Heinonen, Ava; Hellas, Arto

Time-constrained Code Recall Tasks for Monitoring the Development of Programming Plans

Published in:
SIGCSE 2023 - Proceedings of the 54th ACM Technical Symposium on Computer Science Education

DOI:
10.1145/3545945.3569757

Published: 02/03/2023

Document Version
Publisher's PDF, also known as Version of record

Time-constrained Code Recall Tasks for Monitoring the Development of Programming Plans

Ava Heinonen
ava.heinonen@aalto.fi
Aalto University
Espoo, Finland

Arto Hellas
arto.hellas@aalto.fi
Aalto University
Espoo, Finland

ABSTRACT

Programmers rely on the recognition and utilization of reoccurring code sequences to understand and create code. Knowledge of these sequences – *programming plans* – has been shown to be a factor that differentiates novice programmers from experts. Although the information on the development of programming plans would be beneficial to both teachers and students, explicitly following their development over a longer time period is scarce. In this article, we describe an easy-to-apply methodology for monitoring the development of programming plans. The development of programming plans is evaluated with *time-constrained code recall tasks*, where students are shown snippets of code for a short period of time, after which they write the snippets they saw. To determine the existence of programming plans, the short duration is designed so that reading the shown code is not feasible in the given time period. We demonstrate the methodology through an experiment in which we studied the development of programming plans in students in a beginner web programming course.

CCS CONCEPTS

• **Applied computing** → *Interactive learning environments*; • **Social and professional topics** → *Computing education.*

KEYWORDS

code recall, time-constrained code recall, programming plans, focal elements, web development, programming course

ACM Reference Format:
Ava Heinonen and Arto Hellas. 2023. Time-constrained Code Recall Tasks for Monitoring the Development of Programming Plans. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1 (SIGCSE 2023), March 15–18, 2023, Toronto, ON, Canada.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3545945.3569757

1 INTRODUCTION

Learning to program has received a lot of attention in computing education research [20, 22, 27]. It occurs at multiple levels, as students learn to understand both the syntax and semantics of a programming language, pragmatic aspects of programming like working with the available tools [11] and acquire plans that are used to achieve reoccurring goals. Learning and increased expertise are intertwined with an improved ability to handle complex information as organized chunks [9].

In the domain of programming, these organized chunks are referred to as programming plans. Programming plan knowledge refers to programmers' knowledge of reoccurring code patterns such as code sequences and templates. Knowledge of programming plans has been recognized as one of the key types of knowledge for programming expertise [4, 10, 24], and is a key factor in program comprehension [17]. Due to the importance of plan knowledge in program comprehension and programming, the ability to recognize and utilize programming plans has been recognized as an important learning objective in programming education [16, 28]. However, studies into students' programming plans mostly focus on introductory programming, and, are fairly static in that they do not focus on the evolution of the plans (e.g. [14, 23]).

The overall theme of this work is *how do we draw a line between a novice and a more advanced student, and how do we assess the student's expertise?* Prior programming experience has been measured using tests [21, 25] and surveys [12, 15]. However, these approaches are rather coarse-grained, allowing the student to search for answers elsewhere and misrepresent information. Researchers have also used naturally accumulating data, such as keystrokes, as an indicator of experience [19, 26]. These studies, however, have mainly focused on identifying whether a student has programmed previously or not.

In the present work, we explore monitoring programming plan development as a measure of the development of programming expertise. We propose using time-constrained code recall tasks – showing code to students for a limited time and asking them to recall it – for assessing programming plan knowledge. Our research questions for the present study are as follows:

- **RQ1** Can a sequence of time-constrained code recall tasks be used to analyze programming plan development?
- **RQ2** How does studying affect the correctness of students' time-constrained code recall task results?
- **RQ3** What aspects of code elements are most focal in students' web programming plans?

2 BACKGROUND

Researchers have worked for decades to understand the types of skills and knowledge that underlie the performance of expert programmers. These also represent some of the skills and knowledge that we should aim to teach in programming education. Programming plan knowledge has long been at the center of these discussions as an important type of knowledge that underlies programming expertise [1–3, 7].

SIGCSE 2023, March 15–18, 2023, Toronto, ON, Canada

ACM ISBN 978-1-4503-9431-4/23/03.
https://doi.org/10.1145/3545945.3569757

One of the differences between experienced and novice programmers has been shown to be, that instead of reading and mentally representing code as individual statements, experienced programmers have the ability to mentally organize code into meaningful chunks [2, 3]. Thus, experienced programmers can comprehend programs as configurations of meaningful elements, rather than as individual code statements [1–3]. However, these differences between novices and experienced programmers diminish if the code studied is not in a meaningful order [2–4]. This strongly supports the idea that experienced programmers' ability to mentally organize code into meaningful chunks comes from their ability to recognize and comprehend groupings of code statements as instantiations of well-known patterns, programming plans [1, 3]. Thus, experienced programmers have acquired knowledge of these reoccurring patterns, which is often referred to as programming plan knowledge or programming schemata [10, 24]. Knowledge of programming plans has been recognized as one of the types of knowledge important for programming experience [4, 10, 24].

As programming plan knowledge underlies expert performance, the ability to recognize and use programming plans has been described as an important skill for students to learn in programming education [16, 28]. Studies have recognized the importance of programming plan knowledge in computer science education [16, 28], and have shown that some of the plan knowledge important for expert performance may indeed develop due to explicit instruction [7]. Although the promotion of programming plan knowledge and use has been recognized as an important aspect of programming education, many studies on programming plans and their development have been conducted in study settings that use old programming languages and technologies that are no longer used in contemporary settings [14]. Therefore, studies have recognized the need for a better understanding of student programming plans in contemporary contexts [14].

Recall studies have been used for decades to investigate the knowledge, use, and development of programming plans [1, 4, 7]. The effectiveness of these methods comes from the knowledge that programming plan knowledge allows experienced programmers to encode more information in one glance [3, 4]. Thus, experienced programmers can comprehend and memorize code faster and more accurately as they can effectively process code as configurations of meaningful elements instead of relying on the line-by-line reading of individual code statements [1, 4, 7].

Studies have shown that when memorizing code that is presented in the expected order, experienced programmers perform better than novices in recall tasks [1]. These studies have shown that experienced programmers can recall more code and recall the code with more accuracy [1, 3, 4]. Thus, the presence and development of programming plans can be deduced, to some extent, from the participant's recall performance [1, 3, 4]. Recall studies have also shown differences in the areas of code that novices and experienced programmers recall [8]. Some lines of code, the so-called focal lines, seem to be the most indicative of the presence of a certain plan and are used by experienced programmers to recognize plans in code [8]. These lines are recalled with greater accuracy and provide information about which aspects of the code are emphasized in programmers' plan knowledge [8].

3 METHODOLOGY

3.1 Context

The study was carried out in an online introductory course in Web Software Development at Aalto University. The course is typically taken by computer science students during their second year. During our study, the course was continuously ongoing due to the Covid-19 pandemic, and students could join the course at any point in their studies. Therefore, the experiment began in an ongoing course with students at different points in the course, and some students first saw the experiment halfway through the course.

The course uses an online textbook that contains learning materials, programming exercises, quizzes, and project handouts. The course uses JavaScript, which the students had not used in their studies prior to the course.

3.2 Time-constrained code recall system

To answer RQ1, we developed a time-constrained code recall system to perform code recall experiments in an online environment. The key functionality of the system is that it displays a piece of code for a limited time and asks the participant to write the code that they saw. Response code and interactions with the system are stored. Copying and pasting code was disabled within the system.

The system was integrated into the Web Software Development course. Starting the task opened a dialog and grayed out the course materials, focusing the student on the task. The flow to complete a code recall task using the system is shown in Figure 1. First, the student is shown a dialog explaining the task. This is followed by a timed dialog that contains the code that the student needs to remember. Once the time runs out, the student is shown an editor where they can write the code they had been shown.

3.3 Tasks and data collection

During the code recall tasks, the students were asked to replicate a piece of code after viewing it for 10 seconds. We used two pieces of code: the first, shown in Listing 1, was a simple function that prints the text `Hello world!` on the console. The second, shown in Listing 2, was a middleware function that catches and logs errors in a request and adds information about the error to a response.

Listing 1: Hello world recall code

```
const hello = () => {
  console.log("Hello world!");
}
```

Listing 2: Error middleware recall code

```
const errorMiddleware = async (context, next) => {
  try {
    await next();
  } catch (e) {
    console.log(e);
    context.response.body = "Error when processing
     request";
  }
}
```

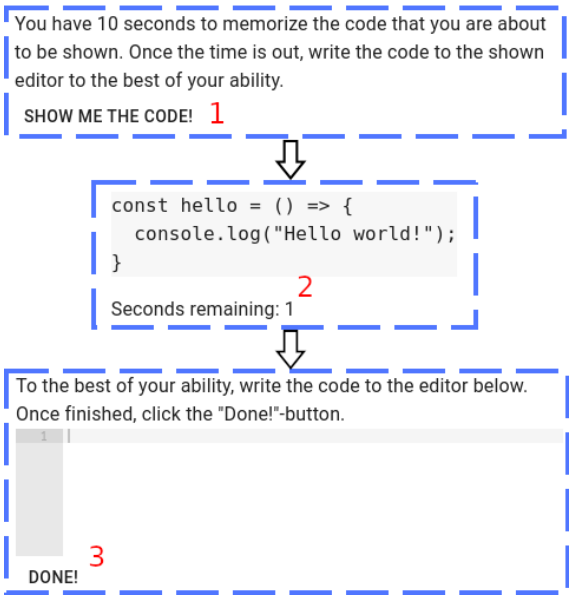

Figure 1: Flow of completing a single recall task. Students are first presented with a dialog that shows them the task. When they click on the button (1), they are shown a dialog with the code that they need to study and later recall, with a countdown timer (2). When the time runs to 0, they are shown an editor for the code that they previously saw. Once they finish a single task by clicking the done-button (3), the next code recall task is shown, given that more tasks are configured to the system.

The time-constrained code recall system was embedded in the course's online textbook at three parts of the course. The first time the students saw it was at the beginning of the course when the students had used the `console.log` command used in Listing 1 but had not worked with middlewares used in Listing 2. The second time the students saw it was midway through the course when students had worked with code similar to both listings. The third time was towards the end of the course when students had more experience with code similar to both listings and beyond. The codes that students were expected to recall were the same each time.

Participation was voluntary and students were not compensated for participating.

3.4 Participants

Students could complete none, one, or both of the recall tasks at any of the three times. 344 students participated in the experiment, completing 802 individual recall tasks. After removing responses with no content at all, we had 616 recall responses from 332 students. Table 1 describes the number of students participating and the number of recall tasks completed each time.

Most of the students participated only one time. 41 students participated in the experiment both the first and second times, 38 students the second and third times, and 14 students at all times. Therefore, we focus on the results within each time as cohorts, and beyond the subpopulation analysis in Section 4.2, we do not consider the progress of individual students.

Table 1: The number of students who participated in the experiment each time and the total number of individual recall tasks completed by the students each time.

Time	Students	Responses
All	322	616
Time 1	228	330
Time 2	116	185
Time 3	68	101

3.5 Analysis

3.5.1 Analysis: RQ2. To answer RQ2, we analyze differences in code writing time and the correctness of the response at the three times. The correctness of the response code was measured using the edit distance between the example code and the response code.

For the edit distance, we used two different approaches. The first is a simple Levenshtein distance (i.e., the number of character changes needed to transform one string to another), and the second is a programming language token distance (i.e., the number of token changes needed to transform one code to another). For the token distance, we use a lexer[1], to transform the written code into tokens. When calculating the character-based edit distance, we took any whitespace characters into account in the calculation. For token-based edit distance, whitespace characters were ignored.

The data is not normally distributed as confirmed using Shapiro-Wilk tests. Therefore, we conduct pairwise Mann-Whitney U tests to study the statistical significance of differences when performing statistical analyzes. When reporting analysis results, we report the Mann-Whitney U test statistic, uncorrected p values, and use Rank Biserial Correlation (RBC) when reporting effect sizes.

3.5.2 Analysis: RQ3. To answer RQ3, we analyze differences in the accuracy of student recall in different parts of code listings. For each student response, we used Python difflib to create a list of non-overlapping matching substrings between the response code and the example code. We discarded all matching sequences that were only one character long to prevent non-authentic matches between characters that occur multiple times in the listings. We then analyzed how frequently each index of the example code was present in the matching sequences. This gave us an approximation of how frequently students were able to recall that part of the code.

4 RESULTS

4.1 Writing time, code length and code correctness

As shown in Table 2, for Listing 1, the mean writing time decreased between the three times. However, for Listing 2, the mean writing time increased between the three times. Table 2 displays the writing time, code length, and character-based and token-based edit distances for both listings at the three times.

[1]We used the *JavascriptLexer* from *Pygments* 2.11.2 (https://pygments.org/)

As shown in Table 2 the average code length (in characters) remained somewhat constant between the times for Listing 1. For Listing 2 the average code length increased from 77.3 to 125.5 from time 1 to time 3.

Overall, as shown in Table 2, there is a clear drop in both character-based and token-based edit distances for both listings between times 1 and 2 and a more subtle drop between times 2 and 3.

To further study these changes, we performed Mann-Whitney U tests that compared character- and token-based edit distances between the times for both listings. For both Listing 1 and Listing 2, there is a statistically significant decrease in both character-based and token-based edit distance between time 1 and time 2. When considering the differences between times 2 and 3, the differences are not statistically significant. The statistical tests and results are summarized in Table 3.

4.2 Changes in recall over individual students

Only a handful of students participated in the experiment three times. To provide further evidence for RQ2 on changes between times 1 and 2, we analyzed the subpopulation of $n = 41$ students who participated in the experiment on the first and second times.

For Listing 1, the average writing time decreased slightly, from 63.6 seconds at time 1 to 61.1 seconds at time 2 (median change from 55.0 to 46.5). The average code length increased from 50.8 to 53.9 characters. The average edit distances decreased from 6.2 to 3.6 for the character-based edit distance and from 2.8 to 2.1 for the token-based edit distances. However, changes in the edit distance are not statistically significant for the character-based edit distance ($p = 0.12$, $RBC = -0.20$) or the token-based edit distance ($p = 0.27$, $RBC = -0.14$).

For Listing 2, the average writing time increased from 66.6 to 96.4 seconds (median change from 66.8 to 88.3) between times 1 and 2. The average length of the code increased from 64.6 to 118.4 characters (median change from 58.0 to 115.0). The average edit distances decreased considerably, from 119.7 to 75.6 (median change from 123.0 to 77.0) for the character-based edit distance and from 26.0 to 14.7 for the token-based edit distance (median change from 29.0 to 11.0). Changes in edit distance are statistically significant for character-based edit distance ($p = 0.0013$) with medium effect size ($RBC = -0.55$) and for token-based edit distance ($p = 0.0012$) also with medium effect size ($RBC = -0.55$).

4.3 Focal elements

For the analysis of focal elements for *RQ3*, we compared the response codes with the sample codes to assess which areas of the sample code were recalled by the participants.

For Listing 1 the analysis shows that the overall correctness of the participants' recall increased from time 1 to time 3. At the first time, some participants were able to recall only the first few characters of the code Listing correctly, as would be expected if they read the code character by character, thus not being able to investigate the whole function in 10 seconds. The second time, the participants were able to recall the first few characters and also most of the function body correctly. However, the second time around, it is still difficult for participants to recall the form of the arrow function. For the third time, almost all participants were able to recall the code Listing correctly, and only a few characters in the function body and the last curly brace were recalled with less than 90% precision.

Figure 2 displays the percentages of correct recall for each character in code Listing 2 at the three times. For the first time, the results are similar to those of Listing 1, showing that participants recall the first few characters correctly before their recall drops. We also see that the async statement and the await statement are recalled with less accuracy than others. The second time, the participants were able to recall the first few characters and also most of the function body correctly up to the last statement of the catch clause. However, the async and the await statements remain less well recalled than other areas of the code, whereas the "try" and "catch" keywords are recalled better. In the third time, the results are similar to time 2, however, the percentage of correct recall of the last statements of the catch clause increases.

5 DISCUSSION

5.1 Code recall and expertise

Our results indicate, that students become more accurate at identifying and recalling code through practice. The results show improvements in code recall at both the character and token levels. This is in line with previous studies, which have shown that experienced programmers can recall more code and recall code more accurately [1, 3, 4]. Similar results have been observed in other domains as well. For example, studies on chess players show that experienced chess players recall board configurations shown to them only for a brief moment more accurately than less experienced chess players [9].

Our results on Listing 1 also show that students became faster in writing the code. The improved writing time is supported by previous studies on links between typing speed and experience. In these studies, more experienced students type particular character pairs faster than less experienced students [19, 26]. Previous studies have also investigated the typing speed of particular constructs over the duration of a course, where the typing speed of often used constructs seems to improve over time [13]. Although we cannot see the same improvement in writing time for the second listing, we see an increase in code length between the times for Listing 2. This shows that the students wrote more as they became more familiar with similar codes throughout the course.

5.2 Focal elements

In our analysis of focal elements in response codes, we can see that some areas of the code are recalled better than others. For Listing 1, the results are less conclusive, as this Listing was recalled well overall. Therefore, we focus primarily on Listing 2, where the evolution of the recall correctness of different areas of the code is shown in Figure 2. In time 1, the structure of a function body and the keywords try and catch are recalled better than other parts of the function. At that point, students have already worked with functions, which probably contributes to the recall performance. Similarly, while the course has not yet touched on error handling at that point, students attending the web software development course have learned to use error handlers in their previous programming courses, which likely reflects in their recall results.

Table 2: Writing time (in seconds), code length (in characters), character-based edit distance (Character ED, in characters), and token-based edit distance (Token ED, in tokens) for Listings 1 and 2 at the three times. Reported using the number of samples *n*, median, mean, and standard deviation *Std*.

		Time 1				Time 2				Time 3			
		n	Median	Mean	Std	n	Median	Mean	Std	n	Median	Mean	Std
L1	Writing time	226	55.7	66.9	55.9	116	49.4	56.9	40.2	68	43.1	50.0	23.8
	Code length	226	53.0	49.2	11.3	116	54.0	53.2	4.4	68	54.0	52.7	4.8
	Character ED	226	4.0	8.8	11.0	116	2.0	3.8	4.7	68	2.0	3.3	5.3
	Token ED	226	2.0	3.2	3.3	116	1.0	2.2	2.3	68	1.0	1.7	2.0
L2	Writing time	104	66.6	73.5	50.2	69	86.8	87.2	36.2	32	92.5	94.3	42.7
	Code length	104	58.5	77.3	58.1	69	126.0	121.2	48.4	32	143.0	125.5	55.2
	Character ED	104	129.0	112.2	52.4	69	70.0	71.3	44.4	32	52.0	63.8	51.1
	Token ED	104	29.0	25.0	13.2	69	11.0	14.1	10.9	32	9.0	13.3	12.3

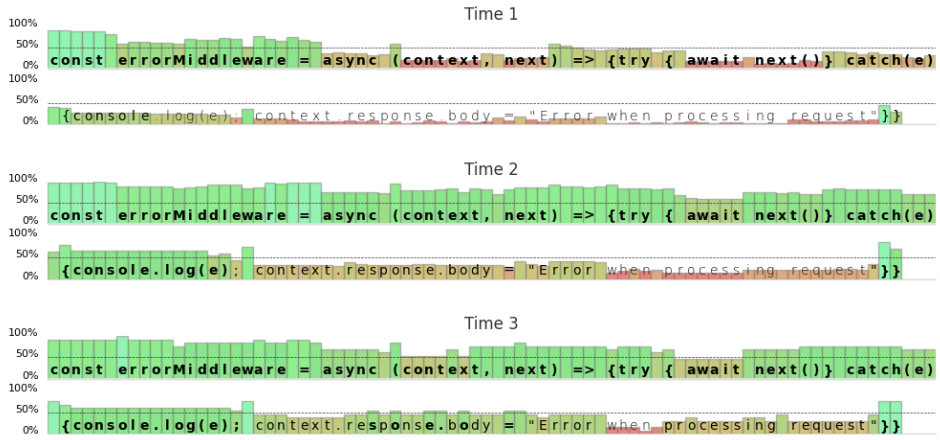

Figure 2: Participant recall of each character in the code Listing 2 (error middleware) at each of the three times.

Table 3: Results of the Mann-Whitney U test comparing the edit distances between the code shown and the code written by the student for both Listing 1 (L1) and Listing 2 (L2) between time 1 and time 2 as well as between times 2 and 3.

		Time 1 vs. 2		Time 2 vs. 3	
		Character	Token	Character	Token
L1	*U*	16977.0	15399.0	4519.5	4467.0
	p	0.000007	0.007072	0.095443	0.124233
	RBC	-0.30	-0.17	-0.15	-0.13
L2	*U*	5220.5	5208.5	1289.0	1252.0
	p	0.0000004	0.0000005	0.283223	0.418466
	RBC	-0.45	-0.45	-0.13	-0.10

When comparing the recall accuracy of different parts of Listing 2 between the first and second times, we see that the students become better at recalling the function parameters and the keywords related to asynchronous processing. They also become better at recalling the function body.

Interestingly, when comparing the second and third times, we see that the students become somewhat worse at recalling the function parameter *context*. This may be due to the course that teaches object decomposition quite early on. Thus, the students do not use the *context* object but rely on decomposition to use the response directly.

We also observe that some students may skip words that they consider redundant. In Listing 2, the word *when* in the error message is poorly remembered. Students often write "Error processing request" instead of "Error when processing request". There is also a minor decline in the recall of async and await between the second and the third times. This could indicate that students start to forget the elements of middleware. Toward the third time, middleware plays a very minor role in the course, and project templates typically include ready-made middlewares, which means that students do not have to practice writing them.

These results are in line with previous studies that have suggested that there are differences in the areas of code that novices and more experienced programmers recall [8]. However, while [8] discusses *focal lines*, based on differences in recalling try-catch, it is

a good question if we should be discussing *focal structures* or *focal patterns*, due to experienced programmers relying on configurations of meaningful elements instead of the line-by-line interpretation of code [1, 4, 7], which has also been observed in eye tracking studies (see, e.g. [5, 6]).

5.3 Towards assessing plan development

Our results provide credibility to the notion of assessing plan development using a time-restrained code-recall task within an online environment. In general, our results were in line with prior lab-based code recall studies. Furthermore, the observation that an online environment can be used to conduct experiments classically done within laboratory settings has been pointed out also in the past in computing education research (see e.g. [18]).

Although the present study was conducted in a Web software development course with advanced students, we see the potential to incorporate similar studies into introductory programming courses. Researchers have previously looked into plan composition in introductory programming courses [14, 23], and using timed code recall tasks with suitably designed code listings would be a good next step in evaluating whether timed code recall would be a good addition to the broader assessment toolkit. We see the possibility of using time-constrained code recall both for research purposes and possibly also in exams as an additional piece of evidence of increasing expertise.

5.4 Limitations of work

First, we acknowledge the selection bias as the participants were volunteers from a specific course. Thus, we cannot state whether or to what extent our observations generalize. However, based on previous research on code comprehension, we believe that the development of programming plans in novice and more experienced students could be quantified in a broader classroom context.

Second, we did not have access to information on students' programming experience or grades. Therefore, we were unable to assess the effect of these on the results. Third, since the listings were the same throughout the study, there is the possibility that students learned (parts of) the listings by the second or third time. Fortunately, only a few students participated more than once, so we have some evidence that this is not the case.

Fourth, we acknowledge that students could have written the code with different symbol names even though the task was to write the shown code. Our analysis focused on the exactness of the written code and did not consider the possibility that the same functionality could be written with different symbol names.

Fifth, we did not make corrections for multiple comparisons. A total of eight statistical significance tests were conducted, although many of them overlapped (character- vs. token-based edit distances, whole population vs. sub-population). If we applied a strict Bonferroni correction, the only result that would no longer be statistically significant would be the token-based change for Listing 1 between times 1 and 2. Instead of relying on the presented statistics in making decisions on classroom activities, we recommend using our results as a starting point for future studies. Finally, we acknowledge that the code recall task requires eyesight and is not suitable for visually impaired students.

6 CONCLUSION

In this study, we evaluated the use of a time-constrained code recall task for monitoring the development of programming plans in an online web software development course. In the code recall task, students were shown a code listing for a short period of time, which they then had to recall and replicate. The experiment was presented three times in the course materials. At each time, students were shown two listings, one easier and one more complex. Our research questions and their answers are as follows:

RQ1: Can a sequence of time-constrained code recall tasks be used to analyze programming plan development? **Answer:** Our results show that we can measure the development of students programming plans using the time-constrained code recall system. We were able to see increased accuracy in students' web programming plans from time 1 to time 3, which indicates an increased ability to recognize and recall code after only a brief glance.

RQ2: How does studying affect the correctness of students' time-constrained code recall task results? **Answer:** The students were quite good at recalling the easier code already the first time, possibly because they were already familiar with some of the constructs. However, there was still a minor improvement in recall correctness from the first to the second time. For the second, more complex, listing, the students had no prior exposure to similar code. Thus, recalling the listing the first time was more challenging, and we observed a considerable improvement in recall correctness from the first time to the second time. The second time, the students also had some exposure to code in the complex listing, which made it easier to recall the code. No major improvements in terms of correctness were observed between the second and third time.

RQ3: What aspects of code elements are most focal in students' web programming plans **Answer:** For the first time, we observed that the key focal code elements were the basic structure of the function and error handling. As the student's expertise grew during the course, the middleware function parameters and the functionality used for handling asynchronous requests became more familiar. We also observed that students omitted redundant content, which was evidenced in the output sent as a response – students often skipped the word 'when', even though it was present.

Our results highlight that time-constrained code recall tasks are a potential addition to the toolkit used for assessing student development. As a part of our future work, we are employing the same methodology in an introductory programming course to assess the development of programming plans in novice programmers. We are also replicating the present study in the web software development course, where we also collect data on students' prior programming background and their progress in the course. In future iterations of the study, we are interested in exploring possibilities to consider the relative importance of recalling different parts of the code. For example, accurately recalling the correct syntax for a function body may be more important in students' learning than the ability to recall symbol names.

ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their valuable feedback that helped us to improve the manuscript.

REFERENCES

[1] Beth Adelson. 1981. Problem solving and the development of abstract categories in programming languages. *Memory Cognition* (1981). https://doi.org/10.3758/bf03197568

[2] Woodrow Barfield. 1997. Skilled performance on software as a function of domain expertise and program organization. *Perceptual and Motor Skills* (1997). https://doi.org/10.2466/pms.1997.85.3f.1471

[3] Woodrow Barfield, Woodrow Barfield, and Woodrow Barfield. 1986. Expert-novice differences for software: implications for problem-solving and knowledge acquisition. *Behaviour Information Technology* (1986). https://doi.org/10.1080/01449298608914495

[4] Allan G. Bateson, Ralph A. Alexander, and Martin D. Murphy. 1987. Cognitive processing differences between novice and expert computer programmers. *International Journal of Human-computer Studies International Journal of Man-machine Studies* (1987). https://doi.org/10.1016/s0020-7373(87)80058-5

[5] Roman Bednarik, Teresa Busjahn, Agostino Gibaldi, Alireza Ahadi, Maria Bielikova, Martha Crosby, Kai Essig, Fabian Fagerholm, Ahmad Jbara, Raymond Lister, et al. 2020. EMIP: The eye movements in programming dataset. *Science of Computer Programming* 198 (2020), 102520.

[6] Martha E Crosby and Jan Stelovsky. 1990. How do we read algorithms? A case study. *Computer* 23, 1 (1990), 25–35.

[7] S. P. Davies. 1990. The nature and development of programming plans. *International Journal of Human-computer Studies International Journal of Man-machine Studies* (1990). https://doi.org/10.1016/s0020-7373(05)80143-9

[8] Simon P. Davies. 1994. Knowledge restructuring and the acquisition of programming expertise. *International Journal of Human-computer Studies International Journal of Man-machine Studies* (1994). https://doi.org/10.1006/ijhc.1994.1032

[9] Adriaan D de Groot. 1978. Thought and choice in chess. (1978).

[10] Françoise Détienne and Elliot Soloway. 1990. An empirically-derived control structure for the process of program understanding. *International Journal of Man-Machine Studies* 33, 3 (1990), 323–342.

[11] Benedict Du Boulay. 1986. Some difficulties of learning to program. *Journal of Educational Computing Research* 2, 1 (1986), 57–73.

[12] Rodrigo Silva Duran, Jan-Mikael Rybicki, Arto Hellas, and Sanna Suoranta. 2019. Towards a common instrument for measuring prior programming knowledge. In *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education*. 443–449.

[13] John Edwards, Juho Leinonen, Chetan Birthare, Albina Zavgorodniaia, and Arto Hellas. 2020. Programming Versus Natural Language: On the Effect of Context on Typing in CS1. In *Proceedings of the 2020 ACM Conference on International Computing Education Research*. 204–215.

[14] Kathi Fisler, Shriram Krishnamurthi, and Janet Siegmund. 2016. Modernizing plan-composition studies. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*. 211–216.

[15] Dianne Hagan and Selby Markham. 2000. Does it help to have some programming experience before beginning a computing degree program?. In *Proceedings of the 5th annual SIGCSE/SIGCUE ITiCSEconference on Innovation and technology in computer science education*. 25–28.

[16] Cruz Izu, Carsten Schulte, Ashish Aggarwal, Quintin Cutts, Rodrigo Duran, Mirela Gutica, Birte Heinemann, Eileen Kraemer, Violetta Lonati, Claudio Mirolo, et al. 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*. 27–52.

[17] Philipp Kather, Rodrigo Duran, and Jan Vahrenhold. 2021. Through (tracking) their eyes: Abstraction and complexity in program comprehension. *ACM Transactions on Computing Education (TOCE)* 22, 2 (2021), 1–33.

[18] Marianne Leinikka, Arto Vihavainen, Jani Lukander, and Satu Pakarinen. 2014. Cognitive flexibility and programming performance. In *Psychology of programming interest group workshop*. 1–11.

[19] Juho Leinonen, Krista Longi, Arto Klami, and Arto Vihavainen. 2016. Automatic inference of programming performance and experience from typing patterns. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*. 132–137.

[20] Andrew Luxton-Reilly, Ibrahim Albluwi, Brett A Becker, Michail Giannakos, Amruth N Kumar, Linda Ott, James Paterson, Michael James Scott, Judy Sheard, and Claudia Szabo. 2018. Introductory programming: a systematic literature review. In *Proceedings Companion of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education*. 55–106.

[21] Miranda C Parker, Mark Guzdial, and Shelly Engleman. 2016. Replication, validation, and use of a language independent CS1 knowledge assessment. In *Proceedings of the 2016 ACM conference on international computing education research*. 93–101.

[22] Arnold Pears, Stephen Seidman, Lauri Malmi, Linda Mannila, Elizabeth Adams, Jens Bennedsen, Marie Devlin, and James Paterson. 2007. A survey of literature on the teaching of introductory programming. *Working group reports on ITiCSE on Innovation and technology in computer science education* (2007), 204–223.

[23] Otto Seppälä, Petri Ihantola, Essi Isohanni, Juha Sorva, and Arto Vihavainen. 2015. Do we know how difficult the rainfall problem is?. In *Proceedings of the 15th Koli Calling Conference on Computing Education Research*. 87–96.

[24] Elliot Soloway and Kate Ehrlich. 1984. Empirical studies of programming knowledge. *IEEE Transactions on software engineering* 5 (1984), 595–609.

[25] Allison Elliott Tew and Mark Guzdial. 2010. Developing a validated assessment of fundamental CS1 concepts. In *Proceedings of the 41st ACM technical symposium on Computer science education*. 97–101.

[26] Richard C Thomas, Amela Karahasanovic, and Gregor E Kennedy. 2005. An investigation into keystroke latency metrics as an indicator of programming performance. In *Proceedings of the 7th Australasian conference on Computing education-Volume 42*. 127–134.

[27] Arto Vihavainen, Jonne Airaksinen, and Christopher Watson. 2014. A systematic review of approaches for teaching introductory programming and their influence on success. In *Proceedings of the tenth annual conference on International computing education research*. 19–26.

[28] Benjamin Xie, Dastyni Loksa, Greg L Nelson, Matthew J Davidson, Dongsheng Dong, Harrison Kwik, Alex Hui Tan, Leanne Hwa, Min Li, and Amy J Ko. 2019. A theory of instruction for introductory programming skills. *Computer Science Education* 29, 2-3 (2019), 205–253.