Investigating Eye Movements in Natural Language and C++ Source Code - A Replication Experiment

Patrick Peachock, Nicholas Iovino, and Bonita Sharif(✉)

Youngstown State University, Youngstown, OH 44555, USA
{prpeachock,nriovino}@student.ysu.edu, bsharif@ysu.edu

Abstract. Natural language text and source code are very different in their structure and semantics. Source code uses words from natural language such as English mainly in comments and identifier names. Is there an inherent difference in the way programmers read natural language text compared to source code? Does expertise play a role in the reading behavior of programmers? In order to start answering these questions, we conduct a controlled experiment with novice and non-novice programmers while they read small short snippets of natural language text and C++ source code. This study is a replication of an earlier study by Busjahn et al. [1] but uses C++ instead of Java source code. The study was conducted with 33 students, who were each given ten tasks: a set of seven programs, and three natural language texts. They were asked one of three random comprehension questions after each task. Using several linearity metrics presented in an earlier study [1], we analyze the eye movements on source code and natural language. The results indicate that novices and non-novices both read source code less linearly than natural language text. We did not find any differences between novices and non-novices between natural language text and source code. We compare our results to the Busjahn study and provide directions for future work.

Keywords: Eye tracking study · C++ · Program comprehension · Natural language

1 Introduction

Programmers are required to not just write source code, but read and comprehend it as well. The better a programmer comprehends code, the better they will be at debugging and finding faults. Programming is not as straightforward as it may seem to the beginner programmer. While it may look linear in its structure, source code is very different from natural text. It is not commonly read and executed left to right, top to bottom. Rather, it skips up and down, only using what part of the program is needed at the time that it is called i.e. the control flow. Natural text such as English prose, on the other hand is read from top to down and left to right.

D.D. Schmorrow and C.M. Fidopiastis (Eds.): AC 2017, Part I, LNAI 10284, pp. 206–218, 2017.
DOI: 10.1007/978-3-319-58628-1_17

We derive our inspiration from a study conducted by Busjahn et al. [1]. They compared eye movements of novice and expert programmers who were asked to read and comprehend natural language text and Java programs. They found that novices read source code less linearly than the natural language texts. In addition, the experts were found to read code even less linearly than the novices. In this paper, we replicate the Busjahn et al. study because we wanted to determine if their results still hold for C++ on a different sample. We do not label our participants as experts and we did not have any participants from industry. Rather, we split our participants as novices and non-novices as detailed in Sect. 3.

The motivation behind this replication is to add to the growing research that is focused in the area of natural language and source code comparison. Current research shows little to no proof that being a novice or expert programmer affects the readability of source code. Since it is very likely that programmers start with reading code written by others, code comprehension plays an important role in programming. In this paper, we propose two different research questions.

- RQ1: Is there an inherent difference in the way novice and non-novice programmers read natural language text compared to source code?
- RQ2: Does expertise play a role in the reading behavior of programmers, in particular, with respect to linear reading?

If we can begin to analyze these patterns and behaviors, it can help to better our methods and practices of teaching and programming overall.

The paper is organized as follows. We discuss some related work in the area in the next section. In Sect. 3, we introduce the study and discuss the results in Sect. 4. Finally we present our conclusions and future work.

2 Related Work

Eye trackers are devices that are able to detect where on a screen a user is looking. An eye tracker will typically record two different types of data, saccades and fixations. A fixation is where an eye has come to rest on part of the screen for a given amount of time. A saccade is the movement from that fixation, to the next fixation. The time for these can vary but is frequently between 200 and 300 ms. Analysis programs will often mark fixations with a dot that grows the longer a user focuses on a spot on the screen. These dots are connected by a line called a scan path [2].

The area of investigation of our study falls under the general area of program comprehension. A program is defined as a set of instructions that are written to perform a specified task. Program comprehension can be defined as the understanding of these lines of code typically written in some language such as Java or C++. There have been several different studies done in the field of program comprehension, many focusing around fragments of code or beacons. While these beacons still play a major part in program comprehension, it has been shown

that they are not often the same between all users. That is to say that different programmers see code in different ways [3].

A more recent study took a look into the way that students can retain information by reading in a less linear pattern. Raina et al. ran a study to see if information presented in a segmented pattern as opposed to a linear pattern would help students to better understand what was presented to them [4]. Using an eye tracker they were able to look at reading scores and reading depth to gather data. The study conducted used a group of 30 students (cut down to 19 due to inaccurate calibrations). Using a control group and a treatment group, they were given the same module in C++. The control group had the module in a linear format. The treatment group used a segmented format. Results showed that when reading in segments, students had higher reading and depth scores. They were able to not only focus on the information presented but understood it better than those reading in a linear fashion.

To expand and study the way that users read programming code, a study was done on the way that programmers read code with syntax highlighting [5]. The study looked to find if it was beneficial to the student to have syntax highlighting (colorization of specific keywords and constructs) from different software development environments. The study included 31 participants using the C# programming language and separated them into two groups: black and white code and code with syntax highlighting. The program having no errors, and the omitting of a time limit gave the programmers adequate time to go through the program to determine its output. The data was recorded with several metrics including fixations, regressions, and scan percentage. In conclusion, the study showed that there was minimal difference between all three metrics. This suggests that syntax highlighting, while possibly more pleasing to the eye, does not make a significant difference in the reading patterns of a programmer. However, in a different study, Sarkar found that highlighting did help with task completion but the effect decreased as programming experience increases [6].

One of the first studies done on natural language versus computer language led to a belief that natural language may be simpler than programming code. Having only been studied on novice programmers, this left room to expand research to include both novice, non-novice, and expert programmers [7]. Following on Crosby's research into beacons [7], Fan's study focused on beacons and comments within programming code and the benefits they have on program comprehension. The study showed the same findings as Crosby in that beacons are noticed differently by each programmer. It also did find that comments within code helped improve code comprehension [8].

Focusing on a different area than natural language and code comprehension, Sharif et al. turned to the comparison of programming languages. The study used students that were studying programming, broken into groups based on their knowledge of both Python and C++ programming languages. The C++ group did have more participants, this was directly related to the amount of programming courses offered to the students. The students had three different tasks that involved finding bugs as well as the overview of two different tasks. Using accuracy, time, fixation counts, and fixation duration, the study showed

that accuracy higher in C++, also that novices took longer overall in C++. This group also had higher fixation counts than the Python group. The students doing Python tasks took longer than those using C++. While these metrics showed different, the final data analysis showed that there was no significant statistical difference when comparing the two programming languages [9].

Sharif and Maletic took a look into using an eye tracker to gather data on the different identifier styles i.e., underscores versus camel case [10]. The eye tracker was an improvement on a previous study [11] that used response time to get data between the two styles. Using the eye tracker, it was found that the different naming styles had a different effect on time as well as effort to detect identifiers within source code. It was shown that camel-casing affected speed of novice programmers. The final data analysis revealed that there was a large improvement when it came to speed and effort when underscores were used.

Focusing on a different aspect of code reading than styles or languages, a group of programmers took a look directly at the reading of code reviewers. Code reviewing has been used and proven to help improve already written code. A study done in 2002 on code reviewing sought to find out if a different code reading technique prevailed over others. They took a look at different metrics such as duration, gazes, glances, line identification and line reading. A lot of the software was developed in house so that the data the eye tracker recorded suited their needs for analysis. The five programmers chosen were to review six programs. Among the programmers it was commonly found that the reviewers would read code in a linear fashion the first time (referred to as a scan) reading and then go back and piece apart the code. Results showed that different code reviewers used different reading patterns involving recursive styles and focus on variables [12].

Using eye-tracking to aid in computer science education is an ever growing field of interest. In 2010 a study was done that looked at program comprehension for educators in the computer science field [13]. It broke down a programmer's thought process into different sections to help better identify learning concepts. External representation is any part of a program outside of the programmers current known knowledge. Cognitive structure is what is already known to the programmer, and assimilation process is how a programmer tackles the current programming problem. They broke down program comprehension into several different models. In the end, the study concluded that there is no single way to learn reading and comprehension of programs. Also that being open to how a program operates by reading code, is like different patches of knowledge coming together to better understand a program as a whole.

In 2013 when Busjahn and Schulte studied code reading, they found with a small group of 6 participants that there was a direct link to comprehension of algorithms and code constructs used [14]. Following this study a workshop was conducted that determined that even after a single programming course it's possible for the reading techniques of a programmer to change. Even from the beginning of a programming class, to the end, a novice programmer can read code different.

Eye-tracking and code comprehension studies are very detail oriented and can often be time consuming. A study done by Marter et al. [15] took a look at reducing study time by doing a study on the readabilty of source code using natural text. Marter's study had a unique setup in that it did not use programmers or programming code. The program was focused on using identifiers and the readability of those identifiers. They primarily did this study due to the claims that programming experiments can be quite costly and time consuming. The study set out to with one primary focus, finding if the similarity and the number of identifiers has a part in the readability of source code. They set out to do this study by having users read short snippets of natural language text with identifiers in it. After having read the text each user was asked two questions. One of these questions would relate to an identifier that was associated to the text, the other fulfilling criteria within the text. Each one of these readings was timed until a correct answer was given. The results of the study showed that while it is easy to produce a quick experiment and get data from multiple subjects, there are strong risks that come in to play. Whether a study of this type can relate directly to source code and the understanding of source code being one of them. The study itself does show a way to create quick lightweight studies, used for specific experiment types. They should not replace a controlled study confirming a hypothesis [7].

3 The Study

This study seeks to analyze and compare the reading and comprehension of natural language text and C++ source code. The main purpose of this study is to determine if natural text is read differently from source code and determine if novice programmers read differently than non-novices.

3.1 Data Collection and Tools

We used a Tobii X60 eye tracker which recorded at 60 Hz and was able to generate 60 samples of eye movements per second. The device is a non-intrusive eye tracker, meaning that a user does not need to wear it. The eye tracker is stationary on a desk between the user and monitor. The eye tracker is capable of compensating for head movements. We used a High Definition 24" monitor at 1920×1080 resolution for the study and an identical monitor for the moderator. Audio and video was recorded via a webcam. We used several different metrics via the Tobii Studio software that included fixations, durations, validity, areas of interest, gaze positions, timestamps, pupil size, validity codes, as well as start and end times for each trial.

3.2 Study Variables

The independent variables are the type of stimulus: source code or natural language text and the expertise (non-novice or novice) of the test participant. The dependent variables being the linearity metrics shown below taken from Busjahn et al. [1].

- Vertical Next Text: The percentage of forward saccades that either stay on the same line or move one line down.
- Vertical Later Text: The percentage of forward saccades that either stay on the same line or move down any number of lines.
- Horizontal Later Text: The percentage of forward saccades within a line.
- Regression Rate: The percentage of backward saccades of any length.
- Line Regression Rate: The percentage of backward saccades within a line.

In addition to the linearity metrics, we also measure *Saccade Length*, which is the average Euclidean distance between every successive pair of fixations and *Element Coverage*, measured by the fraction of words that the participant looked at. Element Coverage and Saccade Length were used to measure the differences between non-novices and novices while the first five measures were used to analyze reading styles and were mostly used to compare source code to natural language text.

3.3 Participants

For the study we used 33 students from Youngstown State University ranging from 0 to 5 years or more of programming experience. We grouped the students into two groups. We consider a non-novice to be a student who was exposed to programming in a prior course. They are typically students enrolled in a higher level course. Novices on the other hand had little or no programming background. The novices were recruited from the Introduction to Programming course and were all given extra credit. They were familiar with various concepts such as variables, looping structures, and data types all done in C++. All participants filled out a questionnaire before the study, we had a total of 18 male and 15 female participants between the ages of 18 and 27. All students were able to speak, read, and write in English, the native language of a majority of participants.

3.4 Tasks

Tasks given to the students were three different natural text paragraphs and seven small C++ programs. The different natural language text programs were all non related topics and were the same ones used by Busjahn et al. [1]. They discussed government and economy, the history of black powder, and the effects of dung beetles into a new environment. The C++ programs were all formulated with the understanding that novice programmers were part of the study. Given that information, we made the programs easy enough for a student going through the Introduction to Programming course to figure out. We included different concepts in each program including loops and nested loops, as well as input statements. While there was a range in difficulty of the programs, they were not overly difficult.

The natural text tasks were presented first (in a random order) followed by the seven source code tasks (in a random order). Subjects announced out loud when they were ready to begin the test and used the mouse to continue on to

the next slide. Subjects selected one of three numbers (in a random order) after each task and answered the question that followed using the mouse or keyboard. After each task, subjects were presented with a short questionnaire about the difficulty of the task and their confidence in their answer. After all tasks were answered, test subjects were presented with a post questionnaire that asked about difficulty, time needed, and if any problems occurred during the test.

There was no time limit given to the participants of the study. They were allowed as much time as needed to finish each Natural language and source code task. Participants were also given as much time as needed to answer the comprehension questions. Each comprehension question was one of three types; a summary of the task, a multiple choice question about the task, or a fact about the text/the output of the source code. The comprehension questions, chosen at random, helped to better understand a participants ability to read a program. The random options helped deter participants from discussing answers with each other. A replication package with all tasks and study material is available at http://sereslweb.csis.ysu.edu/HCII2017.

4 Study Results

We present the results in terms of our two research questions. We first describe how the data was processed followed by comprehension task and timing results. Threats to validity and a discussion is also presented.

Before we ran statistical tests on the data, we needed to map the fixations on source code elements, which was done using eyeCode [16]. eyeCode is able to automatically determine lines and words given an image stimulus. These lines and words form the areas of interest (AOI). In our case the image stimulus is the natural language and source code tasks. eyeCode also maps the fixations on corresponding words so we are able to determine which fixation falls on which word in natural text or source code. We use the abbreviation NT for natural language texts and SC for source code.

4.1 Comprehension Scores

We observe that overall non-novices scored higher than novices. We also observer the gap between novices and non-novices per task. The gaps are larger for difficult programs like SignChecker and PrintPattern. The other programs that fell into the medium and easy difficulty category had less of a gap between novices and non-novices. This indicates that novices had a hard time giving a correct answer for difficult problems. See Fig. 1 for the average comprehension score for novices vs. non-novices within each task.

4.2 Response Time

We recorded the time that it took each user to go through each task. The time differences showed similar results to the comprehension scores. Both novices and

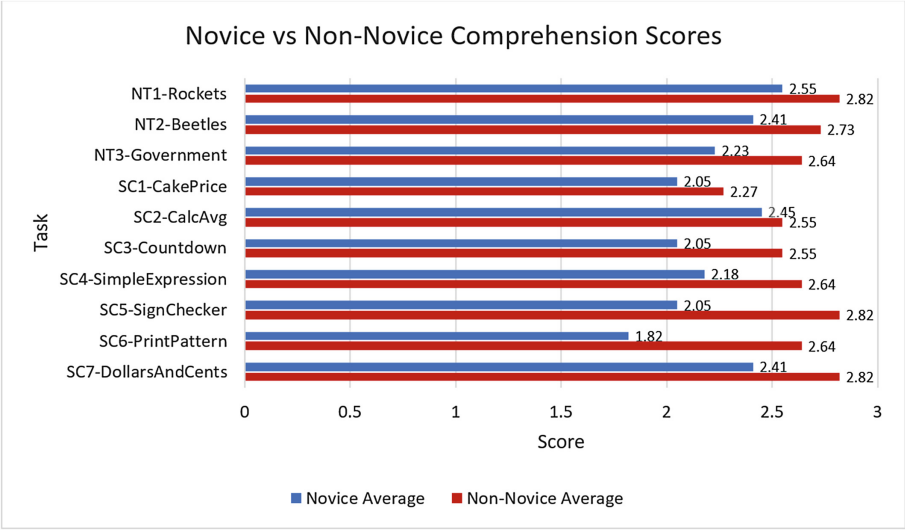

Fig. 1. Average comprehension scores for all participants.

non-novices spent the least amount of time in SC4 which was a small and simple program. Similar to comprehension scores, the gaps between novices and non-novices are much more apparent in difficult programs. Non-novices also tend to take the study more seriously compared to novices and so they spend more time on the tasks. A Mann-Whitney test reveals no significant differences in time between novices and non-novices ($p = 0.866$, $U = 126$). See Fig. 2 for completion time per task for novices vs. non-novices.

4.3 Element Coverage and Saccade Length

Novices: For element coverage, we found that 31.7% of NT in novices were looked at and 34% of elements were looked at for SC. Both saccade length ($p < 0.001$) and element coverage ($p = 0.03$) are significantly different between NT and SC indicating that these two types of stimuli are quite different in terms of their cognitive load for novices.

Non-novices: For element coverage, we found that 28.37% of NT in non-novices were looked at and 33.43% of elements were looked at for SC among non-novices. Both the saccade length ($p = 0.001$) and coverage ($p = 0.01$) were significantly different for NT and SC in the non-novices group as well.

We also notice that the saccade length is higher for NT than SC for novices. This means that in NT they jumped a few lines more than in the SC. The same can be seen in the non-novices group. However, we find that the saccade length in the non-novices SC category is lower (by 9 points) than the novices SC category. For NT, the non-novices have higher (15 points more) saccade length compared to the novices NT category.

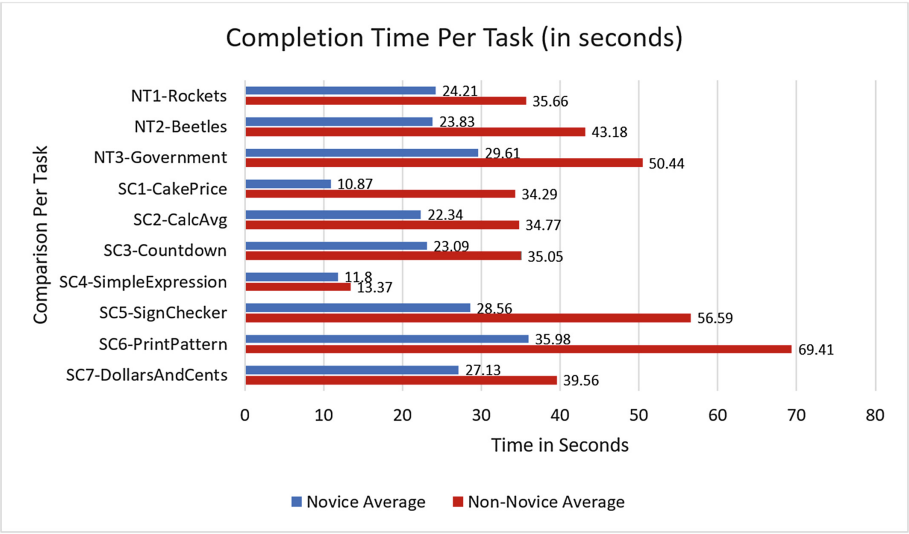

Fig. 2. Average time measurements for all participants

4.4 Research Questions Revisited

In order to answer RQ1, we report on the linearity metrics we introduced in the previous Section. We notice that the measures Vertical Next Text, Vertical Later Text, and Horizontal Later Text are all higher for NT compared to SC. These are all linearity measures indicating that NT is read more linearly than SC. The regression measures deal with non-linear reading. The line regression rates were higher in NT than SC i.e., participants went back to read a line more often in NT than SC. We ran the Wilcoxon paired test within novices and within non-novices to see if the above measures were significantly different in these groups. Table 1 shows the results of the Wilcoxon test. We see that the measures Vertical Next Text, Vertical Later Text, and Horizontal Later Text and Line Regression Rate are all significantly different between NT and SC for

Table 1. Wilcoxon signed ranked test for NT vs. SC for non-novices and novices

Linearity measure	Sum of positive signed ranks	Non-novices	Novices
		p	p
Vertical next text	253	0.001*	<0.001*
Vertical later text	253	0.001*	<0.001*
Horizontal later text	253	0.001*	<0.001*
Regression rate	136	0.01*	0.775
Line regression rate	246	0.002*	<0.001*

non-novices. The Regression Rate which involves regressions of any length is not significant in the novices groups but is significant in the non-novices group. This means that there are significant more line regressions in SC vs. NT reading in the non-novices group.

In order to answer RQ2, we ran the Mann-Whitney test on all participants. We did not find any major differences between the novice group and the non-novice group. These results are shown in Table 2. This could be due to the fact that our non-novices were still students and not what could be considered an expert in the programming industry. This leaves room to expand on this research with expert programmers compared to novices and non-novices. In contrast, Busjahn et al. found significant differences in these linearity measures (except line regression rate) for novices vs. experts in their study.

Table 2. Mann Whitney results for novices vs non-novices over all tasks

Linearity measure	U	p
Vertical next text	546	0.406
Vertical later text	539	0.461
Horizontal later text	536	0.487
Regression rate	540	0.453
Line regression rate	487	0.973

4.5 Story Order Among Novices

We now discuss the alignment of NT and SC to Story Order among novices and show the results of the Needleman-Wunch (N-W algorithm) in Table 3. Story order is basically reading the stimulus one line at a time from top to bottom (typically the way we read natural language text). The N-W algorithm is used as a string matching algorithm to determine story order. It has also been used by Cristino et al. [17] in earlier work on eye movement research. We also use it in order to compare our work with Busjahn et al. [1]. The algorithm gives a similarity score where a high score indicates that two sequences are close to each other. The difference between naïve and dynamic scores is whether repetitions through the code are counted. So if we care about how many times the person

Table 3. Needleman-Wunch results comparing the story order for NT and SC for novices

Story order	NT	SC	p
Naïve N-W score	−8.27	−21.16	<0.001
Dynamic N-W score	18.71	−4.06	<0.001
Repetitions	3.42	2.6	<0.001

read through the code, we keep repeating the string matching with the story order and the eye gaze movements to get a dynamic score, exactly the same as done in [1].

These scores are not close to one another. This means that even novices do not start with an approach that is very top down and left to right (contrary to what Busjahn [1] found for novices). The results indicate that both natural language text and source code were both read multiple times. The NT was read 3.42 times compared to SC which was read 2.6 times. The more read-throughs the higher the N-W alignment score. We found a significant difference between NT and SCs story order for novices. In comparison to the Busjahn study, the NT was read 6.35 times compared to the SC which was read 3.89 times. Busjahn did not find a significant difference between NT and SC which indicates that their novices start out with a primarily linear approach to code reading. We found a clear differences in our novice group as shown by the numbers in Table 3. We leave the same comparison of story order and eventually execution order (how the program is actually executed) for the novices and non-novices group as part of our future work.

4.6 Post Questionnaire Results

After all tasks were complete, each participant was asked if they felt that the given time that it took to go through the tasks was enough. All agreed that they had enough time. Within the group of participants, the difficulty ranking varied. 10% found it to be very easy, 34% easy, 41% average, and 16% found it difficult. The main difficulty that seven participants reported was trying to remember the given stimuli when presented with the comprehension question after the fact.

4.7 Threats to Validity

To account for different control structures in source code as well as different word lengths in natural language text, we used three NT passages and seven source code passages. The fact that we did not find any differences between novices and non-novices indicates that they are possibly at the same level in reading skills. In order to find differences in linearity, it might be necessary to study expert programmers in industry who program on a daily basis and have been working in industry for more than 10 years. One major threat to validity is the skewness that occurs in eye tracking data. The linearity metrics are directly dependent on how accurately the fixations are mapped to words or source code elements. We did not manually correct skewness for this study. We did however make sure our calibrations were done well and since our study didn't last more than 20 min, the drift was minimal. We also discarded all trials with less than 60% mapping on source code elements and found the same significant results. This indicates that we might find even more strength and effect in our findings if all the data was manually corrected. Also, on examining the scan paths, we found that most of them were close to the word that they were looking at. We strongly believe

that after correction, we should see even stronger significance. We have left this as an immediate future exercise.

5 Conclusions and Future Work

The paper presents a study that characterized linearity in eye movements between natural language text and source code in C++. This study replicates an earlier study by Busjahn et al. [1] that looked at Java code. Similar to Busjahn, our results show that both non-novices and novices read source code significantly different than natural language text, while most natural text is read left to right, top to bottom with few regressions, source code is read in a less linear manner with more regressions. Unlike Busjahn our study did not find any significant differences between novices and non-novices. As these findings are different, it calls for more studies to be conducted. It is very likely that this difference was not visible in our study since our non-novices was not comparable to experts from industry used in the Busjahn study.

As part of future work, we are currently conducting a second phase of this study with the same group of students. The purpose is to determine if the eye movements differ at the end of the semester indicating if any learning occurred. The second phase of the study is being conducted during the last week of the semester. We are also taking a look at fixations and durations on specific areas (beacons) in the code provided and want to determine if the difficulty of a task makes a difference on how both novices and experts read the code. Beacons are places in the code that non-novices tend to focus on as one chunk of data. More studies and replications need to be done to add to the body of knowledge and thereby advance the state of the art of eye movement research in programming.

References

1. 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 2015, Piscataway, NJ, USA, pp. 255–265. IEEE Press (2015). http://dl.acm.org/citation.cfm?id=2820282.2820320
2. Rayner, K., Chace, K.H., Slattery, T.J., Ashby, J.: Eye movements as reflections of comprehension processes in reading. Sci. Stud. Read. **10**, 241–255 (2006)
3. Brooks, R.: Towards a theory of the comprehension of computer programs. Int. J. Man-Mach. Stud. **18**(6), 543–554 (1983). http://www.sciencedirect.com/science/article/pii/S0020737383800315
4. Raina, S., Bernard, L., Taylor, B., Kaza, S.: Using eye-tracking to investigate content skipping: a study on learning modules in cybersecurity. In: 2016 IEEE Conference on Intelligence and Security Informatics (ISI), pp. 261–266, September 2016
5. Beelders, T., du Plessis, J.-P.: The influence of syntax highlighting on scanning and reading behaviour for source code. In: Proceedings of the Annual Conference of the South African Institute of Computer Scientists and Information Technologists, SAICSIT 2016, pp. 5:1–5:10. ACM, New York (2016). http://doi.acm.org/10.1145/2987491.2987536

6. Sarkar, A.: The impact of syntax colouring on program comprehension. In: PPIG, July 2015
7. Crosby, M.E.: Natural versus computer languages: a reading comparison. Ph.D. dissertation, University of Hawaii at Manoa (1986)
8. Fan, Q.: The effects of beacons, comments, and tasks on program comprehension process in software maintenance. Ph.D. dissertation, Catonsville, MD, USA (2010)
9. Turner, R., Falcone, M., Sharif, B., Lazar, A.: An eye-tracking study assessing the comprehension of C++ and Python source code. In: Proceedings of the Symposium on Eye Tracking Research and Applications, ETRA 2014, pp. 231–234. ACM, New York (2014). http://doi.acm.org/10.1145/2578153.2578218
10. Sharif, B., Maletic, J.I.: An eye tracking study on camelcase and under_score identifier styles. In: Proceedings of the 2010 IEEE 18th International Conference on Program Comprehension, ICPC 2010, Washington, DC, USA, pp. 196–205. IEEE Computer Society (2010). http://dx.doi.org/10.1109/ICPC.2010.41
11. Binkley, D., Davis, M., Lawrie, D., Maletic, J., Morrell, C., Sharif, B.: The impact of identifier style on effort and comprehension. Empir. Softw. Eng. J. (Invit. Submiss.) **18**(2), 219–276 (2013)
12. Uwano, H., Nakamura, M., Monden, A., Matsumoto, K.-I.: Analyzing individual performance of source code review using reviewers' eye movement. In: Proceedings of the 2006 Symposium on Eye Tracking Research Applications, ETRA 2006, pp. 133–140. ACM, New York (2006). http://doi.acm.org/10.1145/1117309.1117357
13. Schulte, C., Clear, T., Taherkhani, A., Busjahn, T., Paterson, J.H.: An introduction to program comprehension for computer science educators. In: Proceedings of the 2010 ITiCSE Working Group Reports, ITiCSE-WGR 2010, pp. 65–86. ACM, New York (2010). http://doi.acm.org/10.1145/1971681.1971687
14. Busjahn, T., Schulte, C.: The use of code reading in teaching programming. In: Proceedings of the 13th Koli Calling International Conference on Computing Education Research, Koli Calling 2013, pp. 3–11. ACM, New York (2013). http://doi.acm.org/10.1145/2526968.2526969
15. Marter, T., Babucke, P., Lembken, P., Hanenberg, S.: Lightweight programming experiments without programmers and programs: an example study on the effect of similarity and number of object identifiers on the readability of source code using natural texts. In: Proceedings of the 2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2016, pp. 1–14. ACM, New York (2016). http://doi.acm.org/10.1145/2986012.2986020
16. Hansen, M.: GitHub - synesthesiam/eyecode-tools: a collection of tools for analyzing data from my eyeCode experiment. https://github.com/synesthesiam/eyecode-tools
17. Cristino, F., Mathôt, S., Theeuwes, J., Gilchrist, I.D.: ScanMatch: a novel method for comparing fixation sequences. Behav. Res. Methods **42**(3), 692–700 (2010). http://www.springerlink.com/index/10.3758/BRM.42.3.692