2022

Identifying Code Reading Strategies in Debugging using STA with a Tolerance Algorithm

Christine Lourrine S. Tablatin

Ma. Mercedes T. Rodrigo

APSIPA Transactions on Signal and Information Processing, 2022, 11, e6

Original Paper

Identifying Code Reading Strategies in Debugging using STA with a Tolerance Algorithm

Christine Lourrine S. Tablatin[1,2] and Maria Mercedes T. Rodrigo[1*]

[1]*Ateneo de Manila University, Quezon City, Philippines*
[2]*Pangasinan State University, Urdaneta City, Pangasinan, Philippines*

ABSTRACT

The purpose of this study was to identify the common code reading strategies of the high and low performing students engaged in a debugging task. Using Scanpath Trend Analysis (STA) with a tolerance on eye tracking data, common scanpaths of high and low performing students were generated. The common scanpaths revealed differences in the code reading patterns and code reading strategies of high and low performing students. High performing students follow a bottom-up code reading strategy when debugging complex programs with logical and semantic errors. A top-down code reading strategy is employed when debugging programs with simple control structures, few lines of code, and simple error types. These results imply that high performing students use flexible debugging strategies based on the program structure. The generated common scanpaths of the low performing students, on the other hand, showed erratic code reading patterns, implying that no

*Corresponding author: Christine Lourrine S. Tablatin, tablatinchristine@gmail.com.
We thank the Ateneo de Manila, Ateneo de Davao, University of Southeastern Philippines, University of San Carlos, Private Education Assistance Committee of the Fund for Assistance to Private Education for the grant entitled "Analysis of Novice Programmer Tracing and Debugging Skills using Eye Tracking Data" and Ateneo de Manila University's Loyola Schools Scholarly Work Faculty grant entitled "Building Higher Education's Capacity to Conduct Eye-tracking Research using the Analysis of Novice Programmer Tracing and Debugging Skills as a Proof of Concept". We also thank Bobby Roaring for helping us in choosing the most appropriate statistical analysis to use in this study.

Received 30 July 2021; Revised 30 November 2021
ISSN 2048-7703; DOI 10.1561/116.00000040

obvious code reading strategy was applied. The identified code reading strategies of the high performing students could be explicitly taught to low performing students to help improve their debugging performance.

1 Introduction

Programming skill necessitates not just the production of code but also the comprehension of existing source code. Thus, the skill of program comprehension is as important as program writing when learning how to program [16, 14]. Source code comprehension is a vital activity in software development, and code reading strategies affect the programmer's success rate in performing comprehension tasks such as debugging [31].

One of the most intriguing approaches to analyze how programmers perform code comprehension tasks is the use of eye-trackers [3]. Eye-trackers are used to capture visual attention by collecting eye movement data of participants while performing a task and have recently become a common tool used to perform empirical studies in programming [27, 4]. The earliest study that leveraged the use of eye-tracking data to understand the comprehension process of students while reading algorithms was conducted in 1990 [10]. Since then, researchers have become active in exploring the cognitive processes of programmers using eye-tracking data while performing code comprehension tasks and debugging [27, 23]. While efforts have been made to understand the comprehension processes using eye-tracking data, we still have limited knowledge about the code reading patterns and strategies employed in debugging. Most of the approaches used in analyzing eye-tracking data to determine the code reading patterns and strategies were based on visual effort metrics like fixation count and fixation duration on specific Areas of Interest (AOI) [24, 6, 18, 28, 26, 23, 9] while a limited number of studies [32, 7, 21, 19] focused on analyzing the sequential nature of the scanpath [20].

Analyzing the scanpath as a whole entity to determine code reading patterns instead of independently measuring eye movement attributes has become essential to draw explicit conclusions on the nature and interpretation of the cognitive processes [1]. However, finding common code reading patterns to identify visual strategies is challenging since their individual scanpaths tend to be different from each other [4, 13] and are highly individualistic [19]. One study [30] used Scanpath Trend Analysis (STA) with a tolerance to address the challenge of bringing multiple scanpaths together taking into consideration the individual differences of scanpaths to generate common scanpaths that would reveal common code reading patterns of high and low performing students engaged in a debugging task. According to the findings, the generated common

scanpaths are more comparable to the individual scanpaths by adjusting the appropriate tolerance level parameter in the stage of identifying trending scanpaths. The study also confirmed previous findings that high-performing students logically read code while low-performing students make random selection of code statements to find bugs [21].

Understanding how students read and comprehend source code does not only provide information on how they think while performing the task but also provides insights on how we can improve the overall learning process by improving learning materials and knowing when learning intervention is needed. Further, exploring the strategies used by experts or high performing students while performing comprehension tasks such as debugging will allow us to uncover effective strategies that could be explicitly taught to improve code reading and code comprehension skills of students [3]. Thus, the main goal of this study was to identify the code reading strategies of high and low performing students while performing code comprehension to detect bugs. Specifically, this study sought answers to the following questions:

1. How can we identify the common scanpath of high and low performing students to reveal their common code reading patterns?

2. How do the code reading patterns vary between high and low performing students?

3. What code reading strategies can be inferred from the code reading patterns of high and low performing students?

Our previous study focused on identifying the common code reading patterns of only one program using STA with a tolerance. This paper attempts to validate the findings from our prior work by analyzing a larger dataset to infer code reading patterns and strategies of high and low performing students while debugging codes.

2 Methodology

This paper is an analysis of a larger eye tracking study on programmer tracing and debugging skills as well as the development of higher education's capacity to conduct eye tracking research. The experimental setup and procedure discussed in [30], which investigates the use of STA algorithm with a tolerance to determine the common code reading patterns of high and low performing students engaged in a debugging task, is also used in this study. This study, on the other hand, builds on the prior one to validate the initial findings by using a larger dataset.

2.1 Participants

Students aged 18–23 years old who were in their 2nd year to 4th year level in college and had taken the college-level fundamental programming course were recruited to participate in this study. The 64 participants composed of 47 males and 17 females were recruited from four private universities in the Philippines: 16 from Ateneo de Davao University (ADDU), 17 from University of the Cordilleras (UC), 16 from University of San Carlos (USC), and 15 from University of Southeastern Philippines (USeP).

2.2 Dataset

The eye tracking data consisting of 238,733 data points were collected and saved in an individual CSV file composed of information regarding the fixation timestamp, the location of fixations, fixation durations, blinking counts, pupil dilations, and separate values for the left and right eye movements. In the context of this study, only the timestamps, the values of x and y coordinates, and the fixation durations of each participant on 12 programs were extracted from the segmented CSV file to construct the individual scanpaths.

To determine the difference in the code reading patterns and strategies, the study used two participant groupings: high performing and low performing. The scores of the participants in the debugging tasks were used to assign them to a particular group. High performing group consisted of students who scored above and equal to the mean score, while the low performing group consisted of students who scored lower than the mean score. Moreover, two datasets were used in the analysis of the scanpaths using STA algorithm with a tolerance. The first dataset consists of the timestamp, fixation points (x and y coordinates), and fixation duration of each program code by each participant. For the second dataset, instead of using the actual fixation durations on each AOI, the proportional fixation durations for each AOI were used. The proportional fixation duration for each AOI was computed as a ratio of fixation duration on an AOI to the overall fixation duration on all AOIs.

2.3 Experimental Procedure and Setup

We used a Gazepoint GP3 eye tracker in the eye-tracking experiment with a sampling rate of 60 Hz and 0.5–1 degree of accuracy [15]. The source code was presented in a full-screen window with screen resolution set to 1366×768. Then participants were asked to read 12 program codes with known errors, and the errors were marked using the mouse. There is no need for the participants to correct them.

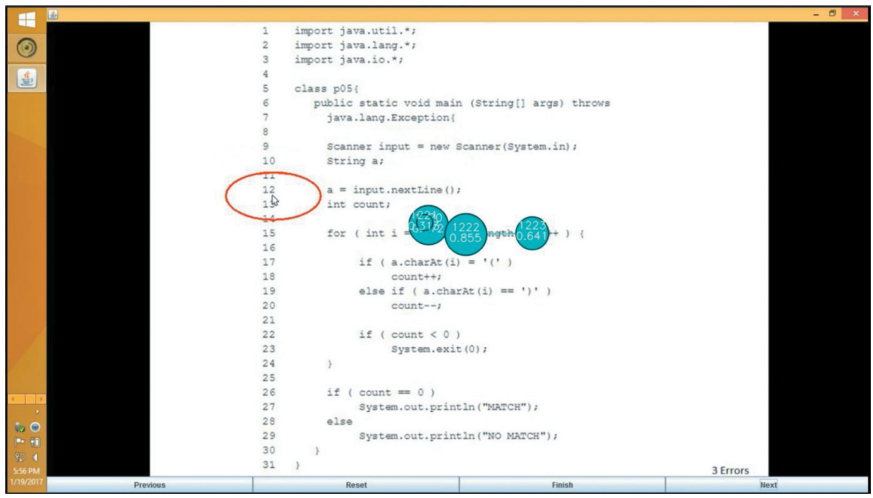


Figure 1: Screenshot of the slide sorter program.

2.3.1 Hardware/Software Setup

The experimental setup consisted of a laptop, a monitor, a mouse, a keyboard, and a GazePoint table-mounted eye-tracking device. The hardware is set up by extending the laptop's display to the monitor connected to it. The eye tracker was placed in front of the 17-inch monitor to allow the participant to view the codes while the eye tracker records the eye movements. The optimal setup between the user and the eye tracker is 100 cm horizontal distance and 40 cm vertical distance. The eye tracker was then put to the test to see if it was already operational. The participant was asked to sit comfortably, and then the eye-tracker was adjusted to detect the eyes of the participant. Calibration of the eye-tracker on the participant was done by asking the participant to follow the white circle/red dot to check if the calibration is proper. Calibration was done since the accuracy and precision of the eye tracker data depend on a successful calibration. When the calibration was successful, the participant started the experiment.

After setting up the hardware, the GazePoint Analysis and Control software was opened from the laptop and brought to the screen for tester viewing. A new project is then created for the experiment before the stimulus is prepared in the background. A slide sorter program (see Figure 1) was created with buttons Previous, Next, Reset, and Finish buttons to display the program description followed by the program code with injected bugs. To navigate across the slides, click the Previous and Next buttons. The Reset button is used to clear the marked error locations on a particular slide, while the Finish button saves the marks and ends the debugging session.

Table 1: Code description, line numbers of injected defects and metric value of codes.

Code	Description	Lines of code	Line(s) with error	Cyclomatic complexity	Nested block depth
P01	What's the next number?	17	10	2	1
P02	Reverse of strings	15	15	2	2
P03	Arrow	16	14	1	1
P04	Q prime	26	15, 22, 29	6	4
P05	Parenthesis matching	24	13, 17, 23	6	2
P06	Palindrome	19	15, 18, 22	3	3
P07	Rock, paper, and scissor	34	20, 26, 30	9.5	1.5
P08	The diamond pattern	27	11, 13, 18	7	2
P09	Paralleloword	29	15, 24, 30	7	3
P10	Consecutive words	16	11, 14, 16	3	2
P11	Earthquake's class	25	11, 18, 24	7	1
P12	Basic calculator	28	11, 22, 27	5	1

2.3.2 Task Stimuli and Injected Defects

The analysis of the eye tracking data focused on 12 short programs written in Java programming language that served as stimuli in the eye tracking experiment. These programs are typically written by novice programmers. Nine of the 12 short Java programs had three defects each, while three programs have one each. The bugs were intentionally added to the program codes, and the participant's task was to find these bugs in the static source codes. In this paper, we refer to this task as debugging. Few of these injected bugs take a minimal number of scans to detect. But quite a number takes a considerable amount of time and may involve the participant's analytical skills and prior knowledge in programming.

Each program was assigned either one or three bugs, with different numbers of lines of codes, cyclomatic complexity, and nested block depth as shown in Table 1. Errors to locate range from easiest to spot (syntax errors) to the hardest (semantic and logical errors). Table 2 presents the characteristics of the errors that were injected into each program code. We can see from the table that the injected errors were of various types and located in different lines and sections of the programs.

2.4 Data Pre-Processing

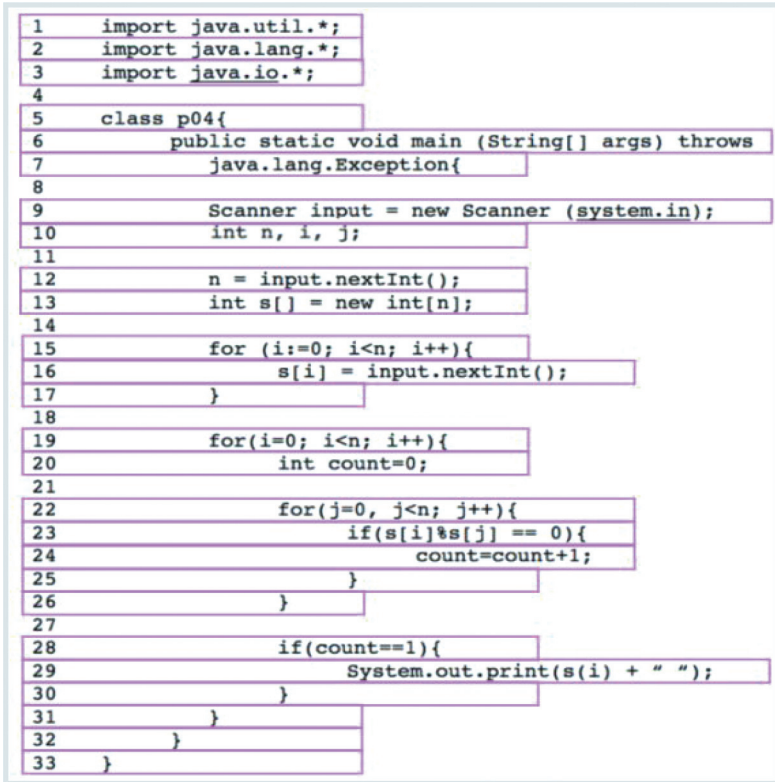
All the eye movements of the participants were stored in a CSV file format. From the CSV file, the timestamp, location of fixations, and fixation durations were extracted. To map the fixations in the stimuli, the AOIs were drawn using

Table 2: Description and type of the injected errors.

Code	Error description	Error type	Error line	Program section
P01	Missing semicolon	Syntax	10	Variable declaration
P02	Additional semicolon	Syntax	15	For loop structure
P03	Use of print instead of println	Logical	14	Program body
P04	Additional colon	Syntax	15	For loop structure
	Comma instead of semicolon	Syntax	22	For loop structure
	Parentheses instead of brackets	Syntax	29	Conditional structure body
P05	Variable not initialized	Semantic	13	Variable declaration and initialization
	Incorrect comparison operator	Semantic	17	Conditional structure
	Incorrect use of command or statement	Logical	23	Conditional structure body
P06	Comma instead of semicolon	Syntax	15	For loop structure
	Incorrect message	Logical	18	Conditional structure body inside a for loop structure
	Incorrect message	Logical	22	Program body
P07	Missing semicolon	Syntax	20	Conditional structure body
	Apostrophe instead of double quotes	Syntax	26	Else conditional structure body
	No parameters in a method	Semantic	30	Method declaration

Table 2: Continued.

Code	Error description	Error type	Error line	Program section
P08	Use of inappropriate data type	Semantic	11	Variable declaration
	Use of inappropriate data type	Semantic	13	Program body
	Additional semicolon	Syntax	18	For loop structure
P09	Variable not decremented	Logical	15	While loop condition
	Incorrect variable was referenced	Logical	24	Second level for loop on first for loop inside while structure
	Incorrect value was printed	Logical	30	Second level for loop on second for loop inside while structure
P10	Use of inappropriate data type	Semantic	11	Variable declaration
	Incorrect argument in print statement	Logical	14	Program body
	No parentheses to indicate that it is a function call	Semantic	16	For loop structure
P11	Use of inappropriate data type	Semantic	11	Variable declaration
	Incorrect comparison operator	Semantic	18	Else if conditional structure
	Use of elseif instead of else if	Syntax	24	Else if conditional structure
P12	Use of inappropriate data type	Semantic	11	Variable declaration
	Incorrect variable was referenced	Logical	22	Case body
	Missing statement	Logical	27	Case body



```
1   import java.util.*;
2   import java.lang.*;
3   import java.io.*;
4
5   class p04{
6       public static void main (String[] args) throws
7           java.lang.Exception{
8
9           Scanner input = new Scanner (system.in);
10          int n, i, j;
11
12          n = input.nextInt();
13          int s[] = new int[n];
14
15          for (i:=0; i<n; i++){
16              s[i] = input.nextInt();
17          }
18
19          for(i=0; i<n; i++){
20              int count=0;
21
22              for(j=0, j<n; j++){
23                  if(s[i]%s[j] == 0){
24                      count=count+1;
25                  }
26              }
27
28              if(count==1){
29                  System.out.print(s(i) + " ");
30              }
31          }
32      }
33  }
```

Figure 2: Areas of interests of QPrime program.

the OGAMA Areas of Interest module [33] to define the Areas of Interests (AOIs) of the 12 programs to obtain the AOI coordinates. Each line of codes of the stimuli were marked as the AOIs, excluding blank lines. Figure 2 shows the AOIs of one of the program codes used in this study. To account for the error on fixation identification, a 20 px boundary was added to increase the size of each AOI rectangle of the stimulus. This process could create overlapping AOIs. The fixations were mapped to the AOI with the nearest center point out of the AOIs that contain the fixation point. These procedures were done for the eye-tracking data of the 64 students who participated in the eye tracking experiment.

2.5 Data Analysis

2.5.1 Common Scanpath Analysis using STA with a Tolerance

Scanpath Trend Analysis (STA) with a tolerance algorithm [13] was employed to generate the common scanpaths. This algorithm gives us the ability to

find an appropriate tolerance level for achieving the highest similarity of the common scanpath to the individual scanpaths by adjusting the tolerance level parameter in the stage of identifying trending AOI instances. Trending AOI instances are AOIs shared by all individual scanpaths or a subset of the individual scanpaths based on the tolerance level specified. The algorithm consists of three main stages: Preliminary Stage, First Pass Stage, and Second Pass Stage.

The first stage is the preparation of the individual scanpaths. The fixation points (x and y coordinates) in the individual scanpaths were mapped to the identified AOIs of the program code. To map the fixations to the code elements or AOIs of the program code, a 20 px boundary was added to increase the size of each AOI rectangle. The fixations were mapped to the AOI with the nearest center point out of those that contain the fixation point. This was done to account for the eye tracker's accuracy during the recording of the eye gazes of the participants. The individual scanpaths in this stage were represented as a series of AOIs with fixation durations for dataset one and proportional fixation for dataset two. For instance, if the student fixates line 6 for 250 ms, then line 9 for 500 ms, and then line 10 for 100 ms, his/her scanpath is represented as 6[250 ms] 9[500 ms] 10[100 ms]. The line numbers were then converted to letters of the alphabet F [250 ms] H[500 ms] I[100 ms] to allow comparison of scanpaths represented as strings using the Levenshtein distance algorithm.

The second stage identifies the trending AOIs in the scanpaths. The trending AOIs were identified using their occurrence frequencies (i.e., the number of fixations on an AOI) and their fixation durations. The same AOI can be fixated consecutively (F H H I) and/or non-consecutively (F H I H) in a particular scanpath. Each fixation to a particular AOI is referred to as an AOI instance. However, consecutive fixations to the same AOI are considered as only one instance of that AOI. For example, an individual scanpath J[200 ms] E[500 ms] J[250 ms] J[100 ms] M[230 ms] L[375 ms] M[190 ms] X[200 ms] M[100 ms] includes two instances of J, three instances of M while E and X have only one instance each. These AOI instances were differentiated with different numbers based on the total fixation duration or proportional fixation duration on each instance. The highest fixation duration or proportional fixation duration in a particular AOI instance receives the first number. The resulting scanpath of the example is represented as J2[200 ms] E1[500 ms] J1[250 ms] J1[100 ms] M1[230 ms] L1[375 ms] M2[190 ms] X1[200 ms] M3[100 ms].

After differentiating the AOI instances, the trending AOI instances were identified. The STA algorithm with a tolerance, added a tolerance level parameter to the original STA Algorithm [13] to lessen the effect of the variances between individual scanpaths in the resulting common scanpath. By default, the tolerance level parameter is equal to one which means that the trending AOI instances are identified based on the shared instances of all

individual scanpaths. If we set the parameter to 0.95, this means that the trending AOI instances are identified based on the instances which are shared by 95% of individual scanpaths. We can adjust the tolerance level parameter based on the goals of the study. This study used three different tolerance level parameters to find the appropriate parameter for achieving the highest similarity of the trending scanpath to individual scanpaths.

An AOI instance becomes a trending instance if it satisfies the following conditions:

1. The total number of fixations on the AOI instance should be greater than or equal to the minimum total number of fixations on the instances which are shared by a subset (adjusted with the tolerance level parameter) of the individual scanpaths; and

2. The total fixation durations or proportional fixation duration on the AOI instance should be greater than or equal to the minimum total fixation duration on the instances shared by a subset (adjusted with the tolerance level parameter) of the individual scanpaths.

The AOI instances that did not satisfy the conditions were removed from the individual scanpaths. The final stage in the algorithm is the identification of the trending scanpath using the trending AOIs based on their positions in the individual scanpaths. The individual scanpaths were collapsed by combining the same AOI instances (J2[200 ms] E1[500 ms] J1[250 ms] J1[100 ms] M1[230 ms] X1[200 ms] → J2[200 ms] E1[500 ms] J1[250 ms] M1[230 ms] X1[200 ms]), and then computation of the priority value of each AOI instance in the individual scanpaths was performed using the following equation (1):

$$\psi = 1 - P\frac{max - min}{L - 1} \tag{1}$$

where:

$$\begin{aligned}(\psi) &= \text{Priority value}\\ P &= \text{The instance position in the scanpath, starting from 0}\\ L &= \text{The length of the user scanpath}\\ max &= \text{Maximum priority value (default value: 1)}\\ min &= \text{Minimum priority value (default value: 0.1)}\end{aligned}$$

When all the priority values were calculated in each of the scanpaths, the total priority value (Ψ) for each AOI instance was calculated with Equation (2) where n is the number of individual scanpaths. The algorithm then positions the instances into the common scanpath based on their total priority values from highest priority to lowest. If multiple instances have the same

total priority value, their total duration, and their total number of fixations on the instances are compared to determine their position in the common scanpath. Once the trending visual element instances are positioned in the common scanpath, their numbers are eliminated (e.g. E1 → E) and then the consecutive repetitions are removed (e.g., JEJJMX → JEJMX). Thus, the common scanpath is represented in terms of the AOIs.

$$\Psi = \sum_{i=1}^{n} \psi i. \tag{2}$$

Since the STA algorithm constructs a common scanpath based on individual scanpaths, the common scanpath should be similar to the individual scanpaths. Therefore, the similarities between the common scanpath and the individual scanpaths are strongly related to the similarities between individual scanpaths. We expect that the median similarity of the common scanpath to the individual scanpaths is equal or greater than the median similarity of the individual scanpaths. String-edit algorithm was used to calculate the distance between the common scanpath and the individual scanpaths of each group of students to determine how similar the common scanpath to the individual scanpaths. The String-edit distance is then used to calculate the similarity between the two scanpaths by using the following equation:

$$S = 100 \times \left(1 - \left(\frac{D}{L}\right)\right), \tag{3}$$

where:

$$\begin{aligned} S &= \text{Similarity} \\ D &= \text{String-edit distance} \\ L &= \text{The length of the longer scanpath} \end{aligned}$$

Statistical tests were used to identify which tolerance level parameters would be selected to generate a common scanpath for each group of students to reveal their code reading patterns. The effect of the tolerance level parameters on the similarity scores for each group of students and school was determined using repeated-measures ANOVA with Greenhouse–Geisser statistics. The result of the statistical tests served as a guide in choosing a common scanpath for each group of high and low performing students from the four schools for each of the 12 program codes. The selected common scanpaths for each school passed through the second stage of the algorithm to generate the common scanpaths that would reveal common code reading patterns of high and low performing students. These patterns can be used to identify the common code reading strategies employed by high and low performing students.

2.5.2 Common Code Reading Patterns Identification

The study of [8] pointed out the difficulty of coding a one-minute eye movement record, which takes a coder two hours to analyze. This led to the suggestion of integrating quantitative and qualitative methods into a combined research design where a quantitative approach is used before deciding if qualitative analysis is feasible. This process aims to reduce the data for qualitative analysis. The quantitative analysis was implemented in this study using the STA algorithm by identifying the relevant fixations from the individual scanpaths and generating the common scanpath based on the number of fixations and fixation durations as described in Section 2.5.1. By reducing the amount of eye movement data, interpretation of the code reading patterns becomes easier. The descriptions of the patterns in Table 3 were used to determine the code reading patterns of high and low performing students. Additional criteria are provided since not all the code reading patterns in the program comprehension coding scheme could be used because of the nature of the generation of the trending AOIs using the algorithm.

Interpreting the eye movement data requires identification of the number of transitions to be used to describe a code reading pattern. The study of [29] regard three-way transitions as one unit of program understanding behavior. Further, the developed coding scheme [3] describes the LinearVertical pattern as following text line by line for at least three lines regardless of program flow and without distinction between signature and body. Flicking on the other hand, is described as gaze transitions that move back and forth between two related items. Furthermore, the capacity of short-term memory is seven plus or minus two items [22]. It implies that most adults can store between five and nine items in their short-term memory while performing a visual task. Since previous studies characterized the patterns using at least three fixations and that the lower bound of short-term memory capacity is five, the common scanpaths of the high and low performing students were divided into sets of five fixations. A common scanpath with 15 fixations, for example, would have three code reading patterns.

Three coders characterized the code reading patterns of the high and low performing students based on the sets of fixations of all programs from both groups. The fixation sets were coded independently based on the coding scheme. After individually coding the sequence of eye movements using the program comprehension coding scheme and the additional criteria indicated in Table 3, the coders convened to discuss the differences and resolved the conflicts in pattern coding. Fleiss kappa was used to determine the level of agreement between the coders in characterizing the sets of fixations. The scanpath length, code coverage, and eye movement sequence were used to describe the differences in the code reading patterns of high and low-performing students.

Table 3: Patterns from the developed coding scheme of Bednarik *et al.* (2014).

Code names	Pattern description
JumpControl	Subjects jumps to the next line according to execution order.
JustPassing-Through	Fixations are on a blank spot and clearly just stop on the way to some- place else. This pattern cannot be identified since blank lines were not identified as AOIs
LinearHorizontal	Subject reads a whole line either from left to right or right to left, all elements in rather equally distributed time. This pattern cannot be identified since a line of code is considered as one AOI and not divided into several AOIs.
LinearVertical	Subject follows text line by line, for at least three lines, no matter of program flow, no distinction between signature and body.
Flicking	Gaze transitions that move back and forth between two related items, such as the formal and actual parameter lists of a method call, gaze transitions between the different locations where the variable is used, and gaze transitions between the use and declaration of a variable [32]. At least three consecutive fixations on a set of fixations must exhibit such gaze transitions to be considered as a Flicking pattern If two consecutive fixations contain the variables identified on the first fixation followed by an upward movement signifying a visual remember eye accessing cue [12] or a fixation to a code segment that is related to the function of the previous fixation and provided that the next fixation is a statement that contain that same variable or another variable from the latter fixation.

Table 3: Continued.

Code names	Pattern description
Scan	Subject first reads all lines of the code from top to bottom briefly. A preliminary reading of the whole program, which occurs during the first 30 % of the review time [32]. In the context of this study, it can only be characterized from the first and subsequent sets of fixations by tracing the relevant code statements to attain the goal of the program. This is because the clustering method used to generate the common scanpath remove fixations that do not receive a significant amount of attention, therefore the common scanpaths could not show brief gaze transitions.
Signatures	Subject looks at all signatures first, before looking into method/constructor body
Thrashing	The gaze moves rapidly and wildly in a sequence that appears to make no sense.
Word(Pattern)-Matching	Gaze transitions exhibit simple visual pattern matching such as tracing program codes with the same code structure or related codes to perform a particular goal. At least three consecutive fixations on a set of fixations are related to the same structure and that comparison of such structures is evident on the gaze transitions. If two consecutive fixations on a set of fixations are followed by an upward eye movement signifying a visual remember eye accessing cue [12] and provided that the next fixation must be of the same structure.

2.5.3 Common Code Reading Strategies Identification

Strategies used by high and low-performing students can be inferred from the code reading patterns derived from the identified common scanpaths. Top-down comprehension strategy describes an assimilation process where the programmer generates hypotheses about the program code using their programming domain knowledge and tries to verify them by mapping his knowledge to the elements of the source code. Comprehension of the program is guided by the hypothesis. In contrast, bottom-up comprehension strategy describes the assimilation process in which programmers start with individual code statements and chunk or group these source code elements into a higher level of abstraction [17]. In this study, the identified code reading patterns were mapped to these comprehension strategies to determine the code reading strategies of the high and low performing students.

3 Results and Discussion

The primary goal of this study was to identify the common code reading strategies of high and low performing students in debugging. Twenty-five (25) students were identified as high performing while thirty-nine (39) students were considered low performing based on their debugging scores. However, after data pre-processing, one eye tracking data from the high performing group was discarded since most of the fixations on the stimuli were recorded with negative x- and y- gaze coordinates and could not be mapped to the identified AOIs. Thus, the analysis only used the eye gaze data of 63 students.

Figure 3 shows that the boxplots of the debugging scores of both high and low performing students are dispersed in most programs. However, the data of high performing students are mostly negatively skewed with several outliers, while low performing students have mostly positively skewed with some symmetric data and no outliers. Further, in all programs except Program four, the median debugging scores of high performing students show great differences from the median debugging scores of low performing students. An independent samples t-test was performed to determine if there is a significant difference in the debugging scores of the high and low performing students. The result of the analysis revealed that there is a significant difference in the debugging scores of the high performing students ($M = 1.530$, $SD = 0.211$) and low performing students ($M = 0.923$, $SD = 0.250$), $t(61) = -10.322$, $p =< 0.001$. The result suggests that the debugging scores of high performing students are considerably higher than low performing students across all programs.

The boxplots in Figure 4 show that several outliers are present in both groups. The boxplots show extreme values on all programs in low performing

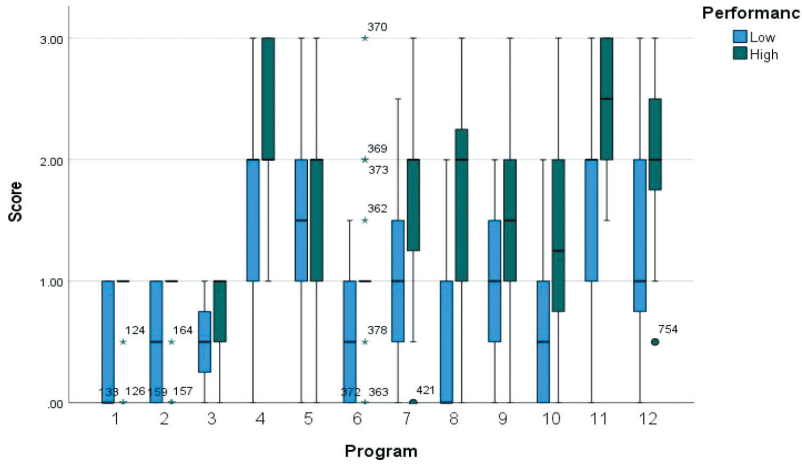


Figure 3: Distribution of debugging scores of high and low performing students.

group while three out of 12 programs had extreme values in high performing group. Further, the median fixation durations of high performing students in Programs 6, 9, 10, and 12 show a great difference from the median fixation durations of low performing students. The boxplots also show that fixation durations of high performing students are more dispersed in most of the programs compared to the data of low performing students. Furthermore, the distribution of data for most of the programs in both groups is positively skewed. An independent samples t-test was performed to determine if there is a significant difference in the average fixation durations of the high and low performing students. The result of the analysis revealed that there is a significant difference in the total fixation durations of high performing students ($M = 151,063.59$, $SD = 36,731.93$) and low performing students ($M = 125,107.42$, $SD = 41,689.59$), $t(61) = -2.508$, $p = 0.015$. The result suggests that the total fixation durations of the high performing students are significantly higher than the low performing students across all programs.

The differences in the debugging score and fixation duration of high and low performing students might have relationships with the pattern similarity presented in Table 6 of Section 3.2. Statistical tests were conducted to determine if a relationship exists between these variables. A Pearson product-moment correlation coefficient was computed to assess the relationship between the difference of the debugging score and pattern similarity of high and low performing students. Results of the Pearson correlation indicated a slight, negative correlation between the two variables, $r(10) = -0.17$, $p = 0.597$; however, the relationship was not significant. This finding suggests that the difference in the debugging score of high and low performing students did not

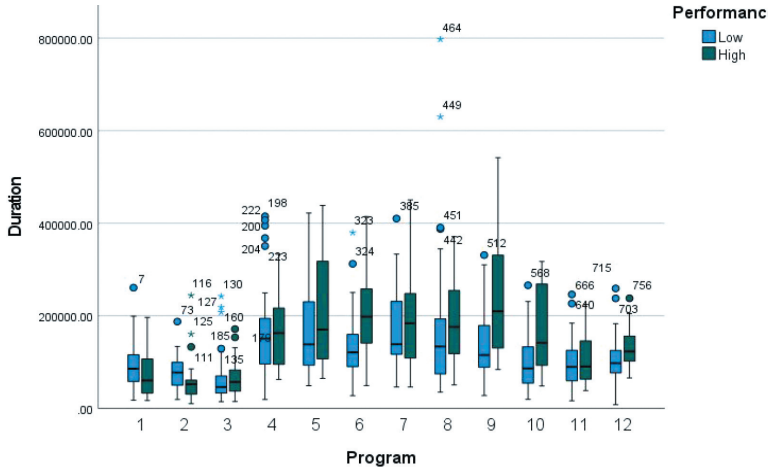


Figure 4: Fixation duration of high and low performing students.

appear to be associated with the pattern similarity. The same correlation test was conducted to assess the relationship between fixation duration and pattern similarity of high and low performing students. Results of the Pearson correlation indicated a moderate, negative correlation between the two variables, $r(10) = -0.53$, $p = 0.074$; however, the relationship was not significant. The difference in the fixation duration of high and low performing students did not appear to be associated with the pattern similarity, although the moderate correlation is nearly significant.

The grouping was used to determine which individual scanpaths belong to the high and low performing groups. After that, the STA algorithm used the fixation count, fixation duration, and position of fixation in the scanpath to generate the common scanpath of the group. The non-significant results of the correlation analysis may be related to the fact that the STA algorithm used several factors to generate the common scanpath of the group to produce the pattern similarity. Therefore, a significant relationship may not be observed using the individual factors.

3.1 Common Scanpath Analysis using Scanpath Trend Analysis with a Tolerance

The three stages of the STA algorithm with a tolerance were employed to the two datasets to generate three common scanpaths for each group of high and low performing students from the four universities. The tolerance level parameter was set first to one which includes all individual scanpaths and then the parameter was adjusted by decreasing it by 0.05 to identify the tolerance

level parameters for each group. The current tolerance level parameter was repeatedly decreased by 0.05 if no changes in the identified trending AOI instances were observed. The first three highest tolerance level parameters were used in the analysis. The three tolerance level parameters that were identified for dataset 1 ranges from 1 to 0.65 for high performing students and from 1 to 0.70 for low performing students. For dataset 2, the identified tolerance level parameters range from 1 to 0.60 for both groups.

After identifying the three common scanpaths of each group with the chosen tolerance level parameters, the similarity scores of the common scanpath with the individual scanpaths and the similarity scores between the individual scanpaths of the 12 programs were computed. The effect of the datasets on the similarity scores for each group of students and school was determined using repeated-measures ANOVA with Greenhouse–Geisser statistics. The statistical analysis used the group of high and low performing students and schools as between-subjects factors. The within-subjects factors are the two similarity scores comparison of the 12 programs on the three tolerance level parameters of the two datasets. The result of the analysis revealed that the similarity scores between the two datasets differed significantly, $F(1, 58) = 32.687$, $p < 0.001$. In comparison to dataset 1, dataset 2 exhibited higher mean similarity scores for all tolerance levels on each program and similarity scores comparison. The higher the similarity scores imply that the generated common scanpath is more similar to the individual scanpaths of the group of participants. Therefore, the common scanpaths of the three tolerance levels from the second dataset were selected to generate the common scanpaths of the high and low-performing students for the 12 program codes.

A statistical test was also conducted to determine which among the three common scanpaths of the second dataset would be selected to generate the code reading patterns of the high and low performing students. The effect of the tolerance level parameters on the similarity scores for each group of students and schools was determined using repeated measures ANOVA with a Greenhouse–Geisser correction. The statistical analysis used the group of high and low performing students and schools as between-subjects factors. The within-subjects factors are the two similarity scores comparison of the 12 programs on the three tolerance level parameters of the second dataset. The result of the statistical test suggests that the selected tolerance level parameters had a significant effect on the similarity scores, $F(1.535, 89.002) = 55.574$, $p < 0.001$. Therefore, it is important to select the appropriate tolerance level where the common scanpath of the group of students from each school would come from. The following criteria were used to determine the appropriate tolerance level:

1. The common scanpath of the first tolerance level would be selected if it has the highest similarity score among the three tolerance levels.

2. The common scanpath of the second tolerance level would be selected if the difference of the similarity scores between the second and first tolerance level is higher than the value of 1 minus the tolerance level parameter of the second tolerance level multiplied by the similarity score of the first tolerance level. Otherwise, the first tolerance level would be selected. For example, the similarity score of the second tolerance level is 29.85, the tolerance level parameter is 0.95, and the similarity score of the first tolerance level is 28.36. We will take the difference of 29.85 and 28.36, which is equivalent to 1.49, and compare it to the value of $((1 - 0.95) \times 28.36)$, which is 1.418. Since 1.49 is greater than 1.418, the common scanpath of the second tolerance level would be selected.

3. The common scanpath of the third tolerance level would be selected if the similarity score is greater than that of the first and second tolerance levels and the difference of the similarity scores between the third tolerance level and the second-highest tolerance level is greater than the value of 1 minus the tolerance level parameter of the third tolerance level multiplied by the similarity score of the first tolerance level. Otherwise, the tolerance level with the second-highest similarity score will be selected. For example, the third tolerance level is 0.9 and the similarity score is 33.50, the similarity score of the second tolerance level is 29.85, and the similarity score of the first tolerance level is 28.36. The third tolerance level has the highest similarity score, so we have to check for the other condition. The second-highest similarity score is 29.85, which is from the second tolerance level. We then get the difference of 33.50 and 29.85, which is 3.65, and calculate the value of $((1 - 0.90) \times 28.36)$, which is 2.836. Since 3.65 is greater than 2.836, the common scanpath of the third tolerance level would be selected.

Table 4 shows a summary of the results of the statistical analysis. The main effects of the similarity scores comparison, programs, tolerance levels, and datasets show statistically significant differences. However, the datasets and tolerance levels have no significant interaction with school and performance.

Table 5 shows the selected tolerance level parameters for each program, school, and group of students. The selected common scanpaths of the high and low-performing students were processed using the second pass stage of the STA with a tolerance to generate the common scanpaths for each program and group. The findings revealed that individual scanpaths tend to be different from each other, and the differences may affect the identification of a representative scanpath of a group which could decrease its similarity to individual scanpaths. Based on the number of fixation and proportional fixation duration data from each school and group, lower tolerance levels were selected for high-performing students on programs where at least one of the members had many fixations and long proportional fixation durations. On the other hand, lower tolerance

Table 4: Summary of the results of statistical analysis.

Measure	*df*	*F*	Sig.
Similarity scores comparison	1, 58	147.023	$p < 0.001$
Program	7.692, 446.111	119.636	$p < 0.001$
Dataset	1, 58	32.687	$p < 0.001$
Dataset × school	3.000, 58	1.150	$p = 0.337$
Dataset × performance	1, 58	0.068	$p = 0.795$
Tolerance level	1.535, 89.002	55.574	$p < 0.001$
Tolerance level × school	4.604, 89.002	0.129	$p = 0.981$
Tolerance level × performance	1.535, 89.002	2.355	$p = 0.114$

Table 5: Tolerance level parameters of the selected common scanpaths.

	ADDU		UC		USC		USeP	
Program	HP	LP	HP	LP	HP	LP	HP	LP
1	1	1	0.95	1	1	1	1	1
2	1	1	0.90	1	0.95	1	1	1
3	1	1	1	1	1	1	1	1
4	1	1	1	1	1	1	1	1
5	1	1	1	1	1	1	1	1
6	1	0.90	1	1	1	0.95	1	1
7	1	1	1	1	1	1	1	1
8	1	1	1	1	1	1	1	0.95
9	1	1	1	1	1	1	1	1
10	0.95	1	1	1	1	1	1	1
11	1	1	0.95	1	1	1	1	1
12	1	1	1	1	1	1	1	0.95

levels were selected for low-performing students on programs where at least one of the members had a lesser number of fixations and shorter proportional fixation durations. These findings suggest that decreasing the tolerance level parameter during the identification of trending AOI instances could generate common scanpaths that are more similar to individual scanpaths.

3.2 Common Code Reading Patterns of High and Low Performing Students

The common scanpaths generated using STA with a tolerance were divided into fixation sets to interpret the code reading patterns. Fleiss' kappa was run to determine if there was agreement between the judgement of the three coders

in the characterization of the 164 sets of fixations of high and low performing students using the code reading patterns described in Table 3. The fixation sets were coded independently by the three coders that were chosen at random from a pool of faculty members teaching Java programming. Fleiss' kappa showed that there was a moderate agreement between the coders' judgements, $k = 0.586(95\%\ \text{CI}, 0.497 \text{ to } 0.674)$, $p < 0.001$.

The differences in the code reading patterns of high and low performing students were characterized using the scanpath length, code coverage, and the sequence of eye movements. Based on the generated common scanpaths using STA with a tolerance on all programs, the results revealed that the code reading patterns vary in terms of the scanpath length, code coverage, and sequence of eye fixations. The qualitative analysis of the sequence of fixations revealed six code reading patterns used by the high and low performing students in debugging. These patterns are Scan (S), Flicking (F), LinearVertical (L), JumpControl (J), Word(Pattern) Matching (W), and Thrashing (T). Table 6 shows the summary of the common code reading patterns of high and low performing students. The pattern similarity of the high and low performing students is 100% in Program 3 but greatly differs from other programs. Notably, the code reading patterns of programs 10 and 12 had a very low pattern similarity since the low performing students had very short scanpaths. The other programs had pattern similarity scores that range from 20% to 67%. This result suggests that the high and low performing students employed different code reading patterns in performing the debugging tasks.

The sequence of eye movements of the high performing students revealed that they start reading the program code by employing a logical Scan pattern or a Flicking pattern. The logical Scan pattern exhibit scanning of code elements according to programming plans and is the initial code reading pattern employed on most of the programs used in this study. Programs 1–4 and 8 were read starting from the main function followed by fixations to code elements related to the sub-goals of the programs. Figures 5 and 6 show the common scanpath of the high performing students in Programs 1 and 2, respectively. Based on the gaze plots, the high performing students focused on code elements related to the functional goal of the programs first which allows them to immediately identify the errors injected in the variable declaration and repetition structure. The logical Scan pattern is immediately followed by Thrashing patterns after fixation to the error line to verify that they had correctly identified the injected error.

Program 6 was read starting from the variable declaration followed by code elements related to the input sub-goal of the program. Program 7 is the most complex in terms of the number of lines of code and code complexity. The common code reading patterns employed in finding the defect in this program is different from all the programs because Thrashing patterns were employed in between focused fixations. Programs 11 and 12 were read from

Table 6: Summary of the characteristics of the common code reading patterns.

Program	Group	Scanpath length	Code coverage	Pattern sequence	Pattern similarity
1	High	21	All lines	SSTT	25%
	Low	20	All lines except line 19	TTLT	
2	High	14	Almost all lines except line 1	STT	67%
	Low	14	Almost all lines except on the output statement	LTT	
3	High	12	Almost all lines except lines 1–2, 17	SWW	100%
	Low	15	Almost all lines except lines 1–3 and 17	SWW	
4	High	35	Almost all lines except 1, 2 and 9	SFTWTTT	43%
	Low	14	Lines 5–17	FTT	
5	High	70	Almost all lines except last two lines	FWFTWTTTTTTTTT	29%
	Low	24	Almost all lines except first and last 3 lines	TTTFT	
6	High	44	Almost all lines except 1–3	STTJTJJTT	22%
	Low	20	Almost all lines except 1–3 and 22	LFTT	
7	High	62	Lines 10–33	STTFTTWTTTTT	50%
	Low	39	Lines 5–19, 25, 30–31	FFFTTTTT	
8	High	53	Lines 6–28	SFTTWWTTJTT	64%
	Low	43	Lines 5–24	TFJWTTTTT	
9	High	72	Lines 11–31	FJFTTTTTTTTTTTT	20%
	Low	21	Lines 5–21	LTTT	
10	High	34	Almost all lines except 1–3	FTJFFJT	14%
	Low	7	Lines 6–16	LT	
11	High	39	Almost all lines except 1–3	SFFTTTTT	38%
	Low	36	Lines 6–22	LLTTTWL	
12	High	85	Almost all lines except 1–4	SSFTTTTTTTWTTTTTT	12%
	Low	11	Lines 7–18 and 22	SF	

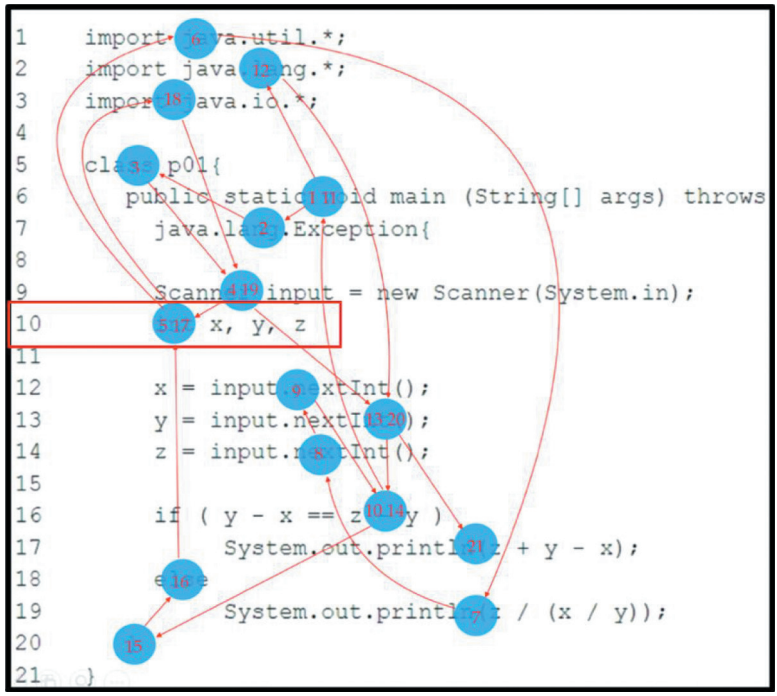


Figure 5: Common scanpath of high performing students in program 1.

the input and assignment statements followed by fixations to conditional structures.

The common scanpaths of low performing students mostly start with LinearVertical patterns while other programs begin with either a Thrashing, a logical Scan, or a Flicking code reading pattern. Among all the programs, the code reading patterns of low performing students in Program 3 is the same as that of the high performing students. The code reading patterns employed by low performing students in Programs 1 (Figure 7) and 2 (Figure 8) show late fixations to the error lines compared to the code reading patterns of high performing students who made early fixation on the error lines by employing the logical Scan pattern first. The common code reading patterns of Programs 4 and 5 employed a combination of a Flicking and several Thrashing patterns. The common scanpath of Program 7 revealed code reading patterns characterized by a combination of Flicking patterns and Thrashing patterns, similar to the code reading patterns of Programs 4 and 5. Among all the programs, the common scanpaths of Programs 6 and 8 revealed more than two patterns of eye movements. Program 6 was read using a LinearVertical code reading pattern followed by a Flicking code reading pattern and succeeded by

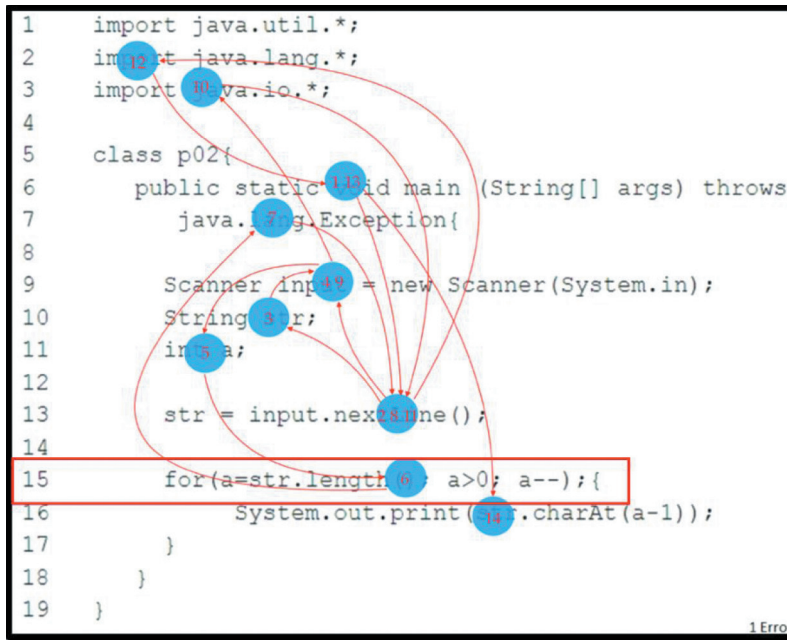


Figure 6: Common scanpath of high performing students in program 2.

two Thrashing code reading patterns. The common scanpath of Program 8 revealed a Thrashing code reading pattern followed by Flicking, JumpControl, and Word(Pattern) Matching patterns. The code reading patterns of Program 9 are more similar to that of Program 6. The similarity might be because both programs had logical errors injected on the output statements. The low performing students employed a LinearVertical code reading pattern followed by a Thrashing pattern in debugging Program 10. The shorter scanpath length and the different sequences of code reading patterns might be due to the errors injected in these programs. The common scanpath of Program 11 consists of a combination of LinearVertical, Thrashing, and Word(Pattern) Matching code reading patterns. Similar code reading patterns with Program 11 is expected for Program 12 since both programs have a similar program structure. However, the common scanpath of Program 12 did not show a Thrashing code reading pattern and had a very short common scanpath similar to Program 10. The difference in the code reading patterns might be because the sequence of eye movements of the participants was very different from each other in Program 12 than in Program 11, which affected the identification of the common scanpath. Thus, the scanpath length in Program 12 is shorter than that of Program 11.

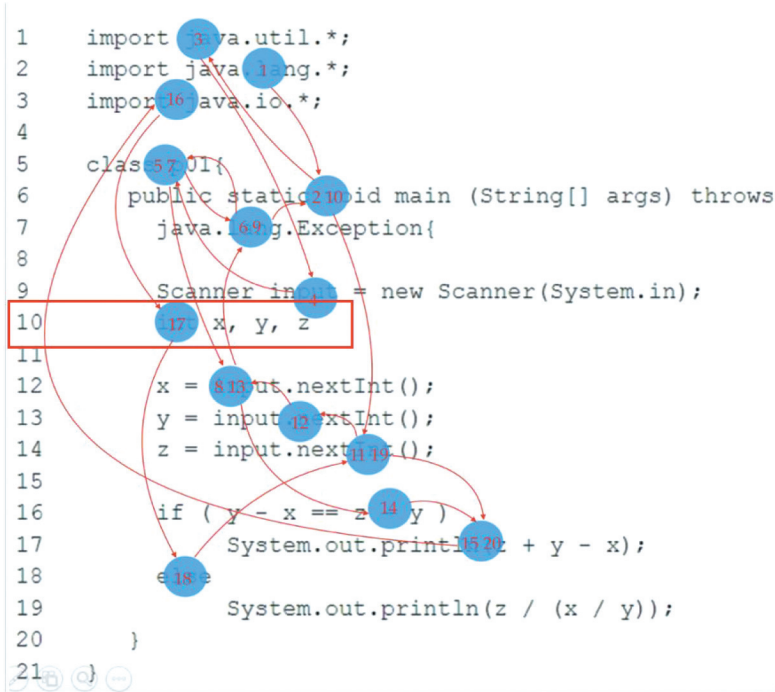


Figure 7: Common scanpath of low performing students in program 1.

3.3 Common Code Reading Strategies

Strategies employed in code reading affect a programmer's success rate of the comprehension tasks. Top-down and bottom-up models are two of the most common program comprehension strategies employed by programmers while performing code comprehension tasks. Table 7 shows the code reading strategy employed by the high and low performing students in each program. A top-down strategy is used by high performing students in most programs, while the low performing students prefer a trial-and-error code reading strategy. Both high and low performing students used the same top-down code reading strategy in Programs 3 and 11.

A top-down code reading strategy was employed to find bugs in programs that had simple or nested conditional structure, simple repetition structure, sequential output statements, and selection structure. However, a bottom-up code reading strategy was employed to find errors in more complex programs that had a repetition structure with multiple conditional structures and nested repetition structures. The deductive approach to code reading might be more appropriate for short programs [3]. But for complex programs, an

Table 7: Summary of code reading strategies of high and low performing students.

Program	Group	Lines of code	Control structures used	Strategy
1	High	17	Conditional structure (if/else)	Top-Down
	Low			Trial-and Error
2	High	15	Repetition Structure with one output statement	Top-Down
	Low			Trial-and Error
3	High	16	Sequential output statements	Top-Down
	Low			Top-Down
4	High	26	Nested for loop	Bottom-Up
	Low			Trial-and Error
5	High	24	Repetition structure (for loop) with conditional	Bottom-Up
	Low		structures	Trial-and Error
6	High	19	Repetition structure (for loop) with one conditional	Top-Down
	Low		structure	Trial-and Error
7	High	34	Repetition structure (for loop) with conditional	Trial-and-Error
	Low		structures and one method	Trial-and Error
8	High	27	Two nested for loops	Bottom-Up
	Low			Bottom-up
9	High	29	Two nested for loops	Bottom-Up
	Low			Trial-and Error
10	High	16	Repetition structure (for loop) with one conditional	Top-Down
	Low		structure	Trial-and Error
11	High	25	Nested conditional structure	Top-Down
	Low			Top-Down
12	High	28	Selection structure	Top-Down
	Low			Trial-and Error

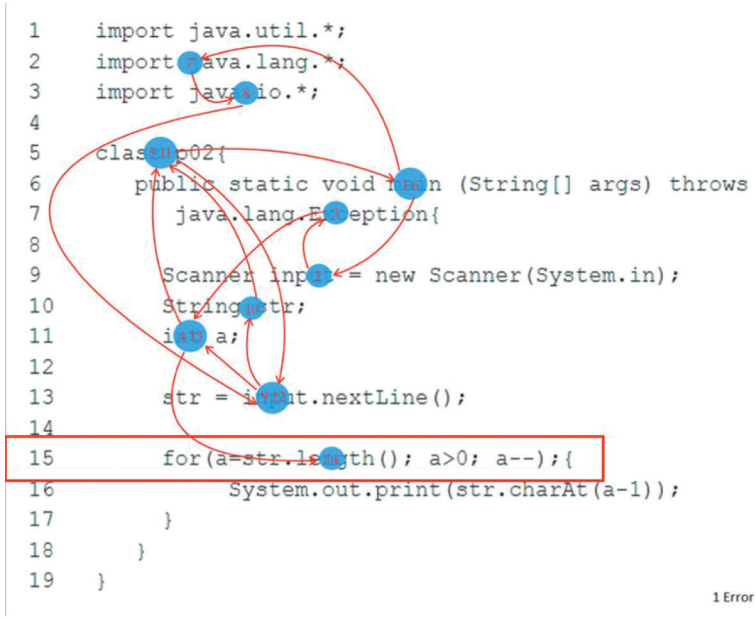


Figure 8: Common scanpath of low performing students in program 2.

inductive approach might be more appropriate to use. Based on the number of lines of code, the programs used in this study can be considered as short programs. Thus, using a deductive approach or a top-down strategy to find the injected errors in these programs might be more effective. As presented in Table 6, the code reading patterns of the high-performing students usually start with a logical Scan pattern to find relevant code segments first, which shows the intention to apply the top-down code reading strategy. However, high performing students also employed a bottom-up code reading strategy in four programs. The change in the employed code reading strategy might be due to the complexity of the programming constructs used besides having more lines of code. Among all the programs, Program 7 was read differently by the high performing students. They employ a trial-and-error code reading strategy instead of a bottom-up strategy. The sequence of eye movements showed more scanning within the program and less focused fixations on related code elements. The high-performing students might have been exhausted after reading six programs and after encountering difficulty in finding the injected logical errors of Program 6. The random fixations might be because of the difficulty of identifying the third error injected in the method definition since only 11 out of 24 students were able to detect this error. Further, Program 7

is the most complex among all the programs used in this study and the only program that had an additional method. These might be the reasons why high performing students employed a trial-and-error strategy.

Unlike high performing students, the low performing students used a trial-and-error code reading strategy for most of the programs. The code reading patterns reveal that the code elements were fixated randomly to find the injected errors. We note that the low performing students were able to employ a top-down code reading strategy in Programs 3 and 11. It might be because the programming constructs of these programs were familiar to most students. Both high and low-performing students employed the same sequence of patterns in Program 3. Programmers mostly use top-down strategy when they are familiar with the application domain [25, 4]. Therefore, we were able to observe this code reading strategy from both groups. Although low performing students employed a bottom-up code reading strategy in Program 8, the sequence of patterns showed more random scanning than a careful examination of chunks of related code segments to find the bugs. This finding suggests that the low performing students did not use evident code reading strategy in the bug-finding task. In contrast, the code reading patterns of the high performing students revealed that they could employ top-down and bottom-up code reading strategies depending on the availability of the information in a particular situation. For instance, the complexity of the programming construct and the type of errors injected into the program might dictate the code reading strategy to be employed. The high performing students demonstrate the use of flexible code reading strategies when performing the debugging task.

4 Discussion

STA algorithm with a tolerance gives us the ability to find an appropriate tolerance level parameter for achieving the highest similarity of the common scanpath to the individual scanpaths. Results show that by adjusting the tolerance level parameter, we were able to generate scanpath that describes the common code reading patterns of high and low performing students while finding bugs in static source codes. These patterns can be used to determine the code reading strategies of the high and low performing students.

The common scanpaths of the high and low performing students using STA with a tolerance on all programs revealed that the code reading patterns vary in terms of the scanpath length, code coverage, and sequence of eye fixations. Both high and low-performing students visited almost the same lines of code in Programs 1, 3, 5, and 6. However, for programs with complex programming constructs, the common scanpaths of high-performing students revealed visual attention to almost all lines of code while low-performing

students had limited and sometimes no visual attention on the lower half of the programs. The sequence of fixations of the high and low performing students also differs from each other except in Program 3. The high performing students employed logical scan patterns by tracing the code elements related to the sub-goals of the program. Non-linear code reading patterns were mostly employed by the high performing students using Scan, Thrashing, Flicking, Word(Pattern) Matching, and JumpControl code reading patterns. Most of the code reading patterns of the high performing students start with a Scan and end with Thrashing patterns. These random eye movements can be considered as a process of verification since a series of careful examination of blocks of related code statements were employed first before the Thrashing code reading patterns. Low performing students mostly employed non-linear code reading patterns using Thrashing patterns and logical Scan patterns that usually start with the main method, variable declaration, or input and assignment statements. However, the fixations of the low performing students in the first set of fixations typically had regressions to previously scanned code elements while high performing students do not make regressions on the initial set of fixations. The findings in the analysis of code reading patterns support a previous study finding that program codes are read less linearly than natural language texts [7]. This finding is evident in the code reading patterns of both high and low performing students because they trace the code elements non-linearly. Further, this study suggests that low-performing students read code without following the program's logic while high-performing students read code in a logical manner, which is in line with [21]. Furthermore, it is suggested that repetitive eye movements may be linked with less expertise [2]. The common scanpaths of the low performing students show frequent regressions while the high performing students had very few regressions within the sets of eye movements in each program. This finding confirms the conclusion that experts could recall more program lines compared to novices [23].

The code reading patterns of the high and low performing students revealed code reading strategies in debugging that could be explicitly taught to students. A top-down approach was employed to find bugs in programs that had simple or nested conditional structure, simple repetition structure, sequential output statements, and selection structure. However, a bottom-up code reading strategy was employed to find errors in more complex programs that had a repetition structure with multiple conditional structures and nested repetition structures. This implies that the high performing students employ flexible strategies in debugging programs with different programming constructs and code complexity. The result of the analysis confirms previous findings that programmers apply flexible code reading strategies depending on the programming tasks [5] and that experts tend to organize code elements into chunks that reflect semantic structures [11, 21]. Low performing students

on the other hand, mostly employ trial-and-error strategy where they tend to debug code erratically. This is in line with the finding of [21] and [28], wherein novices spent more time scanning the program than experts and that novices tinker aimlessly within the program.

5 Threats to Validity

The way in which the common scanpaths are coded might pose a threat to the internal validity of this research. With the absence of a solid description of the optimal number of fixations to be considered as a unit of program understanding behavior when debugging programs, we used 5 fixations to characterize a code reading pattern. To provide more credible characterization of code reading patterns, a standard measure must be established in future studies to increase the reliability of the result. Another threat to the internal validity is the way the eye tracking data is processed to generate the common scanpath using STA algorithm with a tolerance. The clustering method used to generate the common scanpath remove fixations that do not receive a significant amount of attention. Therefore, the common scanpaths could not show brief gaze transitions described as a Scan pattern. However, a Scan pattern can still be characterized if the first and subsequent sets of fixations traces the relevant code statements to attain the goal of the program. This means that the set of fixations or sets of fixations exhibit scanning of code elements according to programming plans. With regard to the external validity of the research, each program was injected with different number and types of error. The participants are aware of the number of errors but not as to what type of errors were injected in the programs. Thus, debugging strategies might be influenced by the interactions between them. Future experiments might be designed to include the same number and type of injected errors to determine how participants read code when finding a specific type of error.

6 Conclusion

Code reading strategies employed by high and low performing students in debugging were identified using eye-tracking data and STA with a tolerance algorithm. The common scanpaths revealed differences in the code reading patterns and strategies of high and low performing students. High performing students apply top-down and bottom-up code reading strategies depending on the complexity of the programs and the type of injected errors. This implies that they use flexible debugging strategies based on the program structure and the injected errors while no evident code reading strategy can be inferred from the common scanpaths of the low performing students. The qualitative

analysis of the common scanpaths of the high and low performing students also confirms previous research findings. High performing students draw more visual attention to code elements related to the sub-goals of the program and can recognize code elements related to programming plans when performing a debugging task. Low performing students, on the other hand, tend to debug the code aimlessly.

This study contributes to the on-going exploration of expert's strategies in code reading by exploring the strategies used by high performing students that can be taught to low performing students to help improve their debugging skills. Computing educators could teach more problem-solving skills and debugging strategies to enhance the student's ability to plan for the debugging task. Students could learn how to identify the goal or sub-goals of the program and map them to the program's code elements. Further, teaching students to consciously employ code reading strategies would help them develop their own effective approach and improve their code reading and debugging skills.

Eye-tracking and computer science education research would also benefit from this study because it would contribute to the existing literature on the use of eye tracking technology and the effectiveness of analyzing eye-tracking data in identifying code reading patterns and strategies. Further, this study validates the findings of our previous work and brings something new to the literature since the analysis of the scanpaths used proportional fixation durations to differentiate trending AOIs instead of the actual fixation duration described in the original algorithm. However, further studies using the proportional fixation duration in identifying trending AOIs need to be conducted to acquire firmer evidence on its effectiveness.

Financial Support

This study was supported in part by the Private Education Assistance Committee of the Fund for Assistance to Private Education (Analysis of Novice Programmer Tracing and Debugging Skills using Eye Tracking Data) and Ateneo de Manila University's Loyola Schools Scholarly Work Faculty (Building Higher Education's Capacity to Conduct Eye-tracking Research using the Analysis of Novice Programmer Tracing and Debugging Skills as a Proof of Concept).

Ethical Standards

The authors assert that all procedures contributing to this work comply with the ethical standards of the relevant national and institutional committees on

human experimentation and with the Helsinki Declaration of 1975, as revised in 2008.

Biographies

Christine Lourrine S. Tablatin received a degree in Master in Information Technology from Colegio de Dagupan in 2010 and a Ph.D. in Computer Science candidate from the Ateneo de Manila University. She is currently the research coordinator of Pangasinan State University – Urdaneta City Campus. Her main research interests are learning analytics and educational data mining.

Maria Mercedes T. Rodrigo received her Ph.D. in Computer Technology in Education from the Nova Southeastern University. She is a professor of the Ateneo de Manila University and teaching in the Department of Information Systems and Computer Science. Her areas of specialization are educational technology, intelligent tutoring systems, and affective computing. She established the Ateneo Laboratory for the Learning Sciences through a grant from the Department of Science and Technology's Engineering Research and Development for Technology program in 2011.

References

[1] M. Andrzejewska, A. Stolinska, A. Blasiak, P. Pkeczkowski, B. Rozek, M. Sajka, and D. Wcislo, "Eye-Tracking Verification of the Strategy Used to Analyse Algorithms Expressed in a Flowchart and Pseudocode," *Interactive Learning Environments*, 24(8), 1981–95.

[2] R. Bednarik, "Expertise-Dependent Visual Attention Strategies Develop Over Time during Debugging with Multiple Code Representations," *International Journal of Human-Computer Studies*, 70, 143–55.

[3] R. Bednarik, T. Busjahn, and C. Schulte, *Eye Movements in Programming Education: Analyzing the Expert's Gaze*, Finland, 2014.

[4] R. Bednarik, N. Myller, E. Sutinen, and M. Tukiainen, *Program Visualization: Comparing Eye-Tracking Patterns with Comprehension and Summaries Performance*, 2006.

[5] R. Bednarik and M. Tukiainen, *Temporal Eye-Tracking Data: Evolution of Debugging Strategies with Multiple Representations*, USA, 2008.

[6] R. Bednarik and M. Tukianen, *Visual Attention and Representation Switching in Java Program Debugging: A Study using Eye Movement Tracking*, Ireland, 2004.

[7] T. Busjahn, R. Bednarik, A. Begel, M. Crosby, J. H. Paterson, C. Schulte, B. Sharif, and S. Tamm, *Eye Movements in Code Reading: Relaxing the Linear Order*, Italy, 2015.

[8] T. Busjahn, C. Schulte, and E. Kropp, *Developing Coding Schemes for Program Comprehension using Eye Movements*, United Kingdom, 2014.

[9] K. R. Chandrika and J. Amudha, "An Eye Tracking Study to Understand the Visual Perception Behavior while Source Code Comprehension," *International Journal of Control Theory and Applications*, 10(19), 169–75.

[10] M. E. Crosby and J. Stelovsky, "How Do We Read Algorithms? A Case Study," *Computer*, 23(1), 25–35.

[11] F. Detinne, "Expert Programming Knowledge: A Schema Based Approach," in, *Psychology of Programming*, ed. J. M. Hoc, T. R. G. Green, R. Samurcay, and D. J. Gillmore, London: Academic Press, 1990, 205–22.

[12] R. Dilts, *Roots of Neuro-Linguistic Programming*, Capitola, CA: Meta Publications, 1983.

[13] S. Eraslan, Y. Yesilada, and S. Harper, *Engineering Web-Based Interactive Systems: Trend Analysis in Eye Tracking Scanpaths with a Tolerance*, Lisbon Portugal, 2017.

[14] S. Eraslan, Y. Yesilada, and S. Harper, "Identifying Patterns in Eye-tracking Scanpaths in Terms of Visual Elements of Web Pages," in, *Web Engineering, ICWE 2014. LNCS, 8541*, ed. S. Casteleyn, G. Rossi, and M. Winckler, Springer International Publishing, 2014, 163–80.

[15] Gazepoint, *Gazepoint Control User Manual Rev 2.0*, Retrieved June 22 2021, 2014, http://apps.usd.edu/coglab/schieber/eyetracking/Gazepoint/pdf/GazepointControlManual.pdf.

[16] J. H. Goldberg and J. I. Helfman, *Scanpath Clustering and Aggregation*, New York, USA, 2010.

[17] E. Harth and P. Dugerdil, *Program Understanding Models: An Historical Overview and Classification*, 2017.

[18] P. Hejmady and N. Hari Narayanan, *Visual Attention Patterns during Program Debugging with an IDE*, Santa Barbara California, 2012.

[19] A. Jbara and D. G. Feitelson, "How Programmers Read Regular Code: A Controlled Experiment using Eye-Tracking," *Empirical Software Engineering*, 22(3), 1440–77.

[20] T. C. A. Kubler, "Algorithms for the Comparison of Visual Scan Patterns," Dissertation. Eberhard Karls University Tübingen.

[21] Y. Lin, C. Wu, T. Hou, Y. Lin, F. Yang, and C. Chang, "Tracking Students Cognitive Processes during Program Debugging: An Eye-Movement Approach," *IEEE Transactions on Education*, 59(3), 175–86.

[22] G. A. Miller, "The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information," *Psychological Review*, 63(2), 81–97.

[23] M. Nivala, F. Hauser, J. Mottok, and H. Gruber, *Developing Visual Expertise in Software Engineering: An Eye Tracking Study*, Abu Dhabi, 2016.

[24] P. Romero, R. Cox, B. Du Buolay, and R. Lutz, *Visual Attention and Representation Switching during Java Program Debugging: A Study using the Restricted Focus Viewer*, Georgia USA, 2002.

[25] T. Shaft and I. Vessey, "The Relevance of Application Domain Knowledge: The Case of Computer Program Comprehension," *Information Systems Research*, 6(3), 286–99.

[26] Z. Sharafi, Z. Soh, Y. Guéhéneuc, and G. Antoniol, *Women and Men—Different but Equal: On the Impact of Identifier Style on Source Code Reading*, Germany, 2012.

[27] Z. Sharafi, Z. Soh, and Y.-G. Guéhéneuc, "A Systematic Literature Review on the Usage of Eye-Tracking in Software Engineering," *Information and Software Technology*, 67, 79–107.

[28] B. Sharif, M. Falcone, and J. I. Maletic, *An Eye-Tracking Study on the Role of Scan Time in Finding Source Code Defects*, California, 2012.

[29] K. Sharma, P. Jermann, M. Nussli, and P. Dillenbourg, *Gaze Evidence for Different Activities in Program Understanding*, London UK, 2012.

[30] C. S. Tablatin and M. M. T. Rodrigo, *Identifying Common Code Reading Patterns using Scanpath Trend Analysis with a Tolerance*, Metro Manila, Philippines, 2018.

[31] J. Tvarozek, M. Konopka, P. Navrat, and M. Bielikova, *Studying Various Source Code Comprehension Strategies in Programming Education*, Finland, 2016.

[32] H. Uwano, M. Nakamura, A. Monden, and K. Matsumoto, *Analyzing Individual Performance of Source Code Review using Reviewers' Eye Movement*, California USA, New York, NY, USA, 2006.

[33] A. Vosskühler, *OGAMA Description (for Version 2.5)*, Berlin, Germany: Freie Universität Berlin, Fachbereich Physik., 2009.