Influencing Attention In Code Reading: An Eye-Tracking Study

Alan T. McCabe
Lund University
alan.mccabe@cs.lth.se

Diederick C. Niehorster
Lund University
diederick_c.niehorster@humlab.lu.se

Emma Söderberg
Lund University
emma.soderberg@cs.lth.se

Abstract

When interacting with other humans, we attempt to develop a shared understanding using various means. One such method is through our eyes: when someone is looking at something, we understand that their attention is focused on that object. In this work, we present the results of an eye-tracking study built upon the Progger tool, in which we used additional code highlighting in an attempt to influence the gaze behaviour of a human programmer, thereby focusing their attention. We found that though it is possible to draw attention towards areas of particular interest to the compiler, this has no apparent effect upon performance when confronted with a bug-finding code comprehension task. We conclude that although this strategy may be of use in the future when attempting to humanise the process of programming, further research is required to establish the efficacy of such interventions.

1. Introduction

In a series of recent research papers (McCabe, Söderberg, & Church, 2021; Church, Söderberg, & McCabe, 2021; McCabe, Söderberg, & Church, 2022), we have explored the idea of viewing the activity of programming as a "conversation" between two participants, namely the developer and their development environment. This "conversational lens" (Church et al., 2021), used as a "tool for thinking with"[1], led to the development of a prototype tool, named Progger (McCabe et al., 2021).

With Progger, we have explored the consequences of making visible the sequence of actions taken by a Java compiler in the lead up to a compiler error. This was an attempt to give the user a means of delving into the "thought process" of the compiler: much as in a human-to-human conversation a misunderstanding could be resolved by asking one participant to clearly and thoroughly explain what they are thinking about. This approach for humanising the interaction between developer and development environment centred around the usage of a linguistic-based strategy. Sections of code were highlighted and explicitly linked with descriptive text in the sidebar, which served as a summary of the computations made using attributes assigned to the code during compilation.

However, in our initial framing paper (Church et al., 2021), we also noted the use of so-called *side-channels* in human interactions. These side-channels are non-verbal cues which can greatly alter the meaning of speech, and can take on a variety of forms: body language, tone of voice, facial expressions, where a person is looking, and so forth.

Having built the Progger prototype, we conducted a pilot study (McCabe et al., 2022) with the aim of validating the design choices made in the development of the tool. Two interesting conclusions were drawn from this study: the additional descriptive text – the linguisic component – was found to be too esoteric for the target demographic of relative programming novices; and that the highlighting – the non-linguistic component – was of significant interest, drawing praise when it worked well and ire when it did not. In other words, Progger users were less interested in the somewhat abstract and complex thought process of the compiler, and much more interested in simply *where the compiler was looking.*

[1]As coined at the industry panel during PPIG 2016 in Cambridge by Steven Clarke.

Being interested in where someone or something is looking is an innate characteristic in humans. The following of another person's gaze is a behaviour that has been found to start developing at a very young age (Frischen, Bayliss, & Tipper, 2007), and is of great significance in the development of social cognition. When two humans align their gaze, it signifies a "joint attention", indicating that both participants in the interaction are focused on the same thing. By aligning the "gaze" of a compiler with that of the developer, it may be possible to encourage the development of joint attention, with a focus on the most relevant areas of code. With this new insight an updated version of Progger was produced (Progger 2.0), stripping out extraneous text information and focusing solely on highlighting as a means of conveying information.

In this paper, we present the results of an eye-tracking study conducted using Progger 2.0, with the aim of understanding how showing where a program analysis tool is "looking" can influence the gaze and code reading behaviour of a human partner in the programming conversation, and help foster joint attention.

2. Background

Most visual tasks performed by humans are bottlenecked by the foveated nature of their visual system. Because the area of highest visual acuity of the human retina only spans a few degrees, visual tasks requiring resolving fine spatial detail often also require eye movements. As such, the study of eye movements has been fruitfully used to study the moment-to-moment unfolding of cognitive processes such as reading (Rayner, 1998) and visual search (Hooge & Erkelens, 1996; Rao, Zelinsky, Hayhoe, & Ballard, 2002; Niehorster, Cornelissen, Holmqvist, & Hooge, 2019).

When viewing static visual scenes, such as pages of text or displays of programming code, humans predominantly exhibit two types of eye movements: fixations (periods during which the eye is still so that a relatively constant area of the visual scene is projected to the fovea to allow for fine visual processing) and saccades (rapid eye movements to bring gaze to the next area of interest in the scene) (Hessels, Niehorster, Nyström, Andersson, & Hooge, 2018). Eye trackers, devices that measure what a person looks at, are used to study such gaze behaviors (Holmqvist et al., 2011). In this study, like many before us (Sharafi, Soh, & Guéhéneuc, 2015; Obaidellah, Al Haek, & Cheng, 2018; Kuang, Söderberg, Niehorster, & Höst, 2023), we make use of eye trackers to study how participants read programming code.

The act of program comprehension, when a person reads and attempts to understand unfamiliar code, has been the focus of multiple studies stretching across decades of research (Crosby & Stelovsky, 1990; Feitelson, 2019). For instance, a recent study (Busjahn et al., 2015) found that novices, as opposed to experts, have a tendency to read through code in a linear fashion, like how we would read a natural language text, and exhibited short average saccade length due to their eyes moving through the code from one line to the next. By comparison, experts were found to have a greater average saccade length and lower element coverage of the code, meaning that they focused on fewer lines of code and made larger jumps between lines as they read the code in a non-linear fashion.

By highlighting multiple non-consecutive lines of code, we hypothesised that analysis tools such as Progger 2.0 may have an affect on this phenomenon. Specifically, we speculate that by visualising the non-linear compiler gaze, we may encourage the user to spend a a greater amount of time dwelling on highlighted lines, leading to the adoption of a similar gaze pattern and a more closely aligned focus of attention.

3. Method

With the aim of increasing our understanding of how showing the "attention" of a program analysis tool can influence human gaze and performance, we conducted an eye-tracking experiment. We broke down our objective into the following research questions:

RQ$_1$ Does the addition of compiler heatmap highlighting affect bug finding performance?

RQ$_2$ Does the addition of compiler heatmap highlighting affect gaze behavior when reading code?

3.1. Participants

In order to maintain consistency with the previous Progger study, as well as to reduce the number of independent variables, we decided to target novice programmers for the experiment. The participants were recruited from the pool of undergraduate computer science students at Lund University. An advertisement was initially made to students taking a course on agile software development, however the scope of the recruitment was later extended to include general advertising to the student cohort using Facebook groups. The only requirement for participation was the completion of at least one programming related class, and a total number of 15 students ultimately took part in the experiment. Of these participants, all were studying at an undergraduate level and none had any industrial experience outside of a summer internship. Participants were compensated for their participation in the form of a gift card for a cinema chain.

3.2. Stimuli

A set of eight stimuli was presented using Tobii Pro Lab, consisting of screenshots of small (8 to 17 lines) Java programs rendered within Progger, each containing at least one compiler error. A single error was identified with a red outline in the code, and the related error-message displayed in the side-bar.

Each stimulus contained either only a simple code snippet without highlighting and with only the error location indicated, or additionally contained highlighting provided by Progger indicating different lines of code which the compiler considered during computation of the error. An example stimulus showing both the non-highlighted and highlighted conditions is shown in Figure 1. Two sets of eight stimuli were created, sets A and B, each of which contained four non-highlighted code snippets and four with additional highlighting. The non-highlighted/highlighted screenshots were swapped between sets A and B. Of the 15 participants, 7 were shown stimulus set A and 8 were shown stimulus set B. The stimuli were presented in random order for each participant. It should be noted that extra whitespace was added between lines of code in order to achieve greater accuracy when determining exactly which line of code a fixation falls on.

3.3. Apparatus and Experimental Procedure

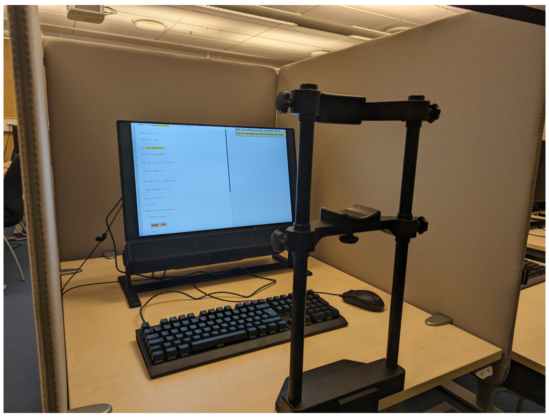
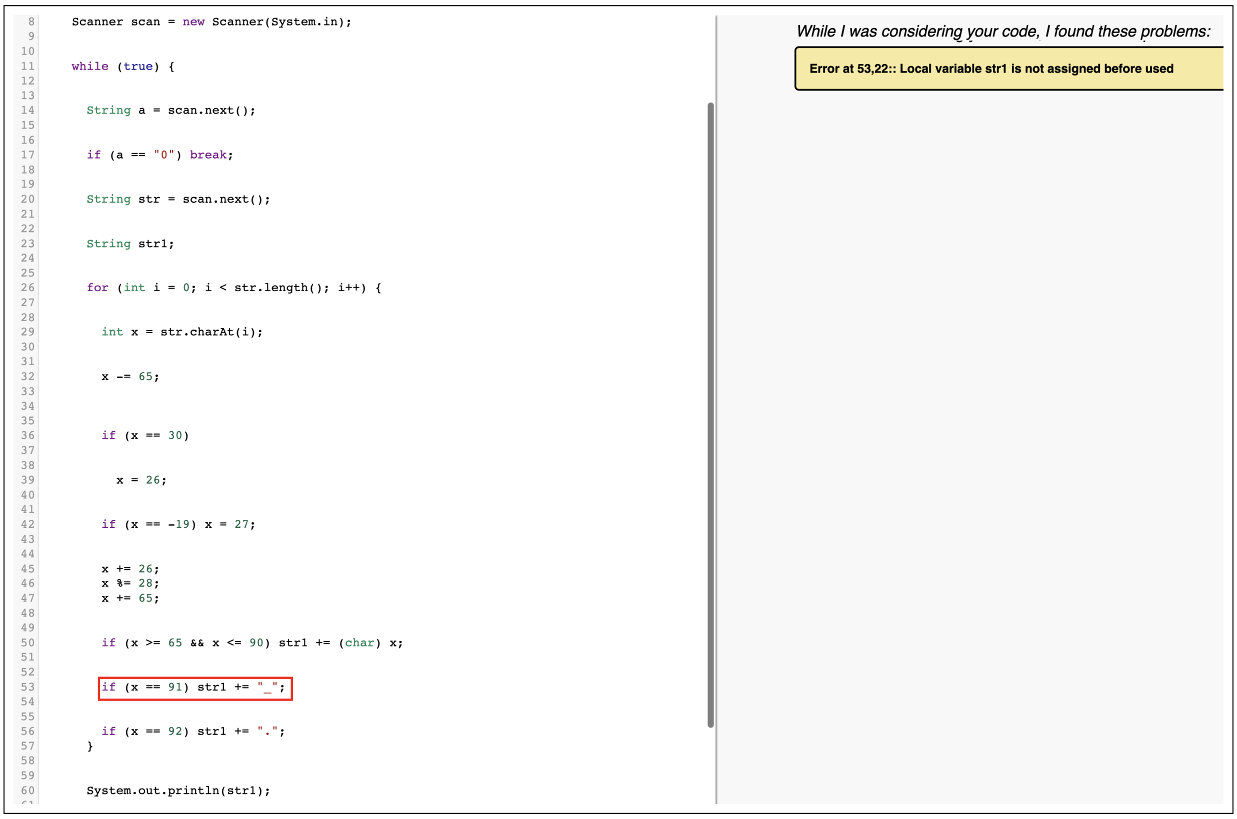


Figure 2 – The experimental setup. The apparatus is contained within a booth, with the eye-tracker visible below the screen and a chin- and forehead rest in the foreground.

The experiment was conducted on-site at the Lund University Humanities Lab, using a Tobii Pro Spectrum that recorded gaze at 600 Hz. Each participant was seated at a booth which constricted their peripheral view, and viewed the stimuli from a viewing distance of approximately 63 cm on a 52.8 x 29.7 cm (47° x 27°) computer screen (EIZO FlexScan EV2451, resolution 1920 x 1080 pixels) attached to the eye tracker while their head was placed on a chin- and forehead rest that was attached to the desk. The headrest and desk height were adjusted until the participant was comfortable. A five-point calibration procedure was executed followed by a four-point validation (mean accuracy 0.501 deg). An example of the experimental setup can be seen in Figure 2.

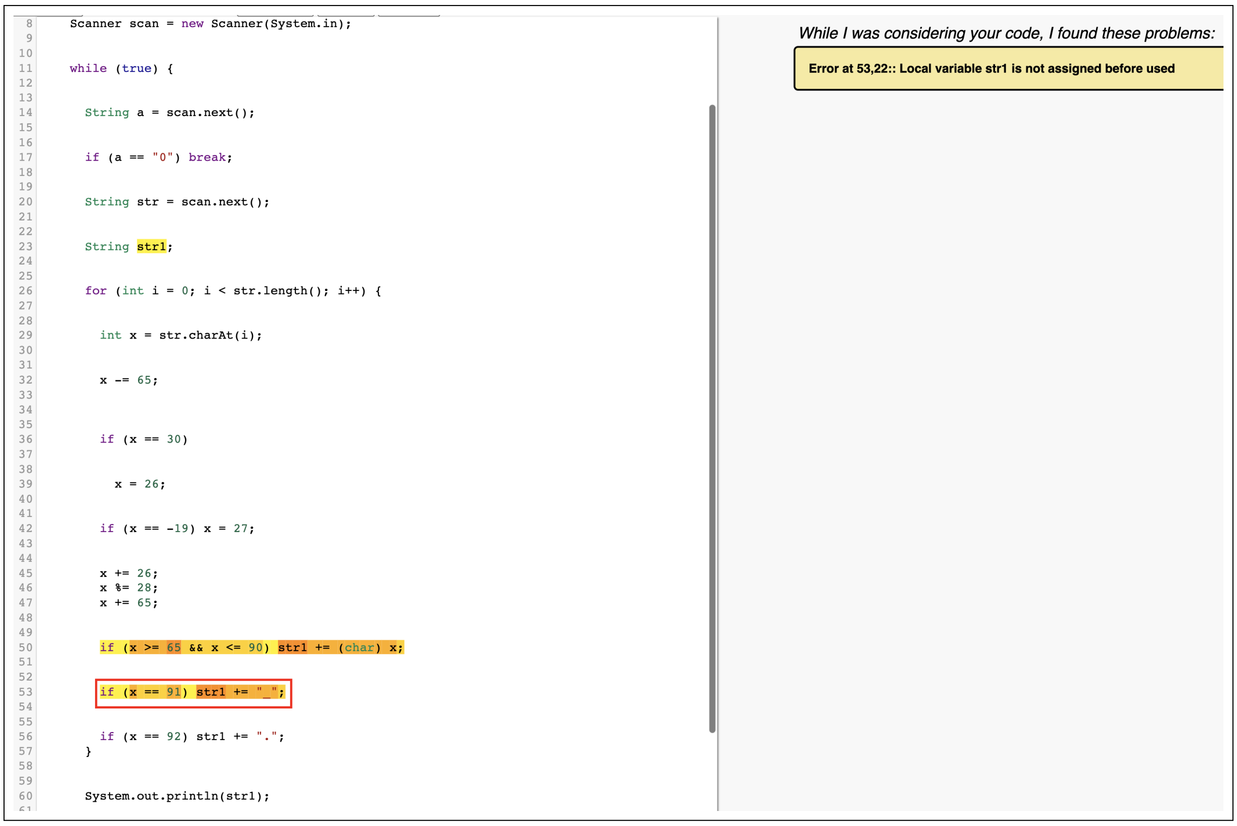
The eight code stimuli were presented during eight separate trials using Tobii Pro Lab. Each trial, the participants were asked to attempt to comprehend the code and determine both why the error had occurred, and how it might be fixed. Once they felt they had a good understanding of these questions, they were instructed to press any key in order to advance to a text-input screen where they were asked to summarise in their own thoughts why they believed the error had occurred, and how to resolve



```
 8    Scanner scan = new Scanner(System.in);
 9
10
11    while (true) {
12
13
14      String a = scan.next();
15
16
17      if (a == "0") break;
18
19
20      String str = scan.next();
21
22
23      String str1;
24
25
26      for (int i = 0; i < str.length(); i++) {
27
28
29        int x = str.charAt(i);
30
31
32        x -= 65;
33
34
35
36        if (x == 30)
37
38
39          x = 26;
40
41
42        if (x == -19) x = 27;
43
44
45        x += 26;
46        x %= 28;
47        x += 65;
48
49
50        if (x >= 65 && x <= 90) str1 += (char) x;
51
52
53        if (x == 91) str1 += "_";
54
55
56        if (x == 92) str1 += ".";
57      }
58
59
60      System.out.println(str1);
```

While I was considering your code, I found these problems:

Error at 53,22:: Local variable str1 is not assigned before used



```
 8    Scanner scan = new Scanner(System.in);
 9
10
11    while (true) {
12
13
14      String a = scan.next();
15
16
17      if (a == "0") break;
18
19
20      String str = scan.next();
21
22
23      String str1;
24
25
26      for (int i = 0; i < str.length(); i++) {
27
28
29        int x = str.charAt(i);
30
31
32        x -= 65;
33
34
35
36        if (x == 30)
37
38
39          x = 26;
40
41
42        if (x == -19) x = 27;
43
44
45        x += 26;
46        x %= 28;
47        x += 65;
48
49
50        if (x >= 65 && x <= 90) str1 += (char) x;
51
52
53        if (x == 91) str1 += "_";
54
55
56        if (x == 92) str1 += ".";
57      }
58
59
60      System.out.println(str1);
```

While I was considering your code, I found these problems:

Error at 53,22:: Local variable str1 is not assigned before used

Figure 1 – An example of non-highlighted (top) versus highlighted (bottom) versions of a stimulus. In the stimulus, a compiler error has been thrown stating that a variable has not been assigned before use due to the initial assignment being dependent upon the results of a scanned input, which is indeterminate at compile time.

it. Once completed, they were able to move on to the next stimulus by pressing a key combination. In order to cater to the target demographic of novices, no advanced language constructs or external libraries were used within the code samples. The whole experiment took approximately 15 to 50 minutes to complete, depending on participant.

3.4. Data Analysis

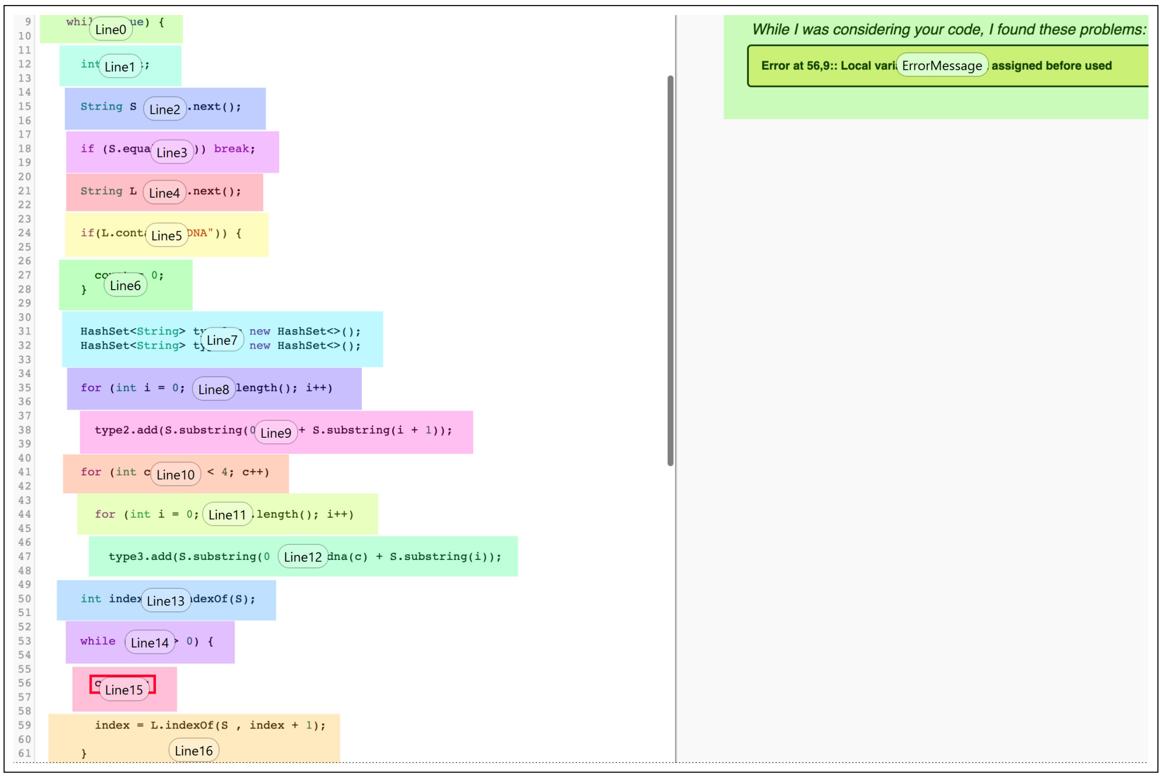
The experiment contained one independent variable: the state of the highlighting, either turned on or off for a given stimulus. Dependent variables included areas-of-interest (AOIs) analyses. To perform these analyses, for each of the experiment stimuli, areas-of-interest (AOIs) were created for each line of code and the error message. In the event of inconsequential lines of code (for example, a trailing "}" to close a block), these lines were grouped with the AOI defined for the preceding line. An example of a stimulus with AOIs defined is provided in Figure 3.

We used Tobii Pro Lab software (Tobii AB, 2023) to classify gaze into fixations using the default fixation filter with default settings, and then to annotate for each fixation whether it was in an AOI or not.

The output from Pro Lab was analyzed using a custom Java program. The program first reads in the data for a single stimulus and from this constructs a "timeline" of AOI hits, corresponding to a list of consecutive AOI fixations. Some of the analyses listed below were performed using this timeline.

In total, six dependent variables are analyzed:

- **Time to completion:** how long a participant takes to solve a task, in seconds.
- **Correctness:** Whether the participant successfully solved a task, with a binary grading of either correct or incorrect. To compute correctness values, the documents where participants recorded their proposed solutions for each problem were analysed by the first author. Each solution was marked using a traffic light system, with red signifying an incorrect solution, yellow an incomplete solution or one where the participant demonstrated understanding but was unable to solve the issue, and green indicating a complete and correct solution. The third author then checked the proposed solution grading for correctness, and any disagreement was discussed until there was a consensus.. For statistical analysis purposes, the possible grades were then recoded to a binary format by coding incorrect or incomplete solutions as unsuccessful task solutions.
- **Hit-rate:** The percentage of AOI fixations that fell on lines that *would have been highlighted by Progger* in a given stimulus. This means that the hit-rate was calculated for the same lines regardless of whether the highlighted or non-highlighted version of a stimulus was presented. For example, in the stimulus shown in Figure 1 on line 23, the variable name is highlighted by Progger in the declaration statement: `String str1`. This AOI was therefore marked as an area of interest for both versions of the stimulus (regardless of the stimulus treatment).
- **Dwell duration:** The average amount of time the participant looked at an area of interest before moving their gaze to another part of the screen, in milliseconds. Like for hit-rate, for the dwell duration calculation only highlighted lines or lines that would have been highlighted by Progger are considered. Dwell times were computed by summing together the duration of consecutive fixations that fell in the same AOI.
- **Saccade length:** The average distance between gaze fixations while reading the code, in degrees. Specifically, the saccade classification output provided by Tobii Pro Lab was used, and the distance between the two fixations that are adjacent to each saccade computed.
- **Linearity:** The percentage of gaze movements corresponding to a linear forward change. Specifically, we counted consecutive AOI fixation pairs that represented a *single-step forward-progression* in corresponding lines. For example, consecutive AOI fixations of lines 5 and 6 would constitute a linear change and thus contribute to this metric. Examples of pairs that would not contribute are fixations on lines 5 and 7 (multiple-step) or lines 6 and 5 (backwards-progression).



```
    whi      ue) {

      int      ;

      String S       .next();

      if (S.equa       )) break;

      String L       .next();

      if(L.cont       DNA")) {

        co      0;
      }

      HashSet<String> ty        new HashSet<>();
      HashSet<String> ty        new HashSet<>();

      for (int i = 0;       length(); i++)

        type2.add(S.substring(0       + S.substring(i + 1));

      for (int c        < 4; c++)

        for (int i = 0;        .length(); i++)

          type3.add(S.substring(0        dna(c) + S.substring(i));

      int index        dexOf(S);

      while        0) {

        c      

        index = L.indexOf(S , index + 1);
      }
```

While I was considering your code, I found these problems:

Error at 56,9:: Local vari   assigned before used

Figure 3 – The areas-of-interest for a stimulus, as defined in Tobii Pro Lab.

The data were analyzed in Jamovi version 2.3 (the jamovi project, 2022; R Core Team, 2021) with the GAMLj module (Galluci, M., 2019). Linear mixed effect modelling was performed with highlighting as a fixed effect and participant and stimulus as random effects for five out of the six dependent variables. Correctness was instead analyzed using a generalized mixed model to perform a logistic regression using a logit link function. As for the other analyses, highlighting was specified as a fixed effect and participant and stimulus as random effects. Data plots were also created using Jamovi and show data for individual participants along with the mean across participants. Error bars denote standard errors of the mean.

3.5. Threats to validity

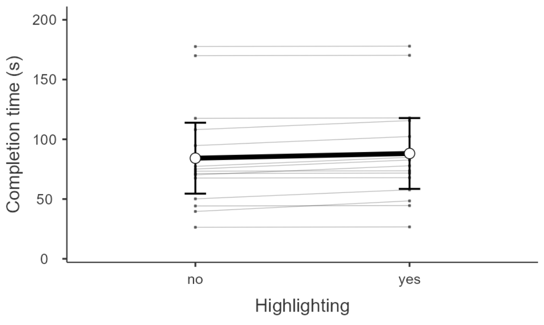
The primary threats to the validity of this study are the low **sample sizes**. To ensure the greatest accuracy of the eye-tracking equipment, we found it necessary to invite participants on-site to the Lund University Humanities Lab, which may have been a deterrent for some people as they were impelled to go out of their way to contribute. We attempted to offset this by offering an incentive, however the total number of participants was ultimately quite low at 15. To improve the internal validity of the study, it would be desirable to recruit a larger pool of participants.

Similarly, the relatively low **stimuli number** could have had an effect on the data. This led to several participants completing the tasks very quickly, however some participants also used the entire 1 hour allotted to them. For future studies, it may be desirable to increase the number of stimuli, although with the caveat that this could impact participation numbers.

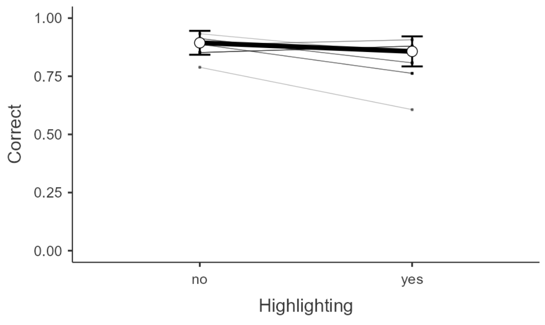
Sample selection may also have affected the internal validity of the study, as only novices were recruited for participation. This led to some difficulties for specific tasks, and may have impacted the validity of the correctness metric in particular.

4. Results

Here we present the results broken down into performance and gaze metrics on AOIs.



(a) Mean **completion times** per participant of non-highlighted and highlighted stimuli.



(b) Mean **correctness** score per participant for non-highlighted and highlighted stimuli.

Figure 4 – Charts detailing the completion time and correctness metrics.

4.1. Time to complete tasks

In order to answer RQ_1, the time taken to solve each trial was analysed. As can be seen in Figure 4 (a), highlighting had no significant effect on the completion time ($F(1,96.5) = 0.258, p = 0.613$).

4.2. Correctness

Across all participants, 101 of 120 trials (84%) yielded a correct solution. Statistical analysis was again performed to investigate whether the addition of highlighting had an effect, as can be seen in Figure 4 (b). The influence of highlighting on mean correctness was found to be insignificant ($\chi^2(1.00) = 0.401, p = 0.527$).

4.3. Highlighted line hit-rate

The mean hit-rates across all participants were then calculated and analysed, as seen in Figure 5 (a). Hit-rate was larger for highlighted stimuli than non-highlighted stimuli ($F(1,95.8) = 10.6, p = 0.002$), indicating that highlighted lines of code are looked at more.

4.4. Dwell duration

The results of dwell time analysis can be seen in Figure 5 (b), and, consistent with the hit rate metric, exhibit an increase of 80 ms in average dwell duration between non-highlighted and highlighted versions of the stimuli ($F(1,96.2) = 4.53, p = 0.036$). This indicates that participants spent more time looking at lines of code that the compiler considered to be of interest when the highlighted versions of the stimuli were presented to them.

4.5. Saccade length

No significant differences in saccade length were found between highlighted and non-highlighted stimuli ($F(1,97.7) = 0.184, p = 0.669$), see Figure 5 (c).

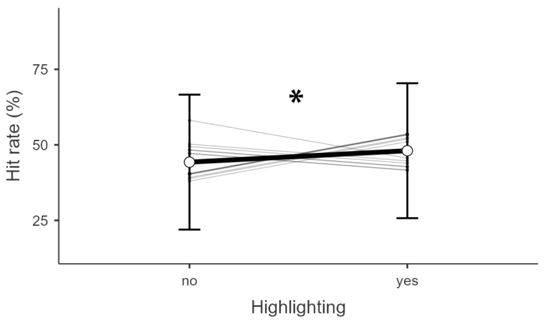
4.6. Reading linearity

As can be seen from Figure 5 (d), there was no significant difference in linearity scores between the non-highlighted and highlighted conditions ($F(1,96.3) = 0.490, p = 0.486$). Together with the saccade length metric, this finding suggest that the way in which participants scanned the stimuli did not differ depending on availability of highlighting.

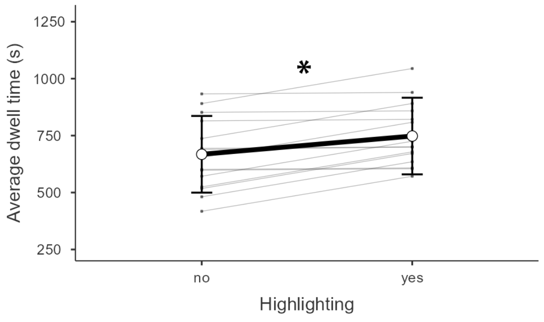
5. Discussion

Of the six dependent variables which were analysed, we used the first two (completion time and correctness) to answer RQ_1 (compiler heatmap highlighting effect on bug finding performance). We found that the addition of compiler heatmap highlighting did not have an effect on both the completion time ($p = 0.613$) and the correctness ($p = 0.527$).

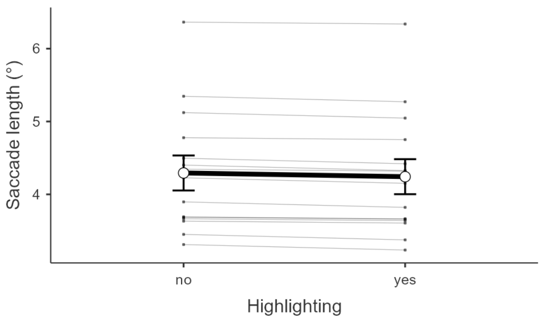
We used the analysis of dependent variables three to six (hit-rate, dwell duration, saccade length, linearity) to answer RQ_2 (compiler heatmap highlighting effect on gaze behaviour when reading code).



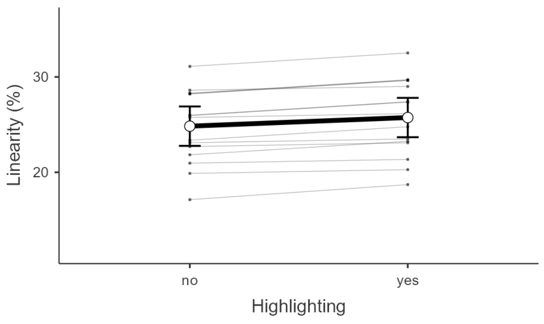
(a) Mean AOI **hit-rate** per participant of non-highlighted and highlighted stimuli.



(b) Mean AOI **dwell time** per participant of non-highlighted and highlighted stimuli.



(c) Mean **saccade length** per participant of non-highlighted and highlighted stimuli.



(d) Mean **linearity score** per participant of non-highlighted and highlighted stimuli.

Figure 5 – Charts detailing the collected eye-tracking data. An asterisk (*) indicates a statistically significant effect of highlighting change.

Of these variables, the closely related metrics of saccade length and linearity displayed no significant difference between the two highlighting conditions. This may suggest that, in the act of program comprehension, novice programmers exhibit similar patterns of gaze movements when reading unfamiliar code, regardless of additional compiler heatmap highlighting.

By contrast, the metrics relating to where (AOI hits) and for how long (dwell duration) the participants looked show a marked difference. From the data, we see that by adding highlighting to a line, novice programmers tend to look more often, and for longer periods of time, at that line. Despite this, as related in the discussion on RQ_1, this difference in gaze behaviour has no significant effect on either time to complete a task or correctness of the solution when presented with a code comprehension problem.

The effect of higher hit-rate and dwell duration ultimately warrants further investigation, as there were several confounding factors which may have influenced the results. It may be that the experimental setup being focused on comprehension was not conducive to the highlighting aiding the participants: a desire to understand the code accurately may have led participants to reading in a very methodical way. We believe that in the future it would be worth studying the effect of compiler heatmap highlighting in different contexts, such as when examining control flow, or when investigating a bug in an already familiar code base. The relatively low number of participants may also have affected our ability to detect effects of highlighting, and it may be worth attempting to re-run the study with a larger number of subjects.

The low level of experience across all participants may also have fed into the results due to their lack of familiarity with certain features of the Java language. For instance, the stimulus which received most incorrect solutions (6 out of 15 responses, or 40%) was centred around the use of the `final` keyword. Some participants were unsure how this would affect the mutability of the relevant variable, and thus offered incorrect solutions.

Considering these potentially confounding factors, it may be of worth to run a follow-up study with steps taken to mitigate their effects. A larger pool of participants, selected from both novices and experts, and a lesser focus on code comprehension within the study may lead to further interesting findings.

Despite these factors, the experiment led to interesting findings when considering the concept of joint attention. The method in which the heatmap highlighting is computed (as described in past papers, (McCabe et al., 2021, 2022) is based on the lines which are considered by the compiler in the code analysis resulting in a found error. When no highlighting exists on a line, the compiler did not consider it to be of importance. In contrast, the darkness of the highlighting on a given line or term is related to the number of times the compiler "gaze" passed over this area of code. The fact that participants looked at highlighted sections more often, and for longer periods, shows that this visualisation of compiler attention had a notable effect on user attention.

One of the main motivations in developing the conversational lens for analysing interactions with a programming environment was to draw upon human characteristics to make the process more natural. We believe that in moving towards more natural, instinctive methods of communication, many of the abstractions of human-computer interaction, for example hiding a complex compilation process behind a simple error message, can be made more clear. In this study we have shown that, although no effects were found on performance in this experimental setup, it is indeed possible to use interaction design to encourage the very human phenomenon of joint attention with a computer.

6. Acknowledgements

The authors gratefully acknowledge Lund University Humanities Lab for providing the facilities for the experiment, and would like to thank Prof. Martin Höst for early feedback on the experiment design. This work has been partially supported by the Swedish Foundation for Strategic Research (grant no. FFL18-0231), the Swedish Research Council (grant no. 2019-05658), ELLIIT - the Swedish Strategic Research Area in IT and Mobile Communications, and the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

7. References

Busjahn, T., Bednarik, R., Begel, A., Crosby, M., Paterson, J. H., Schulte, C., ... Tamm, S. (2015). Eye movements in code reading: Relaxing the linear order. In *2015 ieee 23rd international conference on program comprehension* (p. 255-265). doi: 10.1109/ICPC.2015.36

Church, L., Söderberg, E., & McCabe, A. (2021). Breaking down and making up-a lens for conversing with compilers. In *Psychology of programming interest group annual workshop (ppig).*

Crosby, M., & Stelovsky, J. (1990). How do we read algorithms? a case study. *Computer*, *23*(1), 25-35. doi: 10.1109/2.48797

Feitelson, D. G. (2019). Eye tracking and program comprehension. In *2019 ieee/acm 6th international workshop on eye movements in programming (emip)* (p. 1-1). doi: 10.1109/EMIP.2019.00008

Frischen, A., Bayliss, A. P., & Tipper, S. P. (2007). Gaze cueing of attention: visual attention, social cognition, and individual differences. *Psychological bulletin*, *133*(4), 694-724.

Galluci, M. (2019). *Gamlj: General analyses for linear models.* Retrieved from `https://gamlj.github.io/`

Hessels, R. S., Niehorster, D. C., Nyström, M., Andersson, R., & Hooge, I. T. C. (2018). Is the eye-movement field confused about fixations and saccades? a survey among 124 researchers. *Royal Society Open Science*, *5*. doi: http://doi.org/10.1098/rsos.180502

Hooge, I. T. C., & Erkelens, C. J. (1996). Control of fixation duration in a simple search task. *Perception & Psychophysics*, *58*(7), 969-976. doi: 10.3758/BF03206825

Kuang, P., Söderberg, E., Niehorster, D. C., & Höst, M. (2023). Towards gaxe-assisted developer tools. In *International conference on software engineering: New ideas and emerging results (icse-nier).*

McCabe, A. T., Söderberg, E., & Church, L. (2021). Progger: Programming by errors (work in progress). In *Psychology of programming interest group annual workshop (ppig).*

McCabe, A. T., Söderberg, E., & Church, L. (2022). Visual cues in compiler conversations. In *Psychology of programming interest group annual workshop (ppig).*

Niehorster, D. C., Cornelissen, T., Holmqvist, K., & Hooge, I. (2019). Searching with and against each other: Spatiotemporal coordination of visual search behavior in collaborative and competitive settings. *Atten Percept Psychophys*, *81*, 666-683. doi: https://doi.org/10.3758/s13414-018-01640-0

Obaidellah, U., Al Haek, M., & Cheng, P. C.-H. (2018). A survey on the usage of eye-tracking in computer programming. *ACM Comput. Surv.*, *51*(1).

R Core Team. (2021). *R: A language and environment for statistical computing.* Retrieved from `https://cran.r-project.org`

Rao, R. P., Zelinsky, G. J., Hayhoe, M. M., & Ballard, D. H. (2002). Eye movements in iconic visual search. *Vision Research*, *42*(11), 1447-1463. doi: https://doi.org/10.1016/S0042-6989(02)00040-8

Rayner, K. (1998). Eye movements in reading and information processing: 20 years of research. *Psychological Bulletin*, *124*(3), 372-422. doi: 10.1037/0033-2909.124.3.372

Sharafi, Z., Soh, Z., & Guéhéneuc, Y.-G. (2015). A systematic literature review on the usage of eye-tracking in software engineering. *Information and Software Technology*, *67*, 79-107.

the jamovi project. (2022). *jamovi.* Retrieved from `https://www.jamovi.org`

Tobii AB. (2023). *Tobii pro lab.* Retrieved from `http://www.tobii.com/`