# An Analysis of Program Comprehension Process by Eye Movement Mapping to Syntax Trees

Haruhiko Yoshioka and Hidetake Uwano

**Abstract** To analyze the developers' eye movement to the source code while understanding a program, researchers need to map eye movements to the position of the source code being read. However, the eye movements are recorded as a list of the display's coordinates, hence scrolling and window movement on the editor make mapping difficult. Further, to extract the comprehension behavior for different source codes, it is necessary to consider the differences in control flow, format, identifiers, and other factors, which is time-consuming during analysis. This study analyzed eye movement during program comprehension tasks using our proposed method. Our method identifies the word being looked at and converts it into a corresponding syntax node. The experimental results show a significant difference in the syntax node focused on between the understood/not-understood participant groups. The experiment did not use the conventional eye movement analysis for individual source codes and demonstrated the effectiveness of the proposed method.

**Key words:** Eye tracking, program comprehension, syntax analysis.

## 1 Introduction

Program comprehension is one of the research areas of software engineering. Discovering effective and efficient ways to understand programs contributes to cost savings in development and software testing. There have been several studies measuring the process by which developers read source code to analyze program under-

Haruhiko Yoshioka
Department of Advanced Information Engineering National Institute of Technology, Nara College, Yatacho22, Yamato-koriyama, Nara 639-1080, Japan, e-mail: ai1103@nara.kosen-ac.jp

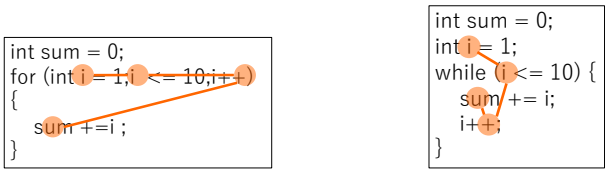
Hidetake Uwano
Department of Information Engineering National Institute of Technology, Nara College, Yatacho22, Yamato-koriyama, Nara 639-1080, Japan, e-mail: uwano@info.nara-k.ac.jp

standing [6, 13, 4, 2]. Extracting the comprehension patterns and strategies of good developers improves education and development efficiency.

Common eye trackers used in the studies record the participants' eye movements as a time series of the display coordinates. Eye movements that remain for a certain period of time in a certain area are integrated as fixations for analysis. Some studies analyzing eye movements relative to source code focused on changes in viewed coordinate positions and found that experts tend to have more up-and-down eye movement than novices[6, 3]. In the usual development environment, source code is displayed in an editor or Integrated Development Environment (IDE). Therefore, the source code displayed at the same coordinates can change by moving the position of the window, scrolling in the code, changing tabs, etc. Some studies convert eye movements into the lines and columns of the source code by recording the eye movement and operation history[12, 5]. Eye movements expressed in lines and columns can identify the eye at the same line or word even when operations differ among participants, which makes it possible to extract features from different participants' eye movements.

Meanwhile, two source code snippets with the same function but different format/control flows are hard to compare because they are recorded as different coordinates or line/column numbers. Figure 1 shows two eye movements for source code snippets implementing the same processing contents but using different control syntaxes. The both codes calculate the sum from 1 to 10 and store it in the variable "sum," but (a) uses a "for" statement, while (b) uses a "while" statement. The circles in the figures represent the fixations of the eye movement (fixation points), and the lines show the connections between consecutive fixation points. Both eye movements show that participants look at the same process (initialization of index variables, condition expressions, increase of the index, and calculation of results) in the same order. However, as their codes have different structures, the eye movements are different: left to right for the "for" statement and up to down for the "while" statement.

In this paper, we analyze eye movements using a method proposed in our previous research [15], converting the coordinate-wise eye movements into transitions to nodes in a syntax tree. The method extracts the syntax tree of the source code and determines the nodes that correspond to the eye movement. In the experiment, we compare fixation ratio for each type of syntax element between the understood/not-understood participant groups.




(a) for statement (b) while statement

**Fig. 1** Eye movements for two source codes

## 2 Related Research

Measurement of eye movements is well used to analyze the difference between novice and skilled programmers. The state where a person's gaze remains within a certain range for a certain period of time is referred to as fixation, and its central coordinate is the fixation point. Successive fixation points represent temporal changes in the point of interest of the reader. Hauser et al. compared differences in eye movement based on fixation points between beginner and expert developers[6]. Their results showed that the experts tended to read the source code non-linearly, while beginners read it linearly.

Another definition used for eye movement analysis is Area of Interest (AOI). The AOI is the rectangle or circle representing part of the region to the image or text. If successive gazes are observed to be on the same AOI, eye movement is integrated as a fixation at that AOI. Rodeghero et al. defined four types of AOIs in source code: method declarations, method calls, control flows, and others, and they analyzed the length of fixation time for each[13]. Their results revealed that skilled Java programmers looked at method calls and control flows for a longer duration than method declarations. Crosby et al. compared time for review between beginners and experts for each AOI defined for words in the source code and found that experts spend less time looking at comments than beginners[4]. Peitek et al. compared the reading of beginners and intermediates by converting coordinate-based data into lines and found that intermediate readers moved their eyes to lines at the top of the source code more frequently[12].

Displaying the source code as an image eliminates the changes of scrolling and tab switching, and many analyses have focused on short source code or code fragments that can be displayed on a single screen. Meanwhile, some studies have created experimental programs to acquire line/column data from eye movements from coordinates based on operation history, such as scrolling or switching between displays of multiple source codes. The open source software iTrace converts eye-tracking information from coordinates to source code line/column numbers[1]. iTrace has been implemented as a plug-in for Eclipse, and it can be used to perform eye movement analysis[5, 1, 14]. Kevic et al. used iTrace on three bug-fixing tasks. They found that developers focus on small parts of methods that are often related to data flow. Eye movement within methods showed developers chase variables flows.[9, 8].

## 3 Eye Movement Mapped to Syntax Trees

Our proposed method converts the eye movement from the coordinate unit into transitions to the nodes of a syntax tree generated from the source code. Figure 2 illustrates an overview of the method. The squares represent the modules constituting the method, and the thin arrows represent the data flow. The eye movement of the par-

[1] https://www.i-trace.org

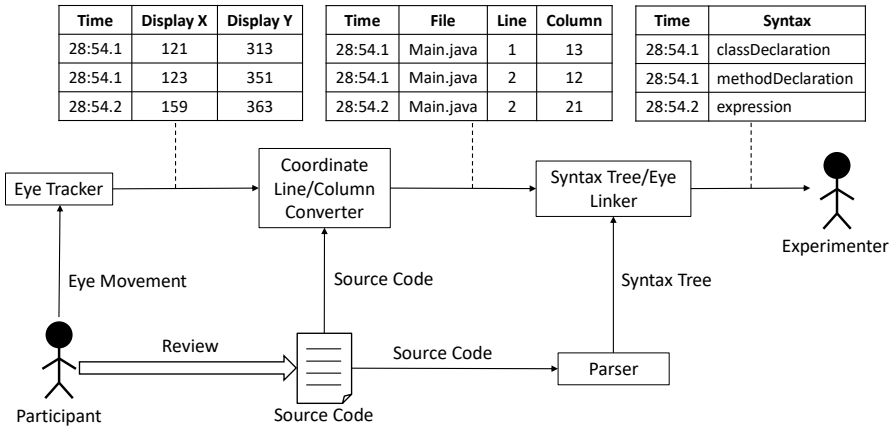



**Fig. 2** Proposed Method [15]

ticipants reading the source code is measured using the eye tracker. The eye tracker outputs the gaze position at each point in time as coordinates on the display (e.g., X:121, Y:313). The Coordinates Line/Column Converter takes the eye movements in units of coordinates and the source code as inputs, then outputs the eye movement as source code name and line/column number (e.g., Main.java, line:1, column:13). The method extracts the word of the source code from the line/column number and maps it to a node on the syntax tree through syntax analysis. Consecutive fixations on the same word are combined into one. The Syntax Tree/Eye Linker receives the syntax tree and line/column eye movement to output the syntax node-represented eye movement. The parser output includes the line/column numbers representing the position of each word in the source code, the number of characters, and the syntax type of the word. The Syntax Tree/Eye Linker combines the eye movement converted into line/column numbers into node units on the syntax tree by mapping the line/column numbers of each node from the parser result.

The proposed method was implemented in Python, using iTrace to implement the "Coordinates Line/Column Converter" to extract line/column numbers from the eye movement coordinate data. ANTLR[2], an open-source parser generator, was used to implement the "Syntax Tree/Eye Linker," with Java[3] as the target language. Using the corresponding parser allowed us to analyze eye movement for source code written in other programming languages. We also implemented the following functions to assist analysis:

- visualization of eye movement on the source code
- calculates fixation time to each syntax tree node and output in formats supported by Graphviz (PNG, PDF, EPS, SVG, etc.)

[2] https://www.antlr.org

[3] https://github.com/antlr/grammars-v4/tree/master/java/java

**Table 1** Output example of eye movement

| ID | Time | Eye Movement |
|---|---|---|
| 1 | 24:54.1 | Main method / For stm / Initialization expression / i |
| 2 | 24:54.1 | Main method / For stm / Conditional expression / i |
| 3 | 24:54.2 | Main method / For stm / Update expression / ++ |
| 4 | 24:54.2 | Main method / For stm / Block / Sum |

```
1  public class Main {
2    static int[] coin = {1, 5, 10, 50,
3                                 100, 500};
4
5    public static void main(String arg[]) {
6      int x = 70;
7
8      int r = method1(x, 5);
9
10     System.out.printf("%d\n", r);
11   }
12
13   static int method1(int x, int i) {
14     if(x == 0)
15       return 1;
16
17     else if(x < 0)
18       return 0;
19
20     else if(i < 0)
21       return 0;
22
23     else {
24       return method1(x-coin[i], i)
25                       + method1(x, i-1);
26     }
27   }
28 }
```

**Fig. 3** Task 13 Source Code

Table 1 presents an example of the eye movement in Figure 1(a) transformed using our method. Each line of the table represents the information of the node corresponding to a word where the eye movement was fixated. The eye movement columns indicate the node representing the fixated word and its parent nodes; ID1 is the fixation on i in the initialization expression of the “for” statement in the main method.

Figures 3 and 4 illustrate examples of the source code used in our experiments and the syntax tree generated from the source code. The syntax tree shows only the node below the node corresponding to the declaration of method1 (methodDeclaration), with the second and subsequent “if” statements (line 17 in Fig. 3) deleted. The leaf nodes in Figure 4 represent words of the source code, and the inner nodes represent syntax elements to which the child nodes belong. For example, the node

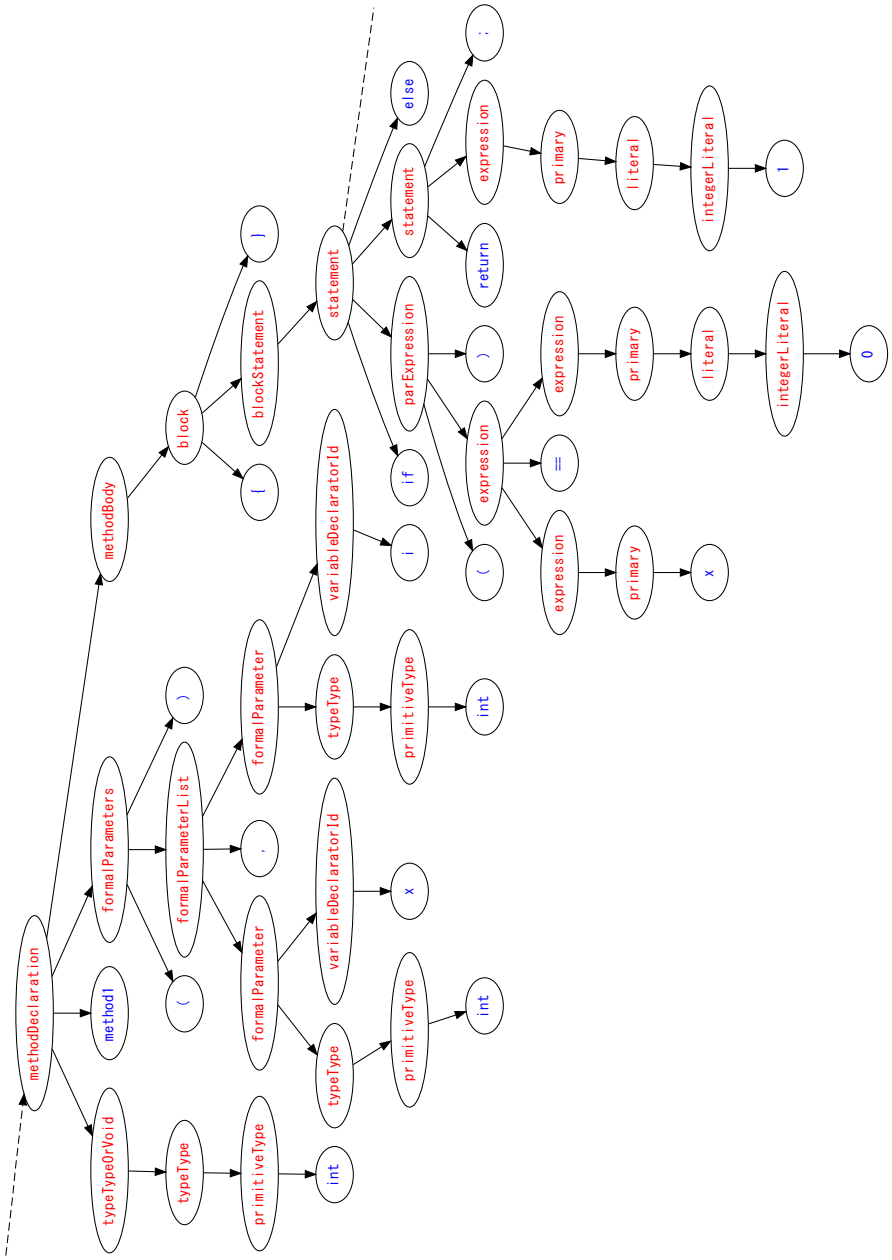



**Fig. 4** Syntax tree from task 13 source code

on the left (formalParameter) represents a temporary parameter consisting of the two words (int, x) in line 13 of Figure 3 and is part of method1 (methodDeclaration)

in line 13. The method converts eye movement into information that combines the word on the fixation, line/column numbers, and all the syntax elements to which the word belongs. Including information about high-level syntax elements allows us to, for example, capture the eye movement for line 15, column 14 in Figure 3 with the following different granularities:

- Fixation at word "1"
- Fixation at return statement
- Fixation at if statement
- Fixation at Method1
- Fixation at Main class

Using this proposed method removes differences in display position and source code format by expressing eye movement as transitions to the nodes of the syntax tree. The output includes information on the block, method, and class to which the fixated word belongs, so it is possible to extract patterns of eye movement at different granularities according to the purpose of analysis. For example, method-wise reading patterns can be extracted by analyzing eye movement for words belonging to MethodDeclaration, MethodBody, and MethodCall. The method also enabled feature analysis and pattern mining from a vector that expresses the multiple syntax nodes for a single word.

## 4 Experiment

Participants were presented with a programming task written in Japanese and Java languages, and their eyes were tracked as they understood the contents of the task. The participants were 14 students from the author's organization, aged between 19 and 21, all of whom had attended courses in basic Java programming.

### *4.1 Experimental environment and tasks*

The experiment was conducted in a quiet room with only one participant and two experimenters. An eye tracker, task presentation PC, and recording PC were used in the experiment. The eye tracker used was the Tobii Eye Tracker 4C.

Each participant was given sixteen tasks, consisting Java source code and corresponding specifications written in Japanese. The participant were asked to verify their understanding to the source code, by asking "What will be the value when line 6 is executed the second time?" verbally. If the answers matched the prepared answers, the participants were assumed to understand the source code correctly. If the answer was incorrect or the participants exceeded the time limit, they were assumed to fail. The participants were not informed their responses were correct or not.

**Table 2** Tasks used in the experiment

| | Task | Specifications |
|---|---|---|
| 1 | Factorial | Calculate factorial |
| 2 | SearchMax | Search for the maximum value |
| 3 | PrimeNum | Determine prime numbers |
| 4 | SearchMedian | Search for the median |
| 5 | Power | Calculate power |
| 6 | Swap | Swap two numbers |
| 7 | Substring | Determine if the specified string is contained |
| 8 | ReverseString | Reverse a string |
| 9 | TowerOfHanoi | Tower of Hanoi |
| 10 | NumOfRoute | Find the number of routes |
| 11 | Permutation | Enumerate all permutations |
| 12 | Combination | Find combinations from the asymptotic formula |
| 13 | PayMoney | Find the combination of coins to pay |
| 14 | StrCombination | Find the combination of strings |
| 15 | CloudSim | Simulate cloud movement |
| 16 | lcm_gcd | Find the least common multiple and greatest common divisor |

The difficulty and time limit of the task were adjusted to measure cases of understanding or not understanding the program to the same extent. Table 2 presents a list of the tasks. Eight easy tasks (1-8 in Table 2) contain easy-to-understand source code, using only the main method, a single iterative statement, and conditional branches. Eight difficult tasks (9-16 in Table 2) contain complex source code expected to be difficult to understand in a short duration, as it uses multiple methods and recursive structures. Using preliminary experiments, the time limit was set to 2 minutes and 30 seconds, which was sufficient for understanding easy tasks and insufficient for difficult tasks. The order of the tasks was counterbalanced to consider the order effect.

## *4.2 Analysis*

We obtain the fixation ratio for each syntax type and compare the participants who answered the task correctly and incorrectly. The parser generated by ANTLR outputs 105 syntax types (semantic nodes) and corresponding words (word nodes) for the Java source code. The tasks used in the experiment contained 41 semantic nodes. The analysis in this study redefines the 21 semantic nodes listed in Table 3 to focus on the control structure and eye movement by sentence. If a word node was connected to a semantic mode of a syntax type that the redefinition excluded, the word node was connected to the parent node, and the fixation time was added to the parent node. Figure 5 shows an example of a syntax tree with the fixation time represented by the redefined syntax types. The first line of each node represents the name of the semantic node, the second line the words corresponding to the node, and the third

**Table 3** Syntax types for analysis

| Syntax type | Description |
|---|---|
| blockStatement | Declaring and initializing variables |
| classBodyDeclaration | Declaring fields and methods |
| forCond | Conditional expressions for “for” |
| forInit | Initialization expression for “for” |
| forProcess | Statement belonging to for block |
| forStm | Word “for” |
| forUpdate | Variable expression for “for” |
| formalParameterList | Temporary parameters for method |
| ifCond | Conditional expression for “if” |
| ifProcess | Statement belonging to an “if” block |
| ifStm | Word “if, else” |
| indent | Indent (blank, tab) |
| newLine | Empty line |
| printStm | print statement, println statement |
| space | White space separating consecutive words |
| statement | Assignment, method call, return statement |
| typeDeclaration | Class declaration |
| whileCond | Condition expression for “while” |
| whileProcess | Statement belonging to a “while” block |
| whileStm | Word “while” |
| whileUpdate | “++” of a “while” block or statement including “-” |

line the total fixation time (seconds and percentage value) for the node. The fixation time for the parent node does not include the time for the child nodes.

## 5 Results and Discussion

224 program comprehension tasks (14 participants $\times$ 16 tasks) were recorded, then we excluded 24 cases of measurement error and analyzed the remaining 200. Figure 6 shows the percentage of fixation time for each syntax type for participants who answered the program comprehension task correctly and incorrectly. The figure illustrates the average fixation time for each syntax type among the participants' fixation times at the words they focused on in each task. The time looking away from the screen or blinking is not included. The figure describes the top 14 with the highest fixation rate, and other syntax types are considered “others.” For both the correct and incorrect, the most attention was given to space (space between words) and blockStatement (declaration and initialization of variables): 16.9% and 10.5–10.9%, respectively, but there was no significant difference between the two groups. The larger fixation ratio for spaces may be attributed to the fact that spaces account for a large percentage of the source code. Eye movements of correct responses a lot of percentages to forInit (initialization expression for “for”), forProcess (statement in for block), ifCond (conditional expression for if), ifProcess (state-

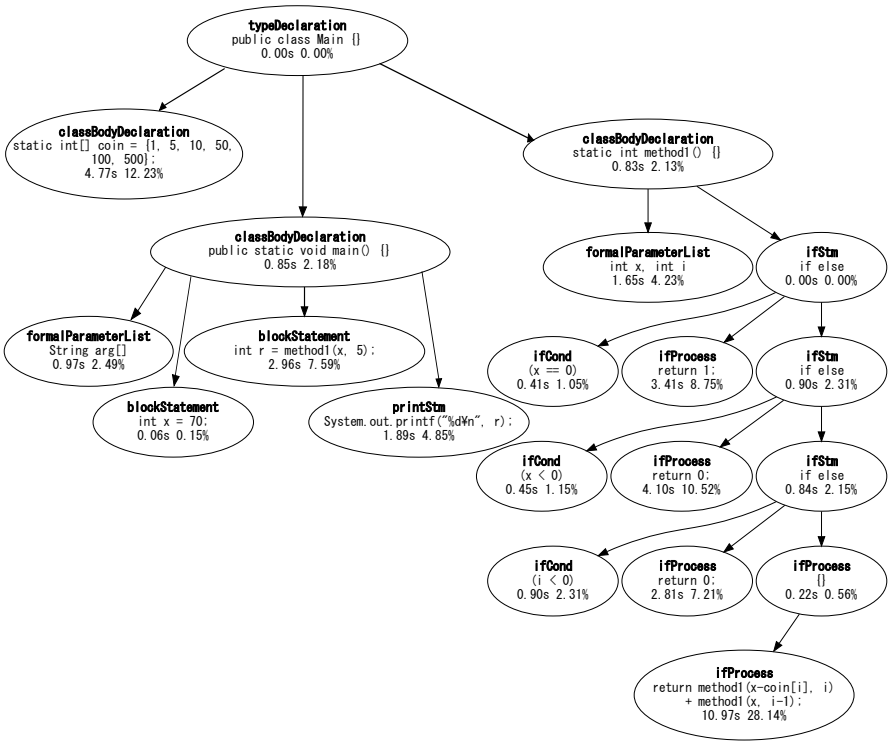



**Fig. 5** Fixation time for each syntax node

ment in if block), and whileProcess (statement in while block). On the other hand, eye movements of incorrect responses a lot of percentages to classBodyDeclaration (declaration of field and method), formalParameterList (temporary parameters for method), and printStm (print statement). Significant differences in the average values for correct and incorrect responses were observed for ifCond (7.8% correct, 5.6% incorrect), formalParameterList (0.8% correct, 3.4% incorrect), and printStm (3.8% correct, 6.3% incorrect) ($p < 0.05$, t-test).

Participants who answered correctly focused more on syntax elements that were all control statements that control program operation and their processing contents, which may have been important elements in the task of understanding the processing contents from the presented source code. Meanwhile, participants who answered incorrectly paid more attention to syntax elements that were class declarations, temporary parameters of methods, and print statements displaying the results of processing, suggesting that they paid more attention to the relationship between input and output for processing.

Next, we compare the distribution of fixation rate between those who gave correct and incorrect answers in one task. The importance of even the same syntax elements for program understanding varies depending on the task, so the proportion

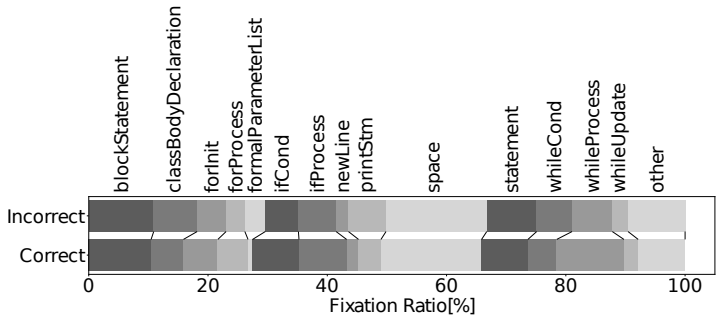



**Fig. 6** Fixation ratio for each syntax type

of attention varies by task. The source code for Task 13 in Figure 3 is a program to find the number of possible coin combinations that can make up the money specified in line 6. For Task 13, 4 participants understood and 10 participants failed. Figure 7 shows the percentage of fixation time for each syntax type for correct and incorrect for Task 13.

For both, there was a high percentage of fixation time for 1)ifProcess, which corresponds to the return statement, 2)classBodyDeclaration, which includes the declaration of a coin array, and 3)spaces. Participants who answered correctly paid the highest percentage of attention to the ifProcess (22.0% correct, 14.7% incorrect) and indent (5.2% correct, 0.9% incorrect), while incorrect responders focused on spaces (21.7% correct, 26.2% incorrect), ifCond (4.8% correct, 9.5% incorrect), and formalParameterList (2.9% correct, 5.8% incorrect). There was a significant difference ($p < 0.05$, t-test) between the average for correct and incorrect responses for ifCond and spaces.

The program for this task calculates the number of coin combinations that will result in the amount of money stored in variable $x$ by recursively calling method1. The first argument for method1 represents the specified amount of money, and the second argument is the starting position of the search in the coin array (Fig. 3, line 2). Line 24 is a recursive call for the specified amount of money minus the number of coins to be searched (argument x-coin[i], i) and a call that moves the search position for coins by one (argument x, i1). IfProcess at lines 15, 18, 21, 24, and 25 had the largest fixation ratio by participants who answered the task correctly. These lines are considered important for determining whether a coin combination can be counted based on the arguments of each recursive call. Conversely, ifCond, which had a significantly larger fixation ratio for those who answered incorrectly, corresponds to the conditionals in lines 14, 17, and 20.

Table 4 compares the fixation ratio of correct and incorrect answers for each element. Participants who gave the correct response had a higher percentage for ifProcess compared to its counterpart ifCond, while participants with incorrect responses showed the same or higher percentage for ifCond. Future research is required to clarify the relationship between the contents of individual tasks and percentage of attention to syntax elements. Further, there was a significant difference between

**Table 4** Fixation ratio for each element of method1

| line | Item | Code | Correct | Incorrect |
|---|---|---|---|---|
| 14 | ifCond | $x == 0$ | 3.2% | 5.2% |
| 15 | ifProcess | return 1 | 9.9% | 5.7% |
| 17 | ifCond | $x < 0$ | 1.4% | 4.3% |
| 18 | ifProcess | return 0 | 3.8% | 2.6% |
| 20 | ifCond | $i < 0$ | 2.1% | 2.3% |
| 21 | ifProcess | return 0 | 5.1% | 1.4% |
| 24 | ifProcess | return method1(...) ... | 14.2% | 11.5% |

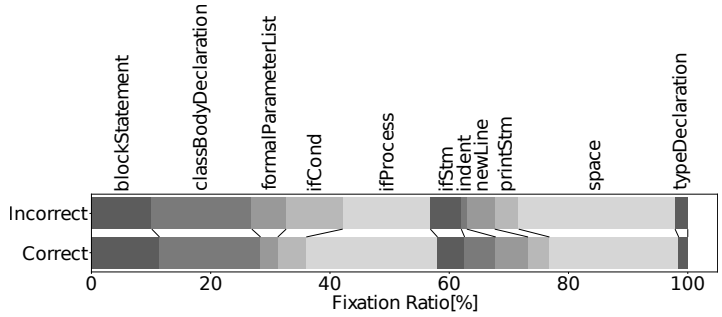



**Fig. 7** Fixation ratio for each syntax type in task 13

the fixation ratio to space. The result suggests that correctly answered participants focused their attention on the specific parts of the source code, while incorrectly answered participants read the entire source code in an exploratory manner; the fixation was separated because the space is the most frequent character in the source code.

## 6 Conclusion

In this study, we analyzed the eye movements while performing a program comprehension task. We used a method that converts the eye movements recorded as the display's coordinates to the eye movements to syntax tree nodes. The percentage of fixation time for each syntax element was compared based on data from 200 eye movements recorded from the experiment. The experiment revealed that participants who responded correctly to the program comprehension tasks spent significantly more time at the conditional expressions "if" and significantly less time on the temporary parameters of the method and the print statement. Complex tasks, such as recursion, also had the participants look more at statements within "if" statements than the conditional expression. The results suggest that there is a relationship between program comprehension and important syntactic elements for understanding,

and we need further research on whether causal relationships and instructions focusing on specific syntactic elements can promote comprehension in programming.

In this study, the purpose was to understand the code. The code reading method depends on the purpose. For example, when participants search for syntax errors, they will scan all words because of searching for inaccurate reserved words. In future work, We compare eye movement patterns during different reading purposes such as fault detection in future work.

The analysis performed in this study summarized the fixation time for each syntax element into a higher syntax element based on the definitions in Section 4. Meanwhile, it is possible to associate each word constituting the source code with multiple meanings on the syntax tree. For example, looking at x in line 14 in Fig. 3 can also be interpreted as looking at the "if" statement that the entire line 14 comprises. Similarly, it can also be seen as looking at the "if" block or method1 in the 14th line. The duration of fixation and number of fixations for a word are therefore counted for all the syntax elements that correspond to that word, which makes it difficult to calculate the percentage. Calculations in this paper were based on fixation for the lowest-level syntax element to which each word belonged to avoid duplicates. Developing and analyzing a method to calculate the percentage of fixations for multiple syntax elements is an important task for future research.

Although there was no significant difference, there was a difference in the fixation ratio at the indent between the correct (5.2%) and incorrect (0.9%) in Task 13 (shown in Section 5). The effects of indentation and its length on the readability of source code have been investigated for many years[10, 7] and are still under research[11]. The source code of Task 13 (Figure 3) consists of two methods (main and method1). Method1 is a program with a complex structure compared to other tasks, consisting of "if-elseif-else" statements and recursive calls. The results of this experiment suggest that participants who correctly understood the source code of Task 13 had a fixation on indents when focusing on the overall program structure. Future research will perform a detailed analysis of the fixation ratio based on the correct/incorrect answers for each task and on the characteristics of the source code.

**Acknowledgements** This research was funded by JSPS Research Grant JP21K11842.

## References

1. Abbad-Andaloussi, A., Sorg, T., Weber, B.: Estimating developers' cognitive load at a fine-grained level using eye-tracking measures. In: 2022 IEEE/ACM 30th International Conference on Program Comprehension (ICPC), pp. 111–121 (2022). DOI 10.1145/3524610.3527890
2. Bertram, I., Hong, J., Huang, Y., Weimer, W., Sharafi, Z.: Trustworthiness perceptions in code review: An eye-tracking study. In: Proceedings of the 14th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), ESEM '20. Association for Computing Machinery, New York, NY, USA (2020). DOI 10.1145/3382494.3422164. URL https://doi.org/10.1145/3382494.3422164
3. Busjahn, T., Bednarik, R., Begel, A., Crosby, M., Paterson, J.H., Schulte, C., Sharif, B., Tamm, S.: Eye movements in code reading: Relaxing the linear order. In: Proceedings of the 2015

IEEE 23rd International Conference on Program Comprehension, ICPC '15, p. 255–265. IEEE Press (2015)
4. Crosby, M., Stelovsky, J.: How do we read algorithms? a case study. Computer **23**(1), 25–35 (1990). DOI 10.1109/2.48797
5. Guarnera, D.T., Bryant, C.A., Mishra, A., Maletic, J.I., Sharif, B.: Itrace: Eye tracking infrastructure for development environments. In: Proceedings of the 2018 ACM Symposium on Eye Tracking Research & Applications, ETRA '18. Association for Computing Machinery, New York, NY, USA (2018). DOI 10.1145/3204493.3208343. URL https://doi.org/10.1145/3204493.3208343
6. Hauser, F., Schreistter, S., Reuter, R., Mottok, J.H., Gruber, H., Holmqvist, K., Schorr, N.: Code reviews in c++: Preliminary results from an eye tracking study. In: ACM Symposium on Eye Tracking Research and Applications, ETRA '20 Short Papers. Association for Computing Machinery, New York, NY, USA (2020). DOI 10.1145/3379156.3391980. URL https://doi.org/10.1145/3379156.3391980
7. Kesler, T.E., Uram, R.B., Magareh-Abed, F., Fritzsche, A., Amport, C., Dunsmore, H.: The effect of indentation on program comprehension. International Journal of Man-Machine Studies **21**(5), 415–428 (1984). DOI https://doi.org/10.1016/S0020-7373(84)80068-1. URL https://www.sciencedirect.com/science/article/pii/S0020737384800681
8. Kevic, K., Walters, B., Shaffer, T., Sharif, B., Shepherd, D., Fritz, T.: Eye gaze and interaction contexts for change tasks observations and potential. J. Syst. Softw. **128**(C), 252–266 (2017). DOI 10.1016/j.jss.2016.03.030. URL https://doi.org/10.1016/j.jss.2016.03.030
9. Kevic, K., Walters, B.M., Shaffer, T.R., Sharif, B., Shepherd, D.C., Fritz, T.: Tracing software developers' eyes and interactions for change tasks. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, p. 202–213. Association for Computing Machinery, New York, NY, USA (2015). DOI 10.1145/2786805.2786864. URL https://doi.org/10.1145/2786805.2786864
10. Miara, R.J., Musselman, J.A., Navarro, J.A., Shneiderman, B.: Program indentation and comprehensibility. Commun. ACM **26**(11), 861–867 (1983). DOI 10.1145/182.358437. URL https://doi.org/10.1145/182.358437
11. Oliveira, D., Santos, R., Madeiral, F., Masuhara, H., Castor, F.: A systematic literature review on the impact of formatting elements on code legibility. Journal of Systems and Software **203**, 111,728 (2023). DOI https://doi.org/10.1016/j.jss.2023.111728. URL https://www.sciencedirect.com/science/article/pii/S0164121223001231
12. Peitek, N., Siegmund, J., Apel, S.: What drives the reading order of programmers? an eye tracking study. In: Proceedings of the 28th International Conference on Program Comprehension, ICPC '20, p. 342–353. Association for Computing Machinery, New York, NY, USA (2020). DOI 10.1145/3387904.3389279. URL https://doi.org/10.1145/3387904.3389279
13. Rodeghero, P., McMillan, C., McBurney, P.W., Bosch, N., D'Mello, S.: Improving automated source code summarization via an eye-tracking study of programmers. In: Proceedings of the 36th International Conference on Software Engineering, ICSE 2014, p. 390–401. Association for Computing Machinery, New York, NY, USA (2014). DOI 10.1145/2568225.2568247. URL https://doi.org/10.1145/2568225.2568247
14. Sharif, B., Mansoor, N.: Humans in empirical software engineering studies: An experience report. In: 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 1286–1292 (2022). DOI 10.1109/SANER53432.2022.00154
15. Yoshioka, H., Uwano, H.: Automatic mapping of syntax trees and eye movement for semantic-based program comprehension pattern extraction. International Symposium on Advances in Technology Education (ISATE) 2022 (2022)