Through (Tracking) Their Eyes: Abstraction and Complexity in Program Comprehension

PHILIPP KATHER, University of Münster, Germany
RODRIGO DURAN, Federal Institute of Mato Grosso do Sul, Brazil and Aalto University, Finland
JAN VAHRENHOLD, University of Münster, Germany

Previous studies on writing and understanding programs presented evidence that programmers beyond a novice stage utilize plans or plan-like structures. Other studies on code composition showed that learners have difficulties with writing, reading, and debugging code where interacting plans are merged into a short piece of code. In this article, we focus on the question of how different code-composition strategies and the familiarity with code affect program comprehension on a more abstract, i.e., algorithmic level. Using an eye-tracking setup, we explored how advanced students comprehend programs and their underlying algorithms written in either a merged or abutted (sequenced) composition of code blocks of varying familiarity. The effects of familiarity and code composition were studied both isolated and in combination. Our analysis of the quantitative data adds to our understanding of the behavior reported in previous studies and the effects of plans and their composition on the programs' difficulty. Using this data along with retrospective interviews, we analyze students' reading patterns and provide support that subjects were able to form mental models of program execution during task performance. Furthermore, our results suggest that subjects are able to retrieve and create schemata when the program is composed of familiar templates, which may improve their performance; we found indicators for a higher element-interactivity for programs with a merged code composition compared to abutted code composition.

CCS Concepts: • **Social and professional topics** → **Computer science education**; *Model curricula*; Student assessment;

Additional Key Words and Phrases: Eye tracking, program comprehension, plan-composition strategies, plans, qualitative content analysis

ACM Reference format:
Philipp Kather, Rodrigo Duran, and Jan Vahrenhold. 2021. Through (Tracking) Their Eyes: Abstraction and Complexity in Program Comprehension. *ACM Trans. Comput. Educ.* 22, 2, Article 17 (November 2021), 33 pages.
https://doi.org/10.1145/3480171

This work was partially supported by Bundesministerium für Bildung und Forschung grant number 01PB18007A and Conselho Nacional de Desenvolvimento Científico e Tecnológico grant number 201365/2015-4.
Authors' addresses: P. Kather and J. Vahrenhold, University of Münster, Münster, Einsteinstraße 62, 48149 Münster, Germany; emails: {philipp.kather, jan.vahrenhold}@uni-muenster.de; R. Duran, Federal Institute of Mato Grosso do Sul, Campo Grande, Rodovia MS–473, km 23, s/nº, Fazenda Santa Bárbara, Nova Andradina, MS, Brazil and Aalto University, Aalto, Finland; email: rodrigo.duran@ifms.edu.br.

1946-6226/2021/11-ART17 $15.00
https://doi.org/10.1145/3480171

1 INTRODUCTION

Programming is a multi-faceted skill. It is widely agreed that programming is more than code writing and encompasses other essential skills such as program comprehension [25, 38, 42, 73, 74], designing, and testing [80]. The aspect we focus on in this article, program comprehension, is emphasized in multiple pedagogical approaches to introductory programming [31, 35, 48, 62, 81], but is also relevant for experts such as professional software developers [34, 69]. Previous studies explored distinct aspects of program comprehension [4, 5, 37, 54, 60] and how students move from textual elements to higher levels of abstraction, and how elements in the program represent overarching goals. Common to all is the notion that comprehension can be described as the development of a (subjective) mental representation of the program [61, p. 66].

Research on program comprehension is heavily influenced by research on human cognition, especially Chunking [12, 45] and Schema Theory [58], which describes how students deal with a large number of concepts and integrate them to form a mental model of the dynamic aspects of program execution.

Central to comprehension is the conceptualization of *plan-schemata*, which was introduced in early **Computing Education Research (CER)** studies (e.g., References [56, 65–67]). *Plan-schemata* serve as a *"library of stereotypical solutions to problems as well as strategies for coordinating and composing them"* [65, p. 850]. Throughout this article, we refer to the following terminology:

- **Plan-schemata** describe (mental representations of) typical solutions to problems, such as finding the largest number in a field or a guard protecting a code section from execution [67].
- **(Standard) Plans** are realizations of a plan-schema in a programming language and therefore refer to code sections. What makes a plan standard varies based on instruction and context. In this work, we consider plans such as *guard, average, counter, sentinel*, as examples of standard plans, extracted from Soloway and Ehrlich [67].
- **Tailored plans**, following the definition of Soloway, refer to modified plans, e.g., using a FindMaximum-plan on an array while storing the index of each candidate [65, p. 856].
- **Unplan-like code** refers to code sections that do not correspond to any plan-schemata.

Literature on how students compose code sections has described several strategies students use: a plan-composition strategy [65]. For example, students might *sequence* (abut) plans, where plan execution is independent or one plan output serves as input to the following plans. Another strategy is to *merge* plans in one code section, where plan execution is interleaved with other plans. An example would be FindMaximum-plan and a FindMinimum-plan sharing the same loop code. It has been reported that students are more likely to produce flawed code if they use merged plan-compositions [16, 67], but consider it to be "better" regardless [17]. It has been argued [15] that sequenced programs may be less difficult to comprehend than merged ones due to lower element interactivity [72], i.e., plans that need to be simultaneously processed by learners. Therefore, it stands to reason that sequenced composition is beneficial not only for code writing [17], but also for comprehension.

We are interested in whether there is a direct influence of plan-schemata and the composition of code sections on code comprehension. While a positive effect of plans in programming, as opposed to non-familiar code, was reported before (see, e.g., Reference [67]), there is no direct investigation of the effects of plans, tailored plans, unplan-like code, and the composition of such code sections on learners' comprehension behavior. To alleviate this gap, we conducted a study with a purposive sample of participants carefully selected to ensure they had plan-schemata at their disposal. Combining eye-tracking and retrospective interviews, we studied the following research questions:

- RQ 1: *How does the availability of plan-schemata affect program comprehension?* Based on the literature, we hypothesized that plans are faster to understand than tailored plans, and these are faster to understand than unplan-like code.
- RQ 2: *How does the composition of code sections affect program comprehension?* Based on the literature, we hypothesized that merged code sections—as opposed to sequenced code sections—are more difficult to comprehend. They would require simultaneous processing of all involved sections and hence resulting in more switches between different sections. Furthermore, we expect that these switches impact the time it takes to comprehend a program.

The contribution of this article is threefold: First, we provide insights into how the process of program comprehension can be affected by plan-schemata. Second, we investigate how the composition of such solutions affects program comprehension. We do so by using innovative investigation methods using eye-tracking data. Instead of focusing on saccades and transitions between keywords or elements of syntax, we focus on more meaningful chunks of information by adopting plans as the unit of measurement. This methodology facilitates the analysis, as it requires less accurate eye-tracking methods and provides more meaningful information about the program comprehension process. Consequently, our third contribution is a setup for a study design: Using the model-building assumption (see Section 3.4) and its verification, it allows us to apply a combination of non-invasive sampling of high-quality eye-tracking data enriched with insights from retrospective interviews [18].

2 RELATED WORK

Plans. Research in the 1980s showed that most of the novices' difficulties originated not from syntax errors, but from difficulties in structuring interconnecting parts of the program, or "putting the pieces together," i.e., composing and coordinating components of a program [65, 66, 68, 70]. Inspired by Schema Theory [58], Soloway and colleagues proposed a novel perspective to investigate difficulties in programming instruction: They argued that "learning to program amounts to learning how to construct mechanisms and how to construct explanations" [65]. The building blocks of programming expertise are plans, essentially schemata in the programming domain, which can be further adapted to solve similar problems using what Soloway described as tailored plans [65, p. 856]. Differences in programming expertise could be attributed to experts having built up large libraries of stereotypical solutions to problems as well as strategies to use them. Rist [56] showed that when schemata are available, both novices and experts resort to higher-level strategies, moving from abstract plans to concrete code (top-down). However, when schemata are not available, both novices and experts resort to a bottom-up strategy.

Program comprehension. Various studies provide evidence that programming skills are organized as a hierarchy [11, 25, 38, 39, 42, 73, 74], where students progress from reading to tracing, summarizing, and finally writing code. Activities involving some of such skills have been generally described as program comprehension tasks [30, 47, 48, 52], i.e., *any activity that supports an individual to construct his or her mental model of a program by interacting with an artifact representing the code* [31]. The BRACElet project [10] findings suggest that a distinction between novices and experts is the ability to abstract from the code and summarize it into a concise description of the program goal and that this ability also correlates positively with code writing skills.

Program comprehension models, such as Pennington's model [54, 55] and the Block Model [60, 61], often emphasize the distinction between textual elements of the code and more abstract representations of goals, such as plans. Such models describe a mental model approach to represent the dichotomy between code and abstractions representing program execution and the goals of parts of the program. Wiedenbeck and Ramalingam [79] show that different programming behaviors could

lead to different mental representations. For example, novices comprehending object-oriented programs tend to develop strong mental representations of the goals of programs' pieces, but weak in terms of data flow and superficial features of the program. Novices, however, develop stronger mental representations of surface features of the code when comprehending procedural programs.

Plan-composition strategies. Plan and goals analysis decoupled the strategies used to solve a problem from its concrete code representation, allowing the comparison of different solutions across programming languages and freeing researchers from the minutiae and specifics of each programming language. Theoretically, plans would allow comparing different problems (and their solutions) based on the similarity of their plan-goal structure [65]. However, there are different ways to "glue together" these plans—a plan-composition strategy that will have a impact on the actual code produced [65]. In general, research has identified three main plan-composition strategies [65, 70]: Abutted or sequenced plan-composition, where plans are presented in sequence, one after the other; Nested plan-composition, where one plan completely surrounds another plan; and Merged plan-composition, where at least two plans are interleaved.

Plan-composition strategies play a crucial role in learning to program. Some studies suggested that students find merged or interleaved programs hard to write or comprehend [19, 65, 70]. There is also evidence that students wrongly concatenate plans instead of merging them and frequently struggle with plans that interact in complex ways [24]. Students sometimes read merged plans as if they were composed sequentially [26], and there is evidence that students' plan composition strategy may lead them to write defective code [16]. Also, there is some evidence favoring a sequenced-plans approach, which may lead to better outcomes when writing programs [66].

Eye-tracking. A pioneer study using eye-tracking in program comprehension was presented by Crosby et al. [13] in 1990. In eye-tracking studies, participants usually work in a highly controlled environment while their eyes are being tracked via an infra-red sensor, which allows computing what their eyes focus on. However, it remained a rarely used methodology [50], even though it can provide insights into processes that are hard to archive using other methods. Just and Carpenter [33] suggest that fixations of the eye indicate cognitive processing and that gaze duration reflects comprehension processes. Recent empirical studies showed connections between cognitive demand and gaze duration [9] and elaborated how other eye-tracking metrics can be used to measure cognitive load [82]. Because of its non-invasive character and the rich insights eye-tracking data can generate, we chose eye-tracking to investigate our research questions.

Previous eye-tracking studies investigated aspects of program comprehension [50, 63]. Differences in natural language and code comprehension were investigated using eye-tracking methods to explore successful reading strategies [6, 57] and strategies to identify bugs [64, 75, 76]. Jbara and Feitelson [32] found that programmers spent less time on repeated code, indicating that the effects of familiarity affect their reading behavior. Studies used the time spent on input-examples to suggest that programming patterns may play a role in solving algorithmic problems [51]. Abid et al. divided code into sections fulfilling a small purpose and leveraged those to compare code summarizing behavior of novices and experts [1]. These code sections, however, did not necessarily have to be familiar to participants and therefore did not qualify as plans. Peitek et al. [53] recently found that linearity of code (e.g., method calls vs. in-line code) and semantic cues have a more significant impact on code reading order than experience and comprehension strategy. Our work continues to investigate this trend in eye-tracking studies, but instead focusing on how the familiarity of code sections and their composition, as opposed to surface-text features, can impact comprehension behavior.

3 METHODOLOGY

For this study, we developed a set of programs based on theoretical considerations, designed "tracing-tasks" based on these, and assessed different aspects of their quality in pilot studies (see Section 3.2 and Section 3.3). We assumed that participants would develop a mental model of the program while working on the first input and later use such mental model in subsequent input-to-output conversions (see Section 3.4). This assumption allowed for a more robust analysis of the eye-tracking data (see Section 3.5 and Section 3.6). We complemented the quantitative analysis of the eye-tracking data by conducting and analyzing retrospective interviews [18] in which we showed the participants a visualization of their gazes and prompted them to comment on it, using deductive qualitative content analysis [43] (see Section 3.7).

3.1 Participants

By the nature of our research questions, participants were required to fulfill the following inclusion criteria: (1) Non-novice status regarding programming and (2) demonstrated experience with writing programs in Java. This way, we ensured that participants possessed some plan-schemata. For this, we used a purposive sample. To ensure that participants indeed had shown proficiency in programming classes, instructors and teaching assistants were asked to recommend participants that fit the inclusion criteria. We acknowledge that this skewed the sample towards high-performing students. While this is in line with the sought expert status, we carefully reflect on this bias when attempting to generalize results beyond the context of this study. Participation in this study was compensated for according to the institutions' standards for trial compensation.

Overall, we recruited 17 subjects for this study. All subjects were either graduate students or had started working in the industry recently and possessed a CS degree. The age range was 22–29 years. Four participants identified themselves as female (a similar ratio of the CS degrees' population in the participants' institutions). We did not record any other demographic data.

3.2 Material

Most programs used in this study were used in previous studies [20–22, 40, 65] to cover a range of difficulties. They reflect a variety of levels in which the sections of the code interact and options to represent the programs in merged or sequenced code composition.

3.2.1 Description of the Programs Used in the Study. We used short programs to avoid unnecessary distractions that may confound the eye-tracking analysis later. Therefore, no program contained checks for empty or invalid input. In each of our experiments, the participants were informed about this and were assured that all inputs presented were valid and unambiguous. We designed four pairs of programs with the goal in mind that the difficulty of understanding the programs in a pair as well as the time to mentally simulating the programs in each pair should be comparable—in Section 3.3, we discuss how we performed such evaluation. Variable names were not obfuscated deliberately but did not carry semantic information, either.

To give a flavor of both sequenced and merged code-compositions, we describe Maximum Count, Rainfall, and Omega in their sequenced version and the rest of the programs in their merged version. A complete description of all programs is available in the Appendix (see Figure 5–7). The first pair was Minimum Sum and Maximum Count, both consisting only of plans.

Minimum Sum (sequenced version): Given an array of integers, this program adds the minimum of all elements to the sum of all elements and returns this value. For this, it uses two (standard) plans realized as two separate `for`-loops: the first loop adds all numbers in the input, the second computes the minimum value in the input. It then adds the sum to the minimum and returns this value.

Maximum Count (merged version): Given an array of integers, this program subtracts the maximum of all numbers from the number of values divisible by three. For this, it uses two plans merged into one `for`-loop. The first plan counts the numbers divisible by three in the input, the second computes the maximum. It returns the difference between the maximum and the count.

The second pair was Rainfall and Rainfall Partner, both containing a tailored plan.

Rainfall (sequenced version): Given an array of integers, this program solves a modified (and simplified) version of Soloway's Rainfall problem [65] in which the aggregate of all negative numbers up to the first occurrence of the number 999 is computed. For this, the program uses three `for`-loops. The first loop creates a list containing all elements in the input up to the first occurrence of the number 999. The second loop filters this list and produces a list containing only the negative numbers. Finally, the third loop computes the sum of all elements in the filtered list and returns this value.

Rainfall Partner (merged version): We designed this program to match Rainfall concerning the plans used and the assumed difficulty to mentally simulate it. Given an array of integers, this program subtracts all positive numbers starting from the first occurrence of 100. For this, the program maintains a variable r (initialized with 100) and uses a single `for`-loop over the input array. Inside this loop, the program executes an `if-else`-statement: If the first occurrence of 100 is encountered, then it sets a flag and, from there on, always executes the `else`-branch. In this branch, it checks whether the current number is positive; if so, then it subtracts it from r. After the last iteration, the program returns the value of r.

The third pair was Omega and Balanced Subset, both containing unplan-like code.

Omega (sequenced version): This program was part of a solution for Ginat's "Omega" problem [20, 21]. The task here is to find the maximum-distance pair of values in a given array such that the left value is larger than the right value. To solve this task, the program proceeds in four steps (see Reference [21] for details): The first step was implemented using a `for`-loop to greedily compute the (indices of) a monotonically increasing sequence of entries in the input starting with the last element in the input. The second step uses a `for`-loop to greedily compute the (indices of) a monotonically increasing sequence of entries in the input starting with the first element in the input. To reduce the tracing difficulty of this task, we presented to students the state of the program before the second code section and asked them to compute the desired output from this point onward. Preempting the discussion of the eye-tracking results, we mention already at this point that our participants indeed did not look back (much) to the first section. The third code section uses two nested `for`-loops for a synchronized scan over the two lists (see Reference [21]); in this scan, the distances between the elements of each candidate pair are computed. The final section then finds the maximum of those distances and returns it. In our implementation, the list containing the indices of a monotonically increasing sequence (see above) was provided as the input as well.

Balanced Subset (merged version): The program was part of a solution for a problem presented by Ginat and Blau [22] (see Figure 3).

The original problem is to find, for a given array A of zeroes and ones, the length of the longest sub-array containing the same number of zeroes and ones. To match the task structure of the Omega-problem discussed above, we also presented the participants with the result of the first step of the algorithm. In this step, each entry $A[i]$ was replaced by the excess or deficit of ones in the original entry $A[0 \ldots i]$. The participants were asked to consider the following code sections: The first section uses a `for`-loop to find the index of the last occurrence of a zero in the "excess array," i.e., the length of a balanced sub-array starting at the beginning of the input. The second section uses another `for`-loop to iterate over the "excess array" and, across each of the values

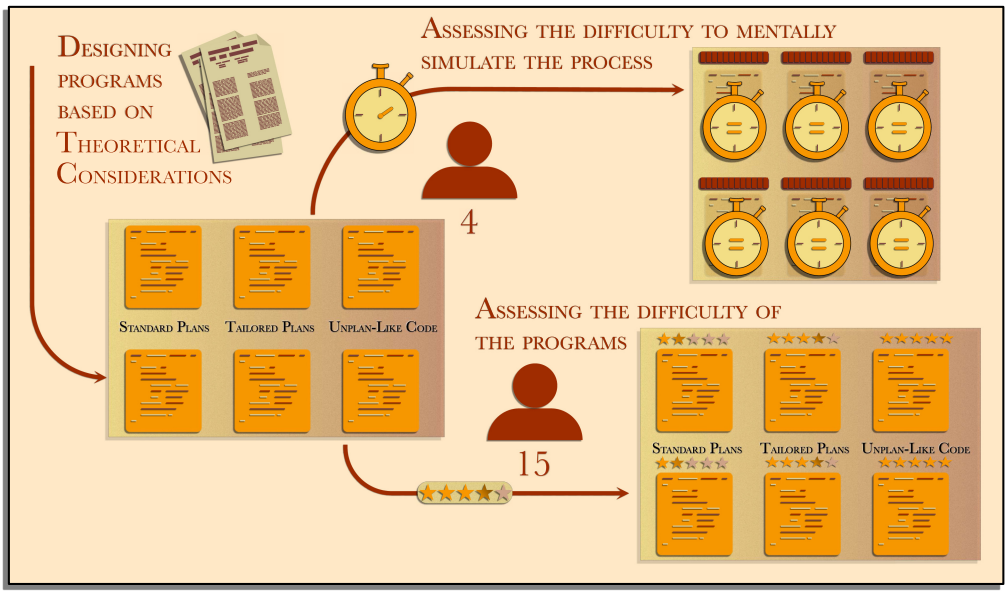

Fig. 1. Visualization of material development: **Left part:** Program candidates including standard plans, tailored plans, and unplan-like code were developed based on theoretical considerations. **Upper right part:** To find inputs that took a similar amount of time to mentally simulate with the program, a high-level summary of each program was handed to participants. The participants were then asked to compute the output of the program. **Lower right part:** Participants were presented to all programs in either a merged or a sequenced version and prompted to rank them according to the difficulty to comprehend them. This ranking was used to confirm the theoretical considerations and mitigate differences within the program pairs.

encountered, record the longest distance to another excess entry with the same value. The third section then returns either this maximum or the length of the longest sub-array starting at the beginning of the input and ending with a zero, whichever is the larger value.

3.3 Pilots

Because the number of participants that can be recruited for an eye-tracking study is very limited, the quality of the material used and the experimental setting is of paramount importance. Hence, we assessed the quality of the material in multiple ways (see Figure 1 for an overview of our process). Besides the assessment of the material that is presented here in detail, we also confirmed the clarity of the instructions for the participants, the recording quality, and the experimental environment using agreed-upon standards [63].

Assessing the difficulty to mentally simulate the process. We used the framework of Duran et al. [15] to differentiate and evaluate the complexity and difficulty of comprehending programs (see the left part of Figure 1). Program's characteristics such as the kind of programming constructs it manipulates, how instruction presented the behavior of such constructs to learners, and how constructs are structured in the code define an absolute, learner-independent metric of program *complexity*. Different learners will display distinct performances when comprehending a given program based on their prior knowledge in the programming domain. Therefore, the same program might yield different *difficulties* even when having the same complexity. For example, according to the framework of Duran et al., comprehending a program that searches for the minimum value in a given input array and a program computing the sum of all values stored in this input would demand a similar cognitive effort from learners to be comprehended, providing learners' have

comparable expertise. However, mentally simulating these programs could lead to rather different results, depending on their inputs. We thus conducted a pilot study to understand and compensate for such effects.

In this pilot study, we aimed to understand how comparable were the processing times for similar problems (see upper-right part of Figure 1). To estimate possible difficulties in the mental simulation, we recruited four students that match the inclusion criteria; however, none of these students participated in the final study. We presented each participant with high-level summaries of sections discussed in Section 3.2.1. To avoid any unwanted effects such as misunderstanding the program, no code but sub-goal labels were provided. For instance, a summary for the second code section in the Balanced Subset problem was *"Considering all elements in the input that have a repeated entry, calculate the maximum distance between the indexes of each repetition for all repeated entries."* For each problem, consisting of several steps, the participants were shown a worked example of how each step contributes to processing the input. Once a participant had provided the correct answer for an example input and was confident to have understood the description, we asked him/her to compute the results for 12 different inputs as quickly as possible. The time for each task was measured.

We repeated this process, adjusting the input for the programs based upon feedback from previous rounds until we were confident that all inputs were adequate and no clear difference in "tracing time" could be detected between pairs of problems (which we added their corresponding code in the next steps of the pilot) we intentionally designed to have similar complexity.

Assessing the difficulty of the programs. To assess the relative difficulty of the programs used in our study, we performed another pilot study with 15 teaching assistants; none participated in the final eye-tracking experiment (see lower left part of Figure 1). Eight of these rated the programs in their merged form and seven rated the programs in their sequenced form. Both groups were asked to rate the difficulty to comprehend the programs using a Likert scale (−7 to +7) while assuming the difficulty of the Rainfall problem as the baseline (0). The instructions explicitly allowed programs to be rated identically. We converted the ratings into rankings [27] and computed their inter-rater reliability. Both groups had at least seven members, therefore the approximation to the chi-squared distribution was valid; we thus can interpret the p-values [36]. *Kendall's W* showed a statistically significant agreement in the rankings for both the sequenced ($W = .71, p < .0005$) and merged ($W = .87, p < .0005$) versions.

The agreed-upon ranking of the sequenced versions of the programs was (in increasing order of difficulty) Minimum Sum, Maximum Count, Rainfall, Rainfall Partner, Balanced Subset, and Omega. For the merged versions, Maximum Count and Rainfall, as well as Balanced Subset, changed their relative order. Therefore, no program was rated more than two ranks apart from its intended pair. In each pair (averaged for all raters), the ranks of two programs differed by strictly less than 1.5 ranks. Since this rating did not contradict the pairing described above, we used (Minimum Sum, Maximum Count), (Rainfall, Rainfall Partner), and (Balanced Subset, Omega) as companion programs.

The pilots confirmed the theoretical considerations: Programs we deemed to be more complex were perceived as more difficult. Additionally, pairs of programs were perceived to be on a similar level of difficulty. Furthermore, for sufficiently comprehended programs, the "tracing-tasks" took a similar amount of time. Hence, differences within pairs of programs may be attributed to code composition (merged/sequenced) or familiarity with parts of code (plans).

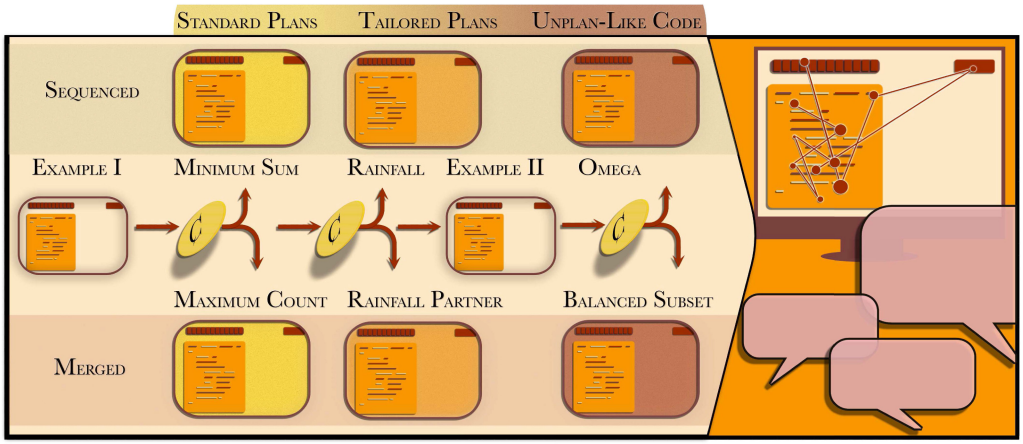

Fig. 2. Study design. Each participant was familiarized with an example before working on the programs. Because the last two programs contained a program section that was not relevant for the task, all participants were presented a second example to familiarize themselves with this setting. **Left part:** Eye-tracking tasks progressing from example programs, programs with plans, programs with tailored parts, and programs with unplan-like code (from left to right). Programs in the upper part are presented in sequenced composition, programs in the lower part are presented in merged composition. The order of plan-composition was randomly assigned (C). **Right part:** Retrospective interviews. The order was randomized for each participant, but the first two programs always consisted only of standard plans, the second two programs always contained standard and tailored plans but no unplan-like code, and the last two programs consisted of all three kinds of code sections.

3.4 Model-building Assumption

We built our study design on the following model-building assumption:

Model-Building Assumption: *While performing the first input-to-output conversion for a program, readers build a mental model of the program and utilize it on the subsequent tasks.*

The model-building assumption, if found to hold, would allow an increase in the quality of eye-tracking data by choosing a short but meaningful interval of eye-tracking data to study, where participants were highly engaged with the intended process and nothing else. As part of our study, we collected and analyzed data concerning validating this assumption.

3.5 Study Design

Our main study consisted of two phases (see Figure 2). In the first phase, participants were successively shown programs along with inputs and were asked to provide the output generated by the program based on the input (see Figure 3). The programs were grouped as described in Section 3.2. We randomized the order (merged-first or sequenced-first) within each pair of programs.

In their textbook on eye-tracking, Holmqvist et al. [29] warn that eye-tracking studies have many confounding factors tied to individual characteristics. To avoid confounding human factors, we thus used a within-subject design for our study. The independent variables were the code composition (sequenced or merged) and the familiarity of the code (plans, tailored plans, or unplan-like code), while the dependent variables were derived from the time the participants' eyes focused on areas of interest and the number of transitions between certain areas.

In the second phase, participants were presented with a screen recording in which the traces of their eye-tracking data were overlaid on top of the respective tasks they had performed during the

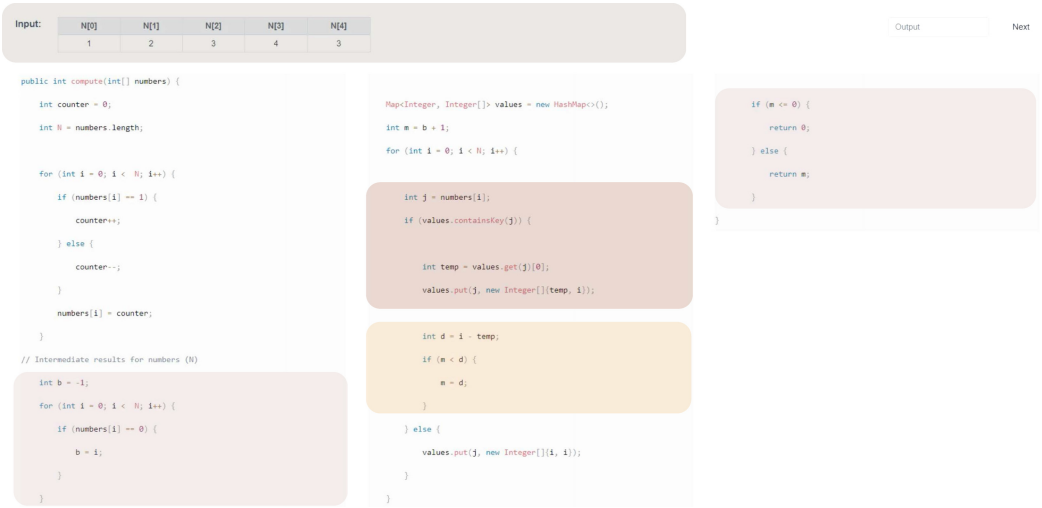

Fig. 3. Areas of interest (AOIs) for the merged version of Balanced Subset.

first phase. They were prompted to comment on their behavior and were asked to describe their behavior during prominent movement phases, such as switching between two code sections, for a better understanding of the eye-tracking data. They were also prompted to comment on how familiar the program felt and to summarize the program briefly.

Eye-tracking. The data was collected in two eye-tracking labs set up according to the recommendations for collecting rich data by Holmqvist et al. [29, pp. 17–21].[1] Both labs were equipped with a state-of-the-art Tobii Pro Spectrum 300 Hz eye-tracker, using the Tobii Pro Lab as recording software. We followed established procedures for seating the participant, calibrating the system, and obtaining high-quality data [29, pp. 116–128]. Per standard procedure, only one participant and a researcher were in the lab simultaneously; the researcher did not interact with participants during the experiment.

To present the programs and their input, we developed a web application showing the material, following recommendations regarding the placement of elements on the screen [63]. Figure 3 shows an example screenshot in which the areas of interest are highlighted.[2] See Section 3.6.1 for how the AOIs were chosen. This environment was in line with recommendations from Sharafi et al. [63, pp. 3144–3155]. The input was presented in an indexed form at the top left corner of the screen to avoid unnecessary attention pulled to that area [63, p. 3144]. An input field for entering the result of the computation was always located at the top right corner. Depending on the length of the program, up to three columns of code were rendered; we ensured that each block of code was presented in one column only. The vertical spacing of lines and blocks of code was chosen with the margins of the areas of interest for the eye-tracking analysis in mind while making the code look natural, as seen in Figure 3.

Previous studies show that allowing participants to familiarize themselves with the environment is beneficial for the experiment [1]. We thus started each session with a phase in which the participants could familiarize themselves with the software, hardware, and the type of tasks to be

[1]We collected the data in two-eye tracking labs, one lab at each participating institution.
[2]Note that these areas were not visible to the participants. Screenshots of all programs, including the used areas of interest, can be found in the Appendix.

performed. A researcher was present to answer questions related to both the application and the task. In the first phase of each experiment, the participant was shown the six programs Minimum Sum (sequenced), Maximum Count (merged), Rainfall (sequenced), Rainfall Partner (merged), Omega (sequenced), and Balanced Subset (merged); within each pair of programs, the order was randomized (see Figure 2). The participants were presented one program at a time with the option to pause between the experiment, e.g., for drinking or if they felt tired.

For each program, we prepared a progression of input-to-output conversion tasks. To make experiments comparable across participants, we limited the time allowed for each task. This limit was derived from the average time needed to mentally simulate the process (see Section 3.3) plus some extra time for comprehending the program in the first place. All programs were presented with the instruction to process as many inputs as possible. The competitive subtext of this task likely engaged the participants and triggered a comprehension process to solve the tasks as fast as possible. To allow participants to relax and focus again, these instructions were repeated before each program. This first phase of each experiment took 30 to 35 minutes per participant.

Interview Setup. After the first phase, we conducted semi-structured reflective interviews. Participants were shown recordings of their gaze movements. For each program, they were asked in a semi-structured interview to explain their thoughts and to comment both on what they did and on the program they were working on. Gaze movements identified as interesting by the researcher during eye-tracking were discussed to gather more detailed impressions. If participants did not comment on their own recording, then we prompted them for each program to describe the program as they understood it, briefly summarize it, and comment on aspects they found familiar or unusual.

3.6 Analysis

In this subsection, we first present the analysis of the eye-tracking data. Then, we explain how we coded and analyzed the qualitative data gathered during the retrospective interviews.

3.6.1 Eye-tracking. In line with standard eye-tracking methods, we analyzed fixation times for **areas of interest (AOIs)**. AOIs are non-overlapping portions of the screen defined by the researchers and separated by sufficiently wide margins such that slight tracking inaccuracies do not cause false-positive identifications of other AOIs [29, pp. 187–192]. Depending on the analysis (see below), we defined the AOIs to encompass blocks of code or the area in which the input was presented. Participants were oblivious to both the AOIs and our analysis goals. As it is customary, we analyzed the number of transitions between different AOIs and the dwelling time within each AOI.

Following established practice [29, pp. 140–143], we manually preprocessed the tracking data to ensure no offset errors occurred for AOIs. To avoid misinterpretation due to spotty data, we removed data traces with prominent gaps; in particular, we made sure that all data traces analyzed had tracking data for more than 80% of the time frame under inspection. These time frames captured the time between first seeing an input and submitting the output (or the time limit was reached). The duration of such a time frame ranged from several seconds to a few minutes.

In the following, we present the AOIs used in our analysis. By design, all programs shared a property that each line contained a declaration, an assignment, or a control structure (see the Appendix), so we could legitimately use the number of lines as a measure of a program's length. Thus, wherever possible, we sought to have AOIs containing the same number of lines of code. Because this was not always possible, we considered the normalized total dwelling time on each code section (plans, tailored plans, or unplan-like code) by dividing the total time participants fixated that section by its number of lines, as recommended by Sharafi et al. [63, pp. 3153–3154]. Doing so, we were able to identify which code section attracted (relatively speaking) the most attention.

Next, we present how these AOIs were used to investigate our research questions. We aimed at designing the program sections within each program to contain the same number of lines. For more details regarding the AOIs and programs, see the Appendix.

AOIs for code sections with sequenced code compositions. The Minimum Sum program contained two loops (and correspondent AOIs), each with the same number of lines. One loop computed the sum of elements in the input array, the other one extracted the minimum of all elements. We assumed that participants would have the corresponding plan-schemata at their disposal.

For Rainfall, we studied two AOIs. The first AOI contained a loop filtering the input to populate a list of all input elements up to the first occurrence of "999." This is an extension of a RunTotal-Loop [65], hence, we considered it to be a tailored plan. The second AOI contained a loop adding all elements in this list; a (standard) plan that can be assumed safely to be familiar to our non-novice participants. Both the (standard) plan and the tailored plan consisted of three lines of code each.

For Omega, we considered three AOIs. The first AOI contained four lines of code[3] describing a (standard) FindMaximum plan [65]. The second AOI contained seven lines of code for a tailored FindMaximum plan that also stored the indices of all candidates in a list. The third AOI contained eight lines of unplan-like code: It identified a maximum-distance tuple of values in the input such that the left value is larger than the right value. For this, two nested `for`-loops comparing indices and values were used. In contrast to what might be expected from two nested loops, the loop indices were updated in a way that resulted in an overall linear runtime [20]; given these features, we considered the code to be unplan-like.

AOIs for code sections with merged code composition. In programs with a merged code composition, two different code sections may share lines of code, such as control structures of a `for`-loop. To avoid overlapping AOIs that in turn could lead to misinterpretation of the dwelling time, we were careful to design the AOIs so each AOI contained only lines of code that could be attributed to exactly one code section. In particular, we did not assign shared control structures, e.g., of `for`-loops, to any AOI.

The Maximum Count program consisted of one `for`-loop containing two plans with the same number of lines of code. One (standard) plan computed a minimum and the other (standard) plan counted the numbers divisible by three. We consider these to be familiar plans.

For Rainfall Partner, we studied two AOIs. The first AOI contained the four lines of code for finding the first occurrence of "100" and then raising a flag. We consider this to be a tailored plan. The second AOI contained the three lines of code that subtracted all negative elements from 100. While this code effectively merges a Filter and a Subtract (standard) plan [65], it did so in a very obvious manner. Hence, we considered this code to be a (standard) plan.

The Balanced Subset program consisted of five code sections (see Figure 3), the first of which initialized the input array. Since we presented our participants with the input array, they did not need to look at this section, and we used only the four remaining code sections for AOIs. One tailored plan (left column) consisted of four lines and scanned an array for the last occurrence of "0." Another code section we considered to be a (standard) plan (right column) contained four lines of code that compute the maximum of two elements. The other two code sections are presented inside a `for`-loop controlling their execution (middle column). The first of these is a four-line, unplan-like code managing data in a hash map to find the first and last occurrence for each value stored. The other (standard) plan is a three-line code that computes the maximum of all differences

[3]We exclude closing brackets, as they do not add information. In fact, they were used to increase the separation between AOIs without making the code look unnatural.

between the current value and its first occurrence using a result from the preceding (upper) code section. Hence, only the code section in the middle column of Figure 3 was merged.

Verification of the model-building assumption underlying the study design. As mentioned before, our study design rests on the model-building assumption (see Section 3.4), which claims that readers build a mental model of the program during the first input-to-output conversion and utilize it on the subsequent tasks. We thus needed to verify this assumption first. Based on the model-building assumption, we would expect participants to spend a higher percentage of their time on the code during the task than during subsequent tasks. For this, we studied the following hypothesis:

H_A: *The ratio of time spent on the input to time spent on the code is highest during the first task.*

To investigate this hypothesis, we studied each participant's reading behavior using two AOIs per program: One AOI encompassed the input, the other AOI encompassed the program. We then computed the ratio of time spent dwelling on the code and the time spent dwelling on the input across the different inputs for each program. In these terms, spending a higher percentage of time on the input than on the code would imply that the code is less important for solving the task and this, in turn, would support the assumption that the reader indeed can resort to a mental representation of the program instead of having to re-read every single instruction of the program.

As we will discuss in Section 4.1, our analyses confirmed this hypothesis and thus supported the model-building assumption underlying our research design. We thus formulated hypotheses for our two research questions.

RQ 1: How does the availability of plan-schemata affect program comprehension? To investigate this research question, we formulated the following two hypotheses:

- $H_{1.1}$: *Within a program, participants spent more time on tailored plans than on plans.*
- $H_{1.2}$: *Within a program, participants spent more time on non-planlike code sections than on tailored plans.*

For $H_{1.1}$, we studied programs that contained plans and tailored plans, and for $H_{1.2}$, we studied programs that also contained unplan-like code sections. Because of the model-building assumption, we limited the investigation to the first task, since we assumed that during this step a mental model of the program was developed and therefore that phase was the most meaningful.

RQ 2: How does the composition of code sections affect program comprehension? To investigate the effects of the composition of program sections, we selected AOIs from each program and counted the number of transitions between these. Following standard procedure, eye movements were classified as transitions if two consecutive fixations occurred in different AOIs.

We assessed this research question by formulating two hypotheses. The first hypothesis we formulated was related to the conjecture that merged program composition leads to simultaneous processing of program sections and therefore more transitions between these. The second hypothesis we formulated was related to the conjecture that this additional processing time led to more time that needs to be spent on code:

- $H_{2.1}$: *The number of transitions between merged program sections is higher than the number of transitions between sequenced program sections.*
- $H_{2.2}$: *The ratio of time spent on the code relative to the input is higher for programs with a merged code composition than for programs with a sequenced code composition.*

Given the confirmation of the model-building assumption, we again limited the investigation to the first task. For $H_{2.2}$, we compared the pairs of programs as described in Section 3.2 (see also Figure 2). For $H_{2.1}$, we compared the number of transitions between these pairs and additionally

within Balanced Subset, because both, merged and sequenced program sections, existed in that program. The exact AOIs we compared were the following:

- Pair Minimum Sum/Maximum Count: For Minimum Sum (sequenced), the AOIs studied corresponded to a FindMinimum and a Sum (standard) plan. For Maximum Count (merged), the AOIs studied corresponded to a Count and a FindMaximum (standard) plan. In this pair, both programs were composed of very similar plans; thus, we assume the number of transitions to be mainly affected by the type of code composition.
- Pair Rainfall/Rainfall Partner: For the (merged) Rainfall Partner program, we considered transitions between the two code sections described in Section 3.2.1. For the (sequenced) Rainfall program, we considered the transitions between the code section extracting the sub-array from the beginning to the first occurrence of "999" and the code section filtering the positive values. For Rainfall, we considered the Sum (standard) plan as well. This (standard) plan interacted with the two other code sections, i.e., we counted interactions between three code sections as opposed to interactions between two code sections in Rainfall Partner. Due to the larger number of plans (and, thus, AOIs), a given approach to reading a program, e.g., a linear scan, would result in more transitions for the sequenced program than for the merged program. Since our research question was operationalized by hypothesizing that the sequenced program induced fewer transitions, this actually made it harder to confirm **H 2.2**. Therefore, we suggest a different reading strategy would have to be used when reading sequenced programs to induce fewer code transitions. We consider the code in both programs to correspond to plan-schemata to the same extent.
- Within Balanced Subset and Pair Balanced Subset/Omega: For the (merged) Balanced Subset program, we had identified four AOIs, two of which were merged inside a `for`-loop (two blocks in the middle column, Figure 3) and two of which were presented in sequenced code composition (left and right column, Figure 3). These were used to compare the effect of merged and sequenced code-composition *within* a program. As with Rainfall, we studied interactions between more sequenced code sections (section in the left and right column and the upper section in the middle column) than between sections in merged composition (two sections in the middle column), again penalizing **H 2.2**. In the same spirit, we studied two AOIs with a maximal number of transitions between them for the (sequenced) Omega program (a tailored plan finding all candidates for a maximum and an unplan-like code finding a maximum-distance tuple). We compared these transitions and the transitions between the plans in the middle column of Balanced Subset.

3.7 Qualitative Content Analysis

We enriched our analysis by collecting additional qualitative data to assist the interpretation of the eye-tracking data [71, p. 161] using retrospective interviews. In such interviews, participants are presented with a video of their eye movement and comment on it.

We chose deductive **qualitative content analysis (QCA)**, "a data analysis technique within a guided research process, and this research process is bound to common (qualitative and quantitative) research standards" [43, p. 10], to analyze these retrospective interviews. The goal of our analysis was to provide insights that support the interpretation of the eye-tracking data and further explore the processes involved in program comprehension.

QCA distinguishes two approaches: *Deductive Content Analysis* aims at "extracting a certain structure from the material based on exact dimensions, that are grounded in theory and derived from the issue/statement of the problem concerned" [43, p. 95]. This method, therefore, builds on theories related to the research question. If no such theoretical background is available, then

Inductive Content Analysis is more suitable. This approach aims at developing categories "that are coming from the material itself, not from theoretical considerations" [43, p. 79].

Both approaches begin with a clearly formulated research question that determines which units of the material are analyzed [43, p. 51]. For example, because all research questions in this study aimed at better understanding program comprehension (its process), the recording units in this study were interview sections for each participant commenting on each program.

Because the research questions in this article are closely related to existing literature, we chose a deductive approach. In this approach, categories (and the respective sub-categories) are defined based on theory. The sub-categories do not have to be found in the literature, verbatim, "but they have to be grounded with theoretical arguments" [43, p. 97]. These definitions are then specified using as unambiguous encoding rules as possible. Additionally, each sub-category is illustrated by one or multiple anchor examples.

By reading the material line-by-line, each part of the material that fulfills the definition is marked and assigned a sub-category. Mayring [43, p. 99] suggests that after coding 10%–50% of the material, the coding rules may be modified until they become smoother and examples for each sub-category are chosen. At this point, or if severe problems with the categories arise, the face validity of the sub-categories and coding rules should be reconsidered in respect to the research question. If changes are necessary at any of these points, then theoretical considerations should be used.

Finally, the sub-categories are applied by multiple researchers. If they derive sufficiently similar codings, as measured by interrater reliability, then the coded material can be further analyzed.

We analyzed the verbatim transcripts of the retrospective interviews using QCA. The coding procedure for all sub-categories followed the suggestions of Campbell et al. [7]. One coder, who was engaged in the development of the category systems, marked for each group of sub-category sections of interest. These sections ranged from a single sentence to one coherent statement covering multiple sentences. For example, if a participant answered the question, or if the program was familiar, then this answer was assigned the category system FAMILIARITY. If multiple sections of the program were described in such a section, then that section was split into multiple parts correspondingly. After that, 20% of each category system was coded by two coders independently by using the categories defined as above. The coding results were compared and ambiguities in the coding rules were resolved. Finally, the whole material was coded again by both coders and the inter-coder reliability was computed. Below, we describe the categories and their sub-categories derived from the literature.

Category – ABSTRACTION OF CODE. The performance during mental simulation may be related to the abstraction of the code [77, p. 239] [78]. Therefore, we investigated the qualitative research question: *On what level of abstraction was the mental model participants created of the programs?* Therefore, we analyzed descriptions of program sections from the retrospective interviews as described below.

A common category system used to analyze the quality of such mental models is the **Structure of the Observed Learning Outcome (SOLO)**. Biggs and Tang describe five levels of abstraction [3, pp. 39–50]. These were applied by Lister et al. to categorize program summaries of novices [41] and by Ginat and Menashe to assess algorithm design [23]. We chose to deductively develop categories based on the SOLO taxonomy reflecting that some programs in our study were longer than programs used in previous works and that our participants were more advanced. Every sentence of an attempt to summarize parts of (or a whole) program was assigned one of the following categories:

- **Prestructural:** Responses on this level "simply miss the point" [3, p. 39]. Responses of participants that failed to summarize a section were categorized as prestructural. Therefore,

statements that merely paraphrased one or two lines of the code or even showed difficulties understanding a line of the code at all were categorized as prestructural. In the following example, a student paraphrased lines of code from Omega while reading them with no attempt to summarize them.

[G:] *If you somehow find a value, um, that is in list 2, but is smaller than or, um, equal to a value from r, then the loop is aborted and nothing is stored in list 3.*

- **Uni- and Multistructural:** A response that meets only one aspect of a task is considered to be on an unistructural level, whereas a "disorganized collection" of such aspects would be considered to be on a multistructural level [3, p. 49]. Biggs and Tang state that answers on this level reflect that a person giving this answer "sees the trees but not the wood" [3, p. 40]. We considered a complete solution of the task a summary of the whole program, including coordination of sub-summaries of different program sections. Hence, using the terminology in the above sub-category, a "tree" would be a summary of a program section. Because single sentences were coded using this category system, it would not be possible to distinguish uni- and multistructural responses directly. Therefore, this category considered both responses. However, after the coding was completed, then this category was refined as follows: If in one document only one program section was described, then the category was changed to unistructural. If two or more program sections in one document were described, then the category was changed to multistructural.
  The following example illustrates a summary of a code section as a maximum but does not relate it to the rest of the program:

  [C:] *Takes the first entry and checks, the first loop checks, what the biggest entry is.*

- **Relational:** Staying in their metaphor, Biggs and Tang describe this level as a response where "the trees have become the woods" [3, p. 40]. Lister et al. [41] describe this level as an understanding of all parts of the program and an integration of these into a coherent structure such as a summary of the whole program. We considered all descriptions to be on a relational level if one or more sections of the program were related to each other. The following example illustrates how a section computing the distance between identical numbers in an array and a section computing a minimum are related to each other:

  [A:] *Eventually my interpretation was, uhm, that it computes the distance between to equal numbers, I mean the smallest distance between two equal numbers.*

- **Extended Abstract:** This is the highest level of abstraction. Biggs and Tong state that responses go "beyond existing principles" [3, p. 41]. However, they note that this also depends on the context, because "today's extended abstract is tomorrow's relational" [3, p. 40]. We considered statements relating program sections to a broader context to be on an extended abstract, e.g., suggestions on how program sections could be rewritten to be more comprehensible or statements on properties such as the runtime of a section. Preempting the results, all documents containing a section we coded as extended abstract also contained at least a section on a relational level. In the following example, a participant comments on how a program section could be rewritten because she identified two independent processes. We note that she also provided a complete summary on a relational level initially.

  [H:] *Here we have a loop with two operations, but those are somewhat independent of each other, so they could, hypothetically, be rewritten as two separate loops.*

Category – Familiarity. The programs in this study were designed according to Soloway's notion of plans [65].

Programs consisted of sections we had designed to (a) correspond to (standard) plans, (b) correspond to tailored plans, or (c) not correspond to any plan-schema at all. To assess whether or not our programs conformed to these criteria, we conducted a pilot study (see Section 3.3).

During the interviews, we prompted the participants to comment on the familiarity of the program, but not explicitly to comment on separate program sections to avoid any bias from social desirability. Responses to these prompts were split into sections commenting on one program section if possible. For the analysis of the qualitative data gathered in this pilot, we used the following category system:

- **Code Corresponding to a Plan-schema.** This category was applied to statements of high familiarity with a program section. Indicators such as the explicit use of plan-names, noting that one is familiar with this code or mentioning that one did encounter the same code before. In the following quote, two code sections are directly referred to as maximum and minimum:

  [I:] *In the end one compares, in order to build the minimum—is it a minimum or a maximum—I don't know.*

- **Modification of a Plan-schema (Tailored Plan).** This category was applied if a code section is described as familiar, but a change from a standard procedure is noted. Indicators are, e.g., the identification of something familiar in otherwise unfamiliar sections or explicit comments on how a section deviates from a standard procedure. The following quote illustrates a comment of a participant on the program section of Rainfall, where two conditions are used in a `for`-loop:

  [E:] *I wasn't totally sure, because, [...] I rarely did it with two conditions and then I had to think how it would probably work.*

- **No Schema Available.** This category was applied if the program code is not perceived as familiar and no references to any schema are made. Indicators are the lack of any references or an explicit comment that the program section is unfamiliar.

  [H:] *This nested* `for`*-loop here with no explanations, it is extremely difficult to find an interpretation for this and to understand what this code does.*

- **Unclear Familiarity.** This category is assigned to statements that do not belong in any of the categories above. Ambiguous statements, comments on rules of discourse, or the inputs provided were assigned this category. For example, some participants commented not on the code, but on the input-to-output conversion tasks:

  [G:] *It seemed unusual to me that I entered 100 every time.*

4 RESULTS

In this section, we present the results of the analysis of both quantitative and qualitative data obtained during our study. We start with presenting the eye-tracking data relevant to answer each of the hypotheses presented in the previous sections. Then, we briefly summarize findings from the QCA. In Section 5, we then combine both—the results from the analysis of the qualitative and the quantitative data—to answer our research questions.

4.1 Eye-tracking Data on How Students Read Programs

Before we present our results, we briefly discuss the question of whether significant results might need to be adjusted. In his review of the use of the Bonferroni correction in clinical research, Armstrong cautions that such a correction "should not be used routinely" [2, p. 504] and lists

Table 1. Ratio of Time Spent on Code and Input over the First Three Tasks Per Program (Mean ± Standard Deviation) for All Traces Meeting the Inclusion Criteria for Data Quality

Program	n	Task 1	Task 2	Task 3
Minimum Sum	9	3.80 ± 1.46	1.85 ± 2.00	.47 ± .50
Maximum Count	8	4.64 ± 1.89	1.61 ± 2.18	.40 ± .30
Rainfall	11	4.82 ± 2.01	1.50 ± 1.87	.68 ± .64
Rainfall Partner	12	16.01 ± 7.54	4.13 ± 8.42	3.71 ± 5.67
Balanced Subset	7	14.73 ± 6.96	0.89 ± 1.00	.96 ± .75
Omega	11	16.78 ± 8.35	3.22 ± 1.26	3.34 ± 3.12

The difference between the first and the second task was always significant ($p < .05$, $d > .8$).

indicators for when it should be used. We discuss these in turn. (1) While, at first, it may appear that the fact that we are using one set of traces for multiple analyses indeed requires such a correction, we point out that each analysis performed pertains to a unique hypothesis (e.g., "The time spent on the input decreases from the first to the second task." or "There is a dependency between the level of familiarity of a code section and the relative amount of time spent on that task."). As such, our study is *not* "a single test of the "universal null hypothesis" (H_0) that all tests are not significant" [2, p. 504]. (2) We did *not* run "a large number of tests without preplanned hypotheses" [2, p. 504]. Instead, all tests were motivated from the literature and exactly those are reported in the following. We thus conclude that adjusting significance levels was not required for our analyses.

H_A: The ratio of time spent on the input to time spent on the code is highest during the first task. We investigated this hypothesis by analyzing the ratio of the overall time spent on gazing at the code and the overall time spent on gazing at the input during the first three input-to-output conversions. The inclusion criterion for data traces, in addition to having no prominent measurement gaps, was that the participant in the corresponding study had computed correct outputs on at least three inputs. We also considered answers to be correct if participants consistently had made the same error, e.g., had confused the minimum with the maximum, because we have no reason to assume that the high-level reading strategy or the time on task might have been affected by such errors. In contrast, such confusions were reported before for experienced programmers, because their experience makes them read code superficially if they expect something [28].

All programs varied in different aspects, i.e., the composition of code sections and how related these were to plan-schemata. Therefore, we chose throughout the analysis to treat programs separately.

We discuss results for the Minimum Sum program; other tasks showed similar patterns (see Table 1). There were no outliers in the data, as assessed by inspection of a box plot for values greater than 1.5 box lengths from the edge of the box. Differences between the ratio of time spent on the different AOIs were normally distributed, as assessed by Shapiro-Wilk's test ($p = .18$).[4] From Task 1 to Task 2, the ratio of time spent on the program to the time spent on the input statistically significant decreased (paired-samples t-test, $t(8) = 2.58, p = .03, d = .86$; here and throughout the rest of this section, we report effect sizes using Cohen's d). Participants spent increasingly less time on the program during the progression through the three tasks: The time to completion (in seconds, shown as mean ± standard deviation) was 3.80 ± 1.50 (Task 1), 1.85 ± 2.00 (Task 2), and .47 ± .51 (Task 3).

[4]For the remainder of this section, we only report outliers if any were detected. Also, we only discuss the normality assumption in case of violations. All analyses were conducted using IBM SPSS™ version 27 using 95% confidence intervals.

Program	SP_S	SP_M	TP_S	TP_M	UC_S	UC_M
Rainfall	10.25 ± 3.14		20.35 ± 6.67			
Rainfall Partner		18.44 ± 9.1		30.18 ± 8.95		
Balanced Subset	9.76 ± 3.00	23.86 ± 11.25	26.40 ± 13.81			49.99 ± 20.71
Omega	18.37 ± 8.19		37.14 ± 16.61		69.63 ± 25.26	

Fig. 4. Results related to hypotheses $H_{1.1}$ and $H_{1.2}$: Normalized time participants spend on different AOIs. The index indicates whether the AOI describes a sequenced (S) or a merged (M) code section. Additionally, SP denotes that the AOI comprises a (standard) plan, TP that it comprises a tailored plan, and UC that it comprises unplan-like code. In each line, a color difference indicates a significant difference.

In summary, after the first input-to-output conversions all subsequent conversions were executed significantly faster. The assumption H_A: *The ratio of time spent on the input to time spent on the code is highest during the first task.* does not need to be rejected.

$H_{1.1}$: Within a program, participants spent more time on tailored plans than on plans. and $H_{1.2}$: Within a program, participants spent more time on non-planlike code sections than on tailored plans. To investigate these hypotheses, we compared the time participants spent on the two AOIs defined for each program (Section 3.6.1) during their first encounter with the program, i.e., during the first input-to-output conversion. We used paired-samples t-tests to assess the mean differences. The time participants spent on the AOIs is presented in Figure 4.

In Rainfall, participants spent significantly more time on the tailored plan that created a sub-array until the first occurrence of "999" (20.35 ± 6.67 seconds) than on the (standard) plan computing the sum (10.25 ± 3.14 seconds), $t(11) = 4.069, p = 0.009, d = 1.17$.

Results for Rainfall Partner were similar. We compensated for differences in lines of code for the (standard) plan (three lines) and the tailored plan (four lines) by normalizing the dwelling time. This increased the weight of the time spent on the (standard) plan and decreased the mean difference. Participants spent more time on the tailored plan (40.24 ± 11.93 seconds effectively, 30.18 ± 8.95 seconds normalized) than on the (standard) plan (18.44 ± 9.1 seconds). We detected three outliers that were more than 1.5 box lengths from the edge of the box in a box plot; these were removed from the analysis. However, all of the participants whose data we removed had focused more on the tailored plan; thus, the removal actually worked against confirming $H_{1.1}$. Even though normalization decreased differences, the mean difference was found to be significant, $t(8) = 11.56, p < 0.0005, d = 3.85$. Thus, in both the sequenced Rainfall and the merged Rainfall Partner program, participants spent significantly less time on (standard) plans, supporting $H_{1.1}$.

For the Balanced Subset program, we considered the difference between plans, tailored plans, and unplan-like code (upper AOI in the middle column, AOIs in the left and right column of Figure 3). Each of these sections consisted of four lines of code, thus, no normalization was needed. Participants spent less time on plans (9.76 ± 3.00 seconds) than on tailored plans (26.40 ± 13.81 seconds) and on unplan-like code (49.99 ± 20.71 seconds). The difference between the time spent on plans and tailored plans was significant, $t(9) = 4.52, p = .001, d = 1.43$. The difference between the time spent on tailored plans and unplan-like code was significant as well, $t(9) = 4.86, p = .001, d = 1.54$.

The internal structure of the Balanced Subset program, however, warrants further investigation. As discussed in Section 3.6.1, the (standard) plan and the tailored plan were presented on

their own, while the middle column of code contained both unplan-like code (upper AOI, four lines) and another (standard) plan (lower AOI, three lines); see Figure 3. Considering these two blocks of code, the time spent on the unplan-like code (49.99 ± 20.71 seconds effectively, 37.49 ± 15.53 seconds normalized) was statistically significant higher that the time spent on the (standard) plan (23.86 ± 11.25 seconds), $t(9) = 2.555, p = .031, d = .81$, confirming $H_{1.2}$ for this case. Interestingly enough, we did not find a statistically significant difference between the time spent on the tailored plan (26.40 ± 13.81 seconds) outside the second block (left column), i.e., presented in a sequenced context, and the time spent on the (standard) plan (23.86 ± 11.25 seconds) inside the middle block (lower AOI), i.e., presented in a merged context, $t(9) = .959, p = .362$. We claim that this missing significance does not contradict $H_{1.1}$. Instead, we attribute it to the complexity added by presenting a (standard) plan in a merged context.

To further study $H_{1.2}$, Omega was presented in a sequenced code composition. Even after normalization, participants spent significantly more time on the tailored plan (seven lines; 65.00±23.81 seconds effectively, 37.14 ± 16.61 seconds normalized) than on the (standard) plan (four lines; 18.37 ± 8.19 seconds), $t(11) = 4.890, p < .0005, d = 1.41$. However, after normalization the time spent on unplan-like code (eight lines; 79.58 ± 23.870 seconds effectively, 69.63 ± 25.26 seconds normalized) was not significantly higher than the time spent on the tailored plan (seven lines; 65.00 ± 23.81, seconds 37.14 ± 16.61 seconds normalized), $t(10) = .407, p = .692$.

In summary, our data suggests support for $H_{1.1}$ across programs of varying difficulty. It also suggests that the time spent on (standard) plans increases if the plan is presented in a merged context. With respect to $H_{1.2}$, the results are not as clear. The implications for $H_{1.2}$ will be discussed in Section 5.

$H_{2.1}$: The number of transitions between merged program sections is higher than the number of transitions between sequenced program sections. To investigate this hypothesis, we analyzed the differences in the number of transitions between relevant AOIs during the first task of each experiment—see Section 3.6.1. The results are presented in Table 2. Differences were significant for both programs in the first pair: Maximum Count (13.14 ± 5.40 transitions) and Minimum Sum (5.14 ± 1.95 transitions), $t(6) = 3.886, p = .008, d = 1.468$.

For the comparison between the Rainfall Partner and the Rainfall ($n = 6$) and the comparison within the Balanced Subset program ($n = 6$) the normality assumption was violated after the removal of outliers. We thus used a Wilcoxon signed-rank test that confirmed the differences to be significant for both comparisons (Rainfall Partner (mean: 12) / Rainfall (mean: 21): $z = 1.997, p = .046, d = .815$; within Balanced Subset (means: 18.5 and 31): $z = 2.1, p = .036, d = .857$).

For the (merged) Omega program, the number of transitions was highest between the two code blocks using indirect array accesses to compare candidate values. However, we could not confirm a statistically significant difference between the mean differences of transitions between the code blocks in the (sequenced) Balanced Subset program (28.17 ± 9.66 transitions) and in the (merged) Omega program (27.50 ± 9.94 transitions), $t(5) = .087, p = .934$.

In summary, three out of four comparisons yielded significant differences for the comparison of the number of transitions between merged and sequenced program sections. We hence partially supported $H_{2.1}$. Limitations are discussed in Section 5.

Table 2. The Transition between AOIs Presented Either as Mean and Standard Deviation or Median Depending on the Distribution of the Differences

Program	Transitions	Significance
Minimum Sum	5.14 ± 1.95	$t(6) = 3.886, p = .008, d = 1.468$
Maximum Count	13.14 ± 5.40	
Rainfall	21	$z = 1.997, p = .046, d = .815$
Rainfall Partner	12	
Balanced Subset$_S$	18.5	$z = 2.1, p = .036, d = .857$
Balanced Subset$_M$	31	
Omega	27.50 ± 9.94	$t(5) = .087, p = .934$
Balanced Subset$_M$	28.17 ± 9.66	

In the rightmost column, the difference in the number of transitions is tested for significance. For Balanced Subset two code sections were considered; a merged code section (Balanced Subset$_M$) consisting of two smaller code sections merged in a `for`-loop (two blocks in the middle column of Figure 3) and a sequenced code section (Balanced Subset$_S$) consisting of three smaller code sections (left and right column and upper part of the middle column in Figure 3). As explained in Section 3.6.1, we considered—if possible—more code sections for the sequenced than for the merged programs and thus penalized H 2.2. Besides the comparison between Omega and Balanced Subset$_M$, all comparisons indicated that the number of transitions is significantly higher for merged than for sequenced programs.

$H_{2.2}$: The ratio of time spent on the code relative to the input is higher for programs with a merged code composition than for programs with a sequenced code composition. To investigate the effects described in the previous section, we compared the ratio of the overall time dwelling on the input and dwelling on the code during the first input-to-output conversion. We did not find any statistically significant difference between Minimum Sum (ratio: 3.58 ± 1.35) and Maximum Count (ratio: 3.70 ± .84), $t(5) = .205, p = .846$. We confirmed a statistically significant difference between the (merged) Rainfall Partner program (ratio: 13.43 ± 5.29) and the (sequenced) Rainfall program (ratio: 4.74 ± 2.02), $t(8) = 3.85, p = .005$. The data for Omega and Balanced Subset shows an effect opposite to what the hypothesis states; the relative time spent on code in Omega was higher than in Balanced Subset (see Table 1).

The results with respect to $H_{2.2}$ are mixed. We will discuss all findings in Section 5.

4.2 Qualitative Data

Using the methods described in Section 3.7, we analyzed the retrospective interviews. The results including the inter-coder reliability between the two coders are presented below.

Category SOLO. The inter-coder reliability for this category system was $\kappa = .746$ after the first iteration of coding. Disagreements could mostly be attributed to an ambiguous description of what is considered a summary. After discussing these issues, the inter-coder reliability increased to $\kappa = .95$, an indicator of a high inter-coder reliability [44].

The analysis revealed that, except for Omega, most participants commented on the programs at a relational level, i.e., most participants were able to relate at least two sections of each program (see Table 3). A closer analysis revealed that all relational answers for Minimum Sum and Maximum Count covered the whole program, and all but one relational answer for Rainfall and all but two answers for Rainfall Partner considered the whole program. In Balanced Subset and Omega, only one relational answer considered the whole program. However, five answers in Balanced

Table 3. The Highest Level of Abstraction Observed in Each Program

Program	Prestructural	Unistructural	Multistructural	Relational	Ext.Abstract
Minimum Sum	0	0	1	9	0
Maximum Count	0	0	1	8	1
Rainfall	0	0	0	9	1
Rainfall Partner	1	1	2	6	0
Balanced Subset	0	2	1	7	0
Omega	2	2	4	2	0

The first two programs consisted both only of standard-plans and the middle two programs also contained tailored plans. The last two programs consisted of (standard) plans, tailored plans, and unplan-like code sections.

Subset considered the whole program except for the last section, which ensures that either a positive number or 0 is returned. A reason for these apparent difficulties with Omega may be the indirect addressing in the code section we considered to be unplan-like:

[F:] *At some point, it occurred to me that here somehow, um, values in list 2 and list 1 were compared, but actually values were compared to the corresponding indices, then I had somehow deduced from my wrong understanding that um values were stored there, so to speak.*

Our analysis of the retrospective interviews shows that each participant was able to summarize all programs, except for Omega, and to relate program sections to each other. Hence, we may assume that they were also able to do so while working on the input-to-output conversions as well. This further *supports* the model-building assumption (Section 3.4). With respect to Omega, we found that the mental representation was incomplete; we attribute this to a code section using indirect addressing, which should be reconsidered in future work.

Category Familiarity. The inter-coder reliability for this category system was $\kappa = .63$ after the first iteration of coding. Disagreements could mostly be attributed to sections of the interviews that commented on the input-to-output rather than on the program, e.g., that the "result was often negative." After discussing these issues, the inter-coder reliability increased to $\kappa = .81$, an indicator of a strong inter-coder reliability [44].

The qualitative data revealed that tailored plans were recognized as such. For example, all comments regarding the loop creating a monotonously increasing sub-array in Omega (three in total, because not every section of Omega was covered, and we did not prompt the participants to comment on each section) described it as an extension of a maximum (standard) plan.

[E:] *In the first* `for`*-loop, entries are added to list2, um (…) the indices are added to list2, that um (…) serve to find a new maximum.*

An analysis of the program section coded with the familiarity section revealed that in seven documents the 999-Filter in Rainfall was described as a tailored plan, where a loop-over-array (standard) plan was extended by an additional condition.

In Rainfall Partner, three participants described the "guard-section" [65] as such but stressed that they would have expected distinct parts, e.g., an else-branch to clarify the guard-off section or a Boolean indicating the guard variable. However, two other participants mentioned severe problems with understanding the guard section and did not recognize any familiar plan-schema. The other half of the participants did not comment on the program in any meaningful way.

This identification of familiar plans was not used to describe unplan-like code sections. For instance, when discussing Omega, participants did identify the concept of nested loops in the problematic section but used this concept only to refer to that section instead of explaining it.

5 DISCUSSION

Our analysis supports the model-building assumption, i.e., that participants developed a mental model of the algorithm while working on the first input-to-output conversion and utilized it later. This simplified the analysis of the two research questions. In addition, the short time frame for which we analyzed the eye-tracking data made it unlikely that the data recorded showed participants engaging with other tasks than trying to comprehend the program.

Regarding the first research question, we were able to observe that standard plans are significantly faster to comprehend than tailored plans and unplan-like code sections. This effect was more nuanced when comparing tailored plans and unplan-like code, since, in general, we could observe differences in comprehension times, but not significant ones. Yet, the analysis of the qualitative data supported our assumption that plans are also used during the comprehension of tailored plans.

The analysis of the second research question revealed that merged code sections increased the number of transitions between code sections as opposed to comparable programs presenting sequenced code sections. However, only one comparison yielded a significant effect *on the time* to comprehend such programs. Below, we discuss factors we assume to have overshadowed the effect on time in these cases.

A more detailed discussion of the research questions and eye-tracking as a methodology to study program comprehension processes will be presented in detail in the following paragraphs. For a more vivid discussion, we also present excerpts from the retrospective interviews.

Model-Building Assumption: While performing the first input-to-output conversion for a program, readers build a mental model of the program and utilize it on the subsequent tasks. The quantitative analyses imply that the focus of attention shifts from the code to the input starting with the second input-to-output conversion. Indeed, many participants did not look at the code at all during later input-to-output conversions. Additionally, the qualitative analysis of the retrospective interviews with respect to the SOLO categories revealed that most participants were able to build an abstraction of at least parts of the program. Together with the eye-tracking data, this indicates that participants developed and actually used the abstractions they developed to mentally simulate the programs; the assumption was thus confirmed to hold.

In line with this interpretation, we quote a participant who stated that the purpose for looking into the code after completing the first task was to re-validate the abstraction he had built:

> [A:] *I generally just simulated and calculated the first input again. And then I tried to generalize again on the second input, or rather to see if what I had came up with as an abstraction of the program would work, or if it was just that one single input.*

During the eye-tracking session, one participant started to think aloud about her approach and explicitly named plans she had identified. During the interview, she explained that this allowed her to work on later tasks more effectively.

> [B:] *Then I look at the series of numbers and recognize the sum very quickly* [...]. *Then I can also change the order, which is, for instance, convenient for negative and positive numbers, which then cancel each other out.*

Participants working on Balanced Subset often focused on the number most frequently occurring in the input and then on this number last occurrence. We interpret this as the use of an abstraction of the actual program: A participant commented this behavior with filtering out numbers "unlikely to be relevant for the result"; of course, the program did not do this.

This behavior resembles observations that programmers spent the most time on the first occurrence of a repeatedly presented and slightly modified code section [32]. Our results suggest a

rationale for this behavior: They build a mental model of the first code section they saw and then recognized its modifications in the next lines. However, we observed another facet of flexibility when students "modified" the program to more efficiently mentally simulate the program. This flexibility is not unusual for mental models [14, 49].

RQ 1: How does the availability of plan-schemata affect program comprehension? Literature suggests that summarizing code is an important process of program comprehension [10]. We thus consider the time participants spent on certain areas of the code to be indicative of how difficult it was to summarize this section. This is in line with the results of Chen et al. [9], who showed that dwelling time significantly increased when tasks were more complex and more attention was needed.

This rationale led to the hypothesis that plans are faster to comprehend than other code sections, which our eye-tracking data confirmed: For each program consisting of plans and tailored plans, participants spent statistically significantly more time on tailored plans. Indeed, participants mentioned that the sections they spent more time on were harder to understand and that it was easier for them to summarize other code blocks. Our findings thus are in line with prior studies that prior knowledge in the form of plans supports comprehension: It allows for faster chunking and consumes fewer cognitive resources that otherwise would be needed for abstraction. Based on this claim, we can further hypothesize that plans, similarly to sub-goal labels [46], can reduce cognitive load.

The results for the comparison between tailored plans and unplan-like code sections is more nuanced: While observing the expected difference comparing the tailored plan and unplan-like code within Balanced Subset, an influencing factor may have been the code composition (see Section 4.1). Hence, we treat those results with caution. Results from comparing the tailored plan in Omega to unplan-like code did not confirm our hypothesis. We saw that participants were struggling more with indirect addressing in the Omega program than could have been expected based on our initial validation study (see Section 4.2). This intrinsic complexity may have increased the time spent on this tailored plan and, consequently, confounded our results. However, from interviews, we gathered evidence that tailored plans were perceived as plan modifications:

[C:] *Well, you skim over the* `for`*-loop if it has the absolute standard form. And then–I think–I just thought: ah yes, there is an extra condition built in.*

The results from qualitative analysis using the Familiarity categories showed that all tailored plans were indeed perceived as such (see Section 4.2). Hence, we suggest that the difference between tailored plans and unplan-like code sections exists, but may not be as prominent as between plans and tailored plans. We suggest that future work could investigate this issue.

RQ 2: How does the composition of code sections affect program comprehension? In all programs, except for Omega, merged code composition resulted in a significantly higher number of transitions between the AOIs compared to sequenced code composition. For the Rainfall, Rainfall Partner, and Balanced Subset programs, we deliberately consider more AOIs for the sequenced version than for the merged ones. If participants had shown similar reading behavior regardless of the composition, then this choice would have shown a higher number of transitions for the sequenced versions. Conversely, our findings show that even under these conditions, merged composition increases the number of transitions.

When looking at the recording of her gazes for the (merged) Rainfall Partner program, one participant commented that the initialization was "rather boring" and then commented on her many transitions between the two code blocks inside the loop.

[B:] *Those are the two interesting areas. And then, yes. I just had the first one pretty quickly. And then, um, the second one isn't that difficult either. And then, I always checked both of them and whether they interact somehow. But they don't, they're completely independent.*

For her, not only the comprehension of code sections alone but also comprehending their interaction contributes to the overall comprehension process. Hence, we assume that the composition of code sections affects the comprehension of the interaction between those.

Previous work from Cognitive Load Theory [8] suggests that some effects cannot be observed when the cognitive load is too low or when the cognitive load is too high. Our results support this notion. The evaluation of the programs designed to be the least complex ones (Minimum Sum and Maximum Count) and the more complex ones (Omega and Balanced Subset) showed no significant differences on times spent on code relative to the input. However, the comparison between Rainfall and Rainfall Partner, designed to be in the middle of the complexity spectrum, showed significant differences. Given the participants' expertise, we suggest that Rainfall and Rainfall Partner did not overwhelm participants' cognitive resources (and were able to be sufficiently cognitively demanding) and enabled us to observe the effects of code composition [15].

Support for this suggestion comes from the qualitative analysis using the SOLO category: All participants were able to fully abstract over the programs Minimum Sum and Maximum Count and—to a lesser extent—over Rainfall and Rainfall Partner; see Table 3. In contrast, participants struggled when abstracting over Omega and Balanced Subset. One take-away from this is that "just" tracking time-on-task may not be sufficient to assess the effect of code composition, while eye-tracking data is nuanced enough to show such effects.

Complementing the results presented by Peitek et al. [53], we found that—similarly to the linearity of program flow and scope—the perceived interaction of (tailored) plans impacts program comprehension. This adds to the body of knowledge of program comprehension, since Peitek et al.'s metric for source code linearity works on the level of lines of code, therefore yielding similar values for programs using either sequenced or merged code compositions. Hence, our work complements theirs by considering similar concepts but on a rather different level of abstraction. A combination of their work and ours, as well as a closer examination of their effects on element interactivity [72], remains subject to further research.

Reflection on the study design and eye-tracking as a methodology. We found eye-tracking to be a valuable approach to gain insights into a comprehension process while being less disruptive than other methods, such as think-aloud interviews. Moreover, it provides rich data on processes not observable through other methods. However, the experimental conditions must be controlled strictly and the study design needs to be carefully tailored to the respective research domain [71].

Our design was based on the model-building assumption. This assumption allowed for tighter control of several factors relevant for eye-tracking, including, but not limited to, attention spans. In particular, the first task of each set of input-to-output conversions allowed for the collection of high-quality data regarding the comprehension process. As per the model-building assumption (which we verified through a comparison of the times spent on subsequent input-to-output conversions and through the analysis of qualitative data), the data collected during the first task represents most of the comprehension process. Due to its limited duration as compared to a "full" set of tasks, this approach leads to data with less confounding factors.

6 THREATS TO VALIDITY

While our sample size is within the range of sample sizes of previous studies (Obaidellah et al. [50] report an average sample size of around 18 participants), we are careful not to make claims regarding generalizability; in particular, the effect sizes reported need to be interpreted carefully.

We sought to provide insights into how rich eye-tracking data could support the investigation of program comprehension in terms of students' development of mental models and the impact of program structure and difficulty on such development.

Even though we recruited participants meeting the inclusion criteria only, we could not control which particular plan-schemata subjects had at their disposal. However, our assumptions of which sections of code may correspond to plan-schemata were not contradicted by the qualitative data. In addition, the low relative number of female participants, while representative of the underlying target population, threatens the validity of the results for this subgroup. Finally, our study excluded beginners by design: The goal was to investigate the effects of plans and their composition, and therefore knowledge of certain plan-schemata was required. This design choice and the fact that all participants successfully completed the tasks for all programs should be kept in mind. This is another limiting factor for generalizability beyond the demographic context of this study, in particular, when including novice programmers.

A final possible threat to validity stems from the material used in this study. The programs used are not (and could not be) identical in all quantitative aspects, such as lines of code, size of areas of interest, number or usage of programming structures, so latent factors may have played an undetected role. To mitigate this threat as much as possible, we had piloted the material and attempted to account for differences through normalization; also, we used retrospective interviews to gather additional information regarding the nature and possible influences of these quantitative differences in the material.

7 CONCLUSION

In this study, we explored a combination of eye-tracking and retrospective interviews to investigate program comprehension on the level of plans. We distinguish ourselves from previous eye-tracking studies used in program comprehension by moving away from text-surface features of the code and instead, for the first time, focus on higher-level plans and their composition. We were able to provide a detailed investigation of program comprehension behavior not presented in previous studies using think-aloud interviews or observations of sketching only: when and how participants were able to form a model of the program and move away from code, how the familiarity with code impacted program comprehension, and how code composition affects participants' behavior. We consider think-aloud and eye-tracking data to be complementary methods that can be used to illuminate the formation of mental models in program comprehension.

Summary. Concerning RQ 1: *How does the availability of plan-schemata affect program comprehension?* We found that plans were understood more quickly than tailored plans. We assume therefore that plan-schemata allowed for faster chunking and therefore accelerated the comprehension process. Qualitative interview data supports that students perceived tailored plans to be similar to plans, which, compared to unplan-like code sections, could help them to comprehend programs. However, a statistically significant effect could not be observed here. Possible reasons for the inconclusive results are discussed below.

Concerning RQ 2: *How does the composition of code sections affect program comprehension?* We observed that code presented in a merged version significantly increases the number of transitions between different code blocks as compared to sequenced code. Interview data provided support that such effect was due to an increased difficulty of identifying interactions between code blocks in a merged composition: Merged code composition increases element interactivity. These effects were found for code blocks on the same level of familiarity as well as for code blocks on different levels of familiarity, indicating they appear regardless of familiarity. However, we did not observe convincing evidence that this also affects the time to comprehend code.

The material we used in this study was piloted (see Section 3.3), but some unwanted effects still affected our study. When applying the setup we presented, we suggest keeping multiple aspects in mind: In Omega, we observed that a certain concept that was not present in the companion program was unexpectedly difficult for the participants. We also observed that an effect of code composition on the time needed to complete the first input-to-output conversion task was not observable for programs that were either too simple or too difficult. If one wants to study the effect of a potential feature that increases its difficulty, then one should match students' cognitive resources with the task difficulty. Students should be challenged, but not overwhelmed [59].

Overall, we observed positive effects of knowledge of plan-schemata on program comprehension. Therefore, this study underlines again the importance of fostering the development of such plan-schemata in novice programmers. We also observed an effect of code composition and found indicators that a merged code-composition strategy may complicate comprehension processes by increasing element interactivity, because different code sections need to be processed in an interleaved manner instead of separately. However, further investigation is required.

A methodological contribution of this article is our presented setup allows studying program comprehension using eye-tracking data. By providing multiple time-critical input-to-output conversion tasks, we observed evidence that participants indeed developed a mental model during the first task and utilized this later. Using the setup we presented, we suggest that this assumption can be used in future works to provide high-quality data in program comprehension studies.

APPENDIX

To support the review process, we added screenshots of the programs used and highlighted the relevant AOIs used in this study. Note that the AOIs were not visible to the participants and were added to the figures below afterwards to increase the transparency of our experiment.

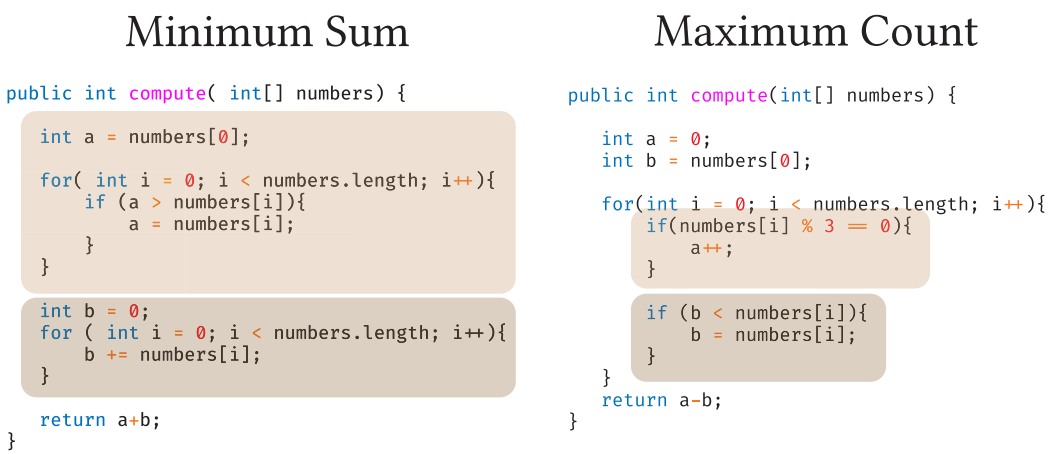

Fig. 5. Example for relevant areas of interest (AOIs). The figure shows AOIs for the sequenced version of Minimum Sum and the merged version of Maximum Count.

Rainfall

```
public int compute(int[] numbers ) {
    List<Integer> list = new ArrayList<>();
    for (int i = 0; i < numbers.length && numbers[i] != 999; i++){
        list.add(numbers[i]);
    }
    List<Integer> list2 = new ArrayList<>();
    for (int i = 0; i < list.size(); i++){
        int v = list.get(i);
        if(v < 0){
            list2.add(v);
        }
    }
    int r = 0;
    for (int i = 0; i < list2.size(); i++){
        r  += list2.get(i);
    }
    return r;
}
```

Rainfall Partner

```
public int compute(int[] numbers) {
    int x = numbers.length;
    boolean y = false;
    int r = 100;
    for (int i = 0; i < numbers.length; i++) {
        if (x == numbers.length) {
            if (numbers[i] == 100 && y == false) {
                x = i + 1;
                y = true;
            }
        } else {
            if (numbers[i] > 0) {
                r = r - numbers[i];
            }
        }
    }
    return r;
}
```

Fig. 6. Example for relevant areas of interest (AOIs). The figure shows AOIs for the sequenced version of Rainfall and the merged version of Rainfall Partner.

Balance Subset

```
Public int compute(int[] numbers) {
    int counter = 0;
    int N = numbers.length;
    for (int i = 0; i <  N; i++) {
        if (numbers[i] == 1) {
            counter++;
        } else {
            counter--;
        }
        numbers[i] = counter;
    }
// Intermediate results for numbers (N)
    int b = -1;
    for (int i = 0; i <  N; i++) {
        if (numbers[i] == 0) {
            b = i;
        }
    }
Map<Integer, Integer[]> values = new HashMap<>();
    int m = b + 1;
    for (int i = 0; i < N; i++) {
        int j = numbers[i];
        if (values.containsKey(j)) {
            int temp = values.get(j)[0];
            values.put(j, new Integer[]{temp, i});
            int d = i - temp;
            if (m < d) {
                m = d;
            }
        } else {
            values.put(j, new Integer[]{i, i});
        }
    }
        if (m <=  0) {
            return 0;
        } else {
            return m;
        }
}
```

Omega

```
Public int compute(int [] numbers) {
    int N = numbers.length;
    ArrayList<Integer> list1 = new ArrayList<>();
    int s = numbers[N - 1];
    list1.add(N - 1);
    for (int d = N - 1; d >=  0; d--) {
        if (numbers[d] < s) {
            s = numbers[d];
            list1.add(0, d);
        }
    }
// Intermediate results for numbers (N) and list1 (R)
    ArrayList<Integer> list2 = new ArrayList<>();
    int b = numbers[0];
    list2.add(0);
    for (int i = 0; i < N; i++) {
        if (numbers[i] > b) {
            b = numbers[i];
            list2.add(i);
        }
    }
    int j = 0;
    ArrayList<Integer> list3 = new ArrayList<>();
    for (int i = 0; i < list2.size() ; i++) {
        for (;  j < list1.size(); j++) {
       if (numbers[list2.get(i)] > numbers[list1.get(j)]){
         list3.add(list1.get(j) - list2.get(j));
     else {
            break;
     }
    }
    int d = 0 ;
    for (int i = 0; i < list3.size(); i++) {
     if (d < list3.get(i)){
         d = list3.get(i)
     }
    }
  return d;
}
```

Fig. 7. Example for relevant areas of interest (AOIs). The figure shows AOIs for the sequenced version of Balanced Subset and the merged version of Omega.

ACKNOWLEDGMENTS

We thank all students who participated in our experiments and Marlena Meyer for her help transcribing the interviews. We are indebted to Marjaana Puurtinen and Stanislaw Schukajlow-Wasjutinski for access to their eye-tracking labs. We thank Juha Sorva and Otto Seppälä for their feedback throughout this project.

REFERENCES

[1] Nahla J. Abid, Jonathan I. Maletic, and Bonita Sharif. 2019. Using developer eye movements to externalize the mental model used in code summarization tasks. In *Proceedings of the 11th ACM Symposium on Eye Tracking Research & Applications*. ACM Press, New York, NY, 1–9. DOI: https://doi.org/10.1145/3314111.3319834

[2] Richard A. Armstrong. 2014. When to use the Bonferroni correction. *Ophthalm. Physiol. Optics* 34, 5 (Sept. 2014), 502–508. DOI: https://doi.org/10.1111/opo.12131

[3] John Burville Biggs and Catherine Tang. 2011. *Teaching for Quality Learning at University* (4th ed.). Open University Press/McGraw Hill, Buckingham.

[4] Ruven Brooks. 1983. Towards a theory of the comprehension of computer programs. *Int. Man-mach. Stud.* 18, 6 (1983), 543–554. DOI: https://doi.org/10.1016/S0020-7373(77)80039-4

[5] Jean-Marie Burkhardt, Françoise Détienne, and Susan Wiedenbeck. 2002. Object-oriented program comprehension: Effect of expertise, task and phase. *Empir. Softw. Eng.* 7, 2 (2002), 115–156. DOI: https://doi.org/10.1023/A:1015297914742

[6] Teresa Busjahn, Roman Bednarik, Andrew Begel, Martha Crosby, James H. Paterson, Carsten Schulte, Bonita Sharif, and Sascha Tamm. 2015. Eye movements in code reading: Relaxing the linear order. In *Proceedings of the IEEE 23rd International Conference on Program Comprehension (ICPC'15)*. IEEE Press, Piscataway, NJ, 255–265.

[7] John L. Campbell, Charles Quincy, Jordan Osserman, and Ove K. Pedersen. 2013. Coding in-depth semistructured interviews: Problems of unitization and intercoder reliability and agreement. *Sociol. Meth. Res.* 42, 3 (2013), 294–320. DOI: https://doi.org/10.1177/0049124113500475

[8] Ouhao Chen, Slava Kalyuga, and John Sweller. 2017. The expertise reversal effect is a variant of the more general element interactivity effect. *Educ. Psychol. Rev.* 29, 2 (01 June 2017), 393–405. DOI: https://doi.org/10.1007/s10648-016-9359-1

[9] Siyuan Chen, Julien Epps, Natalie Ruiz, and Fang Chen. 2011. Eye activity as a measure of human mental effort in HCI. In *Proceedings of the 16th International Conference on Intelligent User Interfaces (IUI'11)*. Association for Computing Machinery, New York, NY, 315–318. DOI: https://doi.org/10.1145/1943403.1943454

[10] Tony Clear, Anne Philpott, Phil Robbins, and Simon. 2009. Report on the Eighth BRACElet Workshop: BRACElet Technical Report 01/08. *Bull. Appl. Comput. Inf. Technol.* 7, 1 (2009). Retrieved from https://www.citrenz.ac.nz/bacit/0701/2009Clear_BRACElet_Report.htm.

[11] Malcolm Corney, Donna Teague, Alireza Ahadi, and Raymond Lister. 2012. Some empirical results for neo-Piagetian reasoning in novice programmers and the relationship to code explanation questions. In *Proceedings of the 14th Australasian Computing Education Conference (ACE'12)*. Australian Computer Society, Inc., 77–86. Retrieved from http://dl.acm.org/citation.cfm?id=2483716.2483726.

[12] Nelson Cowan. 2001. The magical number 4 in short-term memory: A reconsideration of mental storage capacity. *Behav. Brain Sci.* 24, 1 (2001), 87–114. DOI: https://doi.org/10.1017/S0140525X01003922

[13] Martha E. Crosby and Jan Stelovsky. 1990. How do we read algorithms? A case study. *Computer* 23, 1 (1990), 25–35. DOI: https://doi.org/10.1109/2.48797

[14] Kathryn Cunningham, Shannon Ke, Mark Guzdial, and Barbara Ericson. 2019. Novice rationales for sketching and tracing, and how they try to avoid it. In *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education, ITiCSE 2019*, Bruce Scharlau, Roger McDermott, Arnold Pears, and Mihaela Sabin (Eds.). ACM, New York, NY, 37–43. DOI: https://doi.org/10.1145/3304221.3319788

[15] Rodrigo Duran, Juha Sorva, and Sofia Leite. 2018. Towards an analysis of program complexity from a cognitive perspective. In *Proceedings of the ACM Conference on International Computing Education Research (ICER'18)*. ACM, New York, NY, 21–30. DOI: https://doi.org/10.1145/3230977.3230986

[16] Kathi Fisler. 2014. The recurring rainfall problem. In *Proceedings of the 10th Annual Conference on International Computing Education Research (ICER'14)*. Association for Computing Machinery, New York, NY, 35–42. DOI: https://doi.org/10.1145/2632320.2632346

[17] Kathi Fisler, Shriram Krishnamurthi, and Janet Siegmund. 2016. Modernizing plan-composition studies. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education (SIGCSE'16)*. ACM, New York, NY, 211–216. DOI: https://doi.org/10.1145/2839509.2844556

[18] Kerstin Gidlöf, Nils Holmberg, and Helena Sandberg. 2012. The use of eye-tracking and retrospective interviews to study teenagers' exposure to online advertising. *Vis. Commun.* 11, 3 (2012), 329–345. DOI: https://doi.org/10.1177/1470357212446412

[19] David Ginat. 2009. Interleaved pattern composition and scaffolded learning. In *Proceedings of the 14th Annual ACM SIGCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE'09)*. Association for Computing Machinery, New York, NY, 109–113. DOI: https://doi.org/10.1145/1562877.1562915

[20] David Ginat. 2016. Colorful challenges: Linear cat & mouse game. *Inroads* 7, 3 (2016), 14–15. DOI: https://doi.org/10.1145/2943787

[21] David Ginat. 2016. Colorful challenges: Magic swaps. *Inroads* 7, 1 (2016), 32–33. DOI: https://doi.org/10.1145/2851508

[22] David Ginat and Yoav Blau. 2017. Multiple levels of abstraction in algorithmic problem solving. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, Michael E. Caspersen, Stephen H. Edwards, Tiffany Barnes, and Daniel D. Garcia (Eds.). ACM Press, New York, NY, 237–242. DOI: https://doi.org/10.1145/3017680.3017801

[23] David Ginat and Eti Menashe. 2015. SOLO taxonomy for assessing novices' algorithmic design. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, Adrienne Decker, Kurt Eiselt, Carl Alphonce, and Jodi Tims (Eds.). ACM Press, New York, NY, 452–457. DOI: https://doi.org/10.1145/2676723

[24] David Ginat, Eti Menashe, and Amal Taya. 2013. Novice difficulties with interleaved pattern composition. In *Informatics in Schools. Sustainable Informatics Education for Pupils of all Ages*, Ira Diethelm and Roland T. Mittermeir (Eds.). Springer Berlin, 57–67. DOI: https://doi.org/10.1007/978-3-642-36617-8_5

[25] Richard Gluga, Judy Kay, Raymond Lister, and Donna Teague. 2012. On the reliability of classifying programming tasks using a neo-Piagetian theory of cognitive development. In *Proceedings of the 9th Annual International Conference on International Computing Education Research (ICER'12)*. ACM Press, New York, NY, 31–38. DOI: https://doi.org/10.1145/2361276.2361284

[26] Shuchi Grover and Satabdi Basu. 2017. Measuring student learning in introductory block-based programming: Examining misconceptions of loops, variables, and Boolean logic. In *Proceedings of the ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE'17)*. Association for Computing Machinery, New York, NY, 267–272. DOI: https://doi.org/10.1145/3017680.3017723

[27] Kilem Li Gwet. 2014. *Handbook of Inter-rater Reliability: The Definitive Guide to Measuring the Extent of Agreement Among Raters* (4th ed.). Advanced Analytics, Gaithersburg, MD.

[28] Michael Hansen, Robert L. Goldstone, and Andrew Lumsdaine. 2013. What Makes Code Hard to Understand? Retrieved from https://arxiv.org/abs/1304.5257.

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

[30] Cruz Izu, Claudio Mirolo, and Amali Weerasinghe. 2018. Novice programmers' reasoning about reversing conditional statements. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education (SIGCSE'18)*. Association for Computing Machinery, New York, NY, 646–651. DOI: https://doi.org/10.1145/3159450.3159499

[31] Cruz Izu, Carsten Schulte, Ashish Aggarwal, Quintin Cutts, Rodrigo Duran, Mirela Gutica, Birte Heinemann, Eileen Kraemer, Violetta Lonati, Claudio Mirolo, and Renske Weeda. 2019. Fostering program comprehension in novice programmers—learning activities and learning trajectories. In *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education (ITiCSE-WGR'19)*. Association for Computing Machinery, New York, NY, 27–52. DOI: https://doi.org/10.1145/3344429.3372501

[32] Ahmad Jbara and Dror G. Feitelson. 2017. How programmers read regular code: A controlled experiment using eye tracking. *Empir. Softw. Eng.* 22, 3 (2017), 1440–1477. DOI: https://doi.org/10.1007/s10664-016-9477-x

[33] Marcel A. Just and Patricia A. Carpenter. 1980. A theory of reading: From eye fixations to comprehension. *Psychol. Rev.* 87, 4 (1980), 329. DOI: https://doi.org/10.1037/0033-295X.87.4.329

[34] Amy J. Ko, Brad A. Myers, Michael J. Coblenz, and Htet Htet Aung. 2006. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Trans. Softw. Eng.* 32, 12 (Dec. 2006), 971–987. DOI: https://doi.org/10.1109/TSE.2006.116

[35] Clifton Kussmaul. 2012. Process oriented guided inquiry learning (POGIL) for computer science. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education (SIGCSE'12)*. Association for Computing Machinery, New York, NY, 373–378. DOI: https://doi.org/10.1145/2157136.2157246

[36] Laerd Statistics. 2016. Kendall's coefficient of concordance, W using SPSS Statistics. Statistical tutorials and software guides. Retrieved from https://statistics.laerd.com/.

[37] Stanley Letovsky. 1987. Cognitive processes in program comprehension. *J. Syst. Softw.* 7, 4 (1987), 325–339. DOI: https://doi.org/10.1016/0164-1212(87)90032-X

[38] Raymond Lister. 2008. After the gold rush: Toward sustainable scholarship in computing. In *Proceedings of the Tenth Conference on Australasian Computing Education - Volume 78 (ACE'08)*. Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 3–17. DOI: https://doi.org/10.5555/1379249.1379269

[39] Raymond Lister. 2011. Concrete and other neo-Piagetian forms of reasoning in the novice programmer. In *Proceedings of the Thirteenth Australasian Computing Education Conference - Volume 114 (ACE'11)*. Australian Computer Society, Inc., Darlinghurst, Australia, 9–18. DOI: https://doi.org/10.5555/2459936.2459938

[40] Raymond Lister, Collin Fidge, and Donna Teague. 2009. Further evidence of a relationship between explaining, tracing and writing skills in introductory programming. In *Proceedings of the 14th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education, ITiCSE 2009*, Patrick Brézillon, Ingrid Russell, and Jean-Marc Labat (Eds.). ACM, New York, NY, 161–165. DOI: https://doi.org/10.1145/1562877.1562930

[41] Raymond Lister, Beth Simon, Errol Thompson, Jacqueline L. Whalley, and Christine Prasad. 2006. Not seeing the forest for the trees: Novice programmers and the SOLO taxonomy. *SIGCSE Bull.* 38, 3 (June 2006), 118–122. DOI: https://doi.org/10.1145/1140123.1140157

[42] Mike Lopez, Jacqueline Whalley, Phil Robbins, and Raymond Lister. 2008. Relationships between reading, tracing and writing skills in introductory programming. In *International Computing Education Research Workshop, ICER'08*, Michael E. Caspersen, Raymond Lister, and Mike Clancy (Eds.). ACM Press, New York, NY, 101–112. DOI: https://doi.org/10.1145/1404520.1404531

[43] Philipp Mayring. 2014. Qualitative Content Analysis: Theoretical Foundation, Basic Procedures and Software Solution. Retrieved from https://nbn-resolving.org/urn:nbn:de:0168-ssoar-395173.

[44] Mary L. McHugh. 2012. Interrater reliability: The kappa statistic. *Biochem. Med.* 22, 3 (2012), 276–282. DOI: https://doi.org/10.11613/BM.2012.031

[45] George A. Miller. 1956. The magical number seven, plus or minus two: Some limits on our capacity for processing information. *Psychol. Rev.* 63, 2 (1956), 81. DOI: https://doi.org/10.1037/h0043158

[46] Briana B. Morrison, Lauren E. Margulieux, Barbara Ericson, and Mark Guzdial. 2016. Subgoals help students solve parsons problems. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education (SIGCSE'16)*. Association for Computing Machinery, New York, NY, 42–47. DOI: https://doi.org/10.1145/2839509.2844617

[47] Laurie Murphy, Renée McCauley, and Sue Fitzgerald. 2012. "Explain in plain English" questions: Implications for teaching. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education (SIGCSE'12)*. Association for Computing Machinery, New York, NY, 385–390. DOI: https://doi.org/10.1145/2157136.2157249

[48] Greg L. Nelson, Benjamin Xie, and Amy J. Ko. 2017. Comprehension first: Evaluating a novel pedagogy and tutoring system for program tracing in CS1. In *Proceedings of the ACM Conference on International Computing Education Research (ICER'17)*. ACM, New York, NY, 2–11. DOI: https://doi.org/10.1145/3105726.3106178

[49] Donald A. Norman. 1983. Some observations on mental models. In *Mental Models*, Dedre Gentner and Albert L. Steven (Eds.). Lawrence Erlbaum Associates Inc., Mahwah, NJ, 7–14.

[50] Unaizah Obaidellah, Mohammed Al Haek, and Peter C.-H. Cheng. 2018. A survey on the usage of eye-tracking in computer programming. *Comput. Surv.* 51, 1 (2018), 1–58. DOI: https://doi.org/10.1145/3145904

[51] Unaizah Obaidellah, Michael Raschke, and Tanja Blascheck. 2019. Classification of strategies for solving programming problems using AoI sequence analysis. In *Proceedings of the 11th ACM Symposium on Eye Tracking Research & Applications (ETRA'19)*. Association for Computing Machinery, New York, NY. DOI: https://doi.org/10.1145/3314111.3319825

[52] Dale Parsons and Patricia Haden. 2006. Parsons' programming puzzles: A fun and effective learning tool for first programming courses. In *Proceedings of the 8th Australasian Conference on Computing Education - Volume 52 (ACE'06)*. Australian Computer Society, Inc., AUS, 157–163. DOI: https://doi.org/10.5555/1151869.1151890

[53] Norman Peitek, Janet Siegmund, and Sven Apel. 2020. What drives the reading order of programmers? An eye tracking study. In *Proceedings of the 28th IEEE/ACM International Conference on Program Comprehension (ICPC'20)*. Association for Computing Machinery, New York, NY, 342–353. DOI: https://doi.org/10.1145/3387904.3389279

[54] Nancy Pennington. 1987. Comprehension strategies in programming. In *Empirical Studies of Programmers: Second Workshop*. Ablex Publishing, Norwood, NJ, 100–113.

[55] Nancy Pennington. 1987. Stimulus structures and mental representations in expert comprehension of computer programs. *Cogn. Psychol.* 19, 3 (1987), 295–341. DOI: https://doi.org/10.1016/0010-0285(87)90007-7

[56] Robert S. Rist. 1989. Schema creation in programming. *Cogn. Sci.* 13, 3 (1989), 389–414. DOI: https://doi.org/10.1016/0364-0213(89)90018-9

[57] Paige Rodeghero and Collin McMillan. 2015. An empirical study on the patterns of eye movement during summarization tasks. In *Proceedings of the ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM'15)*. IEEE Press, Piscataway, NJ, 1–10. DOI: https://doi.org/10.1109/ESEM.2015.7321188

[58] David E. Rumelhart. 1980. Schemata: The building blocks. In *Theoretical Issues in Reading Comprehension: Perspectives from Cognitive Psychology, Linguistics, Artificial Intelligence and Education*, Rand J. Spiro, Bertram C. Bruce, and William F. Brewer (Eds.). Routledge, London, 33–58.

[59] Wolfgang Schnotz and Christian Kürschner. 2007. A reconsideration of cognitive load theory. *Educ. Psychol. Rev.* 19, 4 (01 Dec. 2007), 469–508. DOI: https://doi.org/10.1007/s10648-007-9053-4

[60] Carsten Schulte. 2008. Block model: An educational model of program comprehension as a tool for a scholarly approach to teaching. In *Proceedings of the 4th International Workshop on Computing Education Research (ICER'08)*. ACM, New York, NY, 149–160. DOI: https://doi.org/10.1145/1404520.1404535

[61] Carsten Schulte, Tony Clear, Ahmad Taherkhani, Teresa Busjahn, and James H. Paterson. 2010. An introduction to program comprehension for computer science educators. In *Proceedings of the 2010 ITiCSE Working Group Reports*, Alison Clear and Lori Russell Dag (Eds.). ACM, New York, NY, 65–86. DOI: https://doi.org/10.1145/1971681.1971687

[62] Sue Sentance and Jane Waite. 2017. PRIMM: Exploring pedagogical approaches for teaching text-based programming in school. In *Proceedings of the 12th Workshop on Primary and Secondary Computing Education (WiPSCE'17)*. Association for Computing Machinery, New York, NY, 113–114. DOI: https://doi.org/10.1145/3137065.3137084

[63] Zohreh Sharafi, Bonita Sharif, Yann-Gaël Guéhéneuc, Andrew Begel, Roman Bednarik, and Martha Crosby. 2020. A practical guide on conducting eye tracking studies in software engineering. *Empir. Softw. Eng.* 25, 5 (2020), 3128–3174. DOI: https://doi.org/10.1007/s10664-020-09829-4

[64] Bonita Sharif, Michael Falcone, and Jonathan I. Maletic. 2012. An eye-tracking study on the role of scan time in finding source code defects. In *Proceedings of the Symposium on Eye Tracking Research and Applications (ETRA'12)*. Association for Computing Machinery, New York, NY, 381–384. DOI: https://doi.org/10.1145/2168556.2168642

[65] Elliot Soloway. 1986. Learning to program = Learning to construct mechanisms and explanations. *Commun. ACM* 29, 9 (Sept. 1986), 850–858. DOI: https://doi.org/10.1145/6592.6594

[66] Elliot Soloway, Jeffrey Bonar, and Kate Ehrlich. 1983. Cognitive strategies and looping constructs: An empirical study. *Commun. ACM* 26, 11 (Nov. 1983), 853–860. DOI: https://doi.org/10.1145/182.358436

[67] Elliot Soloway and Kate Ehrlich. 1984. Empirical studies of programming knowledge. *IEEE Trans. Softw. Eng.* SE-10, 5 (1984), 595–609. DOI: https://doi.org/10.1109/TSE.1984.5010283

[68] James C. Spohrer and Elliot Soloway. 1986. Alternatives to construct-based programming misconceptions. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI'86)*. Association for Computing Machinery, New York, NY, 183–191. DOI: https://doi.org/10.1145/22627.22369

[69] James C. Spohrer and Elliot Soloway. 1986. Novice mistakes: Are the folk wisdoms correct? *Commun. ACM* 29, 7 (July 1986), 624–632. DOI: https://doi.org/10.1145/6138.6145

[70] James C. Spohrer, Elliot Soloway, and Edgar Pope. 1985. Where the bugs are. *SIGCHI Bull.* 16, 4 (Apr. 1985), 47–53. DOI: https://doi.org/10.1145/1165385.317465

[71] Anselm R. Strohmaier, Kelsey J. MacKay, Andreas Obersteiner, and Kristina M. Reiss. 2020. Eye-tracking methodology in mathematics education research: A systematic literature review. *Educ. Stud. Math.* 104, 2 (2020), 147–200. DOI: https://doi.org/10.1007/s10649-020-09948-1

[72] John Sweller. 2010. Element interactivity and intrinsic, extraneous, and Germane cognitive load. *Educ. Psychol. Rev.* 22, 2 (01 June 2010), 123–138. DOI: https://doi.org/10.1007/s10648-010-9128-5

[73] Donna Teague. 2015. *Neo-Piagetian Theory and the Novice Programmer*. Ph.D. Dissertation. Queensland University of Technology.

[74] Donna Teague and Raymond Lister. 2014. Programming: Reading, writing and reversing. In *Proceedings of the Conference on Innovation and Technology in Computer Science Education (ITiCSE'14)*. ACM Press, New York, NY, 285–290. DOI: https://doi.org/10.1145/2591708.2591712

[75] Rachel Turner, Michael Falcone, Bonita Sharif, and Alina Lazar. 2014. An eye-tracking study assessing the comprehension of C++ and Python source code. In *Proceedings of the Symposium on Eye Tracking Research and Applications (ETRA'14)*. Association for Computing Machinery, New York, NY, 231–234. DOI: https://doi.org/10.1145/2578153.2578218

[76] Hidetake Uwano, Masahide Nakamura, Akito Monden, and Ken-ichi Matsumoto. 2006. Analyzing individual performance of source code review using reviewers' eye movement. In *Proceedings of the Symposium on Eye Tracking Research & Applications (ETRA'06)*. Association for Computing Machinery, New York, NY, 133–140. DOI: https://doi.org/10.1145/1117309.1117357

[77] Vesa Vainio and Jorma Sajaniemi. 2007. Factors in novice programmers' poor tracing skills. *ACM SIGCSE Bull.* 39, 3 (2007), 236–240. DOI: https://doi.org/10.1145/1269900.1268853

[78] Anne Venables, Grace Tan, and Raymond Lister. 2009. A closer look at tracing, explaining and code writing skills in the novice programmer. In *Proceedings of the Fifth International Workshop on Computing Education Research, ICER 2009*, Michael J. Clancy, Michael E. Caspersen, and Raymond Lister (Eds.). ACM Press, New York, NY, 117–128. DOI: https://doi.org/10.1145/1584322.1584336

[79] Susan Wiedenbeck and Vennila Ramalingam. 1999. Novice comprehension of small programs written in the procedural and object-oriented styles. *Int. J. Hum.-comput. Stud.* 51, 1 (1999), 71 – 87. DOI: https://doi.org/10.1006/ijhc.1999.0269

[80] John Wrenn and Shriram Krishnamurthi. 2020. Will students write tests early without coercion? In *Proceedings of the 20th Koli Calling International Conference on Computing Education Research (Koli Calling'20)*. Association for Computing Machinery, New York, NY. DOI:https://doi.org/10.1145/3428029.3428060

[81] Benjamin Xie, Dastyni Loksa, Greg L. Nelson, Matthew J. Davidsion, Dongsheng Dong, Harrison Kwik, Alex Tan Hui, Leanne Hwa, Min Li, and Amy J. Ko. 2019. A theory of instruction for introductory programming skills. *Comput. Sci. Educ.* 29, 2-3 (2019), 205–253. DOI: https://doi.org/10.1080/08993408.2019.1565235

[82] Johannes Zagermann, Ulrike Pfeil, and Harald Reiterer. 2016. Measuring cognitive load using eye tracking technology in visual computing. In *Proceedings of the 6th Workshop on Beyond Time and Errors on Novel Evaluation Methods for Visualization (BELIV'16)*. Association for Computing Machinery, New York, NY, 78–85. DOI: https://doi.org/10.1145/2993901.2993908

Received June 2021; revised September 2021; accepted September 2021