017), pp. 87–97. DOI: `10.1111/cgf.13170`.

[16] Scott Blinman and Andy Cockburn. "Program Comprehension: Investigating the Effects of Naming Style and Documentation." In: *Proceedings of the 6th Australasian Conference on User Interface - Volume 40*. AUIC '05. AUS: Australian Computer Society, Inc., 2005, pp. 73–78. ISBN: 1920682228.

[17] Agnieszka Bojko. "Eye Tracking in User Experience Testing: How to Make the Most of It." In: *Proceedings of the 14th Annual Conference of the Usability Professionals' Association (UPA). Montréal, Canada* (Jan. 2005).

[18] Jurgen Börstler and Barbara Paech. "The Role of Method Chains and Comments in Software Readability and Comprehension—an Experiment." In: *IEEE Transactions on Software Engineering* 42.9 (Sept. 2016). not added, pp. 886–898. DOI: `10.1109/tse.2016.2527791`.

[19] Ruven Brooks. "Using a Behavioral Theory of Program Comprehension in Software Engineering." In: *Proceedings of the 3rd International Conference on Software Engineering*. ICSE '78. Atlanta, Georgia, USA: IEEE Press, 1978, pp. 196–201.

[20] Ruven Brooks. "Towards a Theory of the Comprehension of Computer Programs." In: *International Journal of Man-Machine Studies* 18.6 (June 1983), pp. 543–554. DOI: `10.1016/s0020-7373(83)80031-5`.

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

[22] 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*. IEEE, May 2015. DOI: `10.1109/icpc.2015.36`.

[23] 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*. ACM, Nov. 2011. DOI: `10.1145/2094131.2094133`.

[24] Anna Corazza, Valerio Maggio, and Giuseppe Scanniello. "On the Coherence between Comments and Implementations in Source Code." In: *2015 41st Euromicro Conference on Software Engineering and Advanced Applications*. IEEE, Aug. 2015. DOI: 10.1109/seaa.2015.20.

[25] T. A. Corbi. "Program Understanding: Challenge for the 1990s." In: *IBM Systems Journal* 28.2 (1989), pp. 294–306. DOI: 10.1147/sj.282.0294.

[26] Filipe Cristino, Sebastiaan Mathôt, Jan Theeuwes, and Iain D. Gilchrist. "ScanMatch: A Novel Method for Comparing Fixation Sequences." In: *Behavior Research Methods* 42.3 (Aug. 2010), pp. 692–700. DOI: 10.3758/brm.42.3.692.

[27] Martha E. Crosby and Jan Stelovsky. "How Do We Read Algorithms? A Case Study." In: *Computer* 23.1 (Jan. 1990), pp. 24–35. ISSN: 0018-9162.

[28] Martha Crosby, Jean Scholtz, and Susan Wiedenbeck. "The Roles Beacons Play in Comprehension for Novice and Expert Programmers." In: (July 2002).

[29] Matjaž Divjak and Horst Bischof. "Real-Time Video-Based Eye Blink Analysis for Detection of Low Blink-Rate during Computer Use." In: *First Int. Workshop On Tracking Humans For the Evaluation of their Motion In Image Sequences (THEMIS 2008)* (Jan. 2008).

[30] Andrew T. Duchowski. "A Breadth-First Survey of Eye-Tracking Applications." In: *Behavior Research Methods, Instruments, $&$ Computers* 34.4 (Nov. 2002), pp. 455–470. DOI: 10.3758/bf03195475.

[31] Andrew Duchowski. *Eye Tracking Methodology: Theory and Practice*. Springer London, Jan. 2007. ISBN: 978-1-84628-608-7. DOI: 10.1007/978-1-84628-609-4.

[32] A. Dunsmore, M. Roper, and M. Wood. "The Role of Comprehension in Software Inspection." In: *Journal of Systems and Software* 52.2-3 (June 2000), pp. 121–129. DOI: 10.1016/s0164-1212(99)00138-7.

[33] H. E. Dunsmore. "The Effect of Comments, Mnemonic Names, and Modularity: Some University Experiment Results." In: *Empirical Foundations of Information and Software Science* (1985), pp. 189–196.

[34] James L. Elshoff and Michael Marcotty. "Improving Computer Program Readability to Aid Modification." In: *Communications of the ACM* 25.8 (Aug. 1982), pp. 512–521. DOI: 10.1145/358589.358596.

[35] Letha H. Etzkorn, Carl G. Davis, and Lisa L. Bowen. "The Language of Comments in Computer Software: A Sublanguage of English." In: *Journal of Pragmatics* 33.11 (Nov. 2001), pp. 1731–1756. DOI: 10.1016/s0378-2166(00)00068-0.

[36] Sarah Fakhoury, Yuzhan Ma, Venera Arnaoudova, and Olusola Adesope. "The Effect of Poor Source Code Lexicon and Readability on Developers' Cognitive Load." In: *Proceedings of the 26th Conference on Program Comprehension*. ACM, May 2018. DOI: 10.1145/3196321.3196347.

[37] Richard K. Fjeldstad. "Application Program Maintenance Study." In: *Report to Our Respondents, Proceedings GUIDE* 48 (1983).

[38] Beat Fluri, Michael Wursch, and Harald C. Gall. “Do Code and Comments Co-Evolve? On the Relation between Source Code and Comment Changes.” In: *14th Working Conference on Reverse Engineering (WCRE 2007)*. IEEE, Oct. 2007. DOI: 10.1109/wcre.2007.21.

[39] Thomas Fritz, Andrew Begel, Sebastian C. Müller, Serap Yigit-Elliott, and Manuela Züger. “Using Psycho-Physiological Measures to Assess Task Difficulty in Software Development.” In: *Proceedings of the 36th International Conference on Software Engineering*. ACM, May 2014. DOI: 10.1145/2568225.2568266.

[40] Joseph H. Goldberg and Xerxes P. Kotval. “Computer Interface Evaluation Using Eye Movements: Methods and Constructs.” In: *International Journal of Industrial Ergonomics* 24.6 (Oct. 1999), pp. 631–645. DOI: 10.1016/s0169-8141(98)00068-7.

[41] Yann-Gaël Guéhéneuc. “TAUPE: Towards Understanding Program Comprehension.” In: *Proceedings of the 2006 Conference of the Center for Advanced Studies on Collaborative Research - CASCON '06*. ACM Press, 2006. DOI: 10.1145/1188966.1188968.

[42] S. Haiduc and A. Marcus. “On the Use of Domain Terms in Source Code.” In: *2008 16th IEEE International Conference on Program Comprehension*. IEEE, June 2008. DOI: 10.1109/icpc.2008.29.

[43] Prateek Hejmady and N. Hari Narayanan. “Visual Attention Patterns during Program Debugging with an IDE.” In: *Proceedings of the Symposium on Eye Tracking Research and Applications*. ACM, Mar. 2012. DOI: 10.1145/2168556.2168592.

[44] Kenneth Holmqvist, Marcus Nyström, and Fiona Mulvey. “Eye Tracker Data Quality.” In: *Proceedings of the Symposium on Eye Tracking Research and Applications*. ACM, Mar. 2012. DOI: 10.1145/2168556.2168563.

[45] Robert J. K. Jacob and Keith S. Karn. “Eye Tracking in Human-Computer Interaction and Usability Research.” In: *The Mind's Eye*. Elsevier, 2003, pp. 573–605. DOI: 10.1016/b978-044451020-4/50031-1.

[46] Sebastien Jeanmart, Yann-Gaël Guéhéneuc, Houari Sahraoui, and Naji Habra. “Impact of the Visitor Pattern on Program Comprehension and Maintenance.” In: Oct. 2009, pp. 69–78. DOI: 10.1145/1671248.1671255.

[47] Wenbin Ji, Thorsten Berger, Michal Antkiewicz, and Krzysztof Czarnecki. “Maintaining Feature Traceability with Embedded Annotations.” In: *Proceedings of the 19th International Conference on Software Product Line*. ACM, July 2015. DOI: 10.1145/2791060.2791107.

[48] Zhen Ming Jiang and Ahmed E. Hassan. “Examining the Evolution of Code Comments in PostgreSQL.” In: *Proceedings of the 2006 International Workshop on Mining Software Repositories*. ACM, May 2006. DOI: 10.1145/1137983.1138030.

[49] Natalia Juristo and Sira Vegas. “Using Differences among Replications of Software Engineering Experiments to Gain Knowledge.” In: *2009 3rd International Symposium on Empirical Software Engineering and Measurement*. IEEE, Oct. 2009. DOI: 10.1109/esem.2009.5314236.

[50] Marcel A. Just and Patricia A. Carpenter. “A Theory of Reading: From Eye Fixations to Comprehension.” In: *Psychological Review* 87.4 (1980), pp. 329–354. DOI: 10.1037/0033-295x.87.4.329.

[51] Brian W. Kernighan. *The Elements of Programming Style*. McGraw-Hill, 1974, p. 147. ISBN: 0070341990.

[52] D. E. Knuth. “Literate Programming.” In: *The Computer Journal* 27.2 (Feb. 1984), pp. 97–111. DOI: 10.1093/comjnl/27.2.97.

[53] Rainer Koschke, Andrian Marcus, and Gerald C. Gannod. “Guest Editor’s Introduction to the Special Section on the 2009 International Conference on Program Comprehension (ICPC 2009).” In: *Software Quality Journal* 19.1 (Nov. 2010), pp. 3–4. DOI: 10.1007/s11219-010-9119-2.

[54] Jacob Krüger, Wanzi Gu, Hui Shen, Mukelabai Mukelabai, Regina Hebig, and Thorsten Berger. “Towards a Better Understanding of Software Features and Their Characteristics.” In: *Proceedings of the 12th International Workshop on Variability Modelling of Software-Intensive Systems*. ACM, Feb. 2018. DOI: 10.1145/3168365.3168371.

[55] Seolhwa Lee, Danial Hooshyar, Hyesung Ji, Kichun Nam, and Heuiseok Lim. “Mining Biometric Data to Predict Programmer Expertise and Task Difficulty.” In: *Cluster Computing* 21.1 (Jan. 2017), pp. 1097–1107. DOI: 10.1007/s10586-017-0746-2.

[56] M. M. Lehman and F. N. Parr. “Program Evolution and Its Impact on Software Engineering.” In: *Proceedings of the 2nd International Conference on Software Engineering*. ICSE ’76. Washington, DC, USA: IEEE Computer Society Press, 1976, pp. 350–357.

[57] Stanley Letovsky. “Cognitive Processes in Program Comprehension.” In: *Journal of Systems and Software* 7.4 (Dec. 1987), pp. 325–339. DOI: 10.1016/0164-1212(87)90032-x.

[58] Vladimir I. Levenshtein et al. “Binary Codes Capable of Correcting Deletions, Insertions, and Reversals.” In: *Soviet physics doklady*. Vol. 10. 8. Soviet Union. 1966, pp. 707–710.

[59] David C. Littman, Jeannine Pinto, Stanley Letovsky, and Elliot Soloway. “Mental Models and Software Maintenance.” In: *Journal of Systems and Software* 7.4 (Dec. 1987), pp. 341–355. DOI: 10.1016/0164-1212(87)90033-1.

[60] José Luís and Figueiredo de Freitas. “Comment Analysis for Program Comprehension.” PhD thesis. 2011.

[61] Moira Maguire and Brid Delahunt. “Doing a thematic analysis: A practical, step-by-step guide for learning and teaching scholars.” In: *All Ireland Journal of Higher Education* 9.3 (2017).

[62] A. Von Mayrhauser and A. M. Vans. “Program Comprehension during Software Maintenance and Evolution.” In: *Computer* 28.8 (1995), pp. 44–55. DOI: 10.1109/2.402076.

[63] A. von Mayrhauser and A. M. Vans. “Comprehension Processes during Large Scale Maintenance.” In: *Proceedings of the 16th International Conference on Software Engineering*. ICSE ’94. Washington, DC, USA: IEEE Computer Society Press, 1994, pp. 39–48. ISBN: 081865855X.

[64] Paul W. McBurney and Collin McMillan. "An Empirical Study of the Textual Similarity between Source Code and Source Code Summaries." In: *Empirical Software Engineering* 21.1 (Nov. 2014), pp. 17–42. DOI: 10.1007/s10664-014-9344-6.

[65] Paul W. McBurney and Collin McMillan. "Automatic Documentation Generation Via Source Code Summarization of Method Context." In: *Proceedings of the 22nd International Conference on Program Comprehension*. ACM, June 2014. DOI: 10.1145/2597008.2597149.

[66] Seppo Nevalainen and Jorma Sajaniemi. "Comparison of Three Eye Tracking Devices in Psychology of Programming Research." In: *Annual Workshop of the Psychology of Programming Interest Group*. 2004.

[67] Sebastian Nielebock, Dariusz Krolikowski, Jacob Krüger, Thomas Leich, and Frank Ortmeier. "Commenting Source Code: Is It Worth It for Small Programming Tasks?" In: *Empirical Software Engineering* 24.3 (Nov. 2018), pp. 1418–1457. DOI: 10.1007/s10664-018-9664-z.

[68] A. F. Norcio. "Indentation, Documentation and Programmer Comprehension." In: *Proceedings of the 1982 conference on Human factors in computing systems - CHI '82*. ACM Press, 1982. DOI: 10.1145/800049.801766.

[69] E. Nurvitadhi, Wing Leung, and C. Cook. "Do Class Comments Aid Java Program Understanding?" In: *33rd Annual Frontiers in Education, 2003. FIE 2003.* Vol. 1. IEEE, Dec. 2003, T3C–13. ISBN: 0-7803-7961-6. DOI: 10.1109/fie.2003.1263332.

[70] Marcus Nyström and Kenneth Holmqvist. "An Adaptive Algorithm for Fixation, Saccade, and Glissade Detection in Eyetracking Data." In: *Behavior Research Methods* 42.1 (Feb. 2010), pp. 188–204. DOI: 10.3758/brm.42.1.188.

[71] Pavel A. Orlov. "Ambient and Focal Attention during Source-Code Comprehension." In: *FACHBEREICH MATHEMATIK UND INFORMATIK SERIE B INFORMATIK* (2017), p. 12.

[72] Yoann Padioleau, Lin Tan, and Yuanyuan Zhou. "Listening to Programmers Taxonomies and Characteristics of Comments in Operating System Code." In: *2009 IEEE 31st International Conference on Software Engineering*. IEEE, 2009. DOI: 10.1109/icse.2009.5070533.

[73] P. Parkin. "An Exploratory Study of Code and Document Interactions during Task-Directed Program Comprehension." In: *2004 Australian Software Engineering Conference. Proceedings.* IEEE, 2004. DOI: 10.1109/aswec.2004.1290475.

[74] Norman Peitek. "A Neuro-Cognitive Perspective of Program Comprehension." In: *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. ACM, May 2018. DOI: 10.1145/3183440.3183442.

[75] Norman Peitek, Janet Siegmund, and Sven Apel. "What Drives the Reading Order of Programmers?" In: *Proceedings of the 28th International Conference on Program Comprehension*. ACM, July 2020. DOI: 10.1145/3387904.3389279.

[76] Norman Peitek, Janet Siegmund, Sven Apel, Christian Kastner, Chris Parnin, Anja Bethmann, Thomas Leich, Gunter Saake, and Andre Brechmann. "A Look into Programmers' Heads." In: *IEEE Transactions on Software Engineering* 46.4 (Apr. 2020), pp. 442–462. DOI: 10.1109/tse.2018.2863303.

[77] Nancy Pennington. "Comprehension Strategies in Programming." In: *Empirical Studies of Programmers: Second Workshop*. USA: Ablex Publishing Corp., 1987, pp. 100–113. ISBN: 0893914614.

[78] Razvan Petrusel and Jan Mendling. "Eye-Tracking the Factors of Process Model Comprehension Tasks." In: *Notes on Numerical Fluid Mechanics and Multidisciplinary Design*. Springer International Publishing, 2013, pp. 224–239. DOI: 10.1007/978-3-642-38709-8_15.

[79] A. Poole and Linden Ball. "Eye Tracking in Human-Computer Interaction and Usability Research: Current Status and Future Prospects." In: Jan. 2006, pp. 211–219.

[80] C. M. Privitera and L. W. Stark. "Algorithms for Defining Visual Regions-Of-Interest: Comparison with Eye Fixations." In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 22.9 (2000), pp. 970–982. DOI: 10.1109/34.877520.

[81] Inderjot Kaur Ratol and Martin P. Robillard. "Detecting Fragile Comments." In: *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, Oct. 2017. DOI: 10.1109/ase.2017.8115624.

[82] Keith Rayner. "Eye Movements in Reading and Information Processing." In: *Psychological Bulletin* 85.3 (1978), pp. 618–660. DOI: 10.1037/0033-2909.85.3.618.

[83] Keith Rayner. "Eye Movements in Reading and Information Processing: 20 Years of Research." In: *Psychological Bulletin* 124.3 (1998), pp. 372–422. DOI: 10.1037/0033-2909.124.3.372.

[84] Sigrid Rouam. "False Discovery Rate (FDR)." In: *Encyclopedia of Systems Biology*. Ed. by Werner Dubitzky, Olaf Wolkenhauer, Kwang-Hyun Cho, and Hiroki Yokota. New York, NY: Springer New York, 2013, pp. 731–732. ISBN: 978-1-4419-9863-7. DOI: 10.1007/978-1-4419-9863-7_223.

[85] Spencer Rugaber. "Program Comprehension." In: (1995).

[86] Spencer Rugaber. "The Use of Domain Knowledge in Program Understanding." In: *Annals of Software Engineering* 9.1/4 (2000), pp. 143–192. DOI: 10.1023/a:1018976708691.

[87] Felice Salviulo and Giuseppe Scanniello. "Dealing with Identifiers and Comments in Source Code Comprehension and Maintenance." In: *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*. ACM, May 2014. DOI: 10.1145/2601248.2601251.

[88] Dario D. Salvucci and Joseph H. Goldberg. "Identifying Fixations and Saccades in Eye-Tracking Protocols." In: *Proceedings of the Symposium on Eye Tracking Research and Applications*. ACM Press, 2000. DOI: 10.1145/355017.355028.

[89] Carolyn B. Seaman. "Software Maintenance: Concepts and Practice Authored by Penny Grubb and Armstrong A. Takang World Scientific, New Jersey." In: *Journal of Software Maintenance and Evolution: Research and Practice* 20.6 (Nov. 2008), pp. 463–466. DOI: 10.1002/smr.365.

[90] Marcus Seiler and Barbara Paech. "Using Tags to Support Feature Management across Issue Tracking Systems and Version Control Systems." In: *Requirements Engineering: Foundation for Software Quality*. Springer International Publishing, 2017, pp. 174–180. DOI: 10.1007/978-3-319-54045-0_13.

[91] Teresa M. Shaft and Iris Vessey. "The Relevance of Application Domain Knowledge: Characterizing the Computer Program Comprehension Process." In: *Journal of Management Information Systems* 15.1 (June 1998), pp. 51–78. DOI: 10.1080/07421222.1998.11518196.

[92] Zohreh Sharafi, Alessandro Marchetto, Angelo Susi, Giuliano Antoniol, and Yann-Gaël Guéhéneuc. "An Empirical Study on the Efficiency of Graphical vs. Textual Representations in Requirements Comprehension." In: *2013 21st International Conference on Program Comprehension (ICPC)*. IEEE, May 2013. DOI: 10.1109/icpc.2013.6613831.

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

[94] Zohreh Sharafi, Zéphyrin Soh, and Yann-Gaël Guéhéneuc. "A Systematic Literature Review on the Usage of Eye-Tracking in Software Engineering." In: *Information and Software Technology* 67 (Nov. 2015), pp. 79–107. DOI: 10.1016/j.infsof.2015.06.008.

[95] Zohreh Sharafi, Zephyrin 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)*. IEEE, June 2012. DOI: 10.1109/icpc.2012.6240505.

[96] Bonita Sharif, Michael Falcone, and Jonathan I. Maletic. "An Eye-Tracking Study on the Role of Scan Time in Finding Source Code Defects." In: *Proceedings of the Symposium on Eye Tracking Research and Applications*. ACM, Mar. 2012. DOI: 10.1145/2168556.2168642.

[97] Bonita Sharif, Grace Jetty, Jairo Aponte, and Esteban Parra. "An Empirical Study Assessing the Effect of Seeit 3d on Comprehension." In: *2013 First IEEE Working Conference on Software Visualization (VISSOFT)*. IEEE, Sept. 2013. DOI: 10.1109/vissoft.2013.6650519.

[98] Bonita Sharif and Jonathan I. Maletic. "An Eye Tracking Study on camelCase and under_score Identifier Styles." In: *2010 IEEE 18th International Conference on Program Comprehension*. IEEE, June 2010. DOI: 10.1109/icpc.2010.41.

[99] S. B. Sheppard and B. Curtis. "Predicting Programmers' Ability to Modify Software." In: (1978).

[100] Yusuke Shinyama, Yoshitaka Arahori, and Katsuhiko Gondow. "Analyzing Code Comments to Boost Program Comprehension." In: *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, Dec. 2018. DOI: 10.1109/apsec.2018.00047.

[101] Ben Shneiderman and Richard Mayer. "Syntactic/Semantic Interactions in Programmer Behavior: A Model and Experimental Results." In: *International Journal of Computer $&$ Information Sciences* 8.3 (June 1979), pp. 219–238. DOI: 10.1007/bf00977789.

[102] Janet Siegmund. "Program Comprehension: Past, Present, and Future." In: *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, Mar. 2016. DOI: 10.1109/saner.2016.35.

[103] Benoît De Smet, Lorent Lempereur, Zohreh Sharafi, Yann-Gaël Guéhéneuc, Giuliano Antoniol, and Naji Habra. "Taupe : Visualizing and Analyzing Eye-Tracking Data." In: *Science of Computer Programming* 79 (Jan. 2014), pp. 260–278. DOI: 10.1016/j.scico.2012.01.004.

[104] Zéphyrin Soh, Foutse Khomh, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. "Noise in Mylyn Interaction Traces and Its Impact on Developers and Recommendation Systems." In: *Empirical Software Engineering* 23.2 (June 2017), pp. 645–692. DOI: 10.1007/s10664-017-9529-x.

[105] Zephyrin Soh, Foutse Khomh, Yann-Gaël Guéhéneuc, Giuliano Antoniol, and Bram Adams. "On the Effect of Program Exploration on Maintenance Tasks." In: *2013 20th Working Conference on Reverse Engineering (WCRE)*. IEEE, Oct. 2013. DOI: 10.1109/wcre.2013.6671314.

[106] Zephyrin Soh, Zohreh Sharafi, Bertrand Van den Plas, Gerardo Cepeda Porras, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. "Professional Status and Expertise for UML Class Diagram Comprehension: An Empirical Study." In: *2012 20th IEEE International Conference on Program Comprehension (ICPC)*. IEEE, June 2012. DOI: 10.1109/icpc.2012.6240484.

[107] Elliot Soloway, Beth Adelson, and Kate Ehrlich. "Knowledge and Processes in the Comprehension of Computer Programs." In: *The nature of expertise* (1988), pp. 129–152.

[108] Elliot Soloway and Kate Ehrlich. "Empirical Studies of Programming Knowledge." In: *IEEE Transactions on Software Engineering* SE-10.5 (Sept. 1984), pp. 595–609. DOI: 10.1109/tse.1984.5010283.

[109] Peter Sommerlad, Guido Zgraggen, Thomas Corbat, and Lukas Felber. "Retaining Comments When Refactoring Code." In: *Companion to the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*. ACM, Oct. 2008. DOI: 10.1145/1449814.1449817.

[110] Giriprasad Sridhara. "Automatically Detecting the Up-To-Date Status of ToDo Comments in Java Programs." In: *Proceedings of the 9th India Software Engineering Conference*. ACM, Feb. 2016. DOI: 10.1145/2856636.2856638.

[111] Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori Pollock, and K. Vijay-Shanker. "Towards Automatically Generating Summary Comments for Java Methods." In: *Proceedings of the IEEE/ACM international conference on Automated software engineering*. ACM, Sept. 2010. DOI: 10.1145/1858996.1859006.

[112] Thomas A. Standish. "An Essay on Software Reuse." In: *IEEE Transactions on Software Engineering* SE-10.5 (Sept. 1984), pp. 494–497. DOI: 10.1109/tse.1984.5010272.

[113] M.-A. D. Storey, F. D. Fracchia, and H. A. Müller. "Cognitive Design Elements to Support the Construction of a Mental Model during Software Visualization." In: *Proceedings 5th International Workshop on Program Comprehension. IWPC'97*. IEEE Comput. Soc. Press, 1999. DOI: `10.1109/wpc.1997.601257`.

[114] Veronica Sundstedt. "Gazing at Games." In: *ACM SIGGRAPH 2010 Courses*. ACM, July 2010. DOI: `10.1145/1837101.1837106`.

[115] Armstrong A. Takang, Penny A. Grubb, and Robert D. Macredie. "The Effects of Comments and Identifier Names on Program Comprehensibility: An Experimental Investigation." In: *J. Program. Lang.* 4 (1996), pp. 143–167.

[116] Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. "/*icomment." In: *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles*. ACM, Oct. 2007. DOI: `10.1145/1294261.1294276`.

[117] Shin Hwei Tan, Darko Marinov, Lin Tan, and Gary T. Leavens. "$@$tComment: Testing Javadoc Comments to Detect Comment-Code Inconsistencies." In: *2012 IEEE 5th International Conference on Software Testing, Verification and Validation*. IEEE, Apr. 2012. DOI: `10.1109/icst.2012.106`.

[118] T. Tenny. "Program Readability: Procedures Versus Comments." In: *IEEE Transactions on Software Engineering* 14.9 (1988), pp. 1271–1279. DOI: `10.1109/32.6171`.

[119] Ted Tenny. "Procedures and Comments Vs. The Banker's Algorithm." In: *ACM SIGCSE Bulletin* 17.3 (Sept. 1985), pp. 44–53. DOI: `10.1145/382208.382523`.

[120] Rebecca Tiarks. "What Maintenance Programmers Really Do: An Observational Study." In: *Workshop on Software Reengineering*. Citeseer. 2011, pp. 36–37.

[121] 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 and Applications*. ACM Press, 2006. DOI: `10.1145/1117309.1117357`.

[122] Allan Vermeulen, Scott W. Ambler, Greg Bumgardner, Eldon Metz, Trevor Misfeldt, Patrick Thompson, and Jim Shur. *The Elements of Java (TM) Style*. 15. Cambridge University Press, 2000.

[123] Adrian Voßkühler, Volkhard Nordmeier, Lars Kuchinke, and Arthur M. Jacobs. "OGAMA (Open Gaze and Mouse Analyzer): Open-Source Software Designed to Analyze Eye and Mouse Movements in Slideshow Study Designs." In: *Behavior Research Methods* 40.4 (Nov. 2008), pp. 1150–1162. DOI: `10.3758/brm.40.4.1150`.

[124] Alf Inge Wang and Erik Arisholm. "The Effect of Task Order on the Maintainability of Object-Oriented Software." In: *Information and Software Technology* 51.2 (Feb. 2009), pp. 293–305. DOI: `10.1016/j.infsof.2008.03.005`.

[125] Susan Wiedenbeck and Nancy J. Evans. "BEACONS IN PROGRAM COMPREHENSION." In: *ACM SIGCHI Bulletin* 18.2 (Oct. 1986), pp. 56–57. DOI: `10.1145/15683.1044090`.

[126] Frank Wilcoxon. "Individual Comparisons by Ranking Methods." In: *Biometrics Bulletin* 1.6 (1945), pp. 80–83. ISSN: 0099-4987. URL: `http://www.jstor.org/stable/3001968` (visited on 02/09/2024).

[127] Edmund Wong, Jinqiu Yang, and Lin Tan. “AutoComment: Mining Question and Answer Sites for Automatic Comment Generation.” In: *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, Nov. 2013. DOI: 10.1109/ase.2013.6693113.

[128] S. N. Woodfield, H. E. Dunsmore, and V. Y. Shen. “The Effect of Modularization and Comments on Program Comprehension.” In: *Proceedings of the 5th International Conference on Software Engineering*. ICSE ’81. San Diego, California, USA: IEEE Press, 1981, pp. 215–223. ISBN: 0897911466.

[129] Annie T. T. Ying, James L. Wright, and Steven Abrams. “Source Code That Talks: An Exploration of Eclipse Task Comments and Their Implication to Repository Mining.” In: *ACM SIGSOFT Software Engineering Notes* 30.4 (May 2005), pp. 1–5. DOI: 10.1145/1082983.1083152.

[130] S. Yusuf, H. Kagdi, and J. I. Maletic. “Assessing the Comprehension of UML Class Diagrams Via Eye Tracking.” In: *15th IEEE International Conference on Program Comprehension (ICPC ’07)*. IEEE, June 2007. DOI: 10.1109/icpc.2007.10.

[131] Zutao Zhang and Jiashu Zhang. “A New Real-Time Eye Tracking Based on Nonlinear Unscented Kalman Filter for Monitoring Driver Fatigue.” In: *Journal of Control Theory and Applications* 8.2 (Apr. 2010), pp. 181–188. DOI: 10.1007/s11768-010-8043-0.