An eye tracking study assessing the impact of background styling in code editors on novice programmers' code understanding

Kang-il Park
kangil.park@huskers.unl.edu
University of Nebraska-Lincoln
Lincoln, Nebraska, USA

Pierre Weill-Tessier
pierre.weill-tessier@kcl.ac.uk
King's College London
London, UK

Neil C. C. Brown
neil.c.c.brown@kcl.ac.uk
King's College London
London, UK

Bonita Sharif
bsharif@unl.edu
University of Nebraska-Lincoln
Lincoln, Nebraska, USA

Nikolaj Jensen
nikolaj.jensen@kcl.ac.uk
King's College London
London, UK

Michael Kölling
michael.kolling@kcl.ac.uk
King's College London
London, UK

ABSTRACT

Background and Context: The designers of programming editors aimed at learners have long experimented with different styles of code presentation. The idea of *syntax* highlighting – coloring specific words – is very old. More recently, some editors (including text-, frame- and block-based editors) have added forms of *scope* highlighting – colored rectangles to represent programming scope – but there have been few studies to investigate whether this is beneficial for novices when reading and understanding program code.
Objectives: We investigated whether the use of scope highlighting during code comprehension tasks (a) has an impact on where users focus their gaze, (b) affects the accuracy of user's responses to tasks, and/or (c) affects the speed of user's correct responses to the tasks.
Method: We conducted a controlled trial with a crossover design, where all participants completed the same twelve code comprehension tasks, each performed in one of three scope highlighting conditions. The conditions were (a) "plain" Java with no scope highlighting, (b) Java with BlueJ's scope highlighting, and (c) Stride, a frame-based language very similar to Java. We used a combination of eye-tracking hardware and video observation to record where users were looking in the code, the time it took them to answer the task, and the graded correctness of their answer. The data comes from students at two institutions in different countries.
Findings: We found that there was no difference between plain Java, highlighted Java or Stride in terms of correctness, or speed to produce correct answers – and this was unaffected by whether users had seen the latter two interfaces before. There was a difference in eye gaze behavior, especially between Stride and the two Java interfaces. Participants' gaze moved around less, they had shorter saccade lengths, longer fixation durations, fewer regressions, and less line coverage in Stride, which could imply a higher cognitive load. Regardless of condition, all participants who answered correctly read buggy lines toward the end of the task.
Implications: The highlighting conditions showed a difference in low-level gaze behavior, but this did not translate into a difference at the higher level of program comprehension task performance. The gaze behavior difference could be due to the novelty of the design of the Stride interface *other than* the scope highlighting (which showed no such difference when present/not present in the Java display). However, it seems that large changes in the presentation of the program code have only a small (if any) effect on task performance.

CCS CONCEPTS

• **Human-centered computing → Visualization theory, concepts and paradigms**; • **Social and professional topics → Computer science education**.

KEYWORDS

Java, eye-tracking, code understanding, empirical study, Stride

ACM Reference Format:
Kang-il Park, Pierre Weill-Tessier, Neil C. C. Brown, Bonita Sharif, Nikolaj Jensen, and Michael Kölling. 2023. An eye tracking study assessing the impact of background styling in code editors on novice programmers' code understanding. In *Proceedings of the 2023 ACM Conference on International Computing Education Research V.1 (ICER '23 V1), August 07–11, 2023, Chicago, IL, USA.* ACM, New York, NY, USA, 20 pages. https://doi.org/10.1145/3568813.3600133

1 INTRODUCTION

Learning to read and write programs requires using some kind of program editor. There are many options available, whether in the block-based or text-based editing paradigm, or a hybrid option like frames [28]. Although these all use different editing paradigms, their presentation is ultimately lines of text with indentation used to represent scope – but differing in their use of color and other decorations for indicating program structure. As shown in the top left of Figure 1, classic programming displays tend just to highlight keywords: syntax highlighting. Block- and frame-based systems use colored rectangles to indicate scope, but generally without syntax highlighting (see the bottom of Figure 1).

The Java editor in BlueJ [29] has for a long time (since 2010) used such scope highlighting, as well as syntax highlighting: see the top-right of Figure 1. Thus there are many options for code presentation (termed *presentation modality* here), but there has been little investigation into which is beneficial – either for professionals or for novices, the latter being our area of interest.

For clarity, we provide the following definitions of concepts crucial to this paper:

Syntax highlighting is when certain words or characters in the program use a different font color, usually based on the meaning of the token. For example, keywords in the language might be colored blue, string literals might be green. The color affected is the color of the text characters, not the color of the background.

Scope highlighting is when the background of the program code is highlighted according to the lexical scopes of the program. Examples are shown in the top-right and bottom of Figure 1. Such highlighting may be inferred from the program source (as in our Java highlight condition) or determined as part of a structural editor (as in Stride or block-based editors).

We conducted a study with programming novices to compare three different code presentation modalities, which are shown in Figure 1:

1. Java with No scope highlighting **(JN)**. This is a very common presentation modality in text editors. It does feature *syntax* highlighting – coloring of keywords like `if` or `public`, but the background is plain white.
2. Java with scope Highlighting **(JH)**. This mode again features syntax highlighting, but also has coloured rectangles to emphasise where the curly bracket scopes extend from/to.
3. Stride **(S)**. Stride is a frame-based editor with a syntax that is very similar to Java, although it does not need curly brackets and semi-colons for delineation of code (this is determined implicitly by the editor structure, like in block-based editors). Stride uses only minimal syntax highlighting, but its display of scopes is very similar to BlueJ's scope highlighting for Java.

These are three realistic approaches to viewing code, all in use in many classrooms worldwide, with a very similar syntax but differences in how scope is presented. The obvious question for computing education is whether these interfaces have a different effect when novices use them. In this study, we thoroughly investigated the effects of presentation modality on novice programmers during code comprehension (code summarization, and bug finding) tasks. Our research questions are:

RQ1: Does presentation modality affect the accuracy of users on code comprehension tasks?

RQ2: Does presentation modality affect the speed of users on code comprehension tasks?

RQ3: Does presentation modality affect where users look during code comprehension tasks?

The key features of our study are as follows:

- A cross-over trial of three different programming interfaces.
- A combination of eye-tracking data, task correctness, and speed, to provide multiple complementary measures of the differences between interfaces.
- Participants from two institutions in two different countries.
- Pre-registered materials and analysis plan.

The paper is organized as follows. We explore related work in section 2, we describe our method in section 3, then detail our results in section 4, followed by discussion and implications in section 5 and finally conclusions and future work in section 6.

2 RELATED WORK

2.1 Relation between theory and editor design

The designers of code editors frequently add new visual decorations [43]. We are only aware of one theory that addresses visual aspects of program editors, by Conversy [14]. Conversy discusses the idea of "selective" markings that enable a viewer to instantly discern differences at a glance at the whole view. This can be done with using color or large luminosity differences, but would not apply to, for example, font changes. Thus syntax highlighting, scope highlighting (but also indentation) are selective, but curly brackets or semi-colons are not. There are many other theories in program comprehension research – such as the idea of beacons, top-down vs bottom-up comprehension, or semantic chunking [40, 42, 50] – but syntax or scope highlighting are not implemented in alignment with any of these theories.

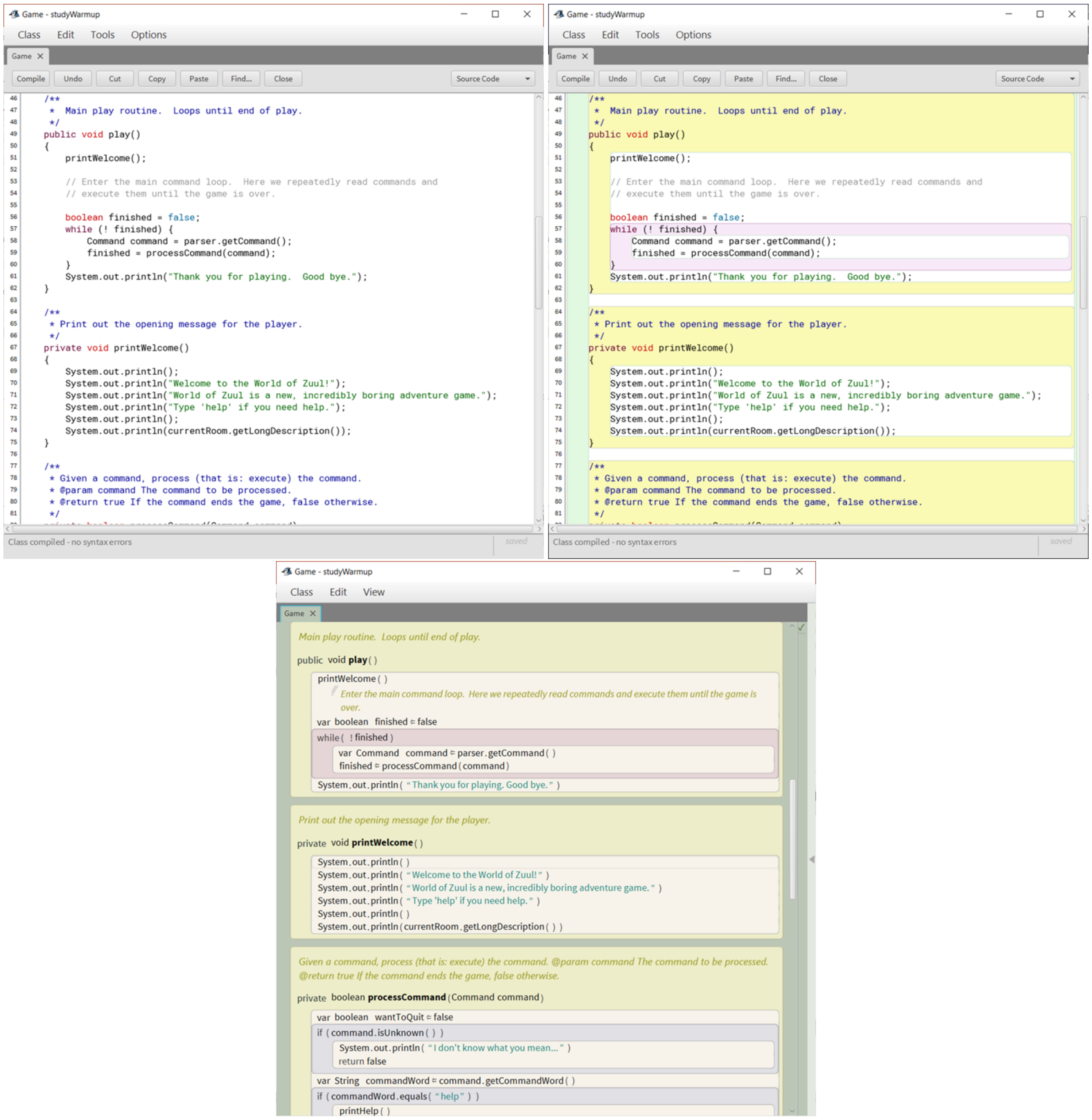
The main program comprehension model to which highlighting may (*post hoc*) relate is the Block Model [34, 35]. The lowest level of the block model are atoms, such as language elements – this would correspond to syntax highlighting of language keywords or literals. The second lowest-level are blocks, which would relate to scope highlighting, emphasizing the block grouping within source code. Thus the Block Model suggests that the highlights will help beginners understand basic program structure at the lower levels of abstraction and understanding.

Another relevant theory for editor design is that of cognitive load. Cognitive load theory has been used in user interface design [22] to explain task performance, and also in computing education [16] to explain human behavior. Various eye-tracking metrics have been found to correspond to cognitive load [15], suggesting that we could observe any interface differences in cognitive load via eye-tracking.

We suggest that it may be unnecessary to come up with detailed theories as to *why* different visual decorations will enhance readability if there is no convincing evidence *that* visual decorations enhance readability. Our aim is that this study will aid in determining whether there is an effect of the different designs, rather than building on a specific theory of highlighting and readability.

2.2 Role of scope highlighting in code understanding

A few studies have investigated the effect of editor highlighting on program comprehension, including eye tracking. Beelders and du Plessis [8] investigated the role of code syntax highlighting, and found that the presence or lack of syntax highlighting does not significantly impact programmers' reading behavior as measured by eye tracking. Hannebauer et al. [19] found the same result in their study, looking only at task performance and not eye tracking. This is despite syntax highlighting being ubiquitous in program editing for many decades. (Curiously, this mirrors a parallel issue of font selection for general text readability; although there are various design guidelines on when to use serif or sans serif fonts, many studies find no effect of font serifs, or lack thereof [2, 31, 45].)



```
    /**
     *  Main play routine.  Loops until end of play.
     */
    public void play()
    {
        printWelcome();

        // Enter the main command loop.  Here we repeatedly read commands and
        // execute them until the game is over.

        boolean finished = false;
        while (! finished) {
            Command command = parser.getCommand();
            finished = processCommand(command);
        }
        System.out.println("Thank you for playing.  Good bye.");
    }

    /**
     * Print out the opening message for the player.
     */
    private void printWelcome()
    {
        System.out.println();
        System.out.println("Welcome to the World of Zuul!");
        System.out.println("World of Zuul is a new, incredibly boring adventure game.");
        System.out.println("Type 'help' if you need help.");
        System.out.println();
        System.out.println(currentRoom.getLongDescription());
    }

    /**
     * Given a command, process (that is: execute) the command.
     * @param command The command to be processed.
     * @return true If the command ends the game, false otherwise.
     */
```

Class compiled - no syntax errors

```
    /**
     *  Main play routine.  Loops until end of play.
     */
    public void play()
    {
        printWelcome();

        // Enter the main command loop.  Here we repeatedly read commands and
        // execute them until the game is over.

        boolean finished = false;
        while (! finished) {
            Command command = parser.getCommand();
            finished = processCommand(command);
        }
        System.out.println("Thank you for playing.  Good bye.");
    }

    /**
     * Print out the opening message for the player.
     */
    private void printWelcome()
    {
        System.out.println();
        System.out.println("Welcome to the World of Zuul!");
        System.out.println("World of Zuul is a new, incredibly boring adventure game.");
        System.out.println("Type 'help' if you need help.");
        System.out.println();
        System.out.println(currentRoom.getLongDescription());
    }

    /**
     * Given a command, process (that is: execute) the command.
     * @param command The command to be processed.
     * @return true If the command ends the game, false otherwise.
     */
```

Class compiled - no syntax errors

```
Main play routine.  Loops until end of play.
public  void play ( )
    printWelcome ( )
    Enter the main command loop.  Here we repeatedly read commands and execute them until the game is
    over.
    var  boolean  finished = false
    while (  ! finished )
        var  Command  command = parser.getCommand ( )
        finished = processCommand ( command )
    System.out.println (  "Thank you for playing. Good bye."  )

Print out the opening message for the player.
private  void printWelcome ( )
    System.out.println ( )
    System.out.println (  "Welcome to the World of Zuul!"  )
    System.out.println (  "World of Zuul is a new, incredibly boring adventure game."  )
    System.out.println (  "Type 'help' if you need help."  )
    System.out.println ( )
    System.out.println ( currentRoom.getLongDescription ( ) )

Given a command, process (that is: execute) the command. @param command The command to be processed.
@return true If the command ends the game, false otherwise.
private  boolean processCommand ( Command command )
    var  boolean  wantToQuit = false
    if ( command.isUnknown ( ) )
        System.out.println (  "I don't know what you mean..."  )
        return false
    var  String  commandWord = command.getCommandWord ( )
    if ( commandWord.equals (  "help"  ) )
        printHelp ( )
```

Figure 1: The three interfaces, all showing the same program code. Top-left is Java No scope highlight (JN), which has only syntax highlighting and no scope highlighting. Top-right is Java Highlight (JH), which has syntax highlighting and scope highlighting. Bottom is Stride (S) which has minimal syntax highlighting, scope highlighting, and, in contrast to Java, no semi-colons or curly brackets.

Talsma et al. [44] explored the impact of the scope highlighting on novice programmers' reading behavior (using eye tracking), and concluded that code comprehension is not affected by the scope highlighting, while the reading pattern is. However, their study used a single hybrid mode of highlighting: a view with Stride's font choice and minimal syntax highlighting, scope highlighting and curly brackets and semi-colons. We expand on this to examine three interfaces: Java without scope highlighting, Java with scope highlighting (but no other changes), and Stride with its font choice, minimal syntax highlighting, but no curly brackets or semi-colons.

This allows us to investigate finer-grained differences in the design elements. In addition, we used a larger sample size (7 participants in their study compared to 62 in ours).

Weintrop et al. [48] conducted a paper-based study comparing an exam with two different code presentations. One is a plain text presentation with no highlighting of any kind. The other adds a block-based form of scope highlighting, with rectangles and colored backgrounds drawn to indicate scope, both of structural scopes (as in our scope highlighting conditions) but also to indicate bracketing within expressions: drawing boxes around, for example, the if condition or bracketed items therein. They found a difference in performance on the exam between these two conditions, with the scope highlighting improving performance.

Stride has been compared to Java in two existing studies of code writing. Price et al. [32] examined the differences in a US school setting and found some differences in the rate of task completion, with Stride being faster. Brown et al. [11] conducted a study involving multiple different UK schools and found no interface effect in a set of code writing tasks, either in task completion time or understanding of object orientation.

In summary, there is reason to believe that there could be a difference in program comprehension based on scope highlighting, but the existing evidence (comprising both controlled laboratory studies and observational school studies) shows a mixture of effects.

2.3 Code reading task choice

In order to conduct a study of program comprehension with different interfaces, we needed to provide our participants with program comprehension tasks to perform. We were informed by many prior studies involving program comprehension, especially in Java.

Busjahn et al. [13] presented the participants six short Java programs and asked them one of three questions: to write a summary of the code, to estimate the value of a variable after the program execution or answer an algorithm-related multiple choice question (MCQ). Writing a summary is another activity chosen by Abid et al. [1], restricted to some reasonably long content methods randomly chosen in various open-source Java systems. They were allowed to open the entire source code of the projects and to see the methods while writing their answers. Bednarik and Tukiainen [6] presented their participants with a set of three short to medium sized Java programs (each program being self-contained in a Java class). They obfuscated the methods and variables of the programs and asked participants to explain what the programs did. In a study comparing the code reading comprehension between novices with and without dyslexia, McChesney and Bond [30] created an open data set in which stimuli consist of three very short to short Java code *snippets*, that participants had to summarize.

To study how plan-schemata and composition of code affect the code comprehension, Kather et al. [26] designed a set of theoretical-driven short programs, but they targeted programmers with enough literacy to no longer be considered novices. Six short C programs were presented to students by Uwano et al. [46]; they discarded any code decoration from their stimuli (i.e. comments) but introduced each small program in the same manner to the reviewers before they read the code. Participants were left aware that each program included a small logical defect that had to be identified. Bednarik and Tukiainen [7] showed their participants three programs containing a few non-syntactic errors to study debugging strategies. Participants did not know how many bugs were in the programs and were given limited time to propose corrections (presumably orally). Similarly, Sharif et al. [38] studied the role of scan time in debugging by asking participants to review four Java code snippets containing a bug. Bug finding activities on a larger scale project (JabRef) have also been used to study gaze behaviour in change tasks [27] – the participants of the study were both experts and students, and the study focused on the action of fixing the bug.

While task code complexity should match the participants' ability, we favored stimuli that show full size Java class code, in the context of a Java application. This choice better reflects programming projects designed for real-life educative purposes. We devised a set of tasks, that required *either* summarization or bug finding. As these types of task are commonly found in code understanding research literature, we decided to devise a study with both. Code understanding and bug finding are also, again, realistic activities that students perform during their programming education.

2.4 Eye tracking

Eye tracking has gained interest for research in different fields partly thanks to the improvement of available technology. It is often used to simply infer what users look at [52], but it also allows researchers to understand better the cognitive load involved in the users' activities, reading and information processing in particular [15, 33]. Therefore, eye-tracking is a relevant technology to the research on code reading activities [6]. Comprehensive guides on conducting eye tracking studies were used as a reference for our method and metrics [23, 37].

2.4.1 *Eye-tracking metrics.*

The principal eye movements extracted from raw gaze data points are fixations and saccades [25]. Fixations strongly indicate attention, and therefore associated metrics are often exploited in code-reading research analysis: fixation count [1, 8, 30, 39, 44], fixation duration [1, 8, 12, 13, 26, 39, 44], fixation rate [39], and fixation location or heatmaps [3, 13, 30, 44]. More advanced metrics include saccade length [12], scan time [8, 38], block sequence diagram [44], and sparse matrix [6].

In this paper, we use the linearity metrics presented by Busjahn et al. [12] as well as the standard fixation counts and durations in our analysis of RQ3. The linearity metrics they validated include saccade length (Euclidean distance between two pairwise fixations), vertical and horizontal reading behavior, regressions, and line coverage. Most eye-tracking studies predefine the stimuli's areas of interests (AOI), which in the context of code reading matches *chunks* of code. We define chunks in our tasks to be contiguous logically related lines of code. They could represent beacons [49] that are understood as a whole and not necessarily at the individual line level.

2.4.2 *Eye tracking tools.*

The preferred eye-tracking devices researchers use are infrared remote eye trackers positioned below the monitor. Such devices offer a good compromise between data quality and ease of usage – being unobtrusive and easy to set up for each participant. The frequency of an eye tracker is a key parameter to consider based on the type of stimuli the research work focuses on. Typical frequency values found in literature can vary significantly:

50 Hz [6], 60 Hz [1, 3, 38, 39, 44], 120 Hz [12, 13, 30], 300 Hz [8, 26]. Andersson et al. [4] do not recommend sampling data on the lower frequency range for reading activities. In a compromise between sampling quality, technical integration and cost, we worked with an infrared remote eye-tracker sampling at 120 Hz.

Although manufacturers can provide in-the-box solutions to run the calibration and tracking sessions, they rely on a stimulus which has fixed dimensions (for example, a fixed-size portion of code). This does not reflect well the real case scenarios of code reading activities when readers often need to scroll through code text that spans across the monitor's height or switch between several files to grasp the content of a program. iTrace (https://www.i-Trace.org) is eye tracking infrastructure that was specifically designed to address these code reading and mapping related issues [18, 36]. Ogama by Voßkühler et al. [47] permits a recording and an analysis of gaze and mouse data with regards to defined static images or slideshows. Crescent has been developed by Uwano et al. [46] to facilitate the estimation of code lines correspondence to points of gaze. Although Crescent made a significant step in designing tools for the gaze analysis of code reviews by including mechanisms to handle scrolling and providing sensible information (lines of code), it lacks the granularity to retrieve the language keywords and possibility to use other IDEs for research studies.

Since iTrace gathers the necessary features we need for both the tracking and analysis of the gaze data on code stimuli, we extended it for our study. iTrace also comes with a post-processing toolkit [9] that we used to generate fixations.

3 METHOD

This section presents the details of our method. This study was approved by the research ethics boards at King's College London (reference LRS/DP-20/21-25453) and the University of Nebraska-Lincoln (IRB# 20211221356EP).

3.1 Pre-registration

The method and materials for this study were voluntarily pre-registered on the Open Science Foundation (OSF) before the data collection began. This can be viewed at https://osf.io/zws7f/. The following changes were made since the pre-registration:

- We changed our use of simple code categories for analysing gaze, to more detailed and classical eye-tracking metrics.
- We added analysis of task correctness, alongside the planned analysis of time to record correct answers.
- In the pre-registration we stated "we are unlikely to have enough correct answers for a 3x12 ANOVA" (and planned a simpler analysis) but as it turned out we did, so we performed the ANOVA analysis.

A separate repository at https://osf.io/3uprw/ contains all the details of the results.

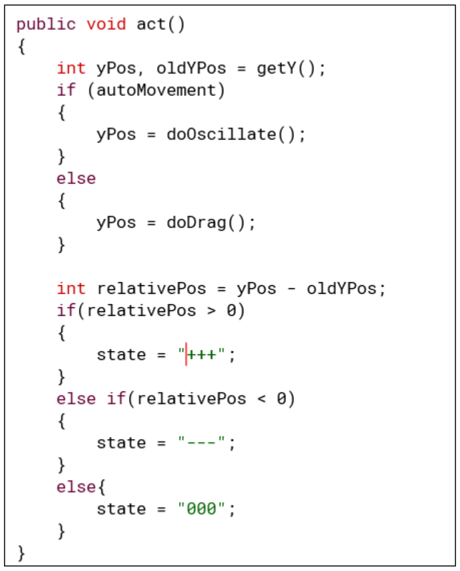
3.2 Conditions and Design

We are interested in the impact of three code presentation conditions: no background scope highlighting (the editor background is plain), scope-highlighting and frames. The first 2 conditions are applied to code written in Java. The last one, frames, is applied to a Java-equivalent language: Stride[1]. In the context of this study, the only differences between frame-based editing and text-editing are the *visual* aspects. Other aspects of frame-based editing (input and navigation aspects) do not impact this study as all the tasks are purely about reading code, not editing it. Figures 2–4 show some tasks and code snippets under each condition (respectively, no highlighting, scope-highlighting, frames).

3.3 Activities

3.3.1 Code Reading tasks. As explained earlier in subsection 2.3, code reading activities usually involve two types of tasks: code summarizing and bug finding. For our study, we designed tasks for these two types of activities and sub-categorized each type. For summarizing, we sub-categorized the type of tasks depending on the scope of the targeted: class-span or method-span. For bug finding, we sub-categorized the type of tasks depending on the nature of the bug: functional (i.e. the code itself is logically sound but does not generate the intended functional behavior) or code-related/logical (i.e. the code is logically not sound, such as an infinite loop). Figures 2-5 illustrate a task example for each aforementioned type.

All tasks were in the form of a question. The participants replied orally and only by **reading** the code; no modification was involved. We did not set a time limit to answer the question but moved to the next task when participants took longer than expected to answer and seemed stuck. We instructed the participants to answer the question in natural language verbally. We forewarned them that if an answer were too generic (for example, replying "The result is wrong." without explanation), we would ask for more details.



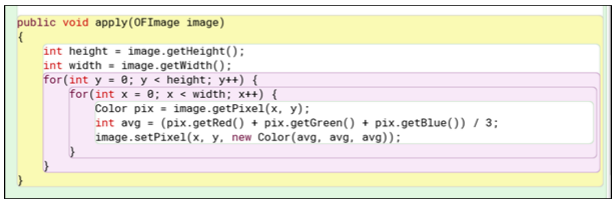
```
public void act()
{
    int yPos, oldYPos = getY();
    if (autoMovement)
    {
        yPos = doOscillate();
    }
    else
    {
        yPos = doDrag();
    }

    int relativePos = yPos - oldYPos;
    if(relativePos > 0)
    {
        state = "+++";
    }
    else if(relativePos < 0)
    {
        state = "---";
    }
    else{
        state = "000";
    }
}
```

Figure 2: Code snippet for a code-scope summarizing task, with the condition "no scope highlighting" (JN)

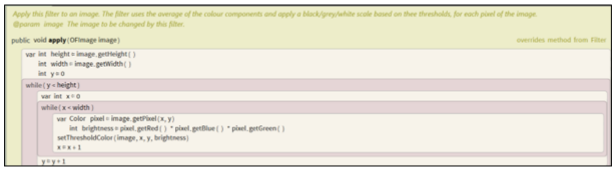
We designed 12 tasks, so that each task category is associated with each condition. We randomly split the 12 tasks across 3 different projects since typical novice programmer's projects are small

[1]https://stride-lang.net/



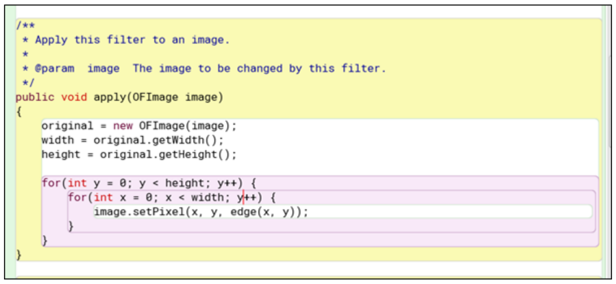
```
public void apply(OFImage image)
{
    int height = image.getHeight();
    int width = image.getWidth();
    for(int y = 0; y < height; y++) {
        for(int x = 0; x < width; x++) {
            Color pix = image.getPixel(x, y);
            int avg = (pix.getRed() + pix.getGreen() + pix.getBlue()) / 3;
            image.setPixel(x, y, new Color(avg, avg, avg));
        }
    }
}
```

Figure 3: Code snippet for a method-scope summarizing task, with the condition "scope highlighting" (JH)



```
Apply this filter to an image. The filter uses the average of the colour components and apply a black/grey/white scale based on thee thresholds, for each pixel of the image.
@param  image  The image to be changed by this filter.
public  void apply(OFImage image)                                   overrides method from Filter
    var int  height = image.getHeight( )
        int  width = image.getWidth( )
        int  y = 0
    while ( y < height )
        var int  x = 0
        while ( x < width )
            var Color  pixel = image.getPixel(x, y)
                int  brightness = pixel.getRed( )  * pixel.getBlue( )  * pixel.getGreen( )
            setThresholdColor(image, x, y, brightness)
            x = x + 1
        y = y + 1
```

Figure 4: Code snippet for a functional bug finding task, in the Stride condition (S). The question asked was "The images produced with the Threshold filter are almost always very bright beyond expectation. Can you find where the issue comes from, in the class ThresholdFilter?"



```
/**
 * Apply this filter to an image.
 *
 * @param  image  The image to be changed by this filter.
 */
public void apply(OFImage image)
{
    original = new OFImage(image);
    width = original.getWidth();
    height = original.getHeight();

    for(int y = 0; y < height; y++) {
        for(int x = 0; x < width; y++) {
            image.setPixel(x, y, edge(x, y));
        }
    }
}
```

Figure 5: Code snippet for a logical/code bug finding task, with the condition "scope highlighting" (JH). The question asked is "Can you identify the bug in the method apply?"

and participants to the study may find working with multiple projects more interesting and stimulating than a large single project. All three projects are Java or Java-like projects, and they were written using BlueJ[2] (which handles both Java and Stride).

The first project, ImageViewer, implemented a simple graphical AWT/Swing Java desktop program that allows the user to load an image file, and apply filters to that image. The second and third projects (SpaceGame and WaveLab) were the source code of a Greenfoot *scenario*[3]. The second project, SpaceGame, was adapted from a space shooting game scenario available on the Greenfoot website[4]. The last project, WaveLab, an interactive wave simulator, was adapted from the Greenfoot book associated projects[5]. Comments in all the projects' source code were preserved, except for the methods that needed to be summarized by the participants.

The distribution of the tasks is summarized in Table 1. We did not fully balance the ordering of tasks among all participants for practical reasons (it would require at least 12! = 479 001 600 participants without grouping by project), so the only balanced variable was the condition: all participants completed the tasks in the same order, with a condition depending on one of the 3 groups they were assigned to.

3.3.2 Post-study Questionnaire & Demographics. In order to understand how the different tasks and conditions of the study were perceived by the participants, we devised an online questionnaire asking the participants their opinions of the study and the conditions. We asked for demographics, consisting of the standard profile-related, education-related and Java/Stride experience-related questions. Demographics data was collected together with the post-hoc questionnaire. We preferred keeping the demographics' questions *after* the data collection to avoid the influence of these questions over the participants' performance and state-of-mind during the data collection. It is possible that being asked gender or ethnicity before the study may invoke stereotype threats [41] or similar priming effects [24]. The combined questionnaire details are available in the aforementioned OSF pre-registration repository.

3.4 Apparatus

3.4.1 Hardware. Our setup comprised a monitor (24.1", 60Hz) on which an eye-tracker is attached (Tobii Eye Fusion Pro, 120Hz, mounted in the usual configuration below the display). Both monitor and eye tracker are connected to a laptop (running Windows 10 Enterprise 21H2 (x64), Intel Core i7-8665U CPU @ 1.9GHz, RAM 16 GB). The display resolution was set to 1920x1200, and the scaling[6] was set to 100%. As the data collection relies on a code reading activity, only the mouse was available to the participant. The session was recorded by 2 webcams (Full HD 1080p/30fps), connected to a separate dedicated laptop. One webcam was recording the participant's face, the other was recording the monitor, from an "over-the-shoulder" control view. Both audio and video was recorded.

3.4.2 Software. al The projects were used with BlueJ v.5.0.3. The eye-tracking calibration and data recording was run by iTrace v.0.2.0, eye tracking infrastructure designed to specifically manage the mapping between gaze points and the code editor areas of interest [18, 36]. In order to allow BlueJ to comply with iTrace data collection, we developed a BlueJ extension to iTrace namely, iTrace-BlueJ. We used OBS Studio 27.2.4 for recording the videos from the webcam.

3.5 Study Protocol

In both investigating institutions, the apparatus was arranged in a similar fashion where the investigators were not in the participants' field of view, once the study effectively started (after consent, and after starting the video-recording). This detail is important as it

[2]https://bluej.org
[3]Greenfoot is an interactive programming environment. A *scenario*, in Greenfoot, is the content fed to this environment (a game for example). From the programmer's point of view, the main parts to implement are *the world* and *the actors* that evolve in the world. Details about Greenfoot can be found at https://www.greenfoot.org/overview
[4]https://www.greenfoot.org/scenarios/27256.
[5]The book's projects are available at https://www.greenfoot.org/book/.
[6]On Windows display settings, the scale of the text, apps and other items.

Table 1: Task types and conditions distribution

Project	Task #	Task Type[(1)]	Condition[(2)]		
			Group 1	Group 2	Group 3
ImageViewer	1	SS	JH	S	JN
	2	SS	JN	JH	S
	3	FB	S	JN	JH
	4	CB	JH	S	JN
SpaceGame	5	SS	S	JN	JH
	6	CS	JH	S	JN
	7	FB	JN	JH	S
	8	CB	S	JN	JH
WaveLab	9	CS	S	JN	JH
	10	CB	JN	JH	S
	11	FB	JH	S	JN
	12	CS	JN	JH	S

(1) SS: method-scope summarizing, CS: code-scope summarizing, CB: logical/code bug, FB: functional bug

(2) JN: no scope highlighting, JH: scope highlighting, S: frames

helped prevent the participants' natural inclination: to turn toward the investigators to answer the questions. We explicitly explained this to the participants and instructed them to avoid moving too much (to avoid disrupting the eye tracker) but to keep a natural sitting posture. We could adjust the distance between the monitor and the seat and the height of the monitor, but we made sure the chair had no wheels to avoid movement during the data collection.

A short introduction to BlueJ and the three conditions (no scope highlighting, scope highlighting and frames) was given to the participants. For Stride, we also introduced the differences to Java. We then gave a description of the four types of tasks (cf. Section 3.3.1), with examples, to the participants. It was followed by a warm-up example without eye-tracking. The warm-up project is a Java text command-based game, World of Zuul, opened in BlueJ. The warm-up activity consisted of two tasks: a code-span summarizing task (with the frames condition) and a functional bug finding task (with the scope-highlighting condition). Once the warm-up activity was completed, we gave a very brief presentation of the three projects and the number of tasks used for the data collection. For each project (ImageViewer, SpaceGame and WaveLab), we showed a snapshot of the project's UI and explaining its purpose, as well the core concepts of the Greenfoot environment for projects 2 and 3, then completed the four tasks.

3.6 Data Collection

3.6.1 Content. The data collected during the study can be split into 3 groups: gaze-related, answer-related and evaluation-related. Gaze data comprises the aggregation of timestamped raw gaze points (generated by the eye-tracker) with the matching code token (provided by iTrace-BlueJ). The latter is primarily word-level: iTrace-BlueJ is able to associate the point of gaze with a specific code token (a word and its role (syntactic category), for example, an operator) [36]. When gaze is not detected against a word, iTrace-BlueJ marks the area of interest as the "background". While in Java this notion applies to the whole document in the BlueJ editor, with Stride this notion is more precise, the "background" is relative to a *frame*.

During the study, we also recorded videos to enable the verification of the participants' activity during the completion of the tasks, and the evaluation of their answers' correctness and speed. The evaluation-related data was collected via the post-hoc questionnaire.

3.6.2 Task answers correctness and duration marking. The videos helped us to assess 1) the duration taken for a participant to answer and 2) categorize the answer given by a participant for each task. Each screen-recording video has been *separately* reviewed by pairs formed from three reviewers. After a few individual trials, the reviewers shared their feedback to set up guidelines indicating how to mark the videos, and ensure consistency for all reviews. For each task, we defined:

- the answer *start time* as the video time when the investigator started to formulate the question;
- the answer *end time* as the video time when the participant concluded their answer; and
- the answer *correctness category*, defined as one of:
  - Y (correct and detailed answer),
  - C (almost correct and detailed answer),
  - P (partially correct and/or partially detailed answer),
  - N (incorrect answer), or
  - X (no answer given or do not know how to answer).

As the different investigators did not consistently ask for details of the inner methods for tasks 6 and 12 while running the study, we did not require this explanation for the task to be marked correct. However, we considered the participant needed to mention the call to an inner method to make a correct answer. For task 10, many participants suspected the bug to be that the index variable (for the array) did not start at 0. Even though this was not the intended bug (which was that the index would overflow beyond the bounds of the array), we considered this answer as partially correct because of the ambiguity of the program.

Once all videos had been *separately* reviewed, the reviewers compared their markings to identify and resolve marking conflicts - different correctness category, or a time difference that exceeded 10 seconds, and produce the final *unified* reviews. For time-related markings, the unified reviews consisted of the average of the separated time markings, unless manually corrected during conflict resolution.

3.6.3 Participants. Participants were recruited at both King's College London in the UK, and University of Nebraska-Lincoln in the USA. For both institutions, the participants were students who had taken Java programming classes and were still relatively novices in Java programming (between 1 term to 2 years passed since taking their Java course). In order to incentivize participants, participation in the study (regardless of completion) was rewarded with an Amazon gift card equivalent to around 10–15 USD. Recruitment was done within the investigating universities via posters, flyers, oral calls for participation in lectures, and email communication on mailing lists.

3.6.4 Areas of Interest. Before analyzing the collected gaze data, two authors collaboratively defined the areas of interest (AOIs) for each task. We only defined the AOIs on the relevant chunks of code of the class mentioned by the task by extracting the key parts of the code in line with the expected answer of the task. Chunks are continuous sets of lines that are logically related. Therefore, the AOIs were usually limited to the chunks of the class methods that needed to be summarized (for summarization tasks) or where the bug occurred (for bug finding tasks). Whenever any other chunk of the code (e.g. fields, other method) was particularly important for the task completion, they were also considered as AOIs. The AOIs were delimited by a start / end lines for Java, and the equivalent in Stride. Note that the code chunks can be nested. We decided to use this chunking method compared to looking at individual line level analysis because it is more intuitive and is based on the concept of beacons [20, 21, 49] (cohesive chunks that make sense together as a whole).

4 RESULTS

4.1 Participation

Combined between the two institutions, 62 participants took part in the study (40 at University of Nebraska-Lincoln, 22 at King's College London, average age 21.2 ± 3.4). We had to turn down 3 other applicants for whom eye tracking could not be performed due to several failed attempts in calibrating the eye tracker. All participants were students proficient in English, but English was identified as a native language for only 36 [58.1%] participants. Table 2 summarizes the other participants' demographic information.

Java and BlueJ are used differently at the two institutions. Table 3 shows how participants reported their experience regarding both Java and BlueJ. Combined experience reports between the two institutions indicate that roughly half the participants had experience with BlueJ for programming in Java. However, only King's College London (and not University of Nebraska-Lincoln) uses BlueJ to teach Java courses in its Bachelor level curriculum. Therefore, participants who had *not* used BlueJ to program in Java before were rare at King's College London (3 participants, 13.6% of the institution), while they conversely were the majority at University of Nebraska-Lincoln (33 participants, 82.5% of the institution).

Table 2: Participant demographics

Gender		
Female	18	(29.0%)
Male	42	(67.7%)
Other / no answer	2	(3.2%)
Ethnicity		
Arab / Middle Eastern	1	(1.6%)
Asian	24	(38.7%)
Hispanic / Latino	1	(1.6%)
Mixed / Multiple Ethnic Groups	4	(6.5%)
White / Caucasian	31	(50%)
No answer	1	(1.6%)
Education		
Bachelors / BSc	51	(82.3%)
year 1	1	(1.6%)
year 2	25	(40.3%)
year 3	20	(32.3%)
year 4	5	(8.1%)
Masters / MSc / MSci	6	(9.7%)
PhD	5	(8.1%)
Vision conditions		
Color blindness (red-green)	2	(3.2%)
Visual impairment	10	(16.1%)

Table 3: Participants' experience with Java and BlueJ

Java (where?)		
In course	36	(58.1%)
Outside course	5	(8.1%)
Both	21	(33.9%)
Java (how long?)		
Less than 1 year	16	(25.8%)
Between 1 and 2 years	32	(51.6%)
More than 2 years	14	(22.6%)
BlueJ (used for Java?)		
Yes	26	(41.9%)
No	36	(58.1%)
BlueJ (used for Stride?)		
Yes	1	(1.6%)
No	61	(98.4%)

4.2 Dataset Errata

As often happens in empirical studies, the actual data set obtained during the study differs from the expected "theoretical" design due to human error. In this section, we transparently report such issues with our study.

A mistake in the WaveLab source code for the group 3 (cf. Table 1) caused the condition JH to be presented instead of the intended JN condition for task 11. For 2 participants assigned to group 3, during the study, the investigator did not toggle the BlueJ eye tracking

Table 4: The mean correctness score by interface, after excluding task 11 (which has no JN data).

Interface	Average
JN	2.23
JH	2.31
S	2.18

correctly for one of the projects (ImageViewer, SpaceGame). Consequently, for those projects the eye tracking data is not available, and the condition JN has also been rendered as JH in the corresponding projects. These mistakes have taken into account in the analysis of the conditions. Across all 62 participants, we have the following condition distribution: 36.4% JH, 33.3% S and 30.2% JN (instead of representing exactly a third for each condition).

For 2 participants, the video recordings were not started properly, during the whole session for the first participant (resulting in no video marking at all for task answer correctness and duration), and later in one of the projects for the second (resulting in no video marking for answer correctness (task 9) and duration (tasks 9 and 10)). For one participant, two tasks in the middle project were accidentally swapped in their order.

4.3 RQ1 and RQ2 Results: Task Correctness and Duration

4.3.1 Conflict resolution. The correctness and duration of task answers were marked based on video examination as explained in Section 3.6.2. Figures 6 and 7 illustrate the percentage of conflicts for each task, respectively for the correctness category and the timings. The nature of the tasks may explain why there were often more conflicts in resolving the correctness of the answers for the summarization tasks than for the bug finding tasks: the explanations given by the participants about a method, and the interpretation made by the investigator is more subjective than bug finding. Not surprisingly, there were very little start time conflicts since the questions were always asked in the same manner (most start time conflicts were due to manual marking mistakes). Overall, time marking was more consistent than correctness marking, as it has less room for the investigators' interpretation. The conflicts were resolved in a series of long meetings between the three investigators. Most were found to be caused by a small number of systemic disagreements or misunderstandings of the agreed categorization.

4.3.2 Correctness analysis. We analyzed the correctness of the answers to each task. We transformed the category to a numerical scale as follows:

- Incorrect (N) and absent (X) answers became 0.
- Partially correct (P) answers became 1.
- Close to correct (C) answers became 2.
- Correct (Y) answers became 3.

The intuition behind our results can be explained from the simple averages shown in Table 4. Firstly, there is a ceiling effect; on a scale of 0–3, an average above 2 reflects that a large proportion (in fact, 66.0%) of the grades are 3, the maximum. Secondly, the interfaces have little difference, as we will see in the statistical results. These effects are also apparent in Figure 8.

Table 5: The median answer time across all tasks, by interface, after excluding task 11 (which has no JN data).

Interface	Median answer time (seconds)
JN	95
JH	96
S	95

Our core analysis is a two-way repeated measures ANOVA with dependent variable grade, and independent variables of Interface (3 levels) and Task (11 levels, after excluding Task 11). Due to the ceiling effect in grade the data is not normally distributed as the test assumes, but there is no non-parametric equivalent to a two-way ANOVA so we proceeded, with caution. This analysis showed no effect of Interface ($F(2, 637) = 0.568, p = 0.567$), which is our key research question. It did show an uninteresting effect of Task ($F(10, 637) = 22.725, p < 0.001$), indicating that there is a difference in difficulty between tasks. It also showed an interaction between Interface and Task ($F(20, 637) = 1.675, p = 0.033$), meaning that the effect of Interface differs by Task. Intuitively, however, if the averages are all the same for the interfaces then any difference in a particular task (Stride is better for this task!) is likely to be balanced out by the opposite effect in other tasks (Stride is worse at this task!). Nevertheless, we conducted follow-up one-way non-parametric ANOVAs (using the Kruskal-Wallis test) for each task, applying FDR correction [10] to the p-values. None of these follow-up analyses were significant.

The ceiling effect in grade is a potential problem, so we conducted a second exploratory analysis. We excluded all tasks where the median performance was 3 (i.e. where over half the responses were completely correct), which left us with only tasks 2, 7, and 10. This data was still not normal. The analysis again showed no effect of Interface ($F(2, 174) = 0.175, p = 0.840$), but again an effect of Task ($F(2, 174) = 10.411, p < 0.001$) and an interaction ($F(4, 174) = 2.643, p = 0.035$). The interaction was not followed up again, as these follow-up tests were already conducted as part of the previous follow-up tests that were not significant.

4.3.3 Timing analysis. We analyzed the time taken to complete an answer. We only included answers that were correct (Y) or almost correct (C), as we felt it was uninformative how long participants took to reach an incorrect answer (and incorrect answers could be affected by a potential speed-accuracy trade-off). The times taken are in seconds and are timed from the beginning of the experimenter's question to the end of the participant speaking. Naturally, some participants will give longer answers than others in an open answer, but since all participants completed all tasks and we are only interested in differences by Interface, this should balance out across the condition allocation.

The times taken are shown as box-plots split by Task and Interface in Figure 9. For intuition, the median times split by interface are shown in Table 5 which shows that the times carry the same pattern as the correctness: no difference between the blocks.

This is reflected in the results of our core timing analysis, which is again a two-way repeated measures ANOVA with dependent variable time, and independent variables of Interface (3 levels) and Task (11 levels, after excluding Task 11). The analysis showed no effect of

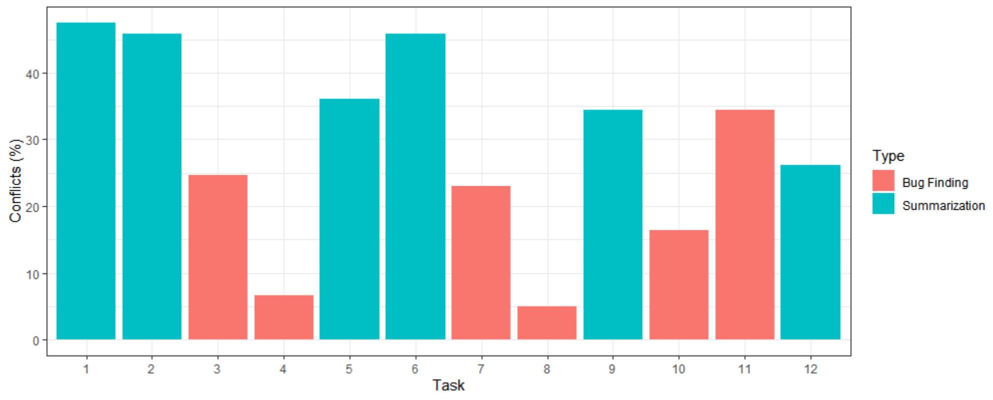


Figure 6: Percentage of answer correctness category evaluation conflicts for each task, per type.

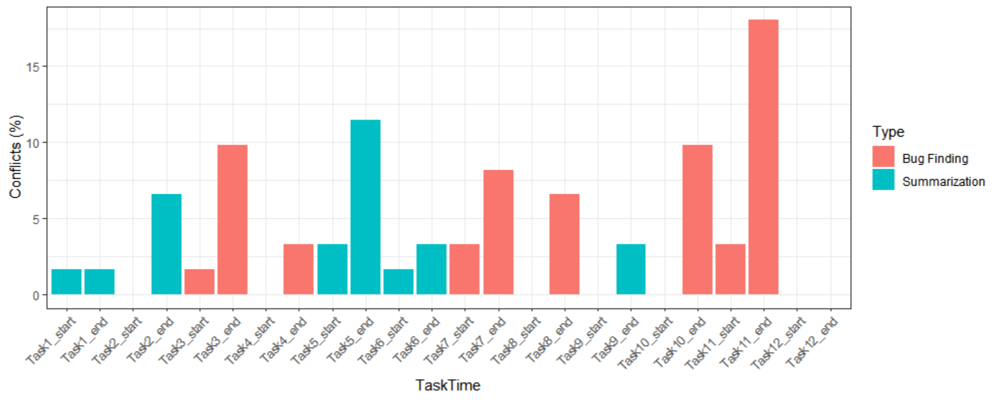


Figure 7: Percentage of answer start/end time evaluation conflicts for each task, per type.

Interface ($F(2, 473) = 0.089, p = 0.915$). It showed an uninteresting effect of Task ($F(10, 473) = 16.381, p < 0.001$) indicating that there is a difference in time taken between tasks. There was no interaction between Interface and Task ($F(20, 473) = 0.537, p = 0.951$).

4.4 Participant Perception of Background Differences

Following the code-reading activities, we asked the participants to give written feedback on the different background styles (i.e. JN, JH and S conditions mentioned in Table 1) they encountered throughout the tasks. The majority of participants indicated a preference for Java with scope highlighting and/or Stride - or a discomfort when there was no highlighting (42 [67.7%]). Only 6 participants [9.7%] indicated a preference for the Java without scope highlighting condition, blaming the impact on cluttering the visual interface with the colors, or familiarity with such a visual style. A small set of participants (14 [22.6%]) did not find any particular differences between the different conditions or did not express an answer that we could interpret. It is worth noting that a few participants mentioned they needed a bit of time to get used to Stride (9 [14.5%]) but none of them considered the Stride condition to be the worst.

4.5 RQ3 Results: Eye Movement Data

We present analysis of the eye tracking data in two different forms in the following subsections. The first are standard metrics we defined in subsubsection 2.4.1 which show overall eye differences in the different conditions. The second is a timeline visualization of the scanpaths (a directed path of fixations) between areas of interest in the code, indicating how participants were navigating the code while solving the tasks.

4.5.1 Eye Movement Metrics. Table 6 shows the results of pairwise t-tests for each metric [12] and each pair of conditions. These are corrected within each row for multiple comparisons using the Benjamini-Hochberg False Discovery Rate (FDR) procedure [10] with a base $\alpha = 0.05$. In practical terms, this means the lowest p-value in each row is significant if it is beneath $\frac{1}{3} \times 0.05$; if so, the second lowest p-value is significant if it is beneath $\frac{2}{3} \times 0.05$; if so, the highest p-value is significant if it is beneath 0.05. Figure 10 shows the fixation counts and durations for each condition which are normalized for the length of time it took for each participant to complete each task. The statistical tests in Table 6 confirm that there is a significant difference between Stride and each of the other two Java interfaces for fixation counts for all tasks ($p < 0.001$). The differences between JH and Stride were very large throughout all

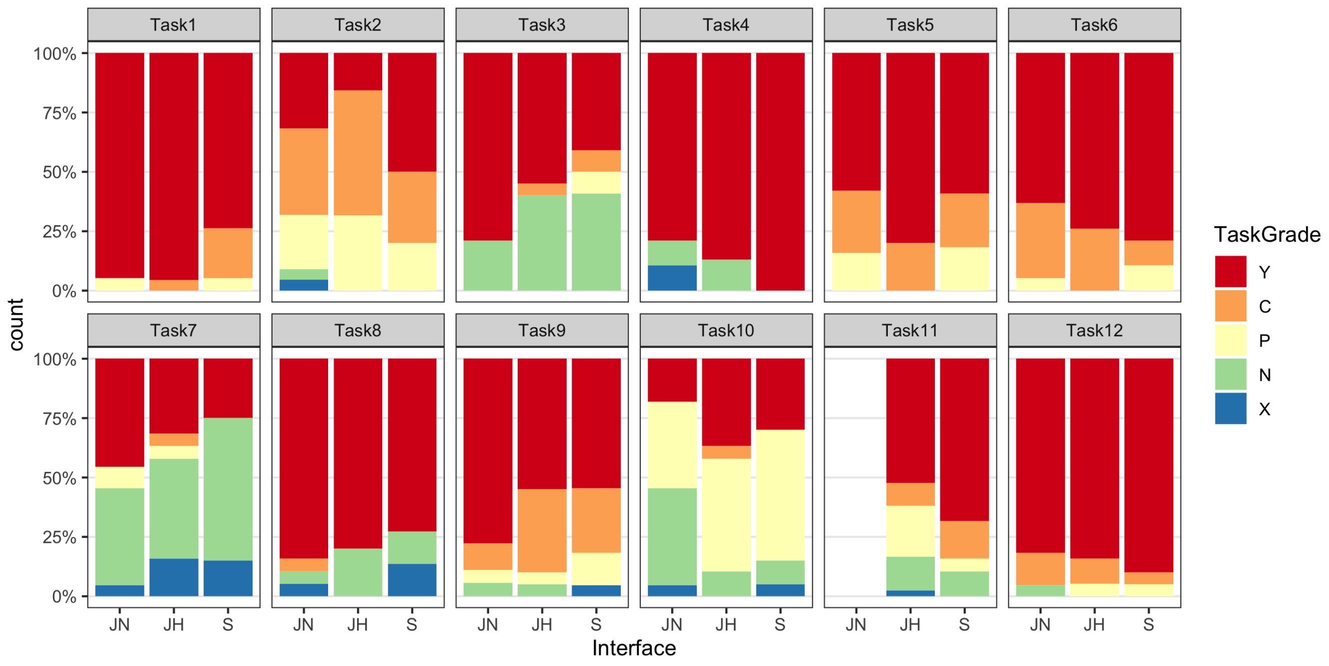


Figure 8: The counts of each grade, split by task (grid square) and by interface (the three bars within each square). More answers in the Y (red) and C (orange) categories indicate better performance.

tasks (Cohen's d = 1.403) and large between JN and Stride (Cohen's d = 1.171). For fixation duration, there is a medium effect size between JN and JH (Cohen's d = -0.780) as well as between JH and Stride (Cohen's d = 0.549). A small effect was observed between JN and Stride (Cohen's d = -0.496).

Figure 11 shows the results of gaze-based measures of linearity split by condition. These metrics are derived from the metrics used by Busjahn et al. [12] to measure the level of linearity in reading source code. Out of these metrics, there were shorter **saccade lengths** across all tasks in Stride with large effect than JH ($p < 0.001$, Cohen's $d = 0.852$) and medium effect with JN ($p < 0.001$, Cohen's $d = 0.606$). When compared to JN, saccade lengths for tasks in JH were longer with a small effect (p<0.021, Cohen's $d = -0.242$). **Vertical next text** results show that gazes are less likely to move forward one line in Stride with a huge effect compared to JH ($p < 0.001$, Cohen's $d = 2.145$) and JN ($p < 0.001$, Cohen's $d = 2.401$). **Vertical later text** results show that gazes are less likely to move forward multiple lines in Stride with a very large effect compared to JH ($p < 0.001$, Cohen's $d = 1.742$) and JN ($p < 0.001$, Cohen's $d = 1.935$). A small effect was observed in JH compared to JN ($p = 0.017$, Cohen's $d = 0.293$) Gazes are less likely to move forward horizontally within a line (**horizontal later text**) in Stride for bug fixing tasks where there was a small effect compared to JH ($p = 0.002$, Cohen's $d = 0.470$). **Regression rate** results show that gazes are less likely to move backwards in Stride with a small effect compared to JH ($p = 0.005$, Cohen's $d = 0.349$).

Line regression rate results show that gazes are more likely to move backwards horizontally within a line in Stride with a small effect size when comparing against JH ($p = 0.025$, Cohen's $d = -0.381$) and a medium effect against JN ($p < 0.001$, Cohen's $d = -0.739$). Comparing JH against JN has a small effect size ($p = 0.022$, Cohen's $d = -0.306$). **Line coverage** shows that there was a lower percentage of total lines in Stride were being looked at. This effect was huge when Stride was compared against JH ($p < 0.001$, Cohen's $d = 2.165$). On the other hand, there was a higher percentage of total lines in JH that were being looked at compared to JN with a very large effect ($p < 0.001$, Cohen's $d = -1.487$).

Based on these results, participants are shown to navigate around elements of the code less when reading code written in the Stride language, perhaps spending more time being focused around the relevant areas of the task.

4.5.2 Timeline Analysis. To determine which parts of the code a participant has been looking at over time, we visualized the scan patterns of the participants by generating augmented scarf plots using the Alpscarf [51] web application. We first plotted the fixations on the different key parts of the task's source code over the entire task duration with the duration-focus and non-normalized option used to visualize the amount of time each participant spent in AOIs. These settings will provide an easier-to-understand visualization of the movement of fixations over time compared to simply analyzing the distribution of how many fixations there were on each AOI as shown in Table 7. However, because one participant may spend more or less time completing a task than others, the normalization of the plot will allow for better viewing of the transitions. The normalized plots for all tasks are available in the results replication package.

We chose a lengthy bug-fixing task that many participants had difficulty completing for this analysis. WelcomeWorld was a task

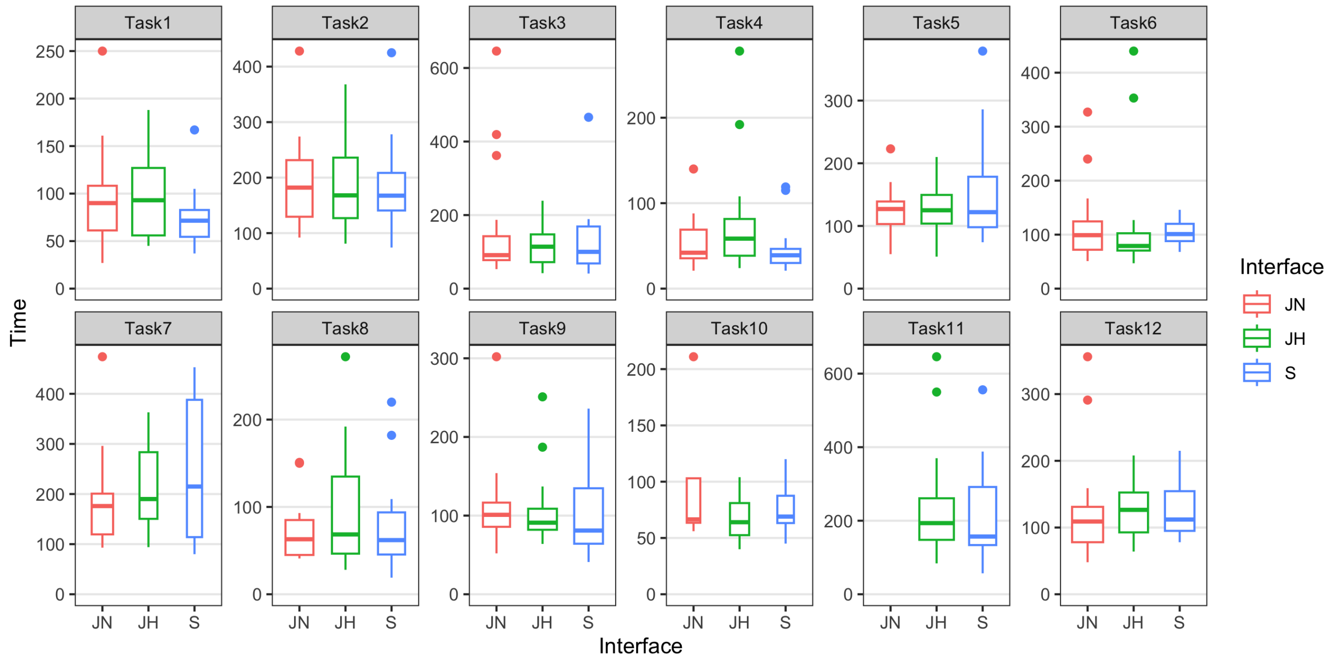


Figure 9: The times taken to give correct (Y) or close to correct (C) answers to the 12 tasks, split by Interface. Smaller time values indicate a faster answer, so lower on the Y axis is better. Each box shows the lower quartile, median, and upper quartile of the data. The lower/upper vertical whiskers extend to the smallest/largest value within -1.5/+1.5 respectively of the inter-quartile range (i.e. height of the box). All other points beyond this range are plotted individually. The width of each plot is proportional to the amount of data (i.e. the amount of C or Y answers to a particular question), which is noticeably lower for the harder tasks of task 7 (n=22) and task 10 (n=18) compared to the median of n=52.

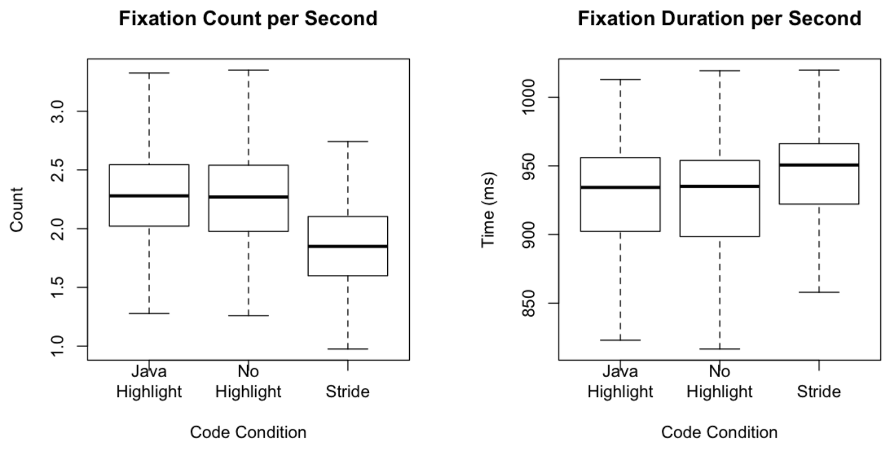


Figure 10: Fixation counts and fixation durations across the tasks, for each condition. The box plot whiskers show the minimum and maximum, with the boxes showing the lower/upper quartiles and the median. Because each participant took a different amount of time to finish the task, we report on the fixation metrics per sec to make a fair comparison.

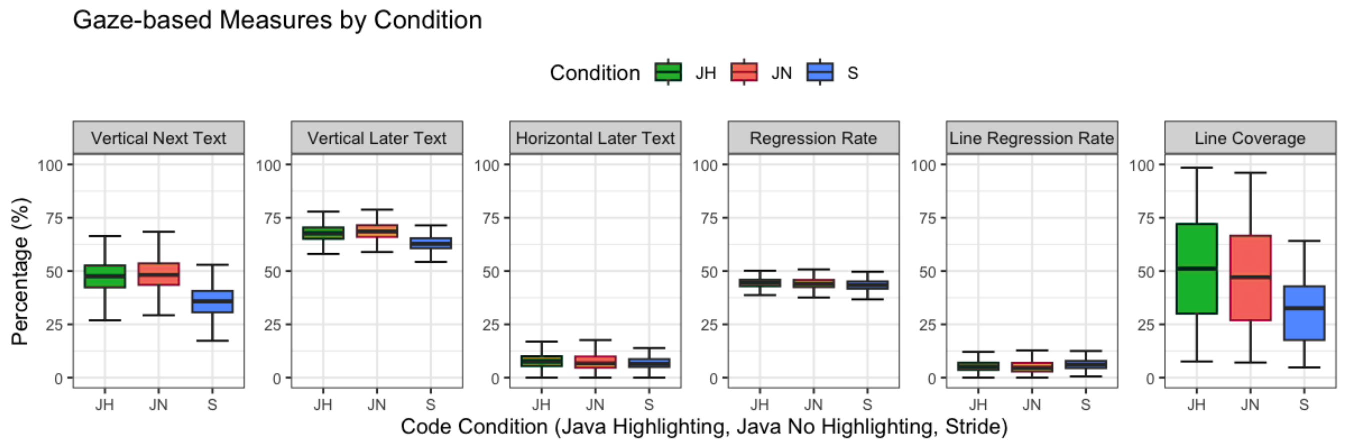


Figure 11: Distribution of gaze-based measures by condition, where JH is Java Highlighting, JN is Java No Highlighting, and S is Stride. Each box represents the individual measures.

Table 6: 2-tailed *t*-Test *p*-values (top numbers) and effect sizes (bottom numbers in parenthesis) between task conditions for each fixation metric. Degrees of freedom is 61. The tests are corrected for multiple comparison on a per-row basis using the False Discovery Rate procedure. Significant comparisons are marked with an asterisk * after the p-value.

Metric \ Comparison	JN vs JH	JN vs S	JH vs S
Fixation Count	0.840 (-0.019)	<0.001* (1.171)	<0.001* (1.403)
Fixation Duration	<0.001* (-0.780)	0.002* (-0.496)	<0.001* (0.549)
Saccade Length	0.021* (-0.242)	<0.001* (0.606)	<0.001* (0.852)
Vertical Next Text	0.209 (0.152)	<0.001* (2.401)	<0.001* (2.145)
Vertical Later Text	0.017* (0.293)	<0.001* (1.935)	<0.001* (1.742)
Horizontal Later Text	0.143 (-0.168)	0.112 (0.236)	0.002* (0.470)
Regression Rate	0.200 (-0.130)	0.102 (0.210)	0.005* (0.349)
Line Regression	0.022* (-0.306)	<0.001* (-0.739)	0.025* (-0.381)
Line Coverage	<0.001* (-1.487)	0.375 (0.182)	<0.001* (2.165)

that was the most difficult for participants due to its length and ambiguity in method names. A screenshot of WelcomeWorld's buggy method and ambiguous class is shown in Figure 12, with each of the rectangles representing key parts that were used as AOIs within the displayed code.

We use the eye tracking data for this task as shown in Figure 13 for participants that completed the task in Java with highlighting, Figure 14 for participants that completed the task in Java with no highlighting, and Figure 15 for completing the task in Stride. Participants that completed the task correctly or mostly correct (Y, C) are colored green, while partial answers as black (P), and incorrect or no answers are colored red (N, X). These markings are visible on the y-axes near each participant's timeline.

When looking at the plots, one notable difference between the plots of participants that correctly completed the task and those that answered incorrectly is the number of fixations on the lines with the bug when a participant looks at the code, highlighted in red. Several participants that answered the task question correctly often had many fixations on the buggy lines of code towards the end of their session. For example, when looking at participants P214 and P217 in Figure 13, P214 spent most of the time looking at the lines of code with the bug in the second half of the plot, whereas P217 spent less time looking at the bug.

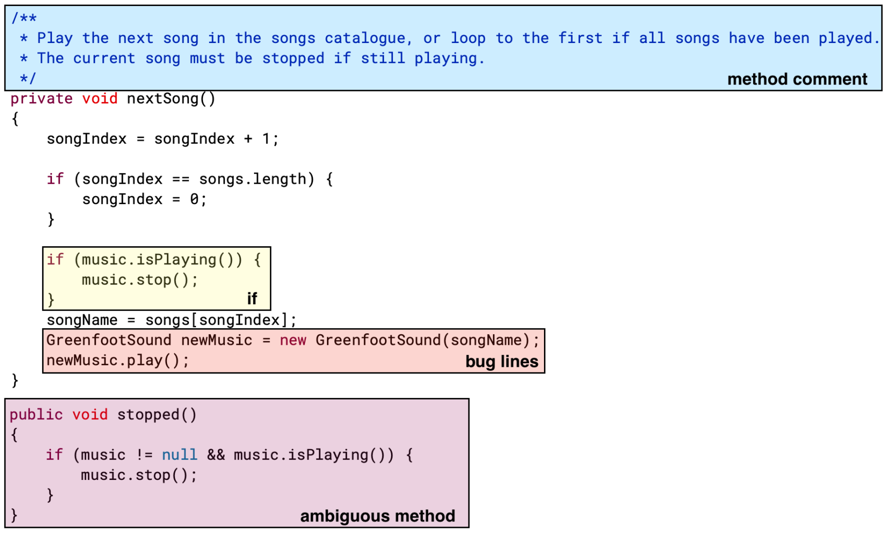
The Alpscarf plots also show how several participants spend a disproportionately large amount of time on the ambiguous method within the WelcomeWorld class called *stopped()* as shown in Figure 12, a method similarly named to the Greenfoot method *stop()*, despite *stopped()* being three lines long and not relevant to the bug. An example of this can be seen with participants P220 and P226 in Figure 13, where two participants spent a noticeable portion of their time reading the *stopped()* method and also give an incorrect answer.

When comparing the different code conditions presented to a participant, there does not appear to be a clear difference in what order participants read code, as there are individual differences within the same condition. One noticeable thing, however, is that all participants who answered the task question correctly were reading the buggy lines toward the end of the task. However, out of the 19 participants that completed Task 7 in Stride, only 4 answered correctly.

5 DISCUSSION AND IMPLICATIONS

5.1 Eye gaze metrics

The main eye gaze differences found in the Stride treatment were that the participants' gaze moved around less, they had shorter saccade lengths, longer fixation durations, fewer regressions, and



```
/**
 * Play the next song in the songs catalogue, or loop to the first if all songs have been played.
 * The current song must be stopped if still playing.
 */
private void nextSong()
{
    songIndex = songIndex + 1;

    if (songIndex == songs.length) {
        songIndex = 0;
    }

    if (music.isPlaying()) {
        music.stop();
    }
    songName = songs[songIndex];
    GreenfootSound newMusic = new GreenfootSound(songName);
    newMusic.play();
}
public void stopped()
{
    if (music != null && music.isPlaying()) {
        music.stop();
    }
}
```

Figure 12: Snippet of method containing bug and ambiguously named method from Task 7 (WelcomeWorld).

Table 7: Percentages of fixation count and fixation duration for each key part of Task 7 (WelcomeWorld) code that is used as an AOI.

Key Part	% Number of Fixations	% Fixation Duration
field declaration	2.37%	2.67%
field assignment	1.62%	1.58%
if / else	13.41%	10.82%
method comments	7.85%	9.26%
if	8.03%	9.83%
bug lines	5.34%	5.65%
call buggy method	1.44%	1.44%
ambiguous method	2.00%	1.83%

less line coverage compared to the other two treatments. Previous research, as reviewed by Debue and van de Leemput [15] and Godwin et al. [17], has suggested that longer fixations and shorter saccades are both associated with higher cognitive load. Thus our results might suggest that Stride has a higher cognitive load than the other interfaces. However, this is unlikely to be due to the scope highlighting, since Stride differed in both these metrics to the Java highlighting condition, which has almost identical scope highlighting. The difference is likely based on other features where Stride differs from Java (as visible in Figure 1), which could be any/all of:

- use of a variable-spaced font,
- less syntax highlighting, or
- reduced structural syntax (no semi-colons or curly brackets).

It is not possible to tell amongst our participants whether this is because these features inherently increase cognitive load, or whether it is related to the novelty of these interface features, as only one participant had seen Stride before. Note again that the scope highlighting, a novel and intrusive feature, does not seem to have produced equivalent effects (although it was not novel to all participants). There is also a perceptual phenomenon where when something is bound by a border, it can lock people perceptionally into the box they are in, making it less likely that people will jump to different areas and stay more constrained to each box, which could explain the short saccade length.

Alternatively, it is possible that the participants were simply better able to find the relevant lines of code in Stride and thus did not need to navigate around the code as much, meaning the gaze data indicates better focus. However, such a difference did not translate into any differences in task accuracy or speed, as we will discuss next.

5.2 Code comprehension performance

We found no difference in code comprehension task performance between the three different interfaces. Looking only at correct answers (to avoid issues with a speed-accuracy trade-off), we also found no difference in speed.

We believe this is a surprising result; the background highlighting that is present in the Java Highlight (JH) and Stride (S) conditions is particularly intrusive compared to the Java No highlight (JN) condition (see Figure 1). There are also several differences between JH and S: Stride has no curly brackets, no semi-colons, uses a different (non-monospace) font and has no syntax coloring. Furthermore, the eye tracking found many significant differences in gaze behavior. Yet, the conditions show no significant difference in performance.

The code in all three interfaces was presented with the same layout, so we could speculate that it is layout that is crucial, not the visual styling of color or background. However, Siegmund et al. [40] found in their study that disrupted layout did not affect comprehension performance, and Bauer et al. [5] found that the level (horizontal amount) of indentation also did not affect performance.

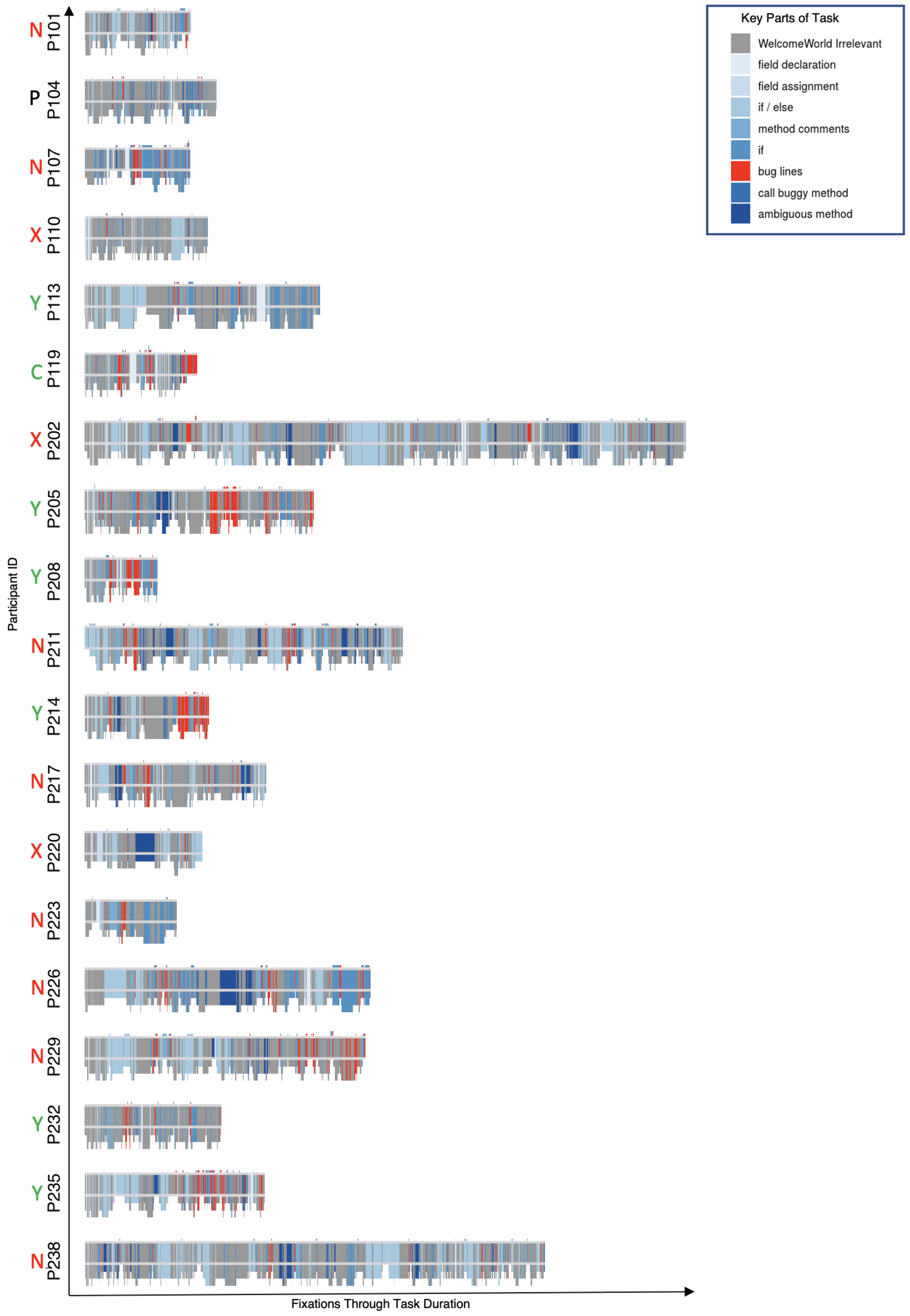


Figure 13: Alpscarf (not normalized) plot showing scanpath and proportional fixation duration on key parts of Java code with highlighting (JH) for each participant during Task 7 (WelcomeWorld), where buggy parts of code are shown in red.

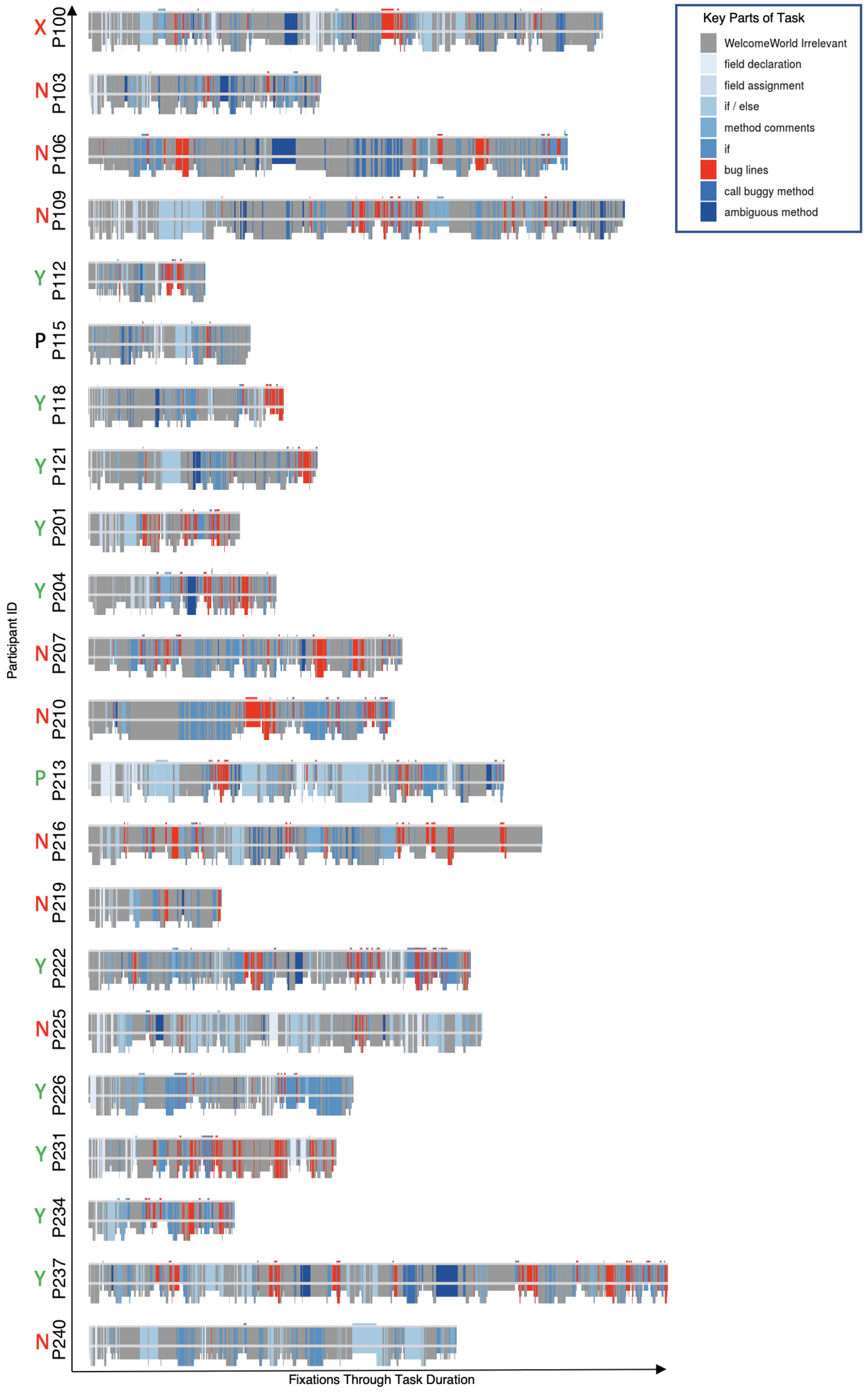


Figure 14: Alpscarf (not normalized) plot showing scanpath and proportional fixation duration on key parts of Java code without highlighting (JN) for each participant during Task 7 (WelcomeWorld), where buggy parts of code are shown in red.

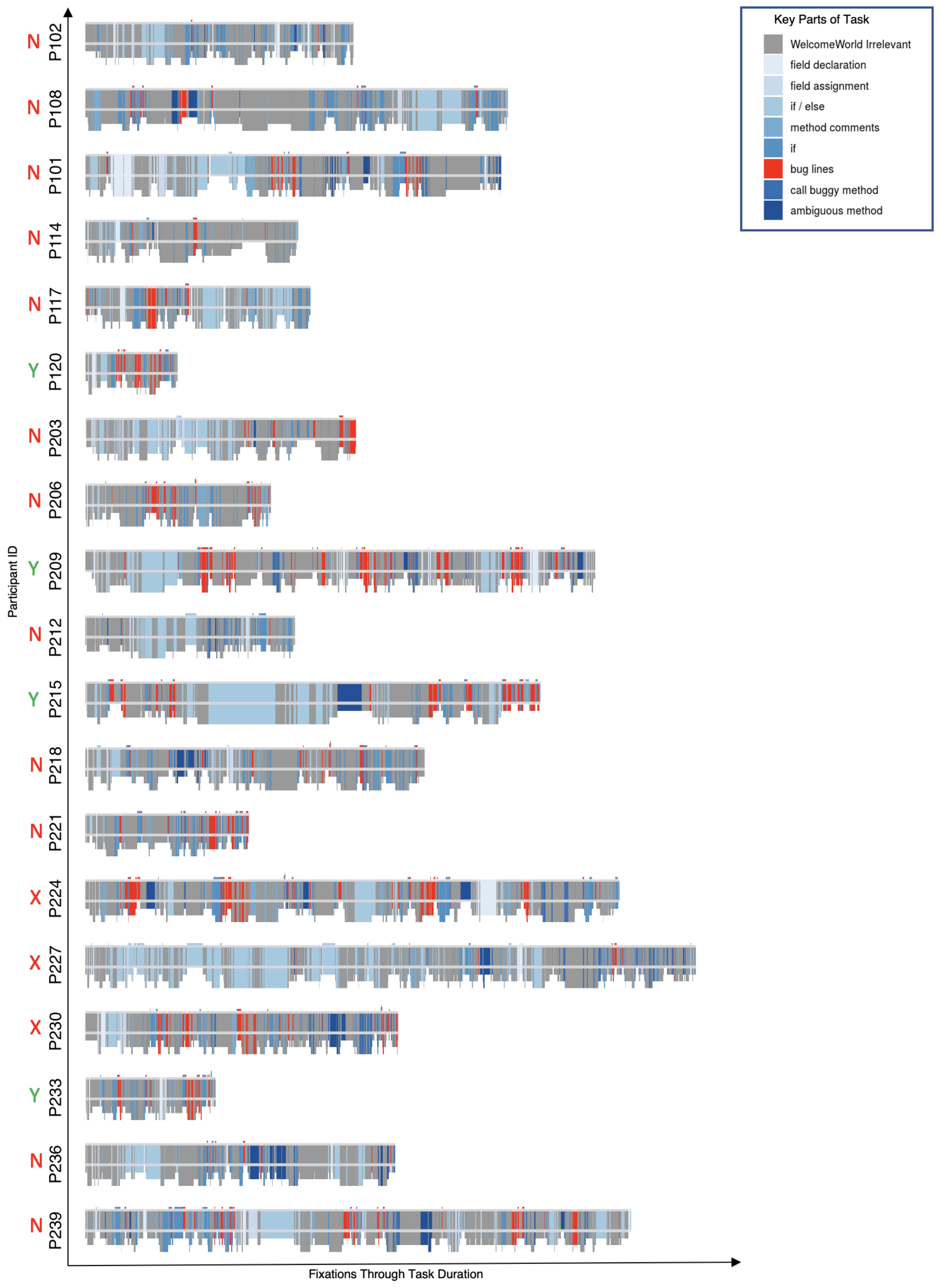


Figure 15: Alpscarf (not normalized) plot showing scanpath and proportional fixation duration on key parts of Stride code (S) for each participant during Task 7 (WelcomeWorld), where buggy parts of code are shown in red.

Combined with this study, it suggests that program comprehension is surprisingly invariant to presentation changes.

One potential factor could have been familiarity with the interface, but this made no difference when included in the analysis. It did not matter whether the participants had used scope highlighting regularly in a course, or saw it for the first time during the experiment: there was still no difference in performance.

There are two ways to act on such a difference. One is to view decorations as a purely personal or aesthetic choice: there is no difference in performance, so use them if you like them, or turn them off if you do not. Another interpretation is that having such an option is futile if it offers no benefit. However, it is important to understand that the decoration has a different purpose in Java compared to Stride. In Java, it is an optional addition that is drawn based on the canonical structure provided by the curly brackets in the text program. In Stride, it reflects the canonical structure of the program; the frames are the only indication of the program's structure. If they were not drawn, the usability of the program may suffer as the extent of the scopes would become less clear when editing the program – just as in block-based programming, if you did not draw outlines for the blocks or color them differently, it would become much harder to use.

Another important observation is that even though we did not find any performance-related differences, the eye-tracking linearity metrics did show some significant differences between the conditions. Put another way, two people might solve the same task correctly but one of them might find it much harder to do and the only way to see this is via the eye tracking metrics. This should also be taken with a grain of salt as we would like to point out that no one eye metric (such as linearity) should be interpreted by itself, rather a suite of metrics should be used to cross-check the results. In our study, these metrics found a consistent difference between Stride and the Java conditions.

6 CONCLUSIONS AND FUTURE WORK

In this study, novice programmers performed code comprehension tasks in one of three interfaces (Java with no scope highlighting, Java with scope highlighting, and Stride – see Figure 1), which differed in how the code was presented in terms of background highlighting, font, syntax highlighting and other decorations. We used eye-tracking hardware to measure participants' gaze behavior, and we recorded their spoken answers which were then graded for correctness and speed. We found several differences in eye behavior between Stride and the two Java conditions, but few between the two Java conditions. We found no difference between the interfaces in terms of answer correctness nor any difference in speed (which was examined for correct answers only).

Designers of program editors have added many different kinds of visual decorations to their editors, which are perceived by educators as efficient assistance for students to learn to program. Most of these additions have not been tested in research studies, but previous research has shown no behavioral effect of syntax highlighting [8, 19], amount of horizontal indentation [5], or misplaced indentation [40]. Nevertheless, we expected that being quite an intrusive change, the scope highlighting (especially combined with the lack of curly brackets and other changes in Stride) would show an effect on code reading in controlled conditions. This expectation was supported by a prior study by Weintrop et al. [48] which found a difference when presenting questions with or without scope highlighting in a paper-based exam. However, we found no effect on task performance in terms of correctness or speed. This matches the result of an observational study that looked at Java vs Stride in a school setting with teenagers and found no difference in performance on a series of code writing task across multiple lessons [11], although a prior study had found some differences in task completion rate [32]. Overall, it seems that high-level behavioral measures are often invariant to the way in which the code is presented, presumably because any high-level effect of presentation differences is small.

We did find a difference in eye gaze behavior, but primarily between Stride and the two Java conditions – the two Java conditions did not differ from each other on most metrics. This is an interesting finding because the scope highlighting background is different between the two Java conditions yet similar between Java with highlighting, and Stride. Therefore one possible explanation is that it is Stride's other features causing the difference: the changes in font, background highlighting, line spacing, or syntax highlighting. The pattern of the metric differences could indicate a higher cognitive load for Stride, but an alternative explanation is that participants needed to navigate around the code less in Stride. It cannot be determined in this study whether these differences are due to novelty of Stride's interface, as only one of the participants had seen Stride before.

As part of our future work, we plan to investigate the eye-tracking results of Stride further. To rule out the novelty factor, we would need to conduct a study with participants for whom Stride was not novel. Alternatively or additionally, we could perform experiments with finer-grained variations in the interface, adjusting the display of Stride or Java without highlighting in order to determine exactly which aspects of Stride's interface caused the difference in eye behavior.

ACKNOWLEDGMENTS

This work is supported in part by the US National Science Foundation under Grant Numbers CNS 18-55753 and CCF 18-55756. We are very grateful to Joshua Behler for his valuable help in adjusting iTrace Toolkit to meet the technical specifications incurred by the iTrace-BlueJ plugin (especially for Stride data). We are grateful to Andrei Balcau, supported by the King's College London Undergraduate Research Fellowship scheme, for doing early pilot tests of the technology. Finally, we are grateful to Jordan Cohen, who gave permission to use his space game Greenfoot scenario for this study.

REFERENCES

[1] Nahla J Abid, Bonita Sharif, Natalia Dragan, Hend Alrasheed, and Jonathan I Maletic. 2019. Developer Reading Behavior While Summarizing Java Methods: Size and Context Matters. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 384–395. https://doi.org/10.1109/ICSE.2019.00052

[2] Ahmad Zamzuri Mohamad Ali, Rahani Wahid, Khairulanuar Samsudin, and Muhammad Zaffwan Idris. 2013. Reading on the Computer Screen: Does Font Type Have Effects on Web Text Readability?. *International Education Studies* 6, 3 (2013), 26–35.

[3] Salwa Aljehane, Bonita Sharif, and Jonathan Maletic. 2021. Determining Differences in Reading Behavior Between Experts and Novices by Investigating Eye Movement on Source Code Constructs During a Bug Fixing Task. In *ACM Symposium on Eye Tracking Research and Applications (ETRA '21 Short Papers)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3448018.3457424

[4] Richard Andersson, Marcus Nyström, and Kenneth Holmqvist. 2010. Sampling frequency and eye-tracking measures: how speed affects durations, latencies, and more. *Journal of Eye Movement Research* 3, 3 SE - Articles (sep 2010). https://doi.org/10.16910/jemr.3.3.6

[5] Jennifer Bauer, Janet Siegmund, Norman Peitek, Johannes C. Hofmeister, and Sven Apel. 2019. Indentation: Simply a Matter of Style or Support for Program Comprehension?. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. 154–164. https://doi.org/10.1109/ICPC.2019.00033

[6] Roman Bednarik and Markku Tukiainen. 2006. An Eye-Tracking Methodology for Characterizing Program Comprehension Processes. In *Proceedings of the 2006 Symposium on Eye Tracking Research & Applications (ETRA '06)*. Association for Computing Machinery, New York, NY, USA, 125–132. https://doi.org/10.1145/1117309.1117356

[7] Roman Bednarik and Markku Tukiainen. 2008. Temporal Eye-Tracking Data: Evolution of Debugging Strategies with Multiple Representations. In *Proceedings of the 2008 Symposium on Eye Tracking Research & Applications (ETRA '08)*. Association for Computing Machinery, New York, NY, USA, 99–102. https://doi.org/10.1145/1344471.1344497

[8] Tanya Beelders and Jean-Pierre du Plessis. 2016. 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* (Johannesburg, South Africa) *(SAICSIT '16)*. Association for Computing Machinery, New York, NY, USA, Article 5, 10 pages. https://doi.org/10.1145/2987491.2987536

[9] Joshua Behler, Praxis Weston, Drew Guarnera, Bonita Sharif, and Jonathan Maletic. 2023. iTrace-Toolkit: A Pipeline for Analyzing Eye-Tracking Data of Software Engineering Studies. In *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering ICSE*. 4 pages to appear. https://doi.org/10.1109/ICSE-Companion58688.2023.00022

[10] Yoav Benjamini and Yosef Hochberg. 1995. Controlling the False Discovery Rate: A Practical and Powerful Approach to Multiple Testing. *Journal of the Royal Statistical Society. Series B (Methodological)* 57, 1 (1995), 289–300. http://www.jstor.org/stable/2346101

[11] Neil Brown, Charalampos Kyfonidis, Pierre Weill-Tessier, Brett Becker, Joe Dillane, and Michael Kölling. 2021. A Frame of Mind: Frame-Based vs. Text-Based Editing. In *Proceedings of the 2021 Conference on United Kingdom & Ireland Computing Education Research* (Glasgow, United Kingdom) *(UKICER '21)*. Association for Computing Machinery, New York, NY, USA, Article 2, 7 pages. https://doi.org/10.1145/3481282.3481286

[12] Teresa Busjahn, Roman Bednarik, Andrew Begel, Martha Crosby, James H Paterson, Carsten Schulte, Bonita Sharif, and Sascha Tamm. 2015. Eye Movements in Code Reading: Relaxing the Linear Order. In *2015 IEEE 23rd International Conference on Program Comprehension*. IEEE, 255–265. https://doi.org/10.1109/ICPC.2015.36

[13] Teresa Busjahn, Carsten Schulte, Bonita Sharif, Simon, Andrew Begel, Michael Hansen, Roman Bednarik, Paul Orlov, Petri Ihantola, Galina Shchekotova, and Maria Antropova. 2014. Eye Tracking in Computing Education. In *Proceedings of the Tenth Annual Conference on International Computing Education Research (ICER '14)*. Association for Computing Machinery, New York, NY, USA, 3–10. https://doi.org/10.1145/2632320.2632344

[14] Stéphane Conversy. 2014. Unifying Textual and Visual: A Theoretical Account of the Visual Perception of Programming Languages. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software* (Portland, Oregon, USA) *(Onward! 2014)*. Association for Computing Machinery, New York, NY, USA, 201–212. https://doi.org/10.1145/2661136.2661138

[15] Nicolas Debue and Cécile van de Leemput. 2014. What does germane load mean? An empirical contribution to the cognitive load theory. *Frontiers in Psychology* 5 (2014). https://doi.org/10.3389/fpsyg.2014.01099

[16] Rodrigo Duran, Albina Zavgorodniaia, and Juha Sorva. 2022. Cognitive Load Theory in Computing Education Research: A Review. *ACM Trans. Comput. Educ.* 22, 4, Article 40 (sep 2022), 27 pages. https://doi.org/10.1145/3483843

[17] Hayward J Godwin, Michael C Hout, Katrín J Alexdóttir, Stephen C Walenchok, and Anthony S Barnhart. 2021. Avoiding potential pitfalls in visual search and eye-movement experiments: A tutorial review. *Atten. Percept. Psychophys.* 83, 7 (Oct. 2021), 2753–2783.

[18] Drew T Guarnera, Corey A Bryant, Ashwin Mishra, Jonathan I Maletic, and Bonita Sharif. 2018. itrace: Eye tracking infrastructure for development environments. In *Proceedings of the 2018 ACM Symposium on Eye Tracking Research & Applications*. ACM, 105.

[19] Christoph Hannebauer, Marc Hesenius, and Volker Gruhn. 2018. Does syntax highlighting help programming novices? *Empirical Software Engineering* 23, 5, 2795–2828. https://doi.org/10.1007/s10664-017-9579-0

[20] Geoffrey L. Herman, Sofia Meyers, and Sarah-Elizabeth Deshaies. 2021. A Comparison of Novice Coders' Approaches to Reading Code: An Eye-tracking Study. *ASEE Annual Conference and Exposition, Conference Proceedings* (26 July 2021). https://doi.org/10.18260/1-2--36567 2021 ASEE Virtual Annual Conference, ASEE 2021 ; Conference date: 26-07-2021 Through 29-07-2021.

[21] Geoffrey L. Herman, Sofia Meyers, and Sarah-Elizabeth Deshaies. 2021. A Comparison of Novice Coders' Approaches to Reading Code: An Eye-tracking Study. *ASEE Annual Conference and Exposition, Conference Proceedings* (26 July 2021). https://doi.org/10.18260/1-2--36567 2021 ASEE Virtual Annual Conference, ASEE 2021 ; Conference date: 26-07-2021 Through 29-07-2021.

[22] Nina Hollender, Cristian Hofmann, Michael Deneke, and Bernhard Schmitz. 2010. Integrating cognitive load theory and concepts of human–computer interaction. *Computers in Human Behavior* 26, 6 (2010), 1278–1288. https://doi.org/10.1016/j.chb.2010.05.031 Online Interactivity: Role of Technology in Behavior Change.

[23] Kenneth Holmqvist and Richard Andersson. 2017. *Eye-tracking: A comprehensive guide to methods, paradigms and measures.* Oxford University Press.

[24] Sherri L Jackson. 2015. *Research methods and statistics: A critical thinking approach.* Cengage learning.

[25] Robert J K Jacob. 1991. The Use of Eye Movements in Human-computer Interaction Techniques: What You Look at is What You Get. *ACM Trans. Inf. Syst.* 9, 2 (apr 1991), 152–169. https://doi.org/10.1145/123078.128728

[26] Philipp Kather, Rodrigo Duran, and Jan Vahrenhold. 2021. Through (Tracking) Their Eyes: Abstraction and Complexity in Program Comprehension. *ACM Trans. Comput. Educ.* 22, 2, Article 17 (nov 2021), 33 pages. https://doi.org/10.1145/3480171

[27] Katja Kevic, Braden M Walters, Timothy R Shaffer, Bonita Sharif, David C Shepherd, and Thomas Fritz. 2015. Tracing Software Developers' Eyes and Interactions for Change Tasks. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*. Association for Computing Machinery, New York, NY, USA, 202–213. https://doi.org/10.1145/2786805.2786864

[28] Michael Kölling, Neil CC Brown, and Amjad Altadmri. 2017. Frame-based editing. *Journal of Visual Languages and Sentient Systems* 3 (2017), 40–67.

[29] Michael Kölling, Bruce Quig, Andrew Patterson, and John Rosenberg. 2003. The BlueJ system and its pedagogy. *Computer Science Education* 13, 4 (2003), 249–268.

[30] Ian McChesney and Raymond Bond. 2021. Eye Tracking Analysis of Code Layout, Crowding and Dyslexia - An Open Data Set. In *ACM Symposium on Eye Tracking Research and Applications (ETRA '21 Short Papers)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3448018.3457420

[31] Manuel Perea. 2013. Why does the APA recommend the use of serif fonts? *Psicothema* 25, 1 (2013), 13–17.

[32] Thomas W. Price, Neil C.C. Brown, Dragan Lipovac, Tiffany Barnes, and Michael Kölling. 2016. Evaluation of a Frame-Based Programming Editor. In *Proceedings of the 2016 ACM Conference on International Computing Education Research* (Melbourne, VIC, Australia) *(ICER '16)*. Association for Computing Machinery, New York, NY, USA, 33–42. https://doi.org/10.1145/2960310.2960319

[33] K Rayner. 1998. Eye movements in reading and information processing: 20 years of research. *Psychological bulletin* 124, 3 (nov 1998), 372–422. https://doi.org/10.1037/0033-2909.124.3.372

[34] Carsten Schulte. 2008. Block Model: An Educational Model of Program Comprehension as a Tool for a Scholarly Approach to Teaching. In *Proceedings of the Fourth International Workshop on Computing Education Research* (Sydney, Australia) *(ICER '08)*. Association for Computing Machinery, New York, NY, USA, 149–160. https://doi.org/10.1145/1404520.1404535

[35] Carsten Schulte, Tony Clear, Ahmad Taherkhani, Teresa Busjahn, and James H. Paterson. 2010. An Introduction to Program Comprehension for Computer Science Educators. In *Proceedings of the 2010 ITiCSE Working Group Reports* (Ankara, Turkey) *(ITiCSE-WGR '10)*. Association for Computing Machinery, New York, NY, USA, 65–86. https://doi.org/10.1145/1971681.1971687

[36] Timothy R. Shaffer, Jenna L. Wise, Braden M. Walters, Sebastian C. Müller, Michael Falcone, and Bonita Sharif. 2015. ITrace: Enabling Eye Tracking on Software Artifacts within the IDE to Support Software Engineering Tasks. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (Bergamo, Italy) *(ESEC/FSE 2015)*. Association for Computing Machinery, New York, NY, USA, 954–957. https://doi.org/10.1145/2786805.2803188

[37] Zohreh Sharafi, Bonita Sharif, Yann-Gaël Guéhéneuc, Andrew Begel, Roman Bednarik, and Martha E. Crosby. 2020. A practical guide on conducting eye tracking studies in software engineering. *Empir. Softw. Eng.* 25, 5 (2020), 3128–3174. https://doi.org/10.1007/s10664-020-09829-4

[38] Bonita Sharif, Michael Falcone, and Jonathan I Maletic. 2012. An Eye-Tracking Study on the Role of Scan Time in Finding Source Code Defects. In *Proceedings of the Symposium on Eye Tracking Research and Applications (ETRA '12)*. Association for Computing Machinery, New York, NY, USA, 381–384. https://doi.org/10.1145/2168556.2168642

[39] Bonita Sharif and Jonathan I Maletic. 2010. An Eye Tracking Study on camelCase and under_score Identifier Styles. In *2010 IEEE 18th International Conference on Program Comprehension*. IEEE, 196–205. https://doi.org/10.1109/ICPC.2010.41

[40] Janet Siegmund, Norman Peitek, Chris Parnin, Sven Apel, Johannes Hofmeister, Christian Kästner, Andrew Begel, Anja Bethmann, and André Brechmann. 2017. Measuring Neural Efficiency of Program Comprehension. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 140–150. https://doi.org/10.1145/3106237.3106268

[41] Steven J. Spencer, Christine Logel, and Paul G. Davies. 2016. Stereotype Threat. *Annual Review of Psychology* 67, 1 (2016), 415–437. https://doi.org/10.1146/annurev-psych-073115-103235 arXiv:https://doi.org/10.1146/annurev-psych-073115-103235 PMID: 26361054.

[42] Margaret-Anne Storey. 2006. Theories, tools and research methods in program comprehension: past, present and future. *Software Quality Journal* 14, 3 (01 Sep 2006), 187–208. https://doi.org/10.1007/s11219-006-9216-4

[43] Matúš Sulír, Michaela Bačíková, Sergej Chodarev, and Jaroslav Porubän. 2018. Visual augmentation of source code editors: A systematic mapping study. *Journal of Visual Languages & Computing* 49 (2018), 46–59. https://doi.org/10.1016/j.jvlc.2018.10.001

[44] Renske Talsma, Erik Barendsen, and Sjaak Smetsers. 2020. Analyzing the influence of block highlighting on beginning programmers' reading behavior using eye tracking. In *Proceedings of the 9th Computer Science Education Research Conference*. ACM, New York, NY, USA, 1–10. https://doi.org/10.1145/3442481.3442505

[45] Dmitry A. Tarasov, Alexander P. Sergeev, and Victor V. Filimonov. 2015. Legibility of Textbooks: A Literature Review. *Procedia - Social and Behavioral Sciences* 174 (2015), 1300–1308. https://doi.org/10.1016/j.sbspro.2015.01.751 International Conference on New Horizons in Education, INTE 2014, 25-27 June 2014, Paris, France.

[46] Hidetake Uwano, Masahide Nakamura, Akito Monden, and Ken-ichi Matsumoto. 2006. Analyzing Individual Performance of Source Code Review Using Reviewers' Eye Movement. In *Proceedings of the 2006 Symposium on Eye Tracking Research & Applications (ETRA '06)*. Association for Computing Machinery, New York, NY, USA, 133–140. https://doi.org/10.1145/1117309.1117357

[47] Adrian Voßkühler, Volkhard Nordmeier, Lars Kuchinke, and Arthur M Jacobs. 2008. OGAMA (Open Gaze and Mouse Analyzer): Open-source software designed to analyze eye and mouse movements in slideshow study designs. *Behavior Research Methods* 40, 4 (2008), 1150–1162. https://doi.org/10.3758/BRM.40.4.1150

[48] David Weintrop, Heather Killen, Talal Munzar, and Baker Franke. 2019. Block-Based Comprehension: Exploring and Explaining Student Outcomes from a Read-Only Block-Based Exam. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education* (Minneapolis, MN, USA) *(SIGCSE '19)*. Association for Computing Machinery, New York, NY, USA, 1218–1224. https://doi.org/10.1145/3287324.3287348

[49] Susan Wiedenbeck. 1986. Beacons in Computer Program Comprehension. *Int. J. Man Mach. Stud.* 25, 6 (1986), 697–709. https://doi.org/10.1016/S0020-7373(86)80083-9

[50] Susan Wiedenbeck and Nancy J. Evans. 1986. BEACONS IN PROGRAM COMPREHENSION. *SIGCHI Bull.* 18, 2 (oct 1986), 56–57. https://doi.org/10.1145/15683.1044090

[51] Chia-Kai Yang and Chat Wacharamanotham. 2018. Alpscarf: Augmenting Scarf Plots for Exploring Temporal Gaze Patterns. In *Extended Abstracts of the 2018 CHI Conference on Human Factors in Computing Systems* (Montreal QC, Canada) *(CHI EA '18)*. Association for Computing Machinery, New York, NY, USA, 1–6. https://doi.org/10.1145/3170427.3188490

[52] Alfred L Yarbus. 1967. *Eye Movements During Perception of Complex Objects*. Springer US, Boston, MA, 171–211. https://doi.org/10.1007/978-1-4899-5379-7_8