
# A replication study of the effect of flowcharts on code comprehension among novice programmers

**Master Thesis**

Submitted in Fulfilment of the
Requirements for the Academic Degree
M.Sc. Web Engineering

Dept. of Computer Science
Chair of Software Engineering

Submitted by: Sumaia Darra

Supervisor: Prof. Dr. -Ing. Janet Siegmund
Dr. Norman Peitek

# Acknowledgment

First and foremost, I would like to express my deepest gratitude to my esteemed supervisor, Prof. Dr.-Ing. Janet Siegmund, for her invaluable guidance, continuous support, and motivating words throughout the course of this thesis. Her insightful feedback, patience, and encouragement have been instrumental in shaping my academic journey and overcoming every challenge along the way.

I extend my heartfelt thanks to my beloved husband, Mohammad Muhra, for his unwavering belief in me and his constant support throughout my entire academic journey. His encouragement during times of doubt, his faith in my ability to persevere, and his constant motivation were the light that guided me through every dark and difficult moment. You have always been my inspiration.

I would also like to pay special tribute to my late grandfather, Nizar Sosan. He was an incredible source of support for my academic aspirations from my earliest school days until his last moments. His words, “Pursue knowledge with passion and determination, I am always proud of you.,” resonate deeply with me and continue to be a source of strength.

Despite the distance between us, I would like to express my sincerest gratitude to my dear mother for her countless sacrifices, unconditional love, and continuous encouragement. Your kind words have always stayed with me.

Furthermore, I extend my deepest gratitude to my family. Their unwavering support, love, and encouragement have always been my greatest source of strength. I am also sincerely thankful to my in-laws for believing in me and for supporting me throughout my academic journey.

This thesis represents not only an academic milestone but also a personal journey of growth, perseverance, and self-discovery. Every challenge I faced along the way became an opportunity to learn, grow stronger, and to believe in my capabilities. I am proud of how far I have come and remain committed to continuous learning and personal development.

# Abstract

Program comprehension is a fundamental skill in computer science education, yet novice programmers often find it difficult to develop accurate mental models that link problem specifications to effective program solutions. Visual aids such as flowcharts have long been suggested as a way of supporting algorithmic thinking and improving comprehension. This study presents a controlled replication of an earlier experiment that investigated the effects of flowcharts on code comprehension of novice programmers. The primary goal was to validate the original findings under similar conditions while exploring potential differences arising from new participants and contextual factors. A within-subjects experimental design was employed, in which participants solved programming comprehension tasks using code snippets alone and with code snippets accompanied by flowcharts. Eye-tracking technology, EEG devices, and well-designed interviews were used to provide a multi-modal analysis of visual attention, cognitive load, response time, comprehension correctness, and subjective preference. The results showed that participants actively used flowcharts while completing the tasks and consistently said that they would prefer them to be included in introductory programming courses. This aligns with the original study's subjective findings. However, no statistically significant improvements in comprehension or response time were observed between the two conditions. This contradicts the original study's findings of improvements in both areas when flowcharts were present. Cognitive load measures also showed no significant differences, which reinforces previous findings. The persistent preference for flowcharts suggests that, while they may not directly enhance performance metrics, they could provide novice programmers with perceived cognitive support or psychological reassurance. The challenges encountered during this replication include a different pool of participants, a small sample size, minor technical updates, and limited access to the original researcher. These issues are consistent with the replication crisis observed in the behavioral sciences, emphasizing the need for careful documentation, broader replication initiatives, and standardized protocols within empirical software research. In conclusion, although flowcharts cannot guarantee measurable improvements in novice code comprehension on their own, integrating them into programming education can enhance the learning experience. Therefore, they should be considered

a useful pedagogical supplement. Future research should involve larger and more diverse samples of participants, explore alternative visual aids and adopt longitudinal designs to evaluate the long-term impact of algorithmic visualizations on learning outcomes.

**Keywords: Replication Study, Program Comprehension, Novice Programmers, Flowcharts, Cognitive Load.**

# Contents

# List of Figures

# List of Tables

# List of Abbreviations

| | |
|---|---|
| **AOI** | Area of Interest |
| **BACCII** | Ben A Calloni Coding for Iconic Interface |
| **CSV** | Comma-Separated Values |
| **EEG** | Electroencephalography |
| **EPU** | External Processing Unit |
| **FITS** | Flowchart-based Intelligent Tutoring System |
| **FLINT** | Flowchart Interpreter |
| **ICA** | Independent Component Analysis |
| **IDE** | Integrated Development Environment |
| **Opal** | Online Platform for Academic Learning |
| **PsychoPy** | Psychological Experiment Software |
| **RAPTOR** | Rapid Algorithmic Prototyping Tool for Ordered Reasoning. |
| **SDK** | Software Development Kit |
| **SFC** | Structured Flow Chart Editor |
| **SOSCISurvey** | Survey System for Social Science Research |
| **STEM** | Science, Technology, Engineering, and Mathematics |
| **TAR** | Theta-to-Alpha Ratio |
| **UML** | Unified Modelling Language |

# 1 Introduction

Since the inception of computing, software systems have progressively advanced from basic sequential programs to sophisticated, large-scale distributed applications capable of real-time operations. As software engineering established itself as a distinct academic discipline during the late 1960s [1], one of its fundamental challenges has consistently been the comprehension of source code, particularly code written by others. Mastery of program comprehension is integral to numerous aspects of software development, such as software maintenance, debugging, and effective team collaboration. Consequently, enhancing the understanding of program comprehension remains a critical objective of ongoing academic research and instructional methodologies. Over the past few decades, researchers have explored how software developers read, interpret, and mentally represent source code. This line of research has contributed to the development of influential theories, including top-down and bottom-up comprehension models [2], [3], mental model frameworks [4], and the use of visual aids to support cognitive processing during code analysis [5], [6].

Despite continuous advances in programming environments and analysis tools, empirical studies consistently report that developers still spend a significant amount of time trying to understand existing code [7], [8]. As a result, improving code understandability has become a central goal in software engineering. In parallel, the global demand for computer science education has increased significantly [9], positioning program comprehension as a critical learning objective in introductory programming courses. However, many novice students continue to face challenges in constructing accurate mental models of code, limiting their ability to solve problems effectively and produce correct programs. These difficulties highlight the need for teaching strategies that bridge the gap between syntax and conceptual understanding. Even experts often report difficulties in teaching programming, which requires carefully designed curricula that go beyond traditional methods [10].

Many educational approaches have been explored to improve the learning experience in introductory programming courses. However, individual empirical studies may not provide generalizable findings. As a result, replication studies are increasingly being used to validate and verify the effectiveness of educational tools and teaching

strategies for novice programmers. In line with this, Bennedsen and Caspersen highlighted the need for reliable, data-driven evidence to guide decisions in computing education. In their study, they examined the success and failure rates of students in first-year programming or computer science courses at the university level. Their results did not indicate a major issue or an alarmingly high failure rate among students in these courses. However, the small number of participants limited how broadly their results could be applied [11].

### 1.1 Problem Statement

Despite the increasing global emphasis on computer science education, a significant number of students continue to struggle with introductory programming courses. Studies have shown that approximately 30% of students enrolled in computer science programs drop out within the first year, and less than half complete their degree requirements [12].

These alarming statistics are closely related to the challenges faced in introductory programming courses, where students must simultaneously develop multiple skills such as syntax mastery, problem solving, and algorithmic thinking. Moreover, learning to program is widely recognized as a cognitively demanding process. Novice learners often struggle to grasp the structural and semantic aspects of programming languages and find it even more challenging to design functional solutions to given tasks [13]. Furthermore, teachers may also misjudge students' conceptual difficulties, limiting their ability to provide appropriate support when needed [13].

In addition, many students fail to build accurate mental models of program behavior, which are essential for meaningful understanding and long-term retention [14]. Research suggests that without such models, students are likely to rely on superficial patterns or misconceptions, leading to poor academic performance and increased frustration [15], [16].

The combined effect of high cognitive load, trustable tools and abstract theoretical content can severely hinder students' learning progress [17]. These barriers highlight the need for efficient tools and teaching strategies that go beyond traditional lecture-based formats.

Previous studies have produced mixed results, often limited by small sample sizes and narrow context [18], [19]. To address this gap, there is a growing interest in conducting replication studies that re-examine previous findings with larger and different groups of participants.

## 1.2 Objective

Program visualization is regarded by numerous authors as a way for developing accurate mental models. Tudoreanu et al. stated that visualization is one of the most common approaches used by students to enhance learning by creating a mental image of how things work [20]. Interactive visualization technologies, as an active learning method, increase the engagement between the learner and the subject matter [21]. Zimmermann et al. conducted a study over three years with pharmacy students, employed both quantitative and qualitative methods to assess the effectiveness of flowcharts in enhancing student learning. The study concluded that flowcharts offer a valuable alternative approach to teaching complex content. They enable students to organize and summarize information, thereby promoting meaningful learning [22]. Levy et al. observed that visualization offers a concrete model of execution essential for all students to comprehend algorithms and programming [23].

Flowcharts are a type of diagram that visually represents the logical flow of a process or algorithm using standardized symbols and arrows to indicate the sequence of operations. Programmers commonly use it to illustrate control structures, e.g., loops, decisions, and sequences, thereby simplifying and communicating complex logic. However, Shneiderman and Mayer noted that flowcharts may be an aid tool in some situations and a hindrance in others [3].

The original study conducted at the technical university Chemnitz in 2022 aimed to explore how novice programmers use flowcharts as a support tool for understanding programs. The effects of flowcharts were assessed in terms of visual attention, cognitive load, response time, comprehension accuracy, and interview responses. In the context of programming instruction, flowcharts have been proposed as effective visual tools to support novice learners in understanding program logic [24]. By replicating the original study under comparable conditions, this study aims to evaluate the robustness and generalizability of its findings. This will help to validate earlier

conclusions and extend our understanding of how visual aids, specifically flowcharts, can support novice programmers in comprehension tasks. While this study follows a direct replication model, minor adjustments have been made to address logistical constraints and improve procedural clarity. These modifications are detailed in the methodology and experimental design sections. The main objective of this study is to replicate the verify the findings of the original study and assess the effectiveness of flowcharts as a cognitive aid in code comprehension among novice programmers. This research aims to:

- Examine whether the use of flowcharts improves code comprehension accuracy in novice programmers.
- Investigate the impact of flowcharts on response time during code-related tasks.
- Assess the effect of flowcharts on perceived cognitive load.
- Explore how and to what extent novice programmers actively use flowcharts during comprehension.
- Either verify the findings of the original study or identify new challenges under similar experimental conditions.

*RQ: Can the impact of flowcharts on novice programmers' code comprehension be verified in a replication study conducted under similar experimental conditions?*

## 1.3 Thesis Structure

This thesis is structured in a way that mirrors the research process, with a logical progression of chapters. Each chapter builds on the previous one to create a coherent and comprehensive study. The present study is structured as follows:

**Chapter 2 (Literature Review):** This chapter establishes the theoretical background necessary to understand the research problem. It discusses core concepts such as program comprehension theories, cognitive strategies of novice programmers, and various approaches to measuring comprehension. In addition, the use of algorithm visualizations, particularly flowcharts, and their role in computing education are examined. The chapter concludes with a review of related work and previous replication efforts in software engineering.

**Chapter 3 (Methodology):** The methodology chapter details the experimental design used in this replication study. It outlines the variables, hypotheses, materials, and tools employed, including eye-tracking and EEG devices. It also describes the participant selection, task structure, and operational definitions to ensure clarity and replicability.

**Chapter 4 (Conduct):** This chapter describes the practical implementation of the experiment. It explains the step-by-step procedure used to conduct the study, including participant recruitment, task administration, and data collection methods.

**Chapter 5 (Data Analysis and Results):** After data collection, this chapter presents how the data were prepared, cleaned, and analyzed. It includes both descriptive and inferential statistics, focusing on fixation time, cognitive load, response time, correctness, and subjective preferences. A direct comparison to the results of the original study is also provided.

**Chapter 6 (Discussion):** The discussion chapter interprets the results considering the research question and objectives. It evaluates the significance of the findings, identifies consistencies and deviations from the original study, and addresses the implications. This chapter also outlines threats to validity that may have influenced the results.

**Chapter 7 (Conclusion and Future Work):** The final chapter summarizes the main contributions of the study and revisits its objectives and research questions. It reflects on the limitations and challenges of conducting replication research, especially in educational and empirical software engineering contexts. Directions for future research are proposed, including methodological improvements and broader replication strategies.

### 1.4 Overview of the Original Study

The original study, entitled "The Effect of Flowcharts on Novice Programmers' Code Comprehension", was a master's thesis conducted at Chemnitz University of Technology in 2022. The primary objective of the study was to investigate the role of flowcharts as a visual aid in improving novice programmers' code comprehension. The research looked at key factors such as visual attention, cognitive load, reaction time, correctness and subjective preference. The following data were extracted directly from the original study in order to provide a clear overview and to enable comparison with the findings of this replication study [24].

The experiment used a within-subjects design involving 11 participants. Each participant completed a total of 14 tasks: 7 using code snippets alone and 7 code snippets accompanied by flowcharts. All tasks were written in Java. The participants had 3 to 5 years of academic programming experience, with only two participants having up to 1.5 years of professional programming experience. Most participants had little or no experience with flowcharts. The demographics of the participants from the original study are summarized in Figure 1.1.

| | |
|---|---|
| Male | 7 |
| Female | 4 |
| Age (in years) | 26 ± 3 |
| Learning Programming (in years) | 4 ± 1 |
| Professional Programming (in years) | 0.5 ± 1 |
| Java Programming (in years) | 2 ± 1 |
| Flowchart Experience | 0.5 ± 0.5 |

**Figure 1.1 Demographic overview of the original study [24].**

The order of the tasks was randomized to minimize learning effects. During the experiment, participants' visual behavior was tracked using an eye-tracking device, and cognitive load was measured using an EEG headset. Prior to the tasks, participants completed a prequestionnaire assessing their experience with programming and flowcharts. After completing the experiment, a post-questionnaire and an interview were conducted to gain insight into their strategies and preferences.

The study was announced in computer science classrooms and social media groups. Students who wished to participate booked an appointment online, then received an automated email with a brief description of the experiment and relevant information. Participants completed a pre-questionnaire to collect demographic information. Figure 1.2 shows the prequestionnaire given in the original study.

Participants received general instructions and a brief explanation of the flowcharts. They then completed two mock tasks: one containing a source code only and another one containing both a source code with a flowchart. After that participant began the

experiment, which included problem-solving tasks in the Java programming language. They were asked to determine the output of the provided snippet, in the next slide four answers were given with a skip option. Each task was followed by a 10-second cross-fixation rest period.

- How long have you been programming for educational purposes? (In years):
- How long have you been programming professionally? (In years):
- How do you estimate your programming experience on a scale from 1 to 10?
- How do you estimate your programming experience compared to fellow students and experts with 20 years of a practical experience on a scale from 1 to 5?
- How do you estimate your experience using flowcharts to visualize or design a program on a scale from 1 to 5?
- How experienced are you with the following programming languages on a scale from 1 to 5? Java, C, Python, JavaScript.
- How many additional programming languages do you have moderate experience with?
- How experienced are you with the following programming paradigms on a scale from 1 to 5? Functional programming, Object-oriented programming, Imperative programming, Logical programming

**Figure 1.2 Pre-questionnaire used in the original study [24].**

The experiment lasted 30 minutes. The experiment ended either upon completion of the 14 tasks or after 30 minutes. At this point, participants were asked to take part in a post-questionnaire interview to gain a better insight into their overall experience and to identify their preferences regarding the use of flowcharts as reported in the study. The post-questionnaire questions are shown in Figure 1.3.

- How did you solve the tasks?
- Did you use a specific strategy to solve the tasks?
- How much did you refer to the flowcharts and how much to the code?
- For which task did you spend more time, when there was a flowchart present or not?
- Do you prefer tasks where the flowchart was present or not?
- Do you think there is an advantage or disadvantage when the flowchart is included along with the code? Why?

**Figure 1.3 Post-questionnaire used in the original study [24].**

### *1.4.1 Results of the Original Study*

The results showed that participants made frequent use of flowcharts and showed improved correctness on tasks where flowcharts were present. Eye-tracking data confirmed that participants actively directed their gaze to the flowcharts throughout the tasks, indicating that these visual aids were indeed used in the comprehension process. However, response times were longer and no significant differences in cognitive load were found. Subjective feedback indicated that most participants preferred the presence of flowcharts, suggesting that they were useful in aiding comprehension.

Despite the promising results, the original study had several limitations that may have affected the internal and external validity of the findings. The small sample size limited statistical power and generalizability. In addition, individual differences in familiarity with flowcharts and Java syntax may have introduced variability into the results. The use of physiological measures, while informative, also presented technical challenges that could affect the accuracy of the data. These threats to validity highlight the need for replication under more diverse conditions.

These findings highlight the potential of flowcharts to enhance mental model formation in novice programmers. However, given the limited sample size and scope, the original author emphasized the need for replication studies to verify the generalizability of these findings [24].

# 2 Literature Review

The purpose of this literature review is to provide a conceptual and empirical basis for the current study by examining relevant research on program comprehension, strategies, approaches to visual aids used in training (e.g. flowcharts) and the cognitive challenges faced by novice programmers. In addition, the technical characteristics of eye tracking and EEG equipment are discussed.

## 2.1 Program Comprehension

Understanding source code is a core cognitive activity in software development and maintenance. Since the 1980s, researchers have studied how programmers read, interpret, and mentally represent code, with the aim of uncovering the cognitive strategies that developers use when interacting with software systems [2],[25]. Program comprehension refers to how developers make sense of existing code, including the structure of components, their relationships, functionality, and dynamic behavior [26].

It involves forming a mental model of the system's purpose and structure, which is essential for tasks such as debugging, maintenance and evolution. Several studies have shown that developers spend a significant amount of time trying to comprehend code. It has been estimated that up to 58% of developers' time is spent understanding code, highlighting its critical role in effective software development [27].

According to Siegmund, this proportion has remained largely unchanged for decades despite advances in development tools, largely due to the increasing complexity of modern software systems. Siegmund also notes that the stagnation of comprehension research since the mid-1990s has led to a lack of rigorous evaluation of tools and techniques, which may contribute to the proliferation of poorly validated features that offer limited practical support to developers [8]. In addition, program understanding is particularly challenging in collaborative environments, where comprehending code written by others is often required. Developers must efficiently construct mental models; cognitive representations built from experience and pattern recognition to navigate unfamiliar code and solve complex problems [28]. While program comprehension remains a critical cognitive process in software engineering, current

challenges including increasing system complexity and insufficient tool evaluation highlight the continued need for empirical research and replication studies to support developer, novices, educator performance and comprehension efficiency.

### 2.1.1 Program Comprehension Strategies

The Mental Model Framework provides a basis for understanding how developers cognitively process source code during program comprehension. Building on this understanding, the top-down, bottom-up and integrated models offer complementary strategic perspectives for approaching comprehension tasks. Together, these strategies, when integrated with the mental model framework, provide a comprehensive view of how developers approach the challenges of program comprehension, offering insights into both their cognitive processes and practical strategies.

### 2.1.2 Top-Down Comprehension

The top-down comprehension model is a hypothesis-driven strategy in which developers use their domain knowledge and experience to form high-level assumptions about the purpose of a program. This approach was extensively studied by Brooks and later extended upon by Soloway and Ehrlich [2], [25].

Brooks proposed in theory that program comprehension begins with a high-level hypothesis, which is then refined through sub-hypotheses in a hierarchical manner moving from general abstractions toward concrete code elements. This iterative refinement helps the programmer systematically narrow down their understanding, ultimately forming a mental model of the program’s behavior. A key concept in Brooks' top-down model is the use of beacons, distinctive patterns or cues in the code that signal specific structures or operations. These beacons help the developer to navigate and interpret the code efficiently [2].

Similarly, Soloway and Ehrlich introduced the notion of programming plans, stereotypical fragments of code that represent common routines or goals. These plans, often used by experts, allow them to associate segments of code with higher-level intentions [29]. Their work suggests that top-down understanding is primarily accessible to expert programmers, who can form accurate hypotheses based on their familiarity with code structures and domain-specific knowledge. This strategy tends to

be less cognitively demanding than the bottom-up approach, as it allows programmers to interpret individual code statements in the context of a broader conceptual framework.

As a result, experienced developers typically prefer top-down comprehension when solving tasks [30]. Although beacons are generally considered to be helpful, Wiedenbeck's research revealed a potential downside: if beacons are inaccurate or misleading, they can lead readers to draw incorrect conclusions, significantly impeding comprehension [31]. This dual role highlights the importance of designing code with clear, consistent cues to aid comprehension.

### 2.1.3 Bottom-Up Comprehension

The bottom-up comprehension model describes how a programmer builds understanding by starting at the lowest levels of abstraction such as individual code statements or basic control structures and gradually combining them into higher-level concepts. This approach is particularly useful for novice or inexperienced developers, who may lack the experience to generate top-down hypotheses effectively. Without a pre-existing mental model, these programmers must examine the code directly to infer its function and build understanding incrementally [32].

A fundamental cognitive mechanism underlying this strategy is chunking, a concept introduced by Miller in 1956. Chunking refers to the grouping of small pieces of information into meaningful units to improve memory and comprehension. Miller showed that working memory has a limited capacity, about seven items, plus or minus two, emphasizing the importance of organizing information efficiently [33].

This principle provided the basis for early cognitive models of programming behavior. Schneiderman and Mayer extended Miller's work to software comprehension, illustrating how programmers’ abstract fragments of code into semantic chunks that are temporarily stored in working memory. These chunks, which may represent syntactic patterns or conceptual operations, are then processed using prior knowledge stored in long-term memory. This process allows developers to associate isolated pieces of code with more abstract goals or structures, enabling higher-level comprehension [3].

Building on this framework, Pennington emphasized that bottom-up comprehension begins with the identification of basic control constructs, such as sequences, loops, and conditionals, which serve as the fundamental units for chunking. These control structures are analyzed for procedural meaning and then reorganized to reflect the broader functional intent of the program. This process supports the construction of a structured mental model and enhances the developer's ability to reason about program behavior [4].

### 2.1.4 Hybrid Program-Comprehension Strategies

Hybrid strategies refer to flexible cognitive models that allow programmers to integrate both top-down and bottom-up approaches to program understanding. Developers dynamically switch between strategies depending on the nature of the task and their level of familiarity with the code. Research suggests that hybrid strategies can improve efficiency by allowing developers to focus only on the relevant parts of the code, reducing the time required to build a coherent mental model. In contrast, relying on a single strategy, especially a bottom-up one, can lead to more comprehensive knowledge, but also increased cognitive load and processing time [34].

Siegmund discusses how experienced programmers often start with a top-down strategy, using their domain knowledge to form high-level hypotheses about the functionality of the program. When these initial hypotheses fail to adequately explain specific code fragments, developers switch to a bottom-up analysis to refine or revise their mental model. This strategy allows them to resolve ambiguities and update their understanding efficiently. Top-down comprehension remains the preferred approach due to its lower cognitive demands, while bottom-up is typically used as a fallback when encountering unfamiliar structures [8].

Similarly, Koenemann and Robertson conceptualize program comprehension as a goal-directed, hypothesis-driven process. They argue that readers selectively focus on code segments that are most relevant to their current goals. Initially, comprehension tends to follow a top-down path, but if hypotheses cannot be validated or inconsistencies arise, the reader switches to a bottom-up process to explore the program structure in more detail [35].

## 2.2 Approaches to measure Program Comprehension

Various methods have been used in empirical software engineering to assess program comprehension. The following sections summarize the main techniques used in this study.

### 2.2.1 Task Performance

Task performance is one of the most widely used indirect methods of assessing code understanding. This approach assesses a participant's ability to complete programming-related tasks, often measuring two key metrics: correctness and response time. Correctness refers to the accuracy of participants' responses, and for some researchers, it is an indicator of their level of understanding. Response time provides insight into the efficiency of cognitive processing and familiarity with the code or domain.

Task performance can be implemented in both face to face and online experimental settings. It allows for both individual and group-based analyses, making it versatile for assessing comprehension in diverse populations. Typically, participants with strong knowledge of domain perform more accurately and efficiently, whereas novices often take longer and make more errors. These performance-based metrics provide valuable, objective insights into the effectiveness of comprehension strategies [36].

### 2.2.2 Interviews

Interviews are a qualitative data collection technique that involve structured or semi-structured dialogues between the researcher and the participant. Unlike quantitative methods, which focus on 'how much' or 'how many', interviews explore 'how' and 'why' participants behave or think in certain ways. This makes them particularly useful for gaining deeper insights into cognitive processes and learning experiences [37]. According to Kvale and Brinkmann, an interview is "a conversation that has a structure and a purpose; it goes beyond the spontaneous exchange of views because it is based on the researcher's agenda and focuses on eliciting specific types of information" [38].

Interview structures range from strictly pre-defined to completely open-ended, depending on the aims of the study. In program comprehension studies, interviews are often used to supplement quantitative data. For example, Xia et al. conducted a large-scale field study with professional programmers, combining observational data with follow-up interviews. Their results showed that comprehension tasks consumed a

significant amount of work time, especially for less experienced developers, highlighting the value of qualitative insights in understanding how developers' approach and experience comprehension challenges [26]. While interviews can provide rich, detailed data, they also present challenges. The risk of interviewer bias, both at the questioning and interpretation stages, can affect validity. Careful design and reflexivity are therefore essential to ensure reliable findings [39].

### 2.2.3 Subjective Ranking

Subjective rating methods aim to capture participants' perceptions and self-assessed levels of understanding. The most used tools include the Likert scale and the semantic differential scale. These tools convert qualitative judgements into quantifiable data that can be analyzed statistically. The Likert scale measures agreement or disagreement with a series of statements [40]. While these scales are valuable for capturing perceptions, they are inherently limited by the specificity and clarity of the questions asked. Subjective rating techniques have been widely used in studies of program comprehension. For example, Apel et al. used Likert-type items to assess participants' familiarity with different programming paradigms [41]. Similarly, Miara et al. used subjective ratings to determine the most comprehensible indentation styles in source code, revealing preferences for two- or four-space indentation levels [42].

## 2.3 New approaches to measure Program Comprehension

Various approaches and technologies have been used to gain further insight into the process of program comprehension. These strategies are discussed in the following sections.

### *2.3.1* Eye Tracking

Eye tracking has become a widely used tool for investigating cognitive processes, particularly in domains where visual attention plays a central role. The eye-mind assumption, proposed by Just and Carpenter, suggests that individuals focus their attention exclusively on the part of the stimulus currently under observation, and that eye movements closely reflect ongoing cognitive processes [43]. This assumption, together with the immediacy assumption, forms the theoretical basis for interpreting eye-tracking data and provides insight into the specific areas of focus, cognitive effort and time required for comprehension.

Rayner highlighted that recent advances in eye-tracking technology have made it possible to collect gaze data with high temporal and spatial resolution [44]. Over the past few decades, eye-tracking has become increasingly popular in software engineering research, providing a powerful means of observing how developers comprehend code. Eye trackers record gaze data, capturing the user's overt visual attention [45]. Lim et al. categorize eye trackers into three main types: mobile eye trackers, virtual reality head-mounted trackers, and desktop-based systems [46].

Eye-tracking data typically consists of horizontal and vertical coordinates that indicate eye positions on a visual stimulus. A calibration process maps sensor input to display coordinates, while event detection algorithms distinguish between different types of eye movements, primarily fixations and saccades [47]. Fixations, where gaze remains stable over a region, are associated with cognitive processing and interpretation. Their duration varies with task complexity, stimulus design and individual factors. Saccades, on the other hand, are rapid eye movements that shift gaze and provide minimal visual input. Data analysis often focuses on Areas of Interest (AOIs), which are predefined based on research objectives [48].

Given the volume of data generated, visualization techniques are critical in eye-tracking analysis. Tools such as heat maps reveal spatial , gaze plots (scan paths) and temporal focus. Gaze plots show the sequence and duration of fixations, while heat maps use color gradients to illustrate the intensity of attention as shown in Figure 2.1.

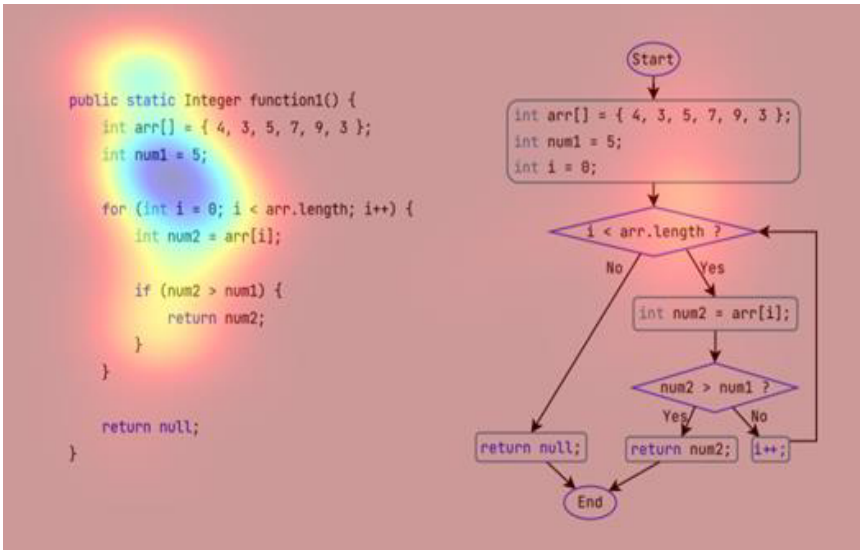



**Figure 2.1 Heatmap visualization over a code snippet with Flowchart.**

Peitek et al. showed that eye tracking can be combined with more advanced methods, such as fMRI, to gain deeper insights into program comprehension [48].

The effect of visual layout and color in UML diagrams has also been studied, with results showing that expert developers use these elements more effectively than novices [49], [50]. Crosby and Stelovsky observed significant differences in gaze behavior between novice and expert programmers [51].

Sharif and Maletic conducted an eye-tracking study to investigate the influence of different naming styles. They found that underscore-style labels were read faster than camel-case labels [52]. Similarly, Park et al. investigated the effect of source code readability rules such as minimizing nesting on comprehension. Using eye tracking, they showed that following these rules increased confidence and reduced reading time [53]. In addition to gaze position, eye-tracking systems can capture other cognitive indicators. For example, pupil dilation is a widely accepted measure of cognitive load [54]. Beatty and Kahneman found that pupil size increased with the difficulty of memory tasks [55], while Hess and Polt linked dilation to the complexity of mathematical problem solving [56]. Behroozi et al. used this metric to assess stress and strain during programming tasks [57]. However, as noted by Doughty, these measures are sensitive to various confounding factors, such as fatigue, humidity and environmental lighting [58].

### 2.3.2 Electroencephalography

The electroencephalogram (EEG), first identified by Hans Berger in 1924, was a breakthrough in neuroscience. Berger was the first to record the brain's electrical potential between 50 and 100 μV from the human cerebral cortex. Berger published his results in 1929, detailing rhythmic brain activity such as alpha and beta waves, and noting how these signals varied according to an individual's state of wakefulness or relaxation [59].

EEG is now widely used in several fields, including neurology to diagnose conditions such as epilepsy and sleep disorders, and cognitive neuroscience to study functions such as perception, attention and cognitive load [60].

EEG is a non-invasive neuroimaging technique that records the brain's electrical activity via electrodes placed on the scalp, allowing real-time monitoring of neural oscillations [61]. An EEG cap is shown in Figure 2.2.

**Figure 2.2 EEG Cap [62].**

Standard electrode placement follows the International 10-20 system, which ensures consistency and accuracy in the positioning of electrodes over specific regions of the scalp, labelled by anatomical zones (e.g. F for frontal, C for central) and numerical indicators for hemisphere location. EEG signals are categorized into frequency bands according to cognitive states [63], [64]. The main bands are described below and illustrated in Figure 2.3.

- **Delta (0.5-4 Hz):** Associated with deep sleep and unconsciousness.
- **Theta (4-8 Hz):** Associated with meditation, creativity and cognitive engagement.
- **Alpha (8-12 Hz):** Reflects relaxed alertness and low cognitive load.
- **Beta (12-30 Hz):** Associated with concentration, problem solving and mental activity.
- **Gamma (>30 Hz):** Associated with perception, awareness and information processing.

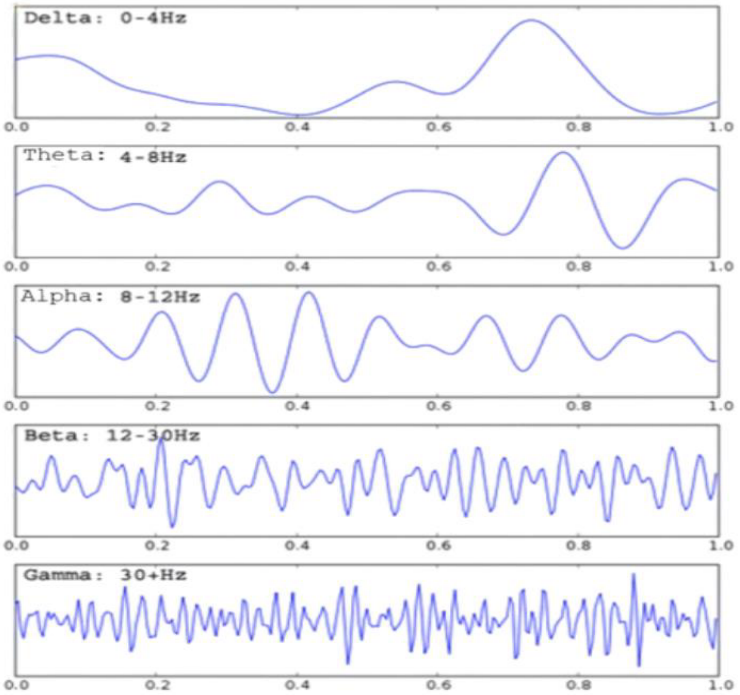



**Figure 2.3 EEG signal bands [65].**

Due to the complexity of brain signals, EEG data is amplified and analyzed using mathematical techniques to visualize the energy distribution across frequencies, commonly referred to as the power spectrum [66].

EEG data are typically displayed as waveforms for real-time interpretation of brain activity. However, EEG signals are susceptible to several artefacts that can distort the analysis as shown in  Figure 2.4These include [67], [68]:

- **Ocular artefacts:** Caused by eye movements; appear below 5 Hz.
- **Muscle artifacts:** Result from facial or scalp muscle activity (50-150 Hz).
- **Respiratory artifacts:** Caused by impedance changes during breathing.
- **Cardiac artifacts:** Caused by heartbeats, typically around 1 Hz.
- **Line noise artefacts:** Caused by electrical noise (50-60 Hz), can be removed by filters.

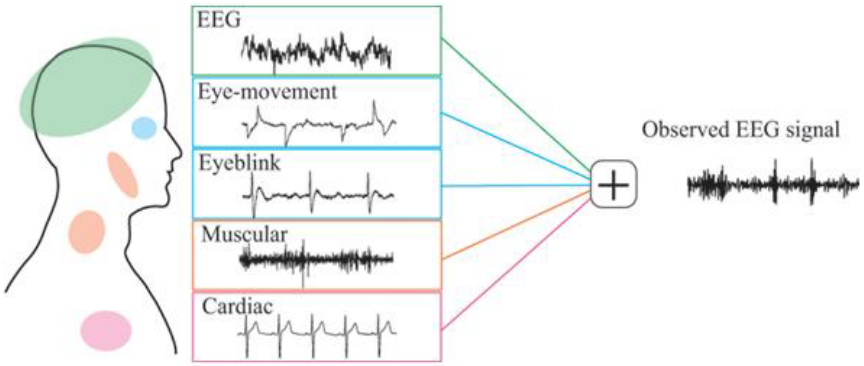



**Figure 2.4 Configuration of an observed EEG signal including biological artifacts [68].**

Artefacts must be removed or reduced to ensure accurate signal interpretation. A well-known physiological indicator extracted from the EEG is the theta/alpha power ratio, which is strongly correlated with cognitive workload. Studies show that theta power increases and alpha power decreases as cognitive demand increases, making the ratio a reliable marker of mental effort [69].

## 2.4 Novice Programmers

The comprehension process of novice programmers has been a major focus of research since the 1970s. Soloway and Spohrer analyzed various aspects of novice programming in their paper “Studying the Novice Programmer”. Their work highlighted common challenges faced by novices, including misconceptions about programming principles, a tendency to prioritize syntax over problem-solving methods, and difficulties in developing effective debugging skills [70]. Similarly, Sheil examined methodological concerns in introductory programming courses, emphasizing how psychological research on programming could improve pedagogical approaches. In addition, Robins and Rountree, in their comprehensive review, explored the challenges faced by novice programmers and the complexities involved in teaching programming. They found that novices often struggle with fundamental concepts, leading to difficulties in both program comprehension and generation [71]. Winslow provided a psychological perspective on programming pedagogy, noting that many challenges have inadequate mental models and use an ineffective "line-by-line" approach to programming [72].

Sheil underscored the complexity of understanding how students learn to program and highlighted the importance of empirical studies to evaluate cognitive processes and

teaching methods in programming education [73]. Lishinski et al. investigated how problem-solving ability relates to programming performance, finding that strong problem-solving skills are a key predictor of success in programming education [74]. Mase and Nel observed that novice programmers often make unnecessary errors when writing code, often due to a poor understanding of basic programming concepts. Their study identified 21 common programming errors, grouped into four categories: syntax, semantic, logic and type errors. Semantic and type errors were the most common. Their results showed that semantic and type errors were the most common [75].

Despite extensive research, there is no consistent definition of novice programmers, as the terms 'novice' and 'expert' are context dependent. A programmer may be a novice in some areas and an expert in others. In general, novices are individuals who are new to programming and lack the experience, depth of knowledge and strategic approaches of experts [76]. Several criteria and methods were used to classify novices and these are presented below:

Duration of experience is a widely used metric. Lister defines novices as programmers who have engaged in programming education for three to four years [76], while Sillito et al. classify those with fewer than two years of professional programming experience as novices [77].

Educational background is another criterion, including education levels, grades, the number of programming languages known, and the quantity of programming courses completed. Ricca et al. classified undergraduates as novices and graduates as experts [78].

The size of programs written also serves as a measure of expertise. Muller categorized programmers based on the number of lines of code in the largest program they authored, with programmers of minimal expertise producing programs of up to 500 lines of code [79].

Research conducted by Kleinschmager et al. and Feigenspan et al. demonstrated a positive relationship between individuals' self-perceived programming competence and their actual performance in comprehension tasks as well as programming

coursework [30], [80]. Building on this approach, Bunse introduced a five-point self-assessment scale to classify programmers, identifying those at levels one and two as novice developers [81].

Pre-testing is another method used to assess programming experience. Biffl et al. used pre-tests to divide subjects into groups categorized as novice, intermediate and expert [82]. Supervisor assessment has also been used, as shown by Hannay et al. where supervisors classified participants into similar categories based on their programming experience [83]. These different perspectives highlight the multidimensional nature of novice programmers, which this study will consider in the analysis.

## 2.5 **Related work**

In computer science education, flowcharts have long been recognized as effective visual tools for illustrating program logic and helping learners to understand control structures. A wide range of tools and techniques have been developed to improve the pedagogical use of flowcharts. In parallel, replication studies have received increasing attention in software engineering research to validate original findings and to explore the challenges associated with reproducing empirical results in educational and experimental contexts.

### 2.5.1 Flowchart Tools Overview

These tools help learners develop the mental models necessary to understand process logic, using shapes such as boxes to represent processes, diamonds to represent decision points, and directed lines to represent control flow, which typically begins and ends at well-defined points. Flowcharts are thought to facilitate the construction of mental models by visually representing the logical structure of a program. Previous research suggests that visualizing algorithms can help learners to abstract away from syntactic details and focus instead on the underlying computational logic.

Originally introduced in the 1940s by Herman Goldstine and John von Neumann as a method for "planning and coding problems," flowcharts became indispensable in the development of computer programs. Analysts used them to visually sketch algorithms, which programmers then translated into machine-readable code [84]. Since then, introductory programming courses have widely adopted flowcharts due to their

effectiveness in improving understanding and performance for visual learners. By providing a structured means of visualizing algorithms, flowcharts help students grasp complex programming concepts more effectively [85].

Empirical studies have shown that flowcharts do not increase the cognitive load associated with understanding the syntax of programming languages. By shifting the focus away from syntax, they allow programmers to concentrate on problem-solving skills.

For example, the original study examined the use of flowcharts alongside code snippets, evaluating their impact on novice programmers' comprehension. The study assessed metrics such as fixation time, cognitive load, response time, correctness, and subjective preference. The results showed that using flowcharts with code snippets enhanced novices' mental models and understanding, resulting in an 18% increase in correctness [24].

Flowchart-based programming environments have evolved to address these challenges, enhancing their utility as educational tools. These environments reduce syntax-related cognitive load, enable step-by-step execution, and provide real-time variable inspection. For instance, BACCII and BACCII++, developed at Texas Tech University, support procedural and object-oriented programming through flowchart creation and automatic code generation. Empirical studies at the university showed significant improvements in students' performance when using these tools compared to traditional methods [86], [87]. Western Kentucky University developed FLINT, which focuses on teaching algorithm design through visual execution. While it effectively aids visual learners, its lack of text representation limits its broader applicability [88], [89]. Similarly, Sonoma State University developed the SFC Editor, which generates pseudocode from flowcharts but lacks execution capabilities, necessitating external tools for program execution [90].

RAPTOR, developed at the United States "Air Force Academy", integrates a drag and drop flowcharting interface with features such as real-time variable inspection and automatic code generation in multiple languages, including Ada, C#, and Java. Research indicates that students overwhelmingly prefer RAPTOR for algorithm

representation, and it enhances their problem-solving skills [91], [92]. Figure 2.5 shows an example of a RAPTOR flowchart.

Andrew S. Scott created Progranimate, a web-based e-learning platform designed to teach programming through dynamic, structured flowchart construction, synchronized code generation in multiple programming languages, and animated execution. The platform offers a user-friendly interface that frees students from the complexities of syntax, enabling them to concentrate on problem-solving and comprehending the abstractions and semantics of programming. Figure 2.6 depicts an example of a programmer's interface, showcasing its features and constructs.

Scott conducted an in-depth empirical study to evaluate the effectiveness of Progranimate in enhancing the programming education of first-year undergraduate students. The study was carried out at the University of Glamorgan and the University of Manchester for at least 11 weeks. A total of 242 students participated, with 99 completing the final questionnaire. The results were statistically significant, demonstrating that Programmer's simplicity enabled students to quickly grasp imperative programming concepts such as variables, input/output, assignment, and calculations. The first tutorial introduced these foundational concepts in an engaging and motivating problem-solving context, surpassing the capabilities of standard development environments. The study's positive outcomes led to the integration of Progranimate into programming courses at the University of Glamorgan and the University of Manchester, showcasing its enduring value as an educational tool. Scott's research reinforces the importance of interactive and adaptive platforms like Progranimate in supporting the development of problem-solving skills and fostering deeper comprehension of programming concepts among novice programmers [93].

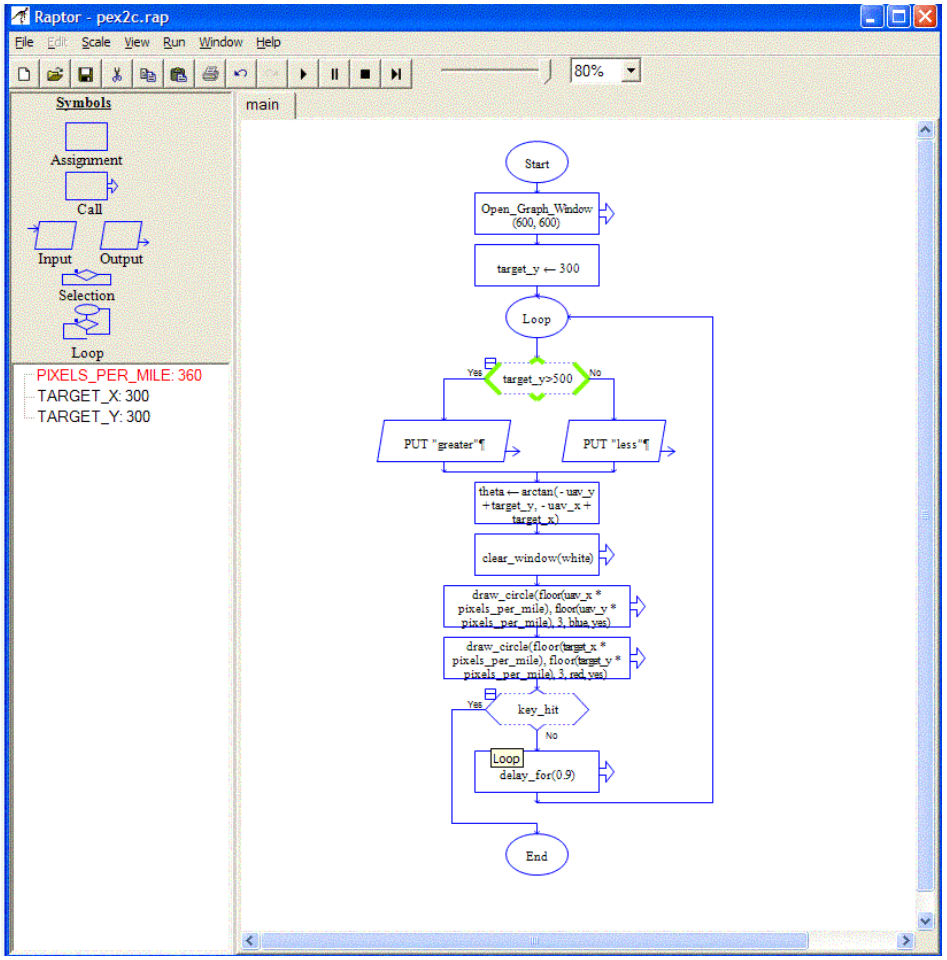



**Figure 2.5 An example of a RAPTOR flowchart [94].**

Atanasova and Hristova developed the Flowchart Interpreter (FCI) at the University of Rousse in Bulgaria to enhance programming education. FCI enables users to construct and execute flowcharts visually or non-either visually, supporting novice programmers in developing essential debugging and testing skills.

However, FCI's reliance on primitive shapes and lack of color differentiation for flowchart construction can increase cognitive load and divert users' attention from core programming concepts. Additionally, a literature search indicates that no published or widely available empirical research has evaluated the effectiveness of FCI, limiting its validation as a robust educational tool [95].

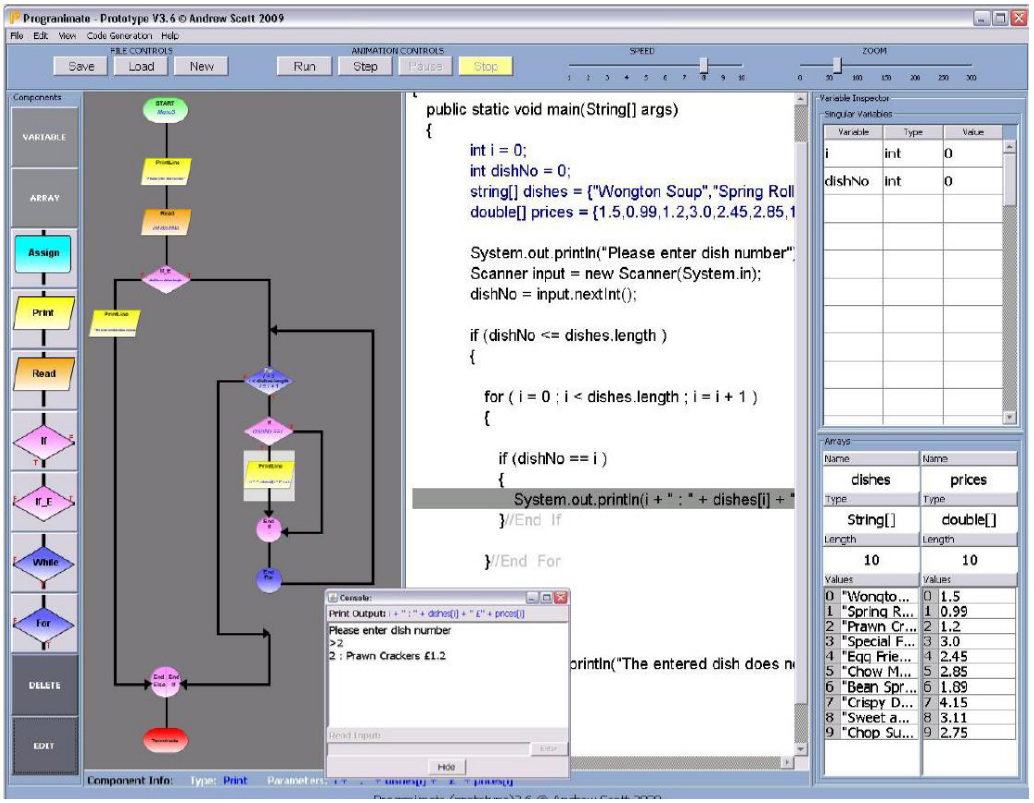



**Figure 2.6 The Programinate programming environment [96]**

Andrzejewska and Stolińska conducted an eye-tracking study to compare the effectiveness of structured flowcharts and pseudocode for algorithm comprehension tasks. They found that participants using flowcharts completed tasks more efficiently, made fewer errors and reported higher levels of confidence, particularly when dealing with complex algorithms. These results suggest that flowcharts can significantly improve cognitive processing in challenging programming scenarios [97].

Dzidek et al. investigated the role of UML in maintenance tasks. Their empirical study concluded that UML could improve the maintainability of software without increasing the time required, highlighting its potential for advanced learners [98].

In a separate study, Cabo examined the integration of flowcharting into Python instruction. He found that 66% of students believed that flowcharts made problem solving easier. This highlights their potential as a tool for developing logical thinking skills before engaging with syntax [99]. Similarly, Xinogalos conducted a survey of flowchart-based programming environments and concluded that these visual platforms improve program comprehension and foster problem-solving abilities by enabling users to construct and understand program logic visually [100].

Together, these studies highlight the importance of flowcharts as cognitive and instructional tools in introductory programming education.

## 2.6 Replication Study

Replication plays a critical role in verifying experimental results, particularly in empirical software engineering, where it is used to assess the consistency of outcomes across different methods, technologies, and populations [93], [101]. In computing education, replication has become increasingly important for validating instructional interventions and for understanding learner behavior in diverse contexts [102].

As evidence-based practices gain prominence, replication studies are essential for developing a cumulative body of reliable knowledge that informs educational and practical advancements. Despite their value, replication studies face several persistent challenges that can influence outcomes and complicate comparisons. Related factors will be mentioned in the following sections.

In experimental research, replication refers to repeating a study under comparable conditions, potentially with variations in sampling or context. The objective is to assess the reliability and external validity of the finding of a previous study.

Moreau and Wiebels stated that replication requires not only duplication of procedures, but also careful attention to the intentions and methodological integrity of the original study. This ensures meaningful interpretation of replication results and reduces ambiguity in assessing their consistency with the original findings [103].

Camerer et al. conducted a large-scale empirical replication of 21 high-impact social science studies originally published in Nature and Science. Their study implemented extended sample sizes and used pre-registered protocols reviewed by the original authors. Only 62% of the experiments replicated statistically significant effects in the same direction, with replicated effect sizes averaging about half those of the originals. These results highlight the inherent challenges in replication, such as statistical power, experimenter bias, and methodological drift. Additionally, the study noted a correlation between expert predictions and actual replication outcomes, suggesting tacit knowledge within the research community about result robustness [104].

Despite its importance, replication is not without limitations. A successful replication does not definitively validate the original findings, nor does a failed replication categorically refute them. Variations in outcome may stem from uncontrolled variables, methodological flaws, system complexity, or random variation. Therefore, the scientific community increasingly evaluates replicability not as a binary measure, but as part of a continuum of evidentiary credibility, informed by a cumulative body of research [105].

### 2.6.1 Challenges in Conducting Replication Studies

Replication studies are essential for validating and strengthening the reliability of empirical research findings. However, they face a variety of methodological, technical, and systemic challenges that can hinder their execution and impact:

- Limited Access to Original Researchers: A frequent obstacle in replication research is the inability to contact or collaborate with the authors of the original study. This lack of communication can impede the replication team's understanding of methodological choices or experimental nuances, even when documentation is provided.
- Technical Obsolescence: Replication efforts may also be compromised by the rapid evolution of technology. Software tools, programming libraries, and experimental platforms used in the original study may become deprecated, incompatible, or unavailable in the intervening years.
- Variability in Participant Characteristics: Replicating studies involving human subjects introduces inherent variability, especially in cognitive and behavioral domains e.g., program comprehension. Differences in participant background, skill level, and demographic profile can significantly affect results.

### 2.6.2 Types of Replications

Replication studies can be broadly categorized based on their methodological alignment with the original study:

- **Direct Replication:** Involves repeating the original study's procedures as closely as possible to verify its findings under nearly identical conditions [106].
- **Conceptual Replication:** Tests the same hypothesis using different methods or operationalizations to evaluate the validity of the underlying theory [107].

- **Systematic Replication:** Involves multiple studies with intentional methodological or contextual variations to assess the robustness of findings. This is more commonly related to systematic reviews than singular empirical replications [103].
- **Constructive Replication:** Retains the core elements of the original study while adding new variables or methodological enhancements to generate additional insight [108].

### 2.6.3 Replication in Computer Science

Replication in software engineering serves several critical functions such as controlling for sampling errors, testing independence of findings from specific researchers or contexts, and confirming protocol consistency. These practices help build a resilient empirical foundation and improve the overall quality of engineering research.

Nosek and Errington stress that replication is essential for verifying scientific claims, including software engineering. They emphasize the value of examining findings across different organizational settings, especially within agile development. Agile practices are now widely adopted and have undergone replication-based evaluations to test their effectiveness across diverse project scopes, team structures, and organizational cultures. These studies reveal both consistent benefits and challenges [109].

Bacchelli and Bird underscore the centrality of code review in quality assurance and argue that replication studies offer valuable insights into how factors such as review tools, reviewer expertise, and team interaction styles affect performance outcomes. Comparisons between formal and lightweight review techniques across different contexts have yielded data that inform best practices and review efficacy [110].

Harman et al. highlights the maturing landscape of software testing research and the critical role of replication in refining testing strategies. Their findings confirm that consistent application of test designs such as mutation testing or regression coverage across different software systems helps to clarify which approaches are reliably effective. This, in turn, enables the software engineering community to extract best practices with empirical support [111].

# 3 Methodology

A controlled experiment was chosen as the fixed design strategy to evaluate the effect of flowcharts on novice programmers' code comprehension. This method involves deliberately manipulating one or more factors, known as independent variables, while holding all other conditions constant. The resulting changes are measured in terms of their impact on one or more dependent variables, allowing researchers to assess causal relationships [112].

At this stage of the study, the research objective, the independent and dependent variables, and the hypotheses are clearly defined. These components directly influence the experimental design and guide how the hypothesis will be tested empirically through structured observation. They also determine how variations in the independent variable are expected to affect the dependent variable under controlled conditions. Careful definition and operationalization of these elements guide the overall research design and inform the procedures for data collection, measurement, analysis and hypothesis testing. Runeson et al. stated that the more thoroughly these preparations are completed, the smoother the experiment will proceed [112].

## 3.1 **Goal**

The primary objective of this replication study is to verify the findings of a previous study that examined the effectiveness of flowcharts as a visual aid in supporting novice programmers' code comprehension. The original study suggested that flowcharts improved understanding of control structures and algorithmic flow by facilitating the development of appropriate mental models [24].

This replication aims to assess whether the benefits observed in the original study such as improved task correctness, increased fixation time, and stronger subjective preference can be reproduced under similar experimental conditions with a new sample of participants and a revised set of three algorithmic tasks. Following the methodology, measurement tools and experimental design of the original study, this research seeks to assess the generalizability and robustness of the findings in the context of visual aids in computer science education. The research question is formulated as follows:

RQ: Can the impact of flowcharts on novice programmers' comprehension be verified through replication under similar experimental conditions?

### 3.2 Modifications to Experimental Material

In this replication, three of the original code snippets (i.e. MultiplicationByAdding, SumOfIntervalNumbers and DiffPosAndNegNumbers) were replaced with BinarySearch, BubbleSort and Power. This change was made to better reflect a wider range of algorithms commonly taught in introductory programming courses, and to provide a more comprehensive test of the effectiveness of flowcharts in supporting code comprehension. These algorithms are standard components of undergraduate science education and are used to introduce key concepts such as iteration, recursion, decision making, and algorithmic efficiency, as outlined in Introduction to Algorithms by Cormen et al. [113].

The substitution was further motivated by observations from the original study where participants showed limited engagement with the flowcharts associated with the replaced algorithms. These tasks also had the highest correct rates, suggesting that their low complexity or familiarity may have led participants to disregard the visual aids. To address this, the newly selected snippets provide a slightly greater cognitive challenge, encouraging more active and purposeful interaction with the flowcharts during the comprehension process.

A pilot study was conducted with the new snippets (BinarySearch, BubbleSort and Power) to ensure that they matched the difficulty of the remaining original tasks. The results confirmed that these snippets were appropriately challenging and consistent with the experimental design. Their inclusion supports a more nuanced examination of how flowcharts influence understanding of different algorithmic structures, thus supporting the broader aim of the study. This replication also uses the same pre-questionnaire as the original research, including the self-estimation item, but introduces specific redesigns to improve clarity, precision, and participant usability. Most notably, the self-estimation question now uses a five-point Likert scale to assess participants' perceived programming experience. This structured and intuitive format facilitates more accurate self-assessment and reduces cognitive load during the questionnaire [40].

Similarly, the post-questionnaire was modelled on the original version but refined to increase both specificity and participant engagement. Whereas the original study relied primarily on open-ended questions about task-solving strategies and the use of flowcharts, this replication used a mixed format approach, including five-point Likert-scale items, open-text responses, and pre-defined reasoning options. These extensions provide a more detailed insight into participants' subjective preferences and cognitive strategies, ultimately supporting a richer interpretation of the experimental results.

### 3.3 Independent Variable

Independent variables are the treatments or conditions intentionally manipulated to observe their effect on the dependent variables [112]. They are the primary factors under studied in experimental research, allowing researchers to explore cause and effect relationships and test specific hypotheses. In this study, the independent variable is the presence of a flowchart accompanying comprehension tasks. More specifically, the study examines whether the inclusion of a flowchart alongside the source code influences the program comprehension processes of novice programmers.

### 3.4 Dependent Variables

Dependent variables are the outcomes measured in an experiment to assess the effect of changes in the independent variable . They reflect the response or effect and are typically observed quantitatively or qualitatively to assess the success of the treatment or condition applied. In this study, the dependent variable is program comprehension, which is assessed using the same metrics defined in the original study. The dependent measures include:

- Visual attention: Measured by eye-tracking, specifically the fixation duration on AOIs.
- Correctness: Defined as the percentage of correct responses given during the comprehension tasks.
- Response time: The total time taken by participants to read and respond to each comprehension task.

- Mental workload: Assessed using EEG data and operationalized as the ratio of theta/alpha brain wave frequencies, a recognized indicator of cognitive load.
- Subjective preference: Based on participants' expressed preference for completing tasks with or without flowcharts.

## 3.5 Hypotheses

A hypothesis is a testable statement that predicts the relationship between one or more variables and serves as the basis for statistical analysis [112]. In this study, as in the original research, hypotheses are formulated to evaluate the effect of flowcharts on novice programmers' code comprehension. These hypotheses are based on the existing literature, which suggests that visual aids such as flowcharts can have a positive impact on program comprehension and reduce cognitive load. To investigate this, several performance indicators will be analyzed, including response time, correctness, visual attention, and cognitive load, to explore how the presence of flowcharts influences comprehension outcomes. The hypotheses used in this replication study were adopted directly from the original research to ensure methodological consistency and to effectively address the objectives of this study [24]:

***H1****: Participants refer to flowcharts in addition to code snippets.*
***Ho1****: Participants do not refer to flowcharts in addition to code snippets.*

***H2****: Participants using flowcharts take less time to complete the comprehension tasks.*
***Ho2****: There is no significant difference in response time of comprehension tasks due to the use of flowcharts.*

***H3****: Participants using flowcharts answer with a higher correction rate.*
***Ho3****: There is no significant difference in correctness of comprehension tasks due to the use of flowcharts.*

***H4:*** *Participants using flowcharts have a lower cognitive load during the comprehension tasks.*
***Ho4:*** *There is no significant difference in cognitive load during comprehension tasks due to the use of flowcharts.*

***H5:*** *Participants prefer flowcharts in addition to code snippets.*
***Ho5:*** *Participants do not prefer flowcharts in addition to code snippets.*

Hypotheses H1 to H4 will be assessed using quantitative data collected through eye tracking, EEG analysis and performance metrics. Hypothesis H5 will be tested qualitatively through a post-questionnaire interview focusing on subjective user preference and perceived helpfulness of flowcharts.

### 3.6 Participants

This replication study focuses on the comprehension process of novice programmers, but the sample of participants differs from that of the original study. Although there is no universally accepted definition of a novice programmer, several criteria such as years of programming experience, education level, project size, and familiarity with programming languages were considered [30]. The original study also relied on a combination of these factors to define novice participants. Accordingly, this study adopts a similar approach, collecting data on participants' academic and professional programming experience, familiarity with programming languages, and relevant demographic information to ensure comparability and validity in the replication process.

A prequestionnaire was used to assess participants' coding background, including self-estimated programming experience, to ensure consistency in participant classification. The questionnaire also assessed familiarity with key concepts such as languages and paradigms. For example, participants were asked to compare their skills with those of their course peers and with professionals with over 20 years' experience. Peitek et al. observed that while traditional measures (e.g., years of experience) do not consistently predict programming efficiency, subjective indicators such as self-estimation and motivation to learn show stronger correlations with performance [114].

To participate in this study, individuals were required to have received a minimal education in computer science domain, including basic knowledge of Java syntax, control structures, and arrays, as well as limited experience with fundamental programming tasks.

Participants were also required to be at least 18 years old and have at least one and a half hours available to complete the study.

Recruitment was conducted via announcements posted in the Opal forum and on social media groups associated with various modules in the Department of Computer Science at Chemnitz University of Technology. To ensure anonymity, no personally identifying information was linked to the collected data.

In the invitation email, available time slots were listed, and appointments were scheduled directly with the experiment supervisor based on students' availability. The prequestionnaire was hosted online on the SoSci platform, distributed via email, and responses were collected alongside participants' demographic information.

### 3.7 Confounding Factors

Confounding factors are extraneous, unwanted variables that can affect both the dependent and independent variables in an experiment [112]. These variables need to be controlled by using appropriate methods and techniques to minimize bias. To ensure the validity of the results, any observed effects should be attributed to the confounding variable and not to the independent variable.

- Program experience is widely recognized as an important confounding factor in software engineering experiments and is defined by researchers using various criteria such as years of experience, number of projects completed, self-assessment, pre-tests, educational level, and supervisor ratings [115]. In this experiment, participants are given a prequestionnaire that includes self-assessment questions, and only those who meet at least one definition of a novice as described in section 2.4 are allowed to proceed with the experiment.
- Prior experience with the material or tools may influence participants' performance, providing an advantage in task comprehension [116]. To control this, familiarity was assessed via a prequestionnaire. In addition, all participants were given a standardized introduction to the flowchart representation before the experiment began to ensure consistent understanding across the sample.
- Long or repetitive sessions can reduce attention and motivation, affecting cognitive performance in later tasks [116]. This experiment was therefore designed as a single, concise session to minimize fatigue.

- Concerns about being judged or evaluated may inhibit participants' performance, particularly if they believe the results may impact their academic standing [116]. Participants were assured of full anonymity and informed that their performance would not influence their course grades.
- Ensuring that all participants follow the experiment protocol precisely is vital to avoid procedural bias [116]. Participants were instructed not to use any external resources and to complete the tasks in one uninterrupted sitting. Completion times were recorded, and outliers caused by distraction were excluded from the analysis.
- Tasks left incomplete can introduce bias. In this study, incomplete responses were flagged and excluded during the analysis phase to maintain data integrity.
- The participant recruitment process was carefully designed to align with the study objectives, as described in the previous section [116].
- The sequence of tasks may influence performance if participants improve with practice [116]. To mitigate this, tasks were presented in randomized order. Additionally, a set of dummy tasks was administered at the beginning of the session to acclimate participants to the experimental environment
- Mono-method bias occurs when only a single measure is used to quantify a variable [116]. To counter this, the study employed multiple assessment measures, including eye tracking, EEG, and subjective questionnaires.
- Inter-individual differences, such as variations in cognitive abilities and personal characteristics can influence performance outcomes [116]. A within-subjects design was employed to control such variability, allowing each participant to serve as their own control.
- Technical problems, such as computer malfunctions or missing participant questionnaires, can occur during experiments [116]. These problems can affect the results as participants may have to repeat tasks or data may be lost. To avoid bias, data from affected participants are excluded from the analysis.
- Intelligence lacks a clear definition and includes skills such as problem solving and memorization [116]. To minimize its impact, task complexity was intentionally kept low to avoid requiring advanced cognitive skills.

- Motivation significantly influences the effort invested in a task and, consequently, performance. Higher individual interest generally improves engagement and learning outcomes [117]. In this study, all participants volunteered after receiving a brief explanation of the study's relevance, fostering intrinsic motivation.

## 3.8 Experiment Material

The materials used in this replication study include code snippets, a pre-questionnaire, and a post-questionnaire. These materials closely follow the structure of the original study, with minor modifications to the questionnaire design for improved clarity. Each component is described in detail in the following sections.

### *3.8.1 Code Snippet Selection*

The selection of code snippets plays a crucial role in shaping participant performance and engagement throughout the study. To ensure methodological consistency and comparability with the original study, most of the original snippets were retained. However, the following three snippets were replaced: MultiplicationByAdding, SumOfIntervalNumbers, and DiffPosAndNegNumbers. The reasons for these replacements are provided in section 3.2.

To guide this replacement, a rigorous selection process was used to identify BinarySearch, BubbleSort and Power as suitable replacements. These new snippets were selected based on the following criteria derived from the original study:

- Avoidance of domain-specific knowledge: As participants are novice programmers, snippets requiring specialized knowledge (e.g., advanced data structures or classes) were excluded. Such complexity could reduce engagement, increase response times, and limit the collection of meaningful data [119].
- Sufficient complexity to require dual-modality comprehension: The selected snippets should be sufficiently complex to encourage participants to consult both the source code and the associated flowcharts. Simpler snippets may allow participants to rely solely on the code, limiting insights into the use of flowcharts.

- Encourage bottom-up comprehension strategies: Snippets should avoid recognizable patterns or "beacons" that allow participants to immediately infer functionality without detailed analysis. This ensures a more authentic measurement of comprehension effort and avoids biased results [32].
- Inclusion of core programming constructs: All snippets include basic concepts commonly taught in introductory programming courses, such as control structures, loops and conditionals. Advanced concepts have been deliberately avoided to maintain accessibility for beginners.
- Prior validation through pilot studies: The snippets were tested in preliminary studies to ensure that they were suitable for assessing program comprehension in novice participants.

### *3.8.2 Pre-questionnaire*

Pre-questionnaires are a fundamental tool in software engineering research to assess participants' experience and background, as emphasized in previous studies [112]. This prequestionnaire follows the same approach as the original study, but with slight modifications to the design. This questionnaire uses a five-point Likert scale to assess participants' perceived programming experience. This scale provides a more intuitive and structured approach to self-assessment. According to Likert (1932), reducing the number of response categories can more effectively capture attitudes and self-perceptions, especially when respondents are not domain experts [40]. Each level on the scale is clearly labelled 'Beginner', 'Novice', 'Competent', 'Proficient', and 'Expert' to help participants quickly identify where they fit without over-analyzing. This simplification minimizes cognitive load and encourages more accurate and reliable responses. In the original study, participants were asked a combined question such as: "How do you estimate your programming experience compared to fellow students and experts with 20 years of practical experience on a scale from 1 to 5?" [24]. This question design may have caused confusion as it involved two different comparison groups, colleagues and experienced professionals within a single item. To address this issue, in this replication study the question was redesigned and split into two separate items: one comparing participants' experience to that of their course peers, and another comparing it to that of experienced professionals.

This modification is based on the understanding that novice programmers may assess their skills differently depending on the reference group. The mean concept of the prequestionnaire was adopted exactly as it appeared in the original study. However, modifications were made to the design layout of the questionnaire to improve clarity and usability. The redesigned pre-questionnaire in this study is as follows:

- How long have you been programming for educational purposes? (In years)
- How long have you been programming professionally? (In years)
- How do you estimate your programming experience on a scale from 1 to 5?

| Beginner | Novice | competent | proficient | Expert |
|---|---|---|---|---|
| O | O | O | O | O |

- How do you estimate your programming experience compared to fellow students?

| Significantly less experience than most students. | Less experience than most students. | About the same level of experience as most students. | More experience than most students. | Significantly more experience than most students. |
|---|---|---|---|---|
| O | O | O | O | O |

- How do you estimate your programming experience compared to experts with 20 years of practical experience?

| I have significantly less experience | I have less experience | I have comparable experience | I have more experience | I have significantly more experience. |
|---|---|---|---|---|
| O | O | O | O | O |

- How do you estimate your experience using flowcharts to visualize or design a program on a scale from 1 to 5?

| Not at all experienced | A little experienced | Moderately experienced | Mostly experienced | Fully experienced |
|---|---|---|---|---|
| O | O | O | O | O |

- How experienced are you with the following programming languages on a scale from 1 to 5?

| Programming Language | No Experience | Basic Understanding | Competent | Experienced | Expert |
|---|---|---|---|---|---|
| Java | O | O | O | O | O |
| C | O | O | O | O | O |
| Python | O | O | O | O | O |
| JavaScript | O | O | O | O | O |

- How many additional programming languages do you have moderate experience with?
- How experienced are you with the following programming paradigms on a scale from 1 to 5?

| | No Experience | Basic Understanding | Capable | Experienced | Expert |
|---|---|---|---|---|---|
| Functional programming | O | O | O | O | O |
| Object-oriented programming | O | O | O | O | O |
| Imperative programming | O | O | O | O | O |
| Logical programming | O | O | O | O | O |

### *3.8.3 Post-questionnaire*

The post-questionnaire plays an important role in this study, providing insight into participants' strategies and preferences during the comprehension tasks [112]. It is designed to determine how subjects engaged with the flowcharts, whether they found them helpful, and what approaches they used to complete the tasks. This post-questionnaire closely follows the structure of the original post-questionnaire but introduces targeted enhancements to increase specificity and participant engagement.

Whereas the original study relied primarily on open-ended questions about task-solving strategies and flowchart use, this study expands the format to include five-point Likert scale items, open-ended questions, and pre-defined reasoning options. These additions allow for a more nuanced analysis of participants' subjective experiences and cognitive processes. At the end of the experiment, participants were shown the tasks they had completed on the first screen, along with the corresponding flowcharts. This allowed them to review and reflect on the material.

The post-questionnaire was then presented on a second screen. If necessary, participants could return to the first screen at any time to refresh their memory before

answering the questions and were encouraged to review these tasks during the interview to identify which flowcharts they found helpful or unhelpful. This interactive, visual method was designed to improve recall and response accuracy, aligning with research showing that visual aids can improve memory retrieval during interviews particularly in task-based assessments [120].

To assess participants' preferences and strategies. The following questions were included in the post-questionnaire:

- Please indicate which tasks you found the use of flowcharts helpful and which you did not, giving a reason for your answer, e.g. on the first screen you can navigate through your completed tasks:

  - ☐ Helped me understand the task.
  - ☐ Avoid ambiguity in the task.
  - ☐ Task was easy.
  - ☐ It is a new strategy for me, and I am not comfortable using it.
  - ☐ I used to program with code.
  - ☐ I am a visual or verbal learner.
  - ☐ Others, provide your own reason.

- Could you describe the approach or strategy you used to solve the tasks?
- How much did you refer to the flowcharts and how much to the code?
- For which task did you spend more time, when there was a flowchart present or not?
- Did the presence of flowcharts in the task impact your understanding of the source code on a scale from 1 to 5?

| Not helpful | Slightly helpful | helpful | very helpful | Extremely helpful |
|---|---|---|---|---|
| O | O | O | O | O |

- If flowcharts were not provided in the tasks, would you have preferred their inclusion to aid task-solving?

    - ☐ Strongly agree – I believe that flowcharts would greatly improve task comprehension and make problem-solving alongside code more efficient.
    - ☐ Somewhat agree – I think flowcharts could offer some help in solving tasks, but they are not essential for understanding.
    - ☐ Neutral – I am indifferent about the inclusion of flowcharts and do not feel they would significantly affect my approach to task-solving.
    - ☐ Somewhat disagree – I find flowcharts somewhat unnecessary and believe they might complicate the task-solving process.
    - ☐ Strongly disagree – I prefer to approach tasks without flowcharts, as I find them unhelpful in understanding or solving problems.

- Did you use flowcharts as a learning tool in your introductory programming classes?

    - ☐ Yes, in all introductory classes.
    - ☐ No, they were not used.
    - ☐ Yes, but only in one specific course.
    - ☐ Yes, in several but not all introductory classes.

### 3.9 Tasks

Program comprehension studies ask participants to complete specific programming tasks. These tasks are designed so that successful completion requires a clear understanding of the code. By measuring the effort and accuracy required to complete these tasks, researchers can gain insight into the difficulty of understanding the code [119]. The program comprehension and software engineering literature use a variety of tasks to assess how well participants understand code, algorithms and software processes. Common tasks include debugging, tracing, predicting output, and modifying code [121], [122].

Variable name obfuscation enforced a bottom-up processing approach, requiring participants to understand the code from basic principles rather than relying on contextual cues and prior knowledge. The comprehension task can be outlined as follows:

- Each comprehension task consisted of two slides. On the first slide, participants are presented with either a standalone code snippet or one accompanied by a flowchart and are instructed to determine the task's output. Each comprehension task includes the input, and all necessary details required to reach the solution, as illustrated in Figure 3.1.

```java
public static String function1() {
    int[] arr = { 3, 0, 1, 0 };
    String res = "";
    int var = 3;

    for (int i = 0; i < arr.length; i++) {
        int value = arr[i];
        if (value == var) {
            res += "x";
        } else if (value < var) {
            res += "m";
        }
        res += "o";
        var--;
    }
    return res;
}
```

**Figure 3.1 The comprehension task.**

- On the second slide, participants are presented with four possible answer choices and are required to select the correct one. If they are uncertain, they have the option to skip the question, as illustrated in Figure 3.2. This feature helps ensure the results remain more accurate and freer from guesswork-related bias.


**Figure 3.2 Answer Options for the Comprehension Task.**

As mentioned above, a flowchart can be presented alongside the corresponding code snippet. To ensure that the flowchart does not introduce unintentional bias, its design must adhere to consistent and neutral standards. Therefore, the following aesthetic criteria have been established to guide the construction of flowcharts:

- Only the universal standard symbols for flowcharts were used to ensure clarity and consistency across all visual aids.
- Program logic was presented in a clear, unidirectional flow from top to bottom and from left to right.
- Each symbol in the flowchart had a single-entry point (start) and exit point (end), except for the decision symbol, also only standers symbols were used. These are illustrated in Table 3.1.
- Syntax highlighting is retained in both the source code and the flowcharts, as it is commonly used in major IDEs. Removing it from the snippet code could change the behavior of the participants.
- The textual content within the flowcharts exactly mirrored the accompanying code snippets, including all syntax elements such as type notations (int), semicolons (;) and method calls (System.print.out).
- The borders of the flowchart shapes were color-matched to the syntax highlighting of the source code, maintaining a visual balance that facilitates comparative analysis.
- Font size and line spacing were carefully chosen to optimize readability and facilitate the analysis of visual attention data, ensuring that participants could comfortably engage with both the code snippets and flowcharts.
- Following established aesthetic and functional criteria ensures that the three replacements mentioned in the snippet selection section of this replication study are unbiased and fully consistent with the methodology of the original study.

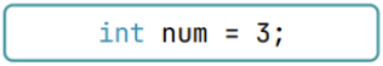
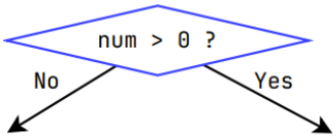
| Shape | Meaning |
|---|---|
|  | **Oval:** Indicates the start or end of a flowchart. |
|  | **Represents** processing steps, such as calculations or data manipulation. |
|  | **Rhombus:** Used for decision points, where the flow branches based on different conditions or choices. |

**Table 3.1 Standard Symbols of flowcharts [24].**

Figure 3.3 illustrates the code snippet and flowchart design used in this study. This design was developed based on the criteria. The design aims to minimize potential bias and facilitate the collection of meaningful visual attention data.

```
public static int[] function1() {
    int[] arr1 = { 3, 4 };
    int[] arr2 = { 5, 2 };
    int length = arr1.length;
    int[] res = new int[length * 2];

    for (int i = 0; i < length; i++) {
        int var1 = arr1[i];
        int var2 = arr2[i];

        res[i] = var1;
        res[i + length] = var2;
    }

    return res;
}
```

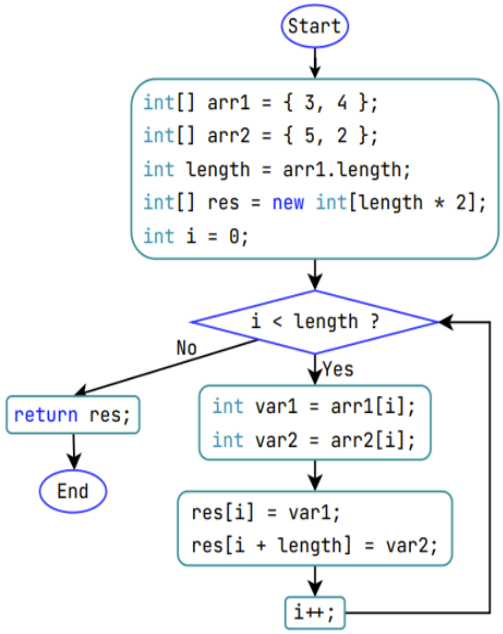



**Figure 3.3 Comprehension task Code snippet with Flowchart.**

## 3.10 Experiment Design

This study employed a within-subjects design, replicating the original study's approach with minor modifications to align with the current research objectives. These modifications are detailed in subsequent sections. Participants whose data were included in the final analysis completed two types of comprehension task. One task involved code snippets presented alongside flowcharts, while the other involved code snippets alone. The key elements of the experimental design are summarized as follows:

- The experiment consisted of 14 comprehension tasks in total. Seven of these were presented with code snippets only, while the other seven were presented with code snippets accompanied by flowcharts.
- Participants were given a total of 30 minutes to complete the tasks, followed by a 5-minute post-task interview.
- To mitigate the effects of order and learning, the order in which the code snippets were selected in Section 3.8.1 was randomized for each participant. Tasks alternated between two conditions: a code snippet on its own, or a code snippet presented alongside a flowchart. Crucially, no participant encountered the same algorithm in both conditions. For instance, if a participant encountered the BinarySearch algorithm in the code-only condition, it would not appear again in the code alongside the flowchart condition. This ensured that familiarity with an algorithm from a previous task would not influence performance.
- After each task, participants were given a five-second rest period with a cross-fixation stimulus. This was an amendment to the original study, which employed a 10-second interval, in order to reduce both the overall session length and participant fatigue.
- Before beginning the comprehension tasks, participants were given instructions on how to interpret the flowcharts and were guided through a practice task. This was done to minimize learning effects during the actual experiment and ensure consistency across participants.

## 3.11 Tools

To ensure consistency and enhance comparability with the original study, the same tools were used throughout the experimental process. The SoSci Survey platform was also used to administer the pre- and post-questionnaires, improving the efficiency of data collection by providing a streamlined, structured digital process. The following sections contribute detailed descriptions of each tool used in the experiment.

### *3.11.1 Eye Tracker*

Eye tracking enables visual attention to be measured by recording where participants focus their gaze. This study used the same Tobii Pro X3-120 EPU eye tracker as the original study [24]. The device projects invisible infrared light into the eyes and uses high-resolution cameras to record corneal reflections. Advanced algorithms then compute gaze direction and subtle eye movements. The Tobii Pro X3-120 captures data at a rate of 120 samples per second, providing detailed insights into participants' visual behavior [123]. The tracker was positioned beneath the stimulus screen, with a screen resolution of 1920×1080 pixels and physical dimensions of 51.1×28.7 cm. Participants were seated approximately 60 cm from the screen and 65 cm from the eye tracker as shown in Figure 3.4. Proper alignment was verified using Tobii's position guide, which displays facial contours and indicates correct eye detection by turning the background green [124]. Communication with the device was managed using the Tobii SDK and controlled via Python on a Windows 10 system.

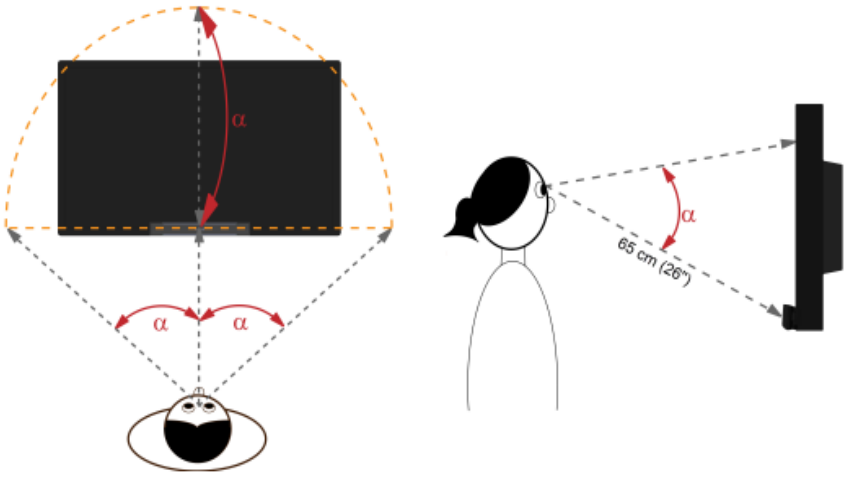



**Figure 3.4 Positioning the participant and the eye tracker [125].**

### *3.11.2 EEG*

As in the original study, EEG data were recorded using the CGX Quick-20r wireless headset. This system features dry sensors and adheres to the 10–20 international electrode placement system. The Quick-20r is equipped with interchangeable dry sensors that comfortably conform to different scalp shapes and sizes.
Prior to recording, the system provides visual feedback to indicate correct electrode placement through green status lights, as illustrated in Figure 3.5. This ensures accurate and reproducible measurements [125].


**Figure 3.5 CGX EEG Headset [126].**

### *3.11.3 PsychoPy*

The experimental stimuli were presented using PsychoPy, which is an open-source psychophysics software written in Python. PsychoPy supports precise visual rendering via OpenGL and was used to display code snippets and flowcharts in full-screen mode, as in the original study. Participants navigated the options using the arrow keys and confirmed their responses using the spacebar. Minor adjustments were made for this replication, such as library updates and the integration of marker signals for synchronizing EEG and eye-tracking data [127].

### *3.11.4 SoSci Survey*

The SoSci Survey platform was used to administer the pre- and post-questionnaires. This web-based tool allows participants to complete surveys without having to install any software. SoSci Survey facilitated the collection of structured data in alignment with the design of the experiment [128].

## 3.12 Ethical Considerations and Academic Integrity

All textual references and paraphrased ideas in this thesis are derived from published works and are properly cited in accordance with academic standards. Any closely aligned wording was used for the purpose of accuracy and clarity, in line with the intent of replication and literature synthesis.

# 4 Conduct

This chapter provides a detailed outline of the implementation process of the present study, covering every step involved. It also describes the participants and the data collection procedures.

## 4.1 Participants Demographics

A total of 11 students participated in the study, each completing the test within 30 minutes or less. Subjects reported between 2 to 5 years of academic programming experience. Only five participants had professional programming experience, with a maximum of 1.5 years, while the rest reported no such experience. Subjects were also asked to assess their programming skills in comparison to their classmates and to experts with 20 years of professional experience. Most participants rated themselves as having intermediate acquaintance relative to their peers but considered their expertise insufficient when compared to seasoned professionals. Additionally, they evaluated their familiarity with Java programming as moderate. Regarding experience with flowcharts, more than half of the participants 7 out of 11 rated themselves as “a little experienced,” indicating limited prior exposure. One participant reported no experience at all, while three participants identified as “moderately experienced”. Table 4.1 presents a detailed overview of the participants’ demographic and background characteristics.

| | |
|---|---|
| **Male** | 6 |
| **Female** | 5 |
| **Age (in years)** | 27.6 ± 4.4 |
| **Learning Programming (in years)** | 3.9 ± 1.4 |
| **Professional Programming (in years)** | 0.8 ± 1.2 |
| **Flowchart Experience** | 2.2 ± 0.6 |

**Table 4.1 Participants Demographics.**

## 4.2 Procedure

This section provides a detailed outline of the experimental procedure. Figure 4.3 Overview of the Experimental Procedure. presents an overview of the experimental workflow, covering each step from participant recruitment to the post-interview phase as follows:

1. An invitation message is posted in the OPAL platform and in social media groups for various modules offered by the Department of Computer Science at Chemnitz University of Technology. The message clarifies that participation is voluntary and will not impact course grades.
2. The invitation includes information about the importance of the participant's involvement in supporting scientific research and their specific role in the study. It provides contact email for questions, offers time slots based on the participant's availability, confirms that the study will take approximately one hour, and states that it will take place in the university's campus laboratory.
3. Once participants select a convenient time slot, they receive an email containing a brief description of the experiment along with general instructions. This email also includes a link to the pre-questionnaire.
4. In the next step, participants complete the pre-questionnaire, which gathers demographic details and asks them to self-assess their programming experience.
5. On arrival at the laboratory, the students are given the following instructions:

- All collected data was anonymized to ensure participant privacy.
- All necessary instructions for the tasks will be displayed on the screen before the experiment begins.
- Participants are encouraged to attempt all tasks, but any task can be skipped if the participant is unsure of the answer. This flexibility is provided to ensure unbiased results.
- A sample Java task and a brief guide explaining the flowchart symbols will be provided to familiarize participants with the procedure before the main experiment begins.

- The visual gaze data will be recorded using an eye tracker. A calibration process is required to configure the device accurately before beginning the experiment.
- An EEG device will be used to record brain activity during the experiment. A calibration process for the EEG device is also required.
- In order to obtain high quality data, it is important that participants remain as relaxed as possible and minimize any body movements throughout the experiment.

6. Participants will review the data protection form, which provides a detailed description of the entire data collection process. Any questions the participants may have will be answered at this stage. If participants agree to the terms and conditions and sign the form, the experiment can proceed.
7. The EEG machine is then calibrated. Each electrode in the EEG cap must be individually adjusted according to the participant's hair density and head size. The duration of this process varies depending on these factors, but it takes approximately 10 to 15 minutes. The CGIX software is used to display the impedance levels for each electrode. When an electrode's impedance is at an acceptable level, it is indicated by a green light as illustrated in Figure 4.1. All electrodes must reach acceptable impedance levels to complete this step.

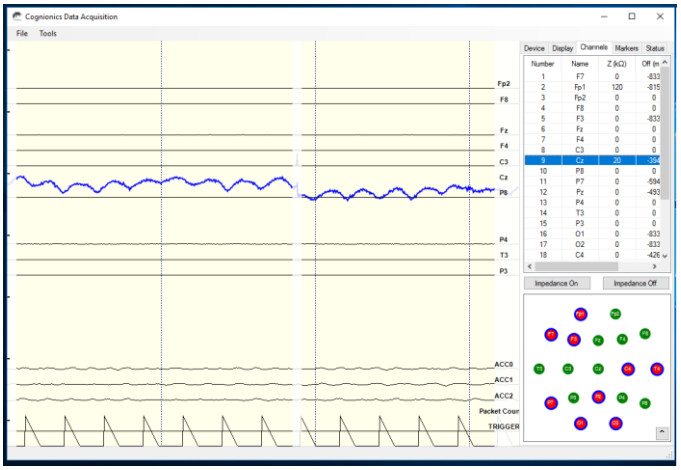



**Figure 4.1 EEG-Signals and Calibration [24].**

8. The eye tracker is then calibrated. The participant sits approximately 60 cm away from the screen, and the monitor height is adjusted for optimal positioning. The Eye-tracker Manager software shows a green screen, as illustrated in Figure 4.2, when the participant is properly aligned. During calibration, the participant is asked to focus on 12 specific points displayed on the screen. Once completed, the system calculates and visualizes the calibration accuracy. If the accuracy is deemed inadequate, the procedure can be repeated. once the participant is correctly positioned

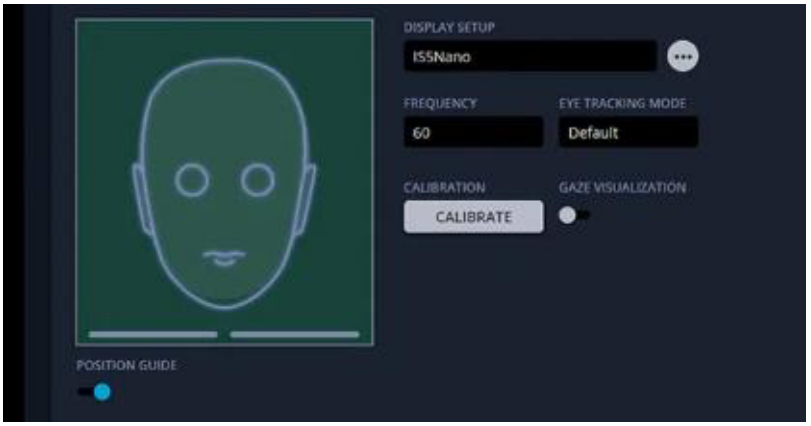



**Figure 4.2 Eye Tracker Calibration [124].**

9. Once the calibration is complete, the participant is ready to begin. The study presentation is displayed on the screen, with instructions and general information.
10. The participant is first presented with training tasks. The introduction phase includes a brief overview of flowcharts, followed by two practice tasks designed to familiarize participants with the experimental setup. The first practice task features only a code snippet, while the second pairs a code snippet with a flowchart.
11. Next, the participant proceeds to the main comprehension tasks. A fixation cross is displayed for five seconds between each task to mark transitions.
12. The experiment concludes either once all 14 tasks are completed or when 30 minutes have passed—whichever occurs first. At that point, the EEG cap is removed, and the eye-tracking system is deactivated.
13. A post-questionnaire interview is conducted immediately after the experiment, this process gathers insights into the participant's experience and preferences, particularly regarding the use of flowcharts.

14. The participant is warmly thanked for their valuable contribution before they leave the lab.

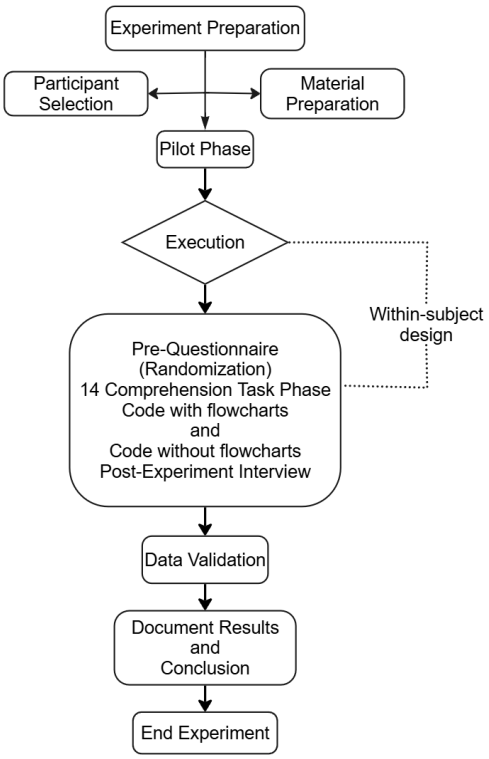



**Figure 4.3 Overview of the Experimental Procedure.**

## 4.3 Data Collection

Once the study is complete, the collected data will be available in the following formats:

1. Participants' responses: Answers and response times are saved in CSV files, automatically generated by PsychoPy at the end of the study.
2. Eye-tracking data: The horizontal (X) and vertical (Y) gaze coordinates for both eyes are recorded and stored in CSV files, generated using the Tobii Eye Tracker SDK for Python.
3. EEG data: Brain activity is captured from 19 channels and saved in FIF forma; this data is recorded in real time using CGX software.
4. Survey data: Responses to the post-questionnaire are collected in CSV files using SoSci Survey, and the recorded responses are saved in M4A and MP3 formats.

# 5 Data Analysis and Results

The primary objective of this replication study is to examine whether using flowcharts improves the code comprehension among novice programmers. Several evaluation metrics were employed to assess this: visual attention, correctness, response time, cognitive load and subjective preference. These metrics were selected to align with those used in the original study, thereby ensuring comparability and consistency. Data for each metric was collected using a controlled experimental design with a within-subjects approach. This section outlines the procedures followed for data pre-processing, characterization and analysis, with the aim of answering the research question and validating the proposed hypotheses.

## 5.1 Data Preparation

This study aims to determine whether the analyzed data supports the hypothesis that flowcharts have a positive impact on the code comprehension of novice programmers, in line with the original study's findings. To ensure methodological alignment and comparable results, this replication employed the same performance measures as the original study: visual attention, cognitive load, response time, correctness, and subjective preference. Data for these measures were collected using a controlled experimental design with a within-subjects approach. This section outlines the steps taken to pre-process and analyze the data in order to evaluate the proposed hypotheses and answer the central research question.

### *5.1.1 Time and Correctness Data Processing*

This phase involves cleaning, processing and transforming raw data into a structured format that is suitable for analysis. This includes removing irrelevant entries, addressing missing values, identifying and mitigating outliers, and minimizing potential sources of bias, in order to ensure the integrity and reliability of the dataset.

The response time and correctness data were recorded in CSV files, with columns for the participant ID, the code snippet, the response time and the correctness. The 'Correctness' column includes three values: 'Yes' for correct answers, 'No' for incorrect ones and 'Skip' for unanswered comprehension tasks.

Response time was measured using PsychoPy, with data collected across different task routines. The first routine measures the time that a participant spends analyzing the code before pressing the spacebar to proceed. The second routine is triggered when the spacebar is pressed, displaying the four multiple-choice options. For the purposes of this analysis, only the response time from the first routine was considered, as this reflects the participant's initial comprehension efforts. The timing data and labels indicating whether the answer was correct were manually extracted from the raw CSV logs and compiled for statistical evaluation.

### *5.1.2 Eye Tracking Data Processing*

The eye tracker data were extracted from CSV files, filtered, and time normalized. Unnecessary columns were removed, and the gaze point coordinates were mapped to the display coordinates according to the predefined screen resolution settings. The column names were manually adjusted to match the format required by the I2MC fixation classification algorithm. The I2MC was then applied to detect fixation events, and the results were saved for further analysis [126]. The fixation data were then integrated with AOIs, allowing AOI-based analysis by determining which fixations occurred within the predefined regions of the screen.

Finally, key eye-tracking metrics were calculated, including the number of fixations, total fixation duration within flowcharts and code, fixations within specific AOIs, and transitions between AOIs. Python scripts were used to process and extract data from the CSV files.

### *5.1.3 EEG Data Processing*

The raw EEG data file contains recordings of electrical brain activity from various regions. First, any unnecessary channels were removed to ensure that only the relevant signals were retained. The standard 10–20 montage was then applied to ensure consistency [129]. Next, a bandpass filter (0.5–40 Hz) was applied to eliminate unwanted signals outside the desired frequency range. Further cleaning of the data was achieved by applying ICA algorithm to identify and remove artefacts such as eye movements and muscle activity that could interfere with the analysis [130], [131].

The iClabel algorithm was then used to remove components unrelated to brain activity, resulting in a more precise signal for subsequent analysis. Finally, the theta-to-alpha ratio (TAR) was calculated to assess brain workload [69].

### *5.1.4 Data Cleaning*

Data cleaning is applied to the results of previous steps to ensure accuracy. Any observations marked as 'skipped' in the 'Accuracy' column are removed. Additionally, comprehension tasks with response times of less than 10 seconds are excluded. The final cleaned data are presented in the following sections.

## 5.2 Descriptive Statistics

Descriptive statistics are used in this section to effectively present the data collected. Various visualization techniques such as tables, boxplots, histograms and percentage calculations are used to highlight key findings. These methods help to summarize the most relevant aspects of the results and support the evaluation of the hypotheses.

### *5.2.1 Eye Tracking Data Results*

The eye-tracking data was examined to assess whether participants directed their attention to the flowcharts during the comprehension tasks. In this study, each participant completed tasks that involved either code snippets alone or code accompanied by flowcharts. For each algorithm, the total fixation time on both the code region and the flowchart area was calculated to evaluate visual attention patterns. Table 5.1 presents the distribution of fixation times recorded during the comprehension tasks. On average, participants spent 34.11 seconds fixating on code snippets when no flowcharts were provided. In contrast, when flowcharts accompanied the code, the average fixation time was 27.04 seconds for the code and 10.08 seconds for the flowcharts. The fixation time over code snippets is reduced by 7.07 seconds (20.73%) when flowcharts are present. The total fixation time over the stimulus increases by 3.01 seconds (8.82%) when flowcharts are present.

A minimum threshold of 10% of the total fixation time is used to determine whether a flowchart was used in each observation. A flowchart was considered to be used if a participant’s fixation time on it exceeded the defined threshold. Table 5.2 summarizes

the number of comprehension tasks in which flowcharts were actively utilized, along with the corresponding fixation time distribution for each participant.

| | *Code Alone* | | *Code in addition to Flowcharts* | | | |
|---|---|---|---|---|---|---|
| Algorithm | Total Responses | Mean Fixation Total (s) | Total Responses | Mean Fixation Time Code (s) | Mean Fixation Time Flowchart (s) | Mean Fixation Total (s) |
| binarySearch | 2 | 19.1 | 5 | 55.36 | 0.87 | 56.23 |
| bubbleSort | 6 | 4.63 | 3 | 13.13 | 8.49 | 21.61 |
| concatlists | 4 | 22.31 | 4 | 35.86 | 14.33 | 50.19 |
| countEvenNumbers | 4 | 15.5 | 5 | 21.73 | 3.02 | 24.76 |
| crosssum | 6 | 47.23 | 2 | 7.72 | 0 | 7.72 |
| decimalToBinary | 5 | 44.46 | 5 | 27.59 | 19.1 | 42.87 |
| dropNumber | 5 | 46.27 | 4 | 70.38 | 1.13 | 71.51 |
| findTheLargest | 1 | 19.23 | 9 | 13.07 | 10.58 | 22.47 |
| firstAboveThreshold | 5 | 30.25 | 5 | 25.19 | 23.59 | 48.78 |
| hindex | 5 | 30.92 | 6 | 32.83 | 21.79 | 49.15 |
| integertoString | 4 | 86.76 | 4 | 47.89 | 12.85 | 57.54 |
| isPrime | 7 | 27.2 | 4 | 23.91 | 5.75 | 29.67 |
| power | 5 | 8.29 | 5 | 9.74 | 6.67 | 15.08 |
| removeDoubleCharacters | 4 | 72.66 | 5 | 2.14 | 0 | 2.14 |
| TOTAL | 63 | 34.11 | 66 | 27.04 | 10.08 | 37,12 |

**Table 5.1 Fixation Time Results**

| | *Code Alone* | | *Code with Flowcharts* | | | |
|---|---|---|---|---|---|---|
| Participant | Total Responses | Mean Fixation Time (s) | Total Responses | No. of tasks were Flowcharts where used | Mean Fixation Time Code (s) | Mean Fixation Time Flowchart (s) |
| Participant 1 | 7 | 29.31 | 6 | 2 | 33.51 | 16.53 |
| Participant 2 | 4 | 41.96 | 5 | 4 | 26.54 | 23.94 |
| Participant 3 | 5 | 46.6 | 6 | 5 | 1.18 | 5.6 |
| Participant 4 | 7 | 29.98 | 6 | 2 | 31.44 | 2.92 |
| Participant 5 | 6 | 15.46 | 7 | 5 | 23.75 | 10.98 |
| Participant 6 | 5 | 13.42 | 6 | 1 | 23.2 | 3.2 |
| Participant 7 | 7 | 13.99 | 7 | 0 | 41.6 | 0.48 |
| Participant 8 | 5 | 42.13 | 5 | 3 | 21.26 | 18.86 |
| Participant 9 | 7 | 34.39 | 7 | 2 | 23.97 | 2.95 |
| Participant 10 | 5 | 90.89 | 6 | 0 | 51.64 | 2.72 |
| Participant 11 | 5 | 33.88 | 5 | 4 | 11.89 | 31.77 |

**Table 5.2 Distribution of Fixation Time across Participants.**

The table shows that two participants used flowcharts extensively (Subjects 3 and 5). Out of the total participants, five made use of flowcharts in five different comprehension tasks. Three others referred to flowcharts in either four or three tasks.

Four participants used them only once or twice, while two did not refer to the flowcharts at all. In total, flowcharts were utilized in 28 out of 66 comprehension tasks.

### *5.2.2 Response Time, Correctness and Cognitive Load Results*

Table 5.3 presents the results related to correctness, response time, and cognitive load for each algorithm under two different task conditions: using code alone and using code accompanied by a flowchart. For the last condition, the analysis includes only the 28 instances in which participants actively engaged with the flowchart. It is also important to note that the number of responses per algorithm varies, as participants were not necessarily exposed to the same set of algorithms.

| | *Code Alone* | | | | *Code with Flowcharts* | | | |
|---|---|---|---|---|---|---|---|---|
| Algorithm | Total Responses | Correctness in % | Mean Response Time (s) | Cog. Load | Total Responses | Correctness in % | Mean Response Time (s) | Cog. Load |
| binarySearch | 2 | 1 | 130.35 | 3.11 | 0 | 0 | 0 | |
| bubbleSort | 6 | 0.67 | 82.55 | 3.71 | 2 | 1 | 71.06 | 3.29 |
| concatlists | 4 | 0.75 | 34.47 | 3.53 | 2 | 0.5 | 84.35 | 3.13 |
| countEvenNumbers | 4 | 0.75 | 23.86 | 2.2 | 2 | 0.5 | 39.07 | 1.75 |
| crosssum | 6 | 0.5 | 57.73 | 3.7 | 0 | 0 | 0 | |
| decimalToBinary | 5 | 0.8 | 62.57 | 3.37 | 2 | 0 | 86.96 | 3.15 |
| dropNumber | 5 | 0.8 | 87.05 | 4.08 | 0 | 0 | 0 | |
| findTheLargest | 1 | 1 | 39.17 | | 5 | 1 | 37.32 | 1.4 |
| firstAboveThreshold | 5 | 1 | 53.68 | 2.17 | 4 | 0.75 | 89.42 | 3.62 |
| hindex | 5 | 0.8 | 98.22 | 2.57 | 4 | 0.25 | 95.68 | 1.1 |
| integertoString | 4 | 0.5 | 107.66 | 4.14 | 1 | 1 | 97.04 | 3.24 |
| isPrime | 7 | 0.71 | 32.57 | 3.21 | 4 | 1 | 48.25 | 4.3 |
| power | 5 | 0.6 | 44.91 | 3.08 | 2 | 1 | 79.01 | 3.96 |
| removeDoubleCharacters | 4 | 0.5 | 104.59 | 2.68 | 0 | 0 | 0 | |
| MEAN_TOTAL | 63 | 0.71 | 66.41 | 3.3 | 28 | 0.71 | 69.21 | 2.94 |

**Table 5.3 Correctness, Response Time and Cognitive Load Results**

Participants completed comprehension tasks with code alone in 10.00 - 154.74 s (mean=66.41s) and comprehension tasks with flowcharts in 13.99 - 161.22 (mean=69.21s). This represents an increase of 2.8 s (4.22%) in the mean response time when flowcharts were present. Furthermore, the median for tasks with code alone is approximately between 45 and 50 seconds and for tasks with flowcharts is approximately between 65 and 70 seconds. The results for the range of response times are shown in Figure 5.1 below.

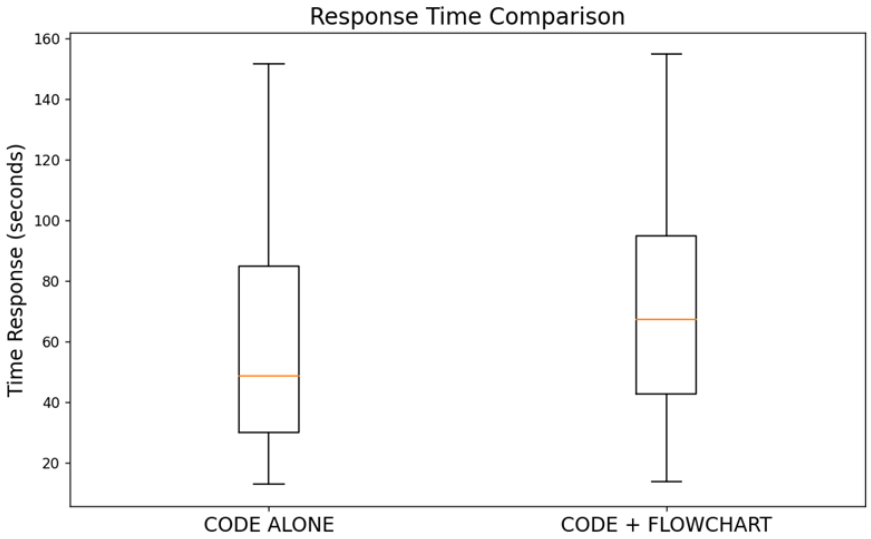



**Figure 5.1 Time Response Distributions.**

The accuracy rate for the comprehension tasks was 0.71 (71%) in the code alone condition and 0.71 (71%) when flowcharts were used. However, it is important to note that the latter calculation is based on fewer observations, as flowcharts were not used in every comprehension task where they were available. As illustrated in Figure 5.2 participants' correctness varied significantly across algorithm types. The results show that the correction rate for comprehension tasks was higher in 4 out of 14 algorithms when flowcharts were used, lower in 9 algorithms, and equal in 1 algorithm.

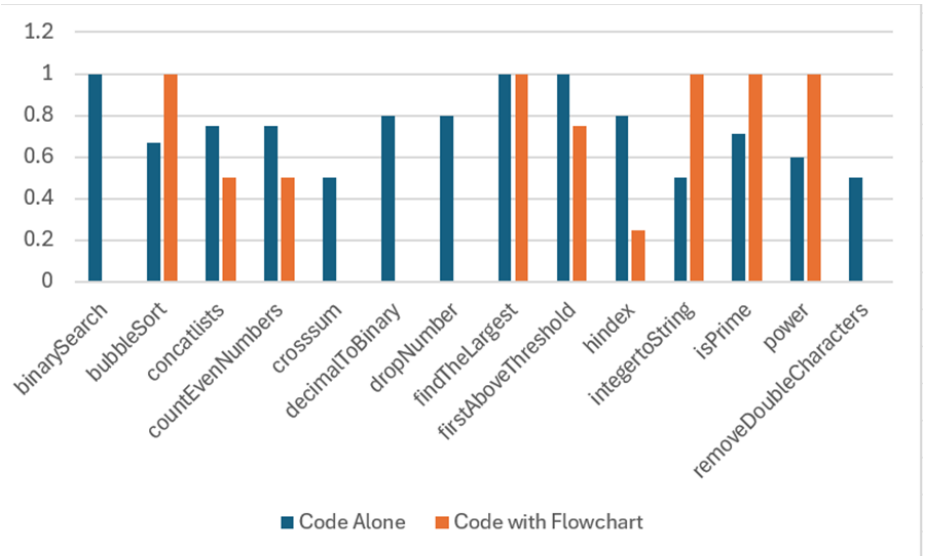



**Figure 5.2 Task Accuracy Across Different Algorithms.**

The cognitive load results are presented in Figure 5.3, using boxplots to show the distribution of cognitive load values. The boxplots compare the cognitive load during comprehension tasks involving code snippets alone with those involving flowcharts.

Cognitive load was measured using EEG signal analysis, specifically the ratio of the relative power of the theta and alpha bands. The mean cognitive load for comprehension tasks with code snippets alone is 3.3 and with code snippets in addition to flowcharts is 2.94.

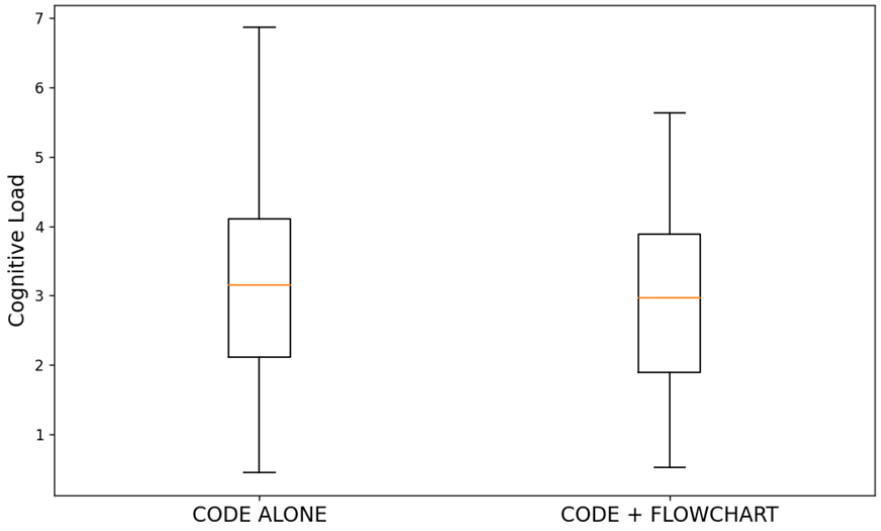



**Figure 5.3 Cognitive Load Distributions.**

### *5.2.3 Post Interview Results*

As described in the methodology, a post-interview questionnaire was conducted after participants completed the comprehension tasks. The responses collected at this stage provide an insight into the participants' subjective preferences regarding the use of flowcharts along with code snippets. The data from the post-questionnaire will be used to evaluate the fifth hypothesis:

**1. Please indicate the tasks where you found the use of flowcharts to be helpful and those where it is not.** Participants provided varied feedback regarding the usefulness of flowcharts across different algorithmic tasks. While some found flowcharts beneficial for understanding complex logic structures, others either ignored them entirely or found them unnecessary for simple tasks. Flowcharts were particularly helpful in tasks involving nested loops and conditional structures, such as FindTheLargest, where participants reported that the flowchart was easier to follow than the code, allowing them to track the algorithm’s flow more intuitively. Similarly, in Crosssum, flowcharts assisted in visualizing the step-by-step summation process, and in integerToString, they aided in comprehending the sequence of operations. Decision-

making tasks also benefited from the presence of flowcharts, such as BinarySearch, where some participants relied on them for condition verification, and DecimalToBinary, where they provided a structured breakdown of the conversion process. On the other hand, several tasks did not significantly benefit from the presence of flowcharts. Some participants ignored the flowchart for BinarySearch because they already recognized the structure from prior knowledge. In H-index, the flowchart was deemed unhelpful in understanding the logic, and in Power Algorithm, participants found the code easier to interpret compared to the flowchart.

Array-based operations such as RemoveDoubleCharacters also did not see significant engagement with the flowchart, as participants felt that reading the code was sufficient. Additionally, some participants expressed discomfort with using flowcharts, noting that learning to reference them mid-experiment slowed them down, while others preferred working solely with code due to familiarity with textual representations.

**2. Could you describe the approach or strategy you used to solve the tasks?** Most participants initially focused on understanding the code using the top-down approach and only referred to the flowchart when they encountered difficulties or needed confirmation. This is consistent with previous research suggesting that novices tend to rely on textual representations by default. Two participants reported switching between the code and the flowchart, particularly when the code was long or unfamiliar. Additionally, one participant highlighted that flowcharts were especially beneficial for understanding the flow of algorithms, particularly in tasks involving loops and conditional structures. They noted that, in certain cases, the flowchart offered more clarity than the code itself due to its structured layout, which was especially helpful when memorization was challenging.

**3. How much did you refer to the flowcharts and how much to the code?** Participants were asked to indicate how often they referred to flowcharts versus code. The responses showed a wide range of preferences. Three participants indicated that they did not refer to the flowchart at all, while four participants indicated a stronger preference for using flowcharts over code snippets. In addition, six participants indicated that they primarily relied on code, but occasionally used flowcharts. On average, flowcharts were referenced in approximately 30-40% of tasks.

**4. For which task did you spend more time on, when was there a flowchart present or not?** Most participants reported spending more time on tasks when a flowchart was present because they had to process two different representations simultaneously. Two participants reported that flowcharts helped them save time, particularly on complex tasks involving nested loops and conditions. A further two participants saw no significant difference in time taken with or without a flowchart. In addition, five participants reported that they spent more time on certain tasks, such as IntegerToString, but attributed this to the complexity of the algorithm rather than the presence of the flowchart. For tasks involving loops, participants found that the extra time spent was unrelated to the use of flowcharts.

**5. Please choose on a scale from 1 to 5, did the presence of flowcharts in the task impact your understanding of the source code?** Participants rated the impact of flowcharts on their understanding of source code using a five-point Likert scale. The responses revealed a mixed perception of the usefulness of flowcharts. Three participants found flowcharts unhelpful, while five found them somewhat helpful. Two found them helpful, and one found them very helpful. Notably, no participant rated flowcharts as extremely helpful.

**6. If flowcharts were not provided in the tasks, would you have preferred their inclusion to aid task-solving?** Participants were also asked whether they would have preferred flowcharts if they had not been included in the tasks. The responses show a variety of preferences. Two participants strongly agreed, stating that flowcharts would have been very helpful in aiding understanding, especially for long tasks and those involving loops. Six participants somewhat agreed, acknowledging that flowcharts could clarify code structure and flow, particularly for visual learners, but were not essential. Three participants remained neutral, stating that flowcharts were useful in some cases, but not consistently helpful in all tasks.

**7. Did you use flowcharts as a learning tool in your introductory programming classes?** The responses indicated that only one participant had encountered flowcharts in all introductory courses, while another had used them in several, but not all, courses. Two participants reported using flowcharts in only one course, while the majority stated that flowcharts were not part of their programming education.

## 5.3 Hypotheses Testing

Once the data have been summarized, significance tests are carried out to determine whether the observed differences in the dependent variable are statistically significant. These tests assess whether variations in the independent variable (e.g. the presence or absence of a flowchart with the code snippet) have a measurable effect on the dependent variables. Since differences can occur due to random variation, significance tests help to assess whether the observed results are meaningful.

The null hypothesis assumes that there is no meaningful difference in the dependent variable across experimental conditions, whereas the alternative hypothesis ($H_1$) suggests the presence of a statistically significant difference. When the calculated p-value representing the probability of observing the data under the null hypothesis falls below the standard significance level of 5%, the null hypothesis is rejected. This indicates that the observed difference is unlikely to have occurred by chance. Conversely, if the p-value exceeds this threshold, the null hypothesis cannot be rejected, implying that any differences may be due to random variation rather than a true effect [125]. To evaluate the data, paired t-tests were applied for normally distributed variables, such as fixation time. For data that did not meet normality assumptions, non-parametric Wilcoxon signed-rank tests were employed. A significance level of $p < 0.05$ was used for all statistical tests.

**Ho1: Participants do not refer to flowcharts in addition to code snippets.**

Eye-tracking data showed that participants spent an average of 10.08 seconds fixating on flowcharts, which accounted for 27.16% of the total fixation time when both code and flowcharts were present. To determine whether participants were actively referencing flowcharts, a paired t-test was conducted comparing fixation time on code alone to fixation time on flowcharts. Therefore, it calculates the difference between the paired values (Fixation time on code when a flowchart was present and Fixation time on flowcharts) for each participant, then computes the mean difference and its standard deviation. The resulting p-value (0.0015) was significantly less than 0.05, indicating a statistically significant difference in fixation times. Therefore, the null hypothesis is rejected, leading to the conclusion that participants referred to flowcharts in addition to code snippets. This finding confirms that flowcharts were actively used

as a visual aid during the comprehension tasks, although they received less fixation time compared to code snippets.

**Ho2: There is no significant difference in response time of comprehension tasks due to the use of flowcharts.**

The analysis of response times indicates that participants took an average of 66.41 seconds to complete comprehension tasks with code snippets alone, compared to 69.21 seconds when flowcharts were included. This represents a 4.22% increase in mean response time when flowcharts were introduced. The Shapiro-Wilk test was used to determine whether the response times results followed a normal distribution. As the data were not normally distributed, a non-parametric test, the rank sum test, was chosen for analysis. The rank sum test was used to compare the response times between the code alone condition and the code with flowchart condition. The test yielded a p-value of 0.4389, which is greater than 0.05. This means that there is no statistically significant difference in response times between the two conditions. As a result, the null hypothesis Ho2 is accepted as this difference is not statistically significant. Therefore, the results suggest that the inclusion of flowcharts does not have a significant effect on the time taken to complete comprehension tasks.

**Ho3: There is no significant difference in correctness of comprehension tasks due to the use of flowcharts.**

The correctness analysis shows that participants achieved a correctness rate of 71% when solving comprehension tasks using code snippets alone and 71% when using flowcharts. This suggests that the correctness rate remained the same regardless of the presence of flowcharts. A chi-squared test was performed to determine whether this difference was statistically significant. The test compared correctness rates between comprehension tasks performed with code snippets alone and those performed with flowcharts. The results yielded a Chi-squared statistic of 0.0 and a p-value of 1.0. Therefore, the null hypothesis $H_{03}$ is accepted. Furthermore, the effect size Phi = 0.0 suggests that the inclusion of flowcharts has no measurable effect on correctness. Therefore, the results of this study do not support the claim that flowcharts improve correctness in comprehension tasks.

**Ho4: There is no significant difference in cognitive load during comprehension tasks due to the use of flowcharts.**

The Shapiro-Wilk test was carried out for both conditions. The test gave a p-value of 0.0728 for the code alone condition and a p-value of 0.0747 for the code with flowchart condition. As both p-values are greater than 0.05, this confirms that the data are normally distributed. The t-test was performed to determine if there was a significant difference in cognitive load between the code alone and code with flowchart conditions. The test yielded a p-value of 0.5209, which is greater than the 0.05 significance level. Therefore, the null hypothesis $H_{04}$ is accepted. This suggests that the use of flowcharts has no significant effect on reducing or increasing cognitive load during comprehension tasks.

**Ho5: Participants do not prefer flowcharts in addition to code snippets.**

The responses showed that 8 out of 11 participants (72.7%) stated a preference for flowcharts, indicating that most participants found them beneficial in aiding comprehension and problem-solving. Based on this majority preference, The null hypothesis $H_{05}$ is rejected.

## 5.4 Answer to Research Question

The research question, as outlined in the methodology section, is as follows:

RQ: *Can the impact of flowcharts on novices' program comprehension be verified in a replication study under similar experimental conditions?*

The results of this replication study indicate that the impact of flowcharts on novice program comprehension could not be fully verified under conditions similar to those of the original study but was only partially confirmed. The results are consistent with the original study in that participants referred to flowcharts in addition to code snippets, as confirmed by the visual attention analysis. In addition, the difference in cognitive load during comprehension tasks due to the use of flowcharts was not statistically significant, although it was slightly lower when flowcharts were presented alongside code snippets. Furthermore, participants' preference for flowcharts over code snippets

was confirmed. However, the integration of flowcharts into the comprehension process did not lead to an improvement in correctness rates, and response time did not increase significantly when flowcharts were used.

# 6 Discussion

This chapter presents the interpretation of the study results, considering both confirmed and rejected statistical hypotheses. The following subsections discuss visual attention, response time, correctness, and cognitive load in relation to the research question: What is the impact of flowcharts on novice code comprehension? Finally, the potential threats to construct, internal, and external validity are addressed.

## 6.1 Visual Attention

The hypothesis relating to this measurement factor was formulated in the methodology section as follows:

*H1. Participants refer to flowcharts in addition to code snippets.*

The results of the visual attention data show that participants actively referred to the flowcharts during the comprehension tasks and the difference in response time when flowcharts were presented was significant. Flowcharts were actively used in 28 out of 66 comprehension tasks, with an average fixation time of 10.08 seconds, which accounted for 27.16% of the total fixation time when flowcharts were present alongside code snippets. Notably, participants spent less time fixating on code snippets in these cases, suggesting that novices tend to seek alternative resources beyond textual code when attempting to understand algorithms. However, the flowchart representing the crosssum and removeDoubleCharacters algorithms was not used extensively. This behavior is consistent with existing research and the original study, which links reliance on additional visual aids with a lack of comprehension and problem-solving skills. When faced with difficulties interpreting algorithms in a programming language, novices seek external aids that provide additional context. The results of this study suggest that flowcharts are a practical approach to support, particularly when code comprehension proves challenging. In addition, feedback from participants confirmed the usefulness of flowcharts for understanding the flow of a program. Several participants noted that the flowcharts supported their ability to visualize how the algorithm executes, particularly in cases involving complex, lengthy, or unfamiliar code. The structured presentation of the flow allowed for a clearer breakdown of iterative

calculations and logical steps, making it easier to follow the progress and behavior of some algorithms.

Eye-tracking data further revealed that two participants did not engage with the flowcharts at all. One possible explanation is that interpreting a new visual format of algorithmic content may have been too cognitively demanding or time-consuming for them, prompting a preference for the textual code. Alternatively, this behavior may reflect individual learning styles particularly the distinction between visual and verbal learners. While visual learners tend to favor diagrams, flowcharts, timelines, and interactive elements, verbal learners may be more comfortable processing information through text. Verbal learners retain and process information more effectively through reading and textual explanations. It may be that the participants who did not refer to flowcharts were verbal learners who found textual representations more appropriate for their understanding. In addition, they may have been unwilling to engage with an unfamiliar tool, preferring a representation with which they were more familiar.

Another possible reason for the lack of use of flowcharts is the simplicity of the task and the complexity of the algorithm. If the algorithms were relatively simple, participants may have felt that they did not need the additional support of flowcharts. The original study suggests that when the complexity of the algorithm is low, textual representations may be sufficient, reducing the need to refer to flowcharts. This suggests that individual learning preferences, familiarity with visual aids and the complexity of the task all play a crucial role in determining whether flowcharts are used as an aid to comprehension.

An analysis of the eye-tracking data in this replication study identified four distinct patterns of flowchart use, extending the three patterns originally defined. While Pattern 1 (Balanced Usage), Pattern 2 (Code-Dominant) and Pattern 3 (Flowchart-Dominant) were consistent with the original findings, a fourth pattern emerged, indicating a unique behavioral shift in some participants.

Pattern 1: Balanced Usage, participants alternated between the code snippet and the flowchart, using both representations to construct their understanding. This pattern suggests that the flowchart served as secondary aid rather than a primary source of

understanding. Participants exhibiting this behavior were likely to refer to the flowchart when they had difficulty understanding the algorithm.

Pattern 2: Code-Dominant, participants spent most of their fixation time on the code snippet, occasionally glancing at the flowchart for confirmation or clarification. This behavior is characteristic of individuals who are more comfortable with textual representations and who use flowcharts as a supplementary tool rather than a necessity. Alternatively, participants following this pattern may have already formed a potential answer to the comprehension task and were only looking at the flowchart to verify or confirm their hypothesis.

Pattern 3: Flowchart dominant, participants focused primarily on the flowchart and only occasionally looked at the code snippet. This suggests that for these individuals the flowchart was the primary resource for understanding, with the code snippet playing a minimal role. This pattern suggests that flowcharts may provide a more efficient approach for some participants, allowing them to solve tasks effectively with some fixation on the code.

Pattern 4: Flowchart Exclusive Usage, a novel pattern unique to this replication study, where some participants relied entirely on the flowchart and completely ignored the code snippet. Several factors could explain this behavior: Task complexity, If the flowchart provided sufficient clarity, participants may not have felt the need to refer to the code; Strong visual learning preference. Some participants may have been highly visual learners who found flowcharts easier to interpret; Fatigue or order effects, when tasks appeared later in the experiment, participants may have defaulted to using flowcharts as a faster alternative to processing code. Heatmaps of some of the comprehension tasks are shown in Table 6.1

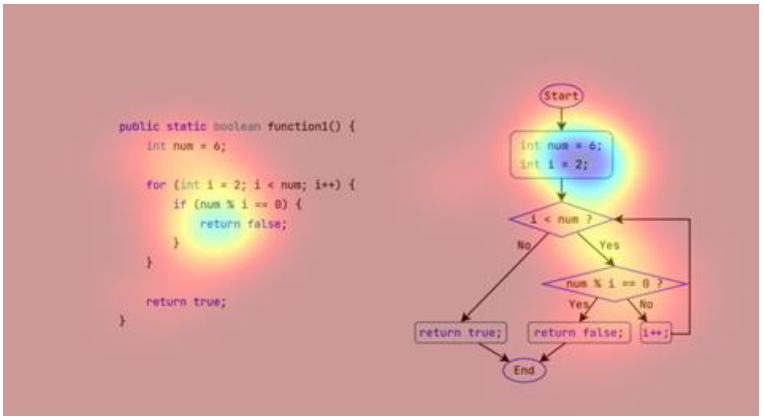
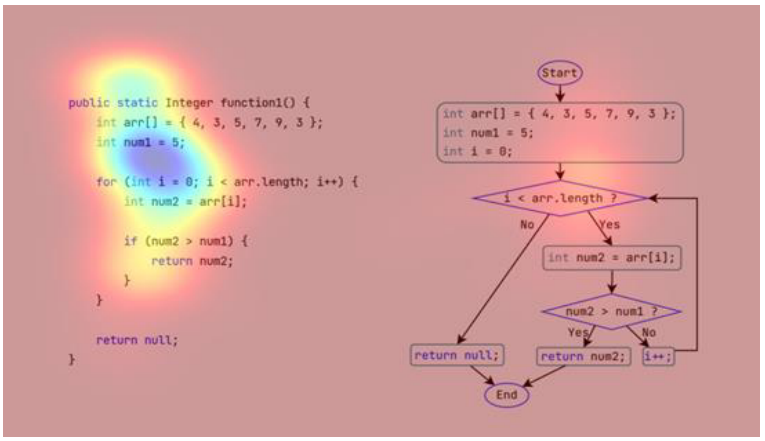
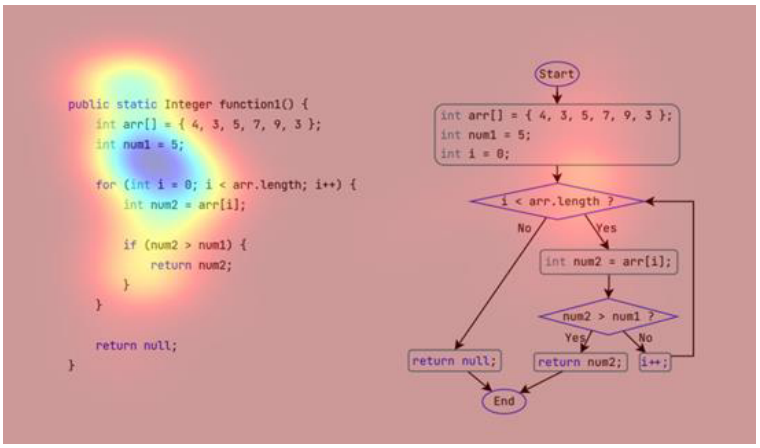
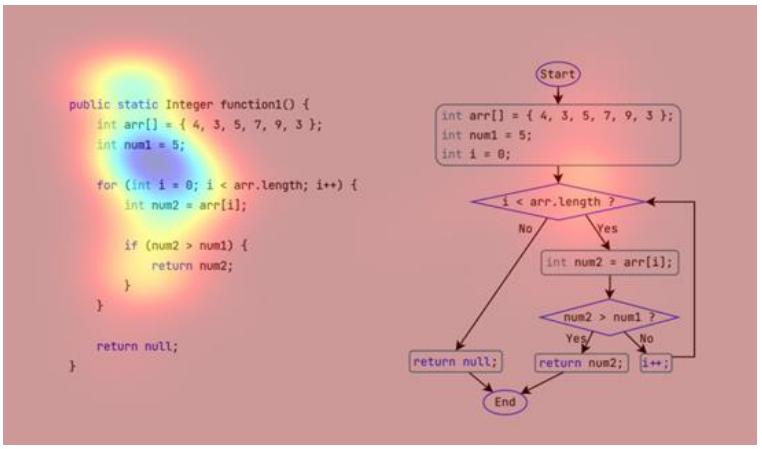
| | |
|---|---|
| **Pattern1** |  |
| **Pattern2** |  |
| **Pattern 3** |  |
| **Pattern 4** |  |

**Table 6.1 Usage Patterns of Flowcharts.**

## 6.2 Response Time

The hypothesis corresponding to this measurement factor was defined in the methodology section as follows:

*H2: Participants using flowcharts take less time to complete the tasks.*

Response time analysis showed that participants took an average of 66.41 seconds to complete comprehension tasks using code snippets alone, whereas the presence of flowcharts resulted in an average response time of 69.21 seconds, an increase of 4.22%. However, statistical tests confirmed that this difference was not statistically significant. These results contrast with those of the original study, where the difference in response time between code snippets alone and code with flowcharts was found to be statistically significant, with participants taking 34.26% more time when using flowcharts. However, in this replication study, the results suggest that students did not take significantly longer when the flowchart was presented alongside the code; rather, for some tasks, the flowchart was simply ignored. In particular, the mean response time for four algorithms (e.g. binarySearch, crosssum, dropNumber and removeDoubleCharacters) was recorded as zero, indicating that participants did not engage with the flowcharts in these tasks. Some students expressed a preference for relying solely on the code, citing discomfort with using a new tool, while others reported that once they became familiar with the flowcharts, they found them useful. As noted in the visual attention analysis, some students relied solely on the flowchart representation and ignored the code altogether, which may explain why the response time differences between comprehension tasks using code snippets alone and comprehension tasks using flowcharts were not significant.

In the original study, novice programmers may have initiated two parallel comprehension processes when presented with both code snippets and flowcharts. In such cases, they attempted to follow the flow of the algorithm in both representations, switching between them when they encountered difficulties or needed additional clarity. This process allowed them to simultaneously check their answers against the alternative representation. Still, the additional time novice programmers spend engaging with flowcharts alongside code snippets may be worthwhile if it contributes to the formation of more accurate mental models, thereby enhancing their overall

understanding. That said, the extent to which this visual aid is utilized can vary significantly based on individual preferences and prior experience, some may rely on it heavily, while others may choose to disregard it entirely.

### 6.3 Correctness

The hypothesis corresponding to this measurement variable was stated in the methodology section as follows:

*H3: Participants using flowcharts answer with a higher correction rate.*

The results of the replication study show that the correctness rate for the comprehension tasks was 0.71 (71%) in both conditions, code alone and code with flowcharts. A chi-squared test confirmed that there was no statistically significant difference in correctness between the two conditions, suggesting that the presence of flowcharts had no measurable effect on participants' accuracy in completing the tasks.

This finding contrasts with the original study, where the use of flowcharts resulted in a significant improvement in correctness, from 50% with code snippets alone to 68% when flowcharts were used, an improvement of 18%. One possible explanation for this discrepancy is the way in which the participants interacted with the flowcharts. In the original study, participants reported that flowcharts facilitated program comprehension by clarifying algorithmic flow, particularly in selection and iteration structures. In addition, participants in the original study used flowcharts as a verification tool, allowing them to refine their understanding before submitting their responses. This iterative process may have contributed to the significant improvement in accuracy.

In contrast, the replication study suggests that while some participants may have benefited from flowcharts, others may have ignored them completely, as mentioned in the discussion of response times above. Furthermore, while certain tasks (e.g., power, isPrime, integertoString, findTheLargest, bubbleSort) were always answered correctly, their accuracy rates were not statistically significant. Even when participants used flowcharts, they were not necessarily effective in helping them find or confirm the correct answer. Table 5.3 Correctness, Response Time and Cognitive Load Results shows that the correctness rate for solving certain tasks (e.g., binarySearch, crosssum, decimalToBinary, dropNumber, removeDoubleCharacters) using flowcharts was 0%.

This suggests that the flowcharts were not seen as an aid to understanding comprehension tasks in this study.

These findings suggest that the presence of flowcharts does not necessarily improve accuracy unless participants actively engage with them in a structured way. The divergence from the original study may be due to differences in participants' familiarity with flowcharts, cognitive strategies or task complexity. While flowcharts provide an additional layer of information, their effectiveness appears to depend not only on their availability, but also on how they are used and whether participants choose to use them as a support tool. This highlights the need for further research to identify the conditions under which flowcharts are most beneficial for novice programmers in comprehension tasks.

### 6.4 Cognitive Load

The hypothesis associated with this measurement factor was established in the methodology as follows:

*H4: Participants using flowcharts have a lower cognitive load when completing the tasks.*

The cognitive load analysis in this study shows that participants had a mean cognitive load of 3.3 when using code snippets alone and 2.94 when using flowcharts in the comprehension tasks. Although the cognitive load appeared to be slightly lower when flowcharts were present, statistical analysis confirmed that this difference was not significant. This is consistent with the original study, which also found no significant increase in cognitive load when flowcharts were introduced. The presence of visual aids may raise concerns that they may increase cognitive load. An important observation is that participants referred to flowcharts when they had difficulty understanding an algorithm or a long algorithm. This suggests that novices were able to use flowcharts strategically to aid their understanding without incurring additional cognitive load. Furthermore, despite their initial unfamiliarity with flowcharts, participants learned how to interpret them through the instructions in the introduction to the experiment. The original study highlighted the low learning curve of flowcharts, making them an effective tool for novice programmers. The results of this study support

this perspective, demonstrating that flowcharts provide an intuitive means of understanding algorithmic flow without imposing a significant cognitive load.

### 6.5 Flowchart Preferences

The hypothesis pertaining to this measurement factor was defined in the methodology as follows:

*H5: Participants prefer to use flowcharts in addition to code.*

The results of the post-interview responses indicate that the majority of participants preferred to use flowcharts in addition to code snippets, which is consistent with the results of the original study. In the original study, 7 out of 11 participants expressed a preference for flowcharts, whereas in this study, 8 out of 11 participants favored their inclusion. This consistency suggests that novice programmers tend to prefer visual aids. However, while the overall preference remained similar, the extent to which participants engaged with flowcharts differed between the two studies.

In the original study, participants not only expressed a preference for flowcharts, but also demonstrated a higher rate of accuracy when using them. They also reported that flowcharts helped them solve comprehension tasks faster by reducing ambiguity and allowing them to check their answers. In this replication study, although most participants expressed a preference for flowcharts, their actual use varied, with some relying on them heavily and others choosing to ignore them entirely. Where several participants stated that the additional presentation did not necessarily improve their efficiency, and in some cases, it increased their reaction time as they processed two different forms of the algorithm. In addition, one participant mentioned that it was difficult to work with an unfamiliar tool. A key difference between the two studies is that in the original study the flowcharts appeared to play a more central role in understanding, whereas in the replication study few participants actively used the flowcharts to understand the algorithm better, while others preferred to have the flowcharts alongside the code to check their answers. Overall, these findings confirm that flowcharts are generally preferred as an aid to understanding, but their effectiveness varies depending on individual problem-solving strategies and task complexity. In addition, participants in this study suggested that flowcharts should be

included as a learning tool in introductory programming courses to help other students understand code more effectively, and that they would have liked to have had the opportunity to use them in their previous programming courses.

## 6.6 Threads to Validity

Threats to validity refer to potential problems or biases that arise during data collection, processing, or analysis that may affect the accuracy and credibility of the study's conclusions. These threats are often influenced by factors related to the experimental environment, participant variability, measurement techniques, or external conditions that may confound the intended results.

### *6.6.1 Construct Validity*

Construct validity refers to the extent to which the study accurately measures the concept it seeks to investigate [112]. In this study, several factors posed potential threats to construct validity, including distortion of the EEG signal, guesswork based on response time, and hypothesis guessing. During EEG data processing, some signals were distorted due to the high sensitivity of EEG signals to noise and interference from other physiological electrical signals, such as heartbeat activity. In addition, participants' movements during the experiment introduced signal artefacts that affected the accuracy of the data. To minimize these distortions, signal processing filters were applied during data analysis; however, the removal of unwanted artefacts also resulted in some signal loss, potentially affecting the measurement of cognitive load [132]. Independent Component Analysis (ICA), recognized as one of the most effective methods for cleaning EEG signals, was applied to improve signal validity and increase the reliability of cognitive load measures [131].

Another threat to construct validity arises from the possibility that short response times may not always reflect accurate understanding, but rather guesswork. To address this issue, participants were given the option to skip tasks if they were uncertain, thus reducing the likelihood of random responses. In addition, a minimum response time threshold and eye fixation duration analysis were used to verify that participants were actively engaged with the flowchart or code before responding. These measures ensured that only meaningful responses contributed to the study results, thereby

reducing the risk of response bias due to guessing. Furthermore, hypothesis guessing poses a social threat to construct validity, as participants may attempt to infer the purpose of the study and adjust their responses accordingly [112]. To mitigate this, participants were explicitly informed that they could skip a question if they were unsure, thus reducing the likelihood of biased responses due to guessing. However, this instruction may have led participants to perceive the study not as an assessment of their programming skills, but rather as an assessment of the impact of flowcharts on code comprehension.

As a result, some participants may have consciously or unconsciously aligned their responses with their assumptions about the study's hypothesis, either positively or negatively. To address this concern, the first question in the post-experiment interview was designed to assess where participants found the flowcharts useful and where they did not. Participants were shown all the flowchart tasks they had completed and asked to give reasons for their choices. This approach provided a deeper insight into their cognitive processes and helped to identify potential biases introduced by hypothesis guessing.

### *6.6.2 Internal Validity*

Internal validity refers to whether the observed effects in the study are really caused by the independent variable (i.e., the flowchart) or whether they are caused by confounding factors [112]. A potential threat to internal validity is stress caused by external factors, such as exam periods or personal pressures, that affect participants' cognitive performance. To minimize this risk, the study was not conducted during exam periods, but rather before or after to avoid this effect. In addition, participants were allowed to choose a convenient time to take part, ensuring that they were able to complete the study without external pressure.

Another factor that could have affected the results is fatigue. Some participants had long and thick hair, which increased the time it took to set up the EEG and, in some cases, led to boredom. Unfortunately, this cannot be controlled experimentally. In addition, the motivation of the participants may have influenced the results. As all participants were volunteers, they were likely to be more motivated and engaged than a sample of the general population [112].

Volunteers tend to be more interested, which may lead to better task performance compared to a less intrinsically motivated group. In the literature, some students are identified as textual learners, while others rely more on visual aids and are classified as visual learners. These differences introduce variations in cognitive strategies that could influence the results of the study. Future research could explore participants' preferred learning styles (visual vs. textual) and analyze whether the effectiveness of flowcharts varies based on these preferences.

### *6.6.3 External Validity*

External validity refers to the extent to which the findings of a study can be generalized beyond the specific conditions of the experiment, such as different populations, settings, and time frames [112]. All participants were novice students from the same university. While this controlled for within-group variability, reducing potential bias, it also limits the generalizability of the findings to other populations, such as professional programmers or novice learners from different educational backgrounds [112].

Another limitation is the nature of the programming tasks. Participants were asked to solve short code snippets, which are consistent with introductory programming courses. However, real-world programming environments, particularly in industry, often involve longer, more complex code bases. This discrepancy raises concerns about whether the effectiveness of flowchart observed in this study would hold up in practical, large-scale software development contexts.

Furthermore, as highlighted in the original study, the experiment, such as this study, was conducted exclusively in the Java programming language. However, as the code snippets used in this study were not highly complex, it is unlikely that changing the programming language would have had a significant impact on comprehension.

# 7 Conclusion

Research has consistently shown that novice programmers often struggle to develop the mental models necessary for effectively understanding programs and solving problems [13]. They encounter considerable difficulties in grasping the flow, structure and purpose of algorithms, hindering their ability to translate problem specifications into working code. Motivated by these challenges, the present study aimed to replicate and extend prior research investigating the impact of algorithmic visualizations specifically flowcharts on novice program comprehension.

This controlled replication study employed eye tracking, EEG monitoring and structured interviews to gain multifaceted insights into the cognitive processes involved. Novices were asked to complete comprehension tasks with code snippets alone or with code snippets accompanied by flowcharts. Metrics such as fixation time, cognitive load, response time, accuracy, and subjective preference were systematically evaluated.

The results of the present study are partly consistent with the original findings [24]. In both studies, participants actively used flowcharts during comprehension tasks, as confirmed by eye-tracking data. Participants also consistently expressed a subjective preference for the inclusion of flowcharts, indicating that they perceived them as beneficial aids in understanding program logic. However, while the original study reported a statistically significant improvement in accuracy with flowcharts, this replication found no statistically significant difference in accuracy rates between conditions.

Similarly, response time did not differ significantly across the two conditions in this replication, whereas the original study found that using flowcharts significantly increased response time.

Notably, no significant effect on cognitive load was observed in either study. This suggests that, although the participants positively received and used flowcharts during the comprehension, they did not necessarily ease cognitive processing or speed up task completion for novice programmers.

Interestingly, despite the lack of improvements in correctness or response time in the replication study, participants' preference for flowcharts indicates that they might perceive them as a helpful scaffold, perhaps offering psychological comfort or reducing the perceived complexity of the task.

The discrepancies between the original and replication results highlight several important considerations. First, replication in software engineering research, particularly studies involving human behavior, remains inherently challenging. Variability in participant characteristics, sample size limitations, changes in technological environments, and the inevitable loss of direct access to original researchers complicate replication efforts [133].

This mirrors broader trends observed in the "replication crisis" across fields like psychology and empirical software engineering, where even carefully conducted replications often fail to reproduce original findings exactly [134].

Furthermore, this study encountered some technical challenges due to updates in libraries and tools between 2022 and 2024/25. This illustrates the rapid pace of technological change in software engineering environments. These factors, alongside different participant demographics and a small sample size, may have contributed to the observed variations.

In conclusion, although flowcharts do not lead to significant improvements in objective measures of program comprehension, novice programmers nevertheless appreciate them as valuable supportive tools. Integrating them into introductory programming courses could enhance students' subjective learning experiences and perceived confidence. However, they should not be viewed as a standalone solution, but rather as one component of a broader pedagogical strategy.

Further research is necessary to better understand the true impact of flowcharts on novice learners, and to validate their effectiveness in different educational contexts. This should include more extensive replication studies.

## 7.1 Future Work

Based on the findings and challenges encountered in this replication study, the following avenues for future research are proposed:

- Investigate the effect of flowcharts on code comprehension at different levels of complexity and among programmers with varying levels of experience.
- Comparative studies could examine the effectiveness of various visual aids, such as pseudocode, UML diagrams and animated traces, compared with traditional flowcharts.
- Expanded sample sizes and diversity, by involving students from different universities and educational backgrounds, would increase the generalizability of results and enhance consistency, reproducibility, and transparency across empirical studies.
- Expert versus novice comparisons could provide valuable insights into how visualization tools are used differently across experience levels.
- Involving students from different universities and educational backgrounds would expand the sample size and diversity, making the results more generalizable and enhancing the consistency, reproducibility and transparency of empirical studies.
- Incorporate a wider range of tools and methodologies for measuring program comprehension, such as functional magnetic resonance imaging (fMRI), recall tasks, think-aloud protocols, and debugging exercises.
- Controlled intervention studies could be conducted by designing two separate courses: one in which flowcharts are systematically integrated into the curriculum, and another in which flowcharts are not used. This approach would allow researchers to observe and compare the long-term academic outcomes between the two groups.

Addressing these areas systematically will enable future research to determine the role of algorithmic visualizations in novice programming education more effectively. This will contribute to the evidence-based enhancement of computer science pedagogy.

# Bibliography


[1] B. Boehm, “A view of 20th and 21st century software engineering,” in *Proceedings of the 28th international conference on Software engineering*, Shanghai China: ACM, May 2006, pp. 12–29. doi: 10.1145/1134285.1134288.

[2] R. Brooks, “Towards a theory of the comprehension of computer programs,” *Int. J. Man-Mach. Stud.*, vol. 18, no. 6, pp. 543–554, 1983, Accessed: Apr. 21, 2025. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0020737383800315

[3] B. Shneiderman and R. Mayer, “Syntactic/semantic interactions in programmer behavior: A model and experimental results,” *Int. J. Comput. Inf. Sci.*, vol. 8, no. 3, pp. 219–238, Jun. 1979, doi: 10.1007/BF00977789.

[4] N. Pennington, “Stimulus structures and mental representations in expert comprehension of computer programs,” *Cognit. Psychol.*, vol. 19, no. 3, pp. 295–341, Jul. 1987, doi: 10.1016/0010-0285(87)90007-7.

[5] T. R. G. Green and M. Petre, “Usability analysis of visual programming environments: a ‘cognitive dimensions’ framework,” *J. Vis. Lang. Comput.*, vol. 7, no. 2, pp. 131–174, 1996, Accessed: Apr. 21, 2025. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1045926X96900099

[6] E. Soloway, J. Bonar, and K. Ehrlich, “Cognitive strategies and looping constructs: an empirical study,” *Commun. ACM*, vol. 26, no. 11, pp. 853–860, Nov. 1983, doi: 10.1145/182.358436.

[7] T. A. Standish, “An essay on software reuse,” *IEEE Trans. Softw. Eng.*, no. 5, pp. 494–497, 1984, Accessed: Apr. 21, 2025. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/5010272/

[8] J. Siegmund, “Program comprehension: Past, present, and future,” in *2016 IEEE 23rd international conference on software analysis, evolution, and reengineering (SANER)*, IEEE, 2016, pp. 13–20. Accessed: Apr. 21, 2025. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/7476769/

[9] C. Areias and A. Mendes, “A tool to help students to develop programming skills,” in *Proceedings of the 2007 international conference on Computer systems and technologies - CompSysTech ’07*, Bulgaria: ACM Press, 2007, p. 1. doi: 10.1145/1330598.1330692.

[10] A. McGettrick, R. Boyle, R. Ibbett, J. Lloyd, G. Lovegrove, and K. Mander, “Grand challenges in computing: Education—a summary,” *Comput. J.*, vol. 48, no. 1, pp. 42–48, 2005, Accessed: Apr. 21, 2025. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8140304/

[11] J. Bennedsen and M. E. Caspersen, “Failure rates in introductory programming,” *ACM SIGCSE Bull.*, vol. 39, no. 2, pp. 32–36, Jun. 2007, doi: 10.1145/1272848.1272879.

[12] J. Figueiredo and F. García-Peñalvo, “Teaching and learning tools for introductory programming in university courses,” in *2021 International Symposium on Computers in Education (SIIE)*, IEEE, 2021, pp. 1–6. Accessed: Apr. 21, 2025. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9583623/

[13] E. Lahtinen, K. Ala-Mutka, and H.-M. Järvinen, “A study of the difficulties of novice programmers,” *ACM SIGCSE Bull.*, vol. 37, no. 3, pp. 14–18, Sep. 2005, doi: 10.1145/1151954.1067453.

[14] Y.-F. Shih and S. M. Alessi, “Mental Models and Transfer of Learning in Computer Programming,” *J. Res. Comput. Educ.*, vol. 26, no. 2, pp. 154–175, Dec. 1993, doi: 10.1080/08886504.1993.10782084.

[15] Z. Ullah, A. Lajis, M. Jamjoom, A. Altalhi, A. Al-Ghamdi, and F. Saleem, “The effect of automatic assessment on novice programming: Strengths and limitations of existing systems,” *Comput. Appl. Eng. Educ.*, vol. 26, no. 6, pp. 2328–2341, Nov. 2018, doi: 10.1002/cae.21974.

[16] K. M. Yusoff, N. S. Ashaari, T. Wook, and N. M. Ali, “Analysis on the requirements of computational thinking skills to overcome the difficulties in learning programming,” *Int. J. Adv. Comput. Sci. Appl.*, vol. 11, no. 3, pp. 244–253, 2020, Accessed: Apr. 21, 2025. [Online]. Available: https://www.academia.edu/download/87004513/Paper_29-Analysis_on_the_Requirements_of_Computational_Thinking_Skills.pdf

[17] O. Kurniawan, C. Jégourel, N. T. S. Lee, M. De Mari, and C. M. Poskitt, “Steps Before Syntax: Helping Novice Programmers Solve Problems using the PCDIT Framework,” presented at the Hawaii International Conference on System Sciences, 2022. doi: 10.24251/HICSS.2022.121.

[18] S. Garner, “Reducing the Cognitive Load on Novice Programmers.,” Jun. 2002. [Online]. Available: https://files.eric.ed.gov/fulltext/ED477013.pdf

[19] F. Schmidt, “Detecting and Correcting the Lies That Data Tell,” *Perspect. Psychol. Sci.*, vol. 5, no. 3, pp. 233–242, May 2010, doi: 10.1177/1745691610369339.

[20] M. E. Tudoreanu, “Designing effective program visualization tools for reducing user’s cognitive effort,” in *Proceedings of the 2003 ACM symposium on Software visualization*, San Diego California: ACM, Jun. 2003, p. 105. doi: 10.1145/774833.774848.

[21] C. Evans and N. J. Gibbons, “The interactivity effect in multimedia learning,” *Comput. Educ.*, vol. 49, no. 4, pp. 1147–1160, 2007, Accessed: Apr. 21, 2025. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0360131506000285

[22] A. E. Zimmermann, E. E. King, and D. D. Bose, “Effectiveness and Utility of Flowcharts on Learning in a Classroom Setting: A Mixed-Methods Study,” *Am. J. Pharm. Educ.*, vol. 88, no. 1, p. 100591, Jan. 2024, doi: 10.1016/j.ajpe.2023.100591.

[23] R. Levy, M. Ben-Ari, and P. Uronen, “The Jeliot 2000 program animation system,” *Comput. Educ.*, vol. 40, pp. 1–15, Jan. 2003, doi: 10.1016/S0360-1315(02)00076-3.

[24] “2022 Master - Effect of Flowcharts on Code Comprehension of Novice Programmers.pdf.” Accessed: Apr. 21, 2025. [Online]. Available: https://www.tu-chemnitz.de/informatik/ST/lectures/Masters%20Thesis%20Pdfs/2022%20Master%20-%20Effect%20of%20Flowcharts%20on%20Code%20Comprehension%20of%20Novice%20Programmers.pdf

[25] E. Soloway and K. Ehrlich, “Empirical Studies of Programming Knowledge,” *IEEE Trans. Softw. Eng.*, vol. SE-10, no. 5, pp. 595–609, Sep. 1984, doi: 10.1109/TSE.1984.5010283.

[26] X. Xia, L. Bao, D. Lo, Z. Xing, A. E. Hassan, and S. Li, “Measuring Program Comprehension: A Large-Scale Field Study with Professionals,” *IEEE Trans. Softw. Eng.*, vol. PP, pp. 1–1, Jul. 2017, doi: 10.1109/TSE.2017.2734091.

[27] Y. Cai, L. Xiao, R. Kazman, R. Mo, and Q. Feng, “Design Rule Spaces: A New Model for Representing and Analyzing Software Architecture,” *IEEE Trans. Softw. Eng.*, vol. 45, no. 7, pp. 657–682, Jul. 2019, doi: 10.1109/TSE.2018.2797899.
[28] A. Tversky and D. Kahneman, “Judgment under Uncertainty: Heuristics and Biases: Biases in judgments reveal some heuristics of thinking under uncertainty.,” *Science*, vol. 185, no. 4157, pp. 1124–1131, Sep. 1974, doi: 10.1126/science.185.4157.1124.
[29] E. Soloway, B. Adelson, and K. Ehrlich, “Knowledge and Processes in The Comprehension of Computer Programs,” in *The Nature of Expertise*, Psychology Press, 1988.
[30] J. Feigenspan, C. Kastner, J. Liebig, S. Apel, and S. Hanenberg, “Measuring programming experience,” in *2012 20th IEEE International Conference on Program Comprehension (ICPC)*, Passau, Germany: IEEE, Jun. 2012, pp. 73–82. doi: 10.1109/ICPC.2012.6240511.
[31] S. Wiedenbeck, “The initial stage of program comprehension,” *Int. J. Man-Mach. Stud.*, vol. 35, no. 4, pp. 517–540, Oct. 1991, doi: 10.1016/S0020-7373(05)80090-2.
[32] J. Siegmund *et al.*, “Measuring neural efficiency of program comprehension,” in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, Paderborn Germany: ACM, Aug. 2017, pp. 140–150. doi: 10.1145/3106237.3106268.
[33] G. A. Miller, “The Magical Number Seven, Plus or Minus Two: Some Limits on our Capacity for Processing Information[1]”.
[34] D. C. Littman, J. Pinto, S. Letovsky, and E. Soloway, “Mental models and software maintenance,” *J. Syst. Softw.*, vol. 7, no. 4, pp. 341–355, Dec. 1987, doi: 10.1016/0164-1212(87)90033-1.
[35] J. Koenemann and S. P. Robertson, “Expert problem-solving strategies for program comprehension,” Proceedings of the SIGCHI conference on Human factors in computing systems Reaching through technology - CHI '91, 1991.
[36] A. Dunsmore and M. Roper, “A Comparative Evaluation of Program Comprehension Measures”.
[37] S. Tenny, J. M. Brannan, and G. D. Brannan, “Qualitative Study,” in *StatPearls*, Treasure Island (FL): StatPearls Publishing, 2025. Accessed: Apr. 21, 2025. [Online]. Available: http://www.ncbi.nlm.nih.gov/books/NBK470395/
[38] S. Kvale, *InterViews : learning the craft of qualitative research interviewing*. Los Angeles : Sage Publications, 2009. Accessed: Apr. 21, 2025. [Online]. Available: http://archive.org/details/interviewslearni0000kval
[39] B. DiCicco-Bloom and B. F. Crabtree, “The qualitative research interview,” *Med. Educ.*, vol. 40, no. 4, pp. 314–321, Apr. 2006, doi: 10.1111/j.1365-2929.2006.02418.x.
[40] “Likert_1932.pdf.” Accessed: Apr. 22, 2025. [Online]. Available: https://legacy.voteview.com/pdf/Likert_1932.pdf
[41] J. Feigenspan, S. Apel, J. Liebig, and C. Kastner, “Exploring Software Measures to Assess Program Comprehension,” in *2011 International Symposium on Empirical Software Engineering and Measurement*, Banff, AB, Canada: IEEE, Sep. 2011, pp. 127–136. doi: 10.1109/ESEM.2011.21.
[42] R. J. Miara, J. A. Musselman, J. A. Navarro, and B. Shneiderman, “Program indentation and comprehensibility,” *Commun. ACM*, vol. 26, no. 11, pp. 861–867, Nov. 1983, doi: 10.1145/182.358437.
[43] M. A. Just, “A theory of reading: From eye ixations to comprehension”.

[44] K. Rayner, “Eye Movements in Reading and Information Processing: 20 Years of Research”.
[45] A. T. Duchowski, *Eye Tracking Methodology*. Cham: Springer International Publishing, 2017. doi: 10.1007/978-3-319-57883-5.
[46] J. Z. Lim, J. Mountstephens, and J. Teo, “Emotion Recognition Using Eye-Tracking: Taxonomy, Review and Current Challenges,” *Sensors*, vol. 20, no. 8, p. 2384, Apr. 2020, doi: 10.3390/s20082384.
[47] B. Kitchenham, “Procedures for Performing Systematic Reviews”.
[48] N. Peitek, J. Siegmund, C. Parnin, S. Apel, J. C. Hofmeister, and A. Brechmann, “Simultaneous measurement of program comprehension with fMRI and eye tracking: a case study,” in *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, Oulu Finland: ACM, Oct. 2018, pp. 1–10. doi: 10.1145/3239235.3240495.
[49] Y.-G. Guéhéneuc, “TAUPE: towards understanding program comprehension,” in *Proceedings of the 2006 conference of the Center for Advanced Studies on Collaborative research - CASCON ’06*, Toronto, Ontario, Canada: ACM Press, 2006, p. 1. doi: 10.1145/1188966.1188968.
[50] 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)*, Banff, Alberta, BC: IEEE, Jun. 2007, pp. 113–122. doi: 10.1109/ICPC.2007.10.
[51] M. E. Crosby and J. Stelovsky, “How do we read algorithms? A case study,” *Computer*, vol. 23, no. 1, pp. 25–35, Jan. 1990, doi: 10.1109/2.48797.
[52] B. Sharif and J. Maletic, *An Eye Tracking Study on camelCase and under_score Identifier Styles*. 2010, p. 205. doi: 10.1109/ICPC.2010.41.
[53] K. Park *et al.*, “An eye tracking study assessing source code readability rules for program comprehension,” *Empir. Softw. Eng.*, vol. 29, no. 6, p. 160, Oct. 2024, doi: 10.1007/s10664-024-10532-x.
[54] J. Beatty and B. Lucero-Wagoner, “The pupillary system,” Oct. 2012.
[55] JACKSON BEATTY and DANIEL KAHNEMAN, “Pupillary changes In two memory tasks,” *Springer*, pp. 371--372, 1966. [Online]. Available: https://link.springer.com/content/pdf/10.3758/BF03328444.pdf
[56] E. H. Hess and J. M. Polt, “Pupil Size in Relation to Mental Activity during Simple Problem-Solving,” *Science*, vol. 143, no. 3611, pp. 1190–1192, Mar. 1964, doi: 10.1126/science.143.3611.1190.
[57] M. Behroozi, S. Shirolkar, T. Barik, and C. Parnin, “Does stress impact technical interview performance?,” in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Virtual Event USA: ACM, Nov. 2020, pp. 481–492. doi: 10.1145/3368089.3409712.
[58] M. Doughty, “Consideration of Three Types of Spontaneous Eyeblink Activity in Normal Humans: during Reading and Video Display Terminal Use, in Primary Gaze, and while in Conversation,” *Optom. Vis. Sci. Off. Publ. Am. Acad. Optom.*, vol. 78, pp. 712–25, Nov. 2001, doi: 10.1097/00006324-200110000-00011.
[59] “The biomedical engineering handbook. 1,” Boca Raton, Fla: CRC Press, 2000.
[60] D. L. Schomer and F. Lopes da Silva, *Niedermeyer’s electroencephalography: Basic principles, clinical applications, and related fields: Sixth edition*. 2012, p. 1269.
[61] M. Teplan, “FUNDAMENTALS OF EEG MEASUREMENT,” *Meas. Sci. Rev.*, vol. 2, 2002.

[62] “Hochdichte EEG-Hauben für Forscher,” BESDATA. Accessed: Apr. 22, 2025. [Online]. Available: https://besdatatech.com/de/high-density-eeg-caps-for-researchers/
[63] G. Buzsáki, *Rhythms of the Brain*. Oxford University Press, 2006. doi: 10.1093/acprof:oso/9780195301069.001.0001.
[64] R. Llinás and U. Ribary, “Coherent 40-Hz Oscillation Characterizes Dream State in Humans,” *Proc. Natl. Acad. Sci. U. S. A.*, vol. 90, pp. 2078–81, Apr. 1993, doi: 10.1073/pnas.90.5.2078.
[65] “Dry Electrode,” Wearable Sensing | Dry EEG. Accessed: Apr. 22, 2025. [Online]. Available: https://wearablesensing.com/dry-electrode/
[66] S. J. Luck, *An Introduction to the Event-Related Potential Technique, second edition*. MIT Press, 2014.
[67] E. Lattari *et al.*, “Corticomuscular coherence behavior in fine motor control of force: A critical review,” *Rev. Neurol.*, vol. 51, pp. 610–23, Nov. 2010.
[68] S. Kanoga and Y. Mitsukura, “Review of Artifact Rejection Methods for Electroencephalographic Systems,” 2017. doi: 10.5772/68023.
[69] W. Klimesch, “EEG alpha and theta oscillations reflect cognitive and memory performance: a review and analysis,” *Brain Res. Rev.*, vol. 29, no. 2–3, pp. 169–195, Apr. 1999, doi: 10.1016/S0165-0173(98)00056-3.
[70] E. Soloway and J. C. Spohrer, *Studying the novice programmer*. Psychology Press, 2013. Accessed: Apr. 21, 2025. [Online]. Available: https://api.taylorfrancis.com/content/books/mono/download?identifierName=doi&identifierValue=10.4324/9781315808321&type=googlepdf
[71] A. Robins, J. Rountree, and N. Rountree, “Learning and Teaching Programming: A Review and Discussion,” *Comput. Sci. Educ.*, vol. 13, no. 2, pp. 137–172, Jun. 2003, doi: 10.1076/csed.13.2.137.14200.
[72] L. E. Winslow, “Programming pedagogy—a psychological overview,” *ACM SIGCSE Bull.*, vol. 28, no. 3, pp. 17–22, Sep. 1996, doi: 10.1145/234867.234872.
[73] B. A. Sheil, “The Psychological Study of Programming,” *ACM Comput. Surv.*, vol. 13, no. 1, pp. 101–120, Mar. 1981, doi: 10.1145/356835.356840.
[74] A. Lishinski, A. Yadav, R. Enbody, and J. Good, “The Influence of Problem Solving Abilities on Students’ Performance on Different Assessment Tasks in CS1,” in *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*, Memphis Tennessee USA: ACM, Feb. 2016, pp. 329–334. doi: 10.1145/2839509.2844596.
[75] B. Mase and L. Nel, “Common Code Writing Errors Made by Novice Programmers: Implications for the Teaching of Introductory Programming,” 2022, pp. 102–117. doi: 10.1007/978-3-030-95003-3_7.
[76] R. Lister, “On the cognitive development of the novice programmer: and the development of a computing education researcher,” in *Proceedings of the 9th Computer Science Education Research Conference*, Virtual Event Netherlands: ACM, Oct. 2020, pp. 1–15. doi: 10.1145/3442481.3442498.
[77] J. Sillito, G. Murphy, and K. Volder, “Asking and Answering Questions during a Programming Change Task,” *Softw. Eng. IEEE Trans. On*, vol. 34, pp. 434–451, Aug. 2008, doi: 10.1109/TSE.2008.26.
[78] F. Ricca, M. Di Penta, M. Torchiano, P. Tonella, and M. Ceccato, “The Role of Experience and Ability in Comprehension Tasks Supported by UML Stereotypes,” in *29th International Conference on Software Engineering (ICSE’07)*, Minneapolis, MN: IEEE, May 2007, pp. 375–384. doi: 10.1109/ICSE.2007.86.

[79] M. M. Müller, “Are Reviews an Alternative to Pair Programming?,” *Empir. Softw. Eng.*, vol. 9, no. 4, pp. 335–351, Dec. 2004, doi: 10.1023/B:EMSE.0000039883.47173.39.
[80] S. Kleinschmager and S. Hanenberg, “How to rate programming skills in programming experiments?: a preliminary, exploratory, study based on university marks, pretests, and self-estimation”.
[81] C. Bunse, “Using patterns for the refinement and translationof UML models: A controlled experiment,” *Empir. Softw. Eng.*, vol. 11, no. 2, pp. 227–267, Jun. 2006, doi: 10.1007/s10664-006-6403-7.
[82] S. Biffl and W. Grossmann, *Evaluating the accuracy of defect estimation models based on inspection data from two inspection cycles*. 2001, p. 154. doi: 10.1109/ICSE.2001.919089.
[83] J. E. Hannay, E. Arisholm, H. Engvik, and D. I. K. Sjoberg, “Effects of Personality on Pair Programming,” *IEEE Trans. Softw. Eng.*, vol. 36, no. 1, pp. 61–80, Jan. 2010, doi: 10.1109/TSE.2009.41.
[84] N. Ensmenger, “The Multiple Meanings of a Flowchart”.
[85] M. d/o Muniandi, N. A. binti M. Saad, and J. d/o M. @ Manickam, “Students’ Perception on RAPTOR Application Implementation in Problem Solving and Program Design,” *Stud. Percept. RAPTOR Appl. Implement. Probl. Solving Program Des.*, vol. 134, no. 1, Art. no. 1, Oct. 2023, Accessed: Apr. 21, 2025. [Online]. Available: https://ijrp.org/paper-detail/5520
[86] B. Calloni and D. Bagert, *ICONIC programming in BACCII vs. textual programming: which is a better learning environment?*, vol. 26. 1994, p. 192. doi: 10.1145/191033.191103.
[87] B. A. Calloni, D. J. Bagert, and H. P. Haiduk, “Iconic programming proves effective for teaching the first year programming sequence,” *ACM SIGCSE Bull.*, vol. 29, no. 1, pp. 262–266, Mar. 1997, doi: 10.1145/268085.268189.
[88] T. Crews and U. Ziegler, “The flowchart interpreter for introductory programming courses,” in *FIE ’98. 28th Annual Frontiers in Education Conference. Moving from “Teacher-Centered” to “Learner-Centered” Education. Conference Proceedings (Cat. No.98CH36214)*, Tempe, AZ, USA: IEEE, 1998, pp. 307–312. doi: 10.1109/FIE.1998.736854.
[89] T. Crews, “Using a Flowchart Simulator in a Introductory Programming Course”.
[90] T. Watts, “The SFC editor a graphical tool for algorithm development,” *J Comput Sci Coll*, vol. 20, no. 2, pp. 73–85, Dec. 2004.
[91] M. C. Carlisle, “RAPTOR: A VISUAL PROGRAMMING ENVIRONMENT FOR TEACHING OBJECT-ORIENTED PROGRAMMING”.
[92] M. C. Carlisle, T. A. Wilson, J. W. Humphries, and S. M. Hadfield, “Raptor: introducing programming to non-majors with flowcharts,” *J. Comput. Sci. Coll.*, vol. 19, no. 4, pp. 52–60, 2004, Accessed: Apr. 21, 2025. [Online]. Available: https://raptor.martincarlisle.com/raptor_paper.doc
[93] A. Scott, *Using Flowcharts, Code and Animation for Improved Comprehension and Ability in Novice Programming*, Ph.D. dissertation, Univ. of South Wales, 2010. [Online].Available: *https://pure.southwales.ac.uk/files/991736/Dr_Andrew_Scott_PhD_Thesis.pdf*
[94] “RAPTOR - Flowchart Interpreter.” Accessed: Apr. 22, 2025. [Online]. Available: https://raptor.martincarlisle.com/
[95] G. Atanasova and P. Hristova, “Flow chart interpreter: an environment for software animation representation,” in *Proceedings of the 4th international conference conference on Computer systems and technologies e-Learning -*

*CompSysTech '03*, Rousse, Bulgaria: ACM Press, 2003, pp. 453–458. doi: 10.1145/973620.973696.
[96] D. Hooshyar, R. Ahmad, M. Md Nasir, S. Band, and S.-J. Horng, "Flowchart-Based Programming Environments for Improving Comprehension and Problem-Solving Skill of Novice Programmers: A Survey," *Int. J. Adv. Intell. Paradig.*, vol. 7, Nov. 2014, doi: 10.1504/IJAIP.2015.070343.
[97] M. Andrzejewska and A. Stolińska, "Do Structured Flowcharts Outperform Pseudocode? Evidence From Eye Movements," *IEEE Access*, vol. 10, pp. 132965–132975, Dec. 2022, doi: 10.1109/ACCESS.2022.3230981.
[98] W. J. Dzidek, E. Arisholm, and L. C. Briand, "A Realistic Empirical Evaluation of the Costs and Benefits of UML in Software Maintenance," *IEEE Trans. Softw. Eng.*, vol. 34, no. 3, pp. 407–432, May 2008, doi: 10.1109/TSE.2008.15.
[99] C. Cabo, *Effectiveness of Flowcharting as a Scaffolding Tool to Learn Python*. 2018, p. 7. doi: 10.1109/FIE.2018.8658891.
[100] S. Xinogalos, *Using Flowchart-based Programming Environments for Simplifying Programming and Software Engineering Processes*. 2013. doi: 10.1109/EduCon.2013.6530276.
[101] R. Neild, "Common Guidelines for Education Research and Development".
[102] R. Rosenthal, *Meta-Analytic Procedures for Social Research*. 2455 Teller Road, Thousand Oaks California 91320 United States of America: SAGE Publications, Inc., 1991. doi: 10.4135/9781412984997.
[103] D. Moreau and K. Wiebels, "Ten simple rules for designing and conducting undergraduate replication projects," *PLOS Comput. Biol.*, vol. 19, no. 3, p. e1010957, Mar. 2023, doi: 10.1371/journal.pcbi.1010957.
[104] C. F. Camerer *et al.*, "Evaluating the replicability of social science experiments in Nature and Science between 2010 and 2015," *Nat. Hum. Behav.*, vol. 2, no. 9, pp. 637–644, Aug. 2018, doi: 10.1038/s41562-018-0399-z.
[105] Committee on Reproducibility and Replicability in Science *et al.*, *Reproducibility and Replicability in Science*. Washington, D.C.: National Academies Press, 2019, p. 25303. doi: 10.17226/25303.
[106] O. S. Gómez, N. Juristo, and S. Vegas, *Replications types in experimental disciplines*. 2010. doi: 10.1145/1852786.1852790.
[107] Open Science Collaboration, "Estimating the reproducibility of psychological science," *Science*, vol. 349, no. 6251, p. aac4716, Aug. 2015, doi: 10.1126/science.aac4716.
[108] C. Chambers, *The Seven Deadly Sins of Psychology: A Manifesto for Reforming the Culture of Scientific Practice*. Princeton University Press, 2019. Accessed: Apr. 21, 2025. [Online]. Available: https://www.perlego.com/book/859652/the-seven-deadly-sins-of-psychology-a-manifesto-for-reforming-the-culture-of-scientific-practice-pdf
[109] B. A. Nosek and T. M. Errington, "Making sense of replications," *eLife*, vol. 6, p. e23383, Jan. 2017, doi: 10.7554/eLife.23383.
[110] A. Bacchelli and C. Bird, "Expectations, outcomes, and challenges of modern code review," in *2013 35th International Conference on Software Engineering (ICSE)*, San Francisco, CA, USA: IEEE, May 2013, pp. 712–721. doi: 10.1109/ICSE.2013.6606617.
[111] M. Harman, S. A. Mansouri, and Y. Zhang, "Search-based software engineering: Trends, techniques and applications," *ACM Comput. Surv.*, vol. 45, no. 1, pp. 1–61, Nov. 2012, doi: 10.1145/2379776.2379787.

[112] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering*. Berlin, Heidelberg: Springer, 2024. doi: 10.1007/978-3-662-69306-3.
[113] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*, Third edition. Cambridge, Massachusetts London, England: MIT Press, 2009.
[114] N. Peitek *et al.*, "Correlates of programmer efficacy and their link to experience: a combined EEG and eye-tracking study," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Singapore Singapore: ACM, Nov. 2022, pp. 120–131. doi: 10.1145/3540250.3549084.
[115] J. Siegmund and J. Schumann, "Confounding parameters on program comprehension: a literature survey," *Empir. Softw. Eng.*, vol. 20, no. 4, pp. 1159–1192, Aug. 2015, doi: 10.1007/s10664-014-9318-8.
[116] M. Cauchoix *et al.*, "The repeatability of cognitive performance: a meta-analysis," *Philos. Trans. R. Soc. B Biol. Sci.*, vol. 373, no. 1756, p. 20170281, Aug. 2018, doi: 10.1098/rstb.2017.0281.
[117] G. Moreno, "Mook, Douglas.G. (1996). Motivation: The Organization of Action. Nueva York: Norton.," *Rev. Psicol.*, vol. 20, pp. 156–159, Jun. 2002, doi: 10.18800/psico.200201.008.
[118] R. B. Cattell, *Abilities: their structure, growth, and action*. Boston, Mass.: Houghton Mifflin, 1971.
[119] D. Feitelson, *Considerations and Pitfalls in Controlled Experiments on Code Comprehension*. 2021, p. 117. doi: 10.1109/ICPC52881.2021.00019.
[120] M. Lichtman, *Qualitative Research in Education: A User's Guide*, 4th ed. New York: Routledge, 2023. doi: 10.4324/9781003281917.
[121] C. Sun, S. Yang, and B. Becker, "Debugging in Computational Thinking: A Meta-analysis on the Effects of Interventions on Debugging Skills," *J. Educ. Comput. Res.*, vol. 62, no. 4, pp. 867–901, Jul. 2024, doi: 10.1177/07356331241227793.
[122] M. P. O'Brien, "Software Comprehension – A Review & Research Direction".
[123] "manual-tobii-pro-x3-120_23122019.pdf." Accessed: Apr. 25, 2025. [Online]. Available: https://www.staff.universiteitleiden.nl/binaries/content/assets/sociale-wetenschappen/faculteitsbureau/solo/research-support-website/equipment/manual-tobii-pro-x3-120_23122019.pdf
[124] "Tobii Customer Portal." Accessed: Apr. 22, 2025. [Online]. Available: https://connect.tobii.com
[125] "Dry EEG Headset | Quick-20r," CGX. Accessed: Apr. 25, 2025. [Online]. Available: https://www.cgxsystems.com/quick-20r-v2
[126] R. S. Hessels, D. C. Niehorster, C. Kemner, and I. T. C. Hooge, "Noise-robust fixation detection in eye movement data: Identification by two-means clustering (I2MC)," *Behav. Res. Methods*, vol. 49, no. 5, pp. 1802–1823, Oct. 2017, doi: 10.3758/s13428-016-0822-1.
[127] "Home — PsychoPy v2025.1.0." Accessed: Apr. 22, 2025. [Online]. Available: https://www.psychopy.org/
[128] "SoSci Survey ▸ Onlinebefragung, DSGVO-konform, deutsches Unternehmen." Accessed: Apr. 22, 2025. [Online]. Available: https://www.soscisurvey.de/
[129] U. Herwig, P. Satrapi, and C. Schönfeldt-Lecuona, "Using the International 10-20 EEG System for Positioning of Transcranial Magnetic Stimulation," *Brain Topogr.*, vol. 16, pp. 95–9, Feb. 2003, doi: 10.1023/B:BRAT.0000006333.93597.9d.

[130] B. Coffman *et al.*, *Using independent components analysis (ICA) to remove artifacts associated with transcranial direct current stimulation (tDCS) from electroencephalography (EEG) data: A comparison of ICA algorithms*, vol. 7. 2013. doi: 10.1016/j.brs.2014.01.025.
[131] M. Ullsperger and S. Debener, “Simultaneous EEG and fMRI : Recording, Analysis, and Application,” Apr. 2010, doi: 10.1093/acprof:oso/9780195372731.001.0001.
[132] A. Widmann, E. Schröger, and B. Maess, “Digital filter design for electrophysiological data – a practical approach,” *J. Neurosci. Methods*, vol. 250, pp. 34–46, Jul. 2015, doi: 10.1016/j.jneumeth.2014.08.002.
[133] “Why Many Psychology Studies Fail to Replicate,” Verywell Mind. Accessed: Apr. 27, 2025. [Online]. Available: https://www.verywellmind.com/what-is-replication-2795802
[134] “Threats of a Replication Crisis in Empirical Computer Science – Communications of the ACM.” Accessed: Apr. 28, 2025. [Online]. Available: https://cacm.acm.org/research/threats-of-a-replication-crisis-in-empirical-computer-science/