Automated Fixation Error Correction to Support Eye Tracking Studies on Source Code

ANONYMOUS AUTHOR(S)

A significant challenge in eye-tracking studies is detecting and fixing errors in data collection that happen for various reasons (drift, calibration issues, etc.). Many errors cannot be fully mitigated and require manual correction, which is intensively time-consuming, or automated correction. The work presented in this paper focuses on error correction, primarily on eye-tracking data on source code written in programming languages such as C++, Java, and C#. Many automated correction solutions are general-purpose, computationally inefficient, and use little information about the stimulus. To bridge this gap, we introduce *srcGaze*, a heuristic algorithm explicitly developed for correcting fixation gaze events in eye-tracking data from studies using source code as a stimulus. A golden dataset is manually constructed and verified to establish the heuristics. Results show a ≈40% improvement compared to no fixation correction. The approach has a multi-linear complexity and can correct over 44K fixations in approximately 6 seconds.

CCS Concepts: • **Hardware**; • **Software and its engineering**; • **Human-centered computing** → **Empirical studies in HCI**; **Heuristic evaluations**; • **Applied computing**;

Additional Key Words and Phrases: AOI Methods, Automated analysis methods, Tools for eye tracking analysis

ACM Reference Format:

Anonymous Author(s). 2025. Automated Fixation Error Correction to Support Eye Tracking Studies on Source Code. In *Proceedings of May 26–29, 2025 (ETRA '25).* ACM, New York, NY, USA, 16 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Eye tracking is an integral part of software engineering research [Obaidellah et al. 2018; Sharafi et al. 2015]. There are numerous efforts to understand the cognitive process of developers and their approach to program comprehension and maintenance tasks. This work shows much promise, but inherent challenges are associated with eye-tracking technology for software development research. Even when fixating on a single point, the human eye is never entirely still. Data produced by an eye tracker is constantly in flux, with data points scattered in the vicinity of an area of focus, which introduces a non-trivial amount of noise. To help mitigate this issue, gaze processing algorithms [Andersson et al. 2017; Liu et al. 2018; Olsen 2012; Olsson 2007; Salvucci and Goldberg 2000; Trapp et al. 2019; Zemblys et al. 2018] help filter and group gaze into an approximate region or fixation. While many techniques exist from scholarly work [Liu et al. 2018; Olsson 2007; Salvucci and Goldberg 2000; Zemblys et al. 2018] and proprietary commercial products [Olsen 2012], they are mainly geared toward activities that involve observing small static images, watching video, or reading short natural language prose. While these approaches are used with positive effect in those domain, research shows that developers do not read or view source code the same way as natural language prose [Binkley et al. 2013; Busjahn et al. 2015, 2014, 2011; Ko et al. 2006]. That is, reading source code is different than reading natural language text, hence gaze processing approaches need to be specifically developed for source code stimuli. Textual tokens in a source-code

Manuscript submitted to ACM

document have different semantic meanings that impact the order in which a developer reads the content, allowing them to skip large sections or review a specific section multiple times, in the context of data and control flow paths.

Collecting eye-tracking data has inherent imperfections. Issues arise from the participant, the experiment facilitator, the environment, the study design, and even the eye-tracking device. Any of these issues can cause erroneous data to be recorded. While some of these problem can be mitigated, error correction for eye-tracking data is necessary to ensure proper results are obtained during collection and subsequent analysis. This time-consuming task is often done manually during or after recording eye-tracking data [Busjahn 2021]. While some automated approaches exist for correcting eye-tracking data, most focus on prose as the intended stimulus rather than source code.

These challenges and limited support for source-code-based stimulus studies motivate the need for an algorithm designed explicitly for fixation correction when using source-code stimuli presented here as *srcGaze*. To study how developers address various software engineering tasks it is important to have fine grained Areas of Interests (AOIs). Thus, AOIs are normally at the *token level* for source code when studying developer eye movements. In the context of this work, source code tokens are anything a lexer (i.e., compiler) separates as a token in the programming language. This includes all keywords, identifier names, and symbols (e.g., "(", ")", ";", etc.). However, this fine grained level AOI dramatically increases the overall number of AOIs making it difficult and algorithmically complex to manage and process the data.

In the work presented here, we present a new fixation error correction method specifically for source code stimuli. The approach, *srcGaze*, leverages the syntactic information of tokens (AOIs) in the source code to ascertain heuristics on when and what tokens are most likely to be viewed. This results in a more accurate clustering of gaze data into fixations. The approach proposed here considers 1) spatial information for AOIs, 2) fixation locations in two-dimensional Euclidean space on the screen, and 3) the syntactic meaning of each token in the source code. To develop this approach we need to identify the most syntactically relevant tokens in source code (RQ1) and use these elements with positional information to perform a syntactic fixation correction for the source-code stimulus. Additionally, we compare the results to determine if the approach is improved (RQ2). Lastly, given the shear number of AOIs, the scalability of the approach (RQ3) is critical. A slow running algorithm will mean hours of processing versus seconds. Thus, this research addresses the following questions that build on top of each other as explained above.

- RQ1: What are the syntactic types of the tokens fixated on most frequently?
- RQ2: Are heuristics effective for enhancing fixation event correction?
- RQ3: Is using a heuristic approach at scale feasible?

The paper is organized as follows. Section 2 discusses related work on addressing errors in fixations. The data set used along with how we processed the data is presented in Section 3. This data is then used to construct a golden set and develop heuristics in Section 4. The *srcGaze* algorithm is presented in Section 5 with results given in Section 6. Threats to validity are given in Section 7 followed by conclusions and future work in Section 8.

2 Related Work

Eye-tracking devices are susceptible to error while recording eye movement data [Feit et al. 2017; Holmqvist et al. 2012; Hyrskykari 2006; Niehorster et al. 2018; Nyström et al. 2013; Palmer and Sharif 2016; Zhang and Hornof 2011]. These errors can come from participant movements, eye physiology, eyewear, makeup, and changes to the testing environment, such as ambient lighting. When areas of interest (AOIs) in a study are large, spatial errors are less problematic than those with smaller AOIs, such as when reading prose or source code [Busjahn 2021]. With small AOIs,

the margin for error narrows as spatial errors in gaze data can cause fixations to be attributed to incorrect textual tokens, negatively impacting analysis and results. Issues with errors in eye-tracking data are compounded as the data stream an eye-tracking device collects has a temporal element, as the degree of error can vary throughout a study due to drift[Busjahn 2021; Palmer and Sharif 2016].

While researchers attempt to mitigate the impact of potential error sources in eye-tracking studies, the collected data will always have a degree of error. One of the most common methods for addressing data errors in eye-tracking studies is manual correction [Busjahn 2021] using visualizations or assistive tools [Al Madi et al. 2025] to reposition gaze data for the stimulus. Manual validation is intensively time-consuming, and the time required for manual correction is not possible for studies at scale. Another critical issue is that manual validation can introduce subjectivity or bias from the data reviewers. While bias can be limited by having multiple reviewers correct each trial and find agreements between them, the time required for validation increases multiplicative.

The preferred approach to gaze event correction is automatic correction using various algorithmic means. Automated algorithms are faster than manual correction and produce consistent deterministic results. They can also be run as many times as needed while experimenting with threshold parameters for event detection, which is impossible to accomplish manually in a reasonable time frame. Automatic correction approaches in the research literature primarily focus on standard prose reading tasks[Frank et al. 2012; Hornof and Halverson 2002; Mishra, Abhijit et al. 2012]. These approaches operate under some assumptions about reading behavior in that domain and are not necessarily well suited for a reading style that leverages heavy use of skimming, regressions, and non-linear reading patterns that follow code execution order rather than reading line-by-line sequentially [Busjahn et al. 2015].

Compared to natural text reading, existing work for automated fixation correction on source code is limited. Lohmeier conducted a program comprehension study to model comprehension using anaphors in Java [Lohmeier 2015]. The approach to gaze event correction uses statistics and prior research about vision to develop an automated algorithm. This algorithm places a bounding box around potential fixation locations at the token level. An error function with parameters for horizontal and vertical offsets along with a linear factor applied to the vertical component. These parameters aim to find values such that the resulting error value is minimized for the fixation. This process is computationally intensive as these parameters must be checked in a *brute force* manner to test all possible combinations of values with the range defined by Lohmeier. To mitigate the computations, the horizontal parameter is chosen by Lohmeier so that only two of the three parameters will need to change for the brute force computation. Once correction offsets are found to minimize the error value, the fixation is positioned to the nearest target, and fixed fixations that are not near any targets are removed. The correction results are outlined with descriptive statistics and indicate that the error, as calculated by the function, was lowered.

The approach closest to our work is that of Palmer and Sharif [Palmer and Sharif 2016] who present an iterative method for automatically correcting fixations for source code-based stimuli. This approach focuses on line-based AOIs to correct vertical displacement in 68 trials. The first part of this method involves an algorithm to determine clusters of fixations using a time window parameter. When a cluster is detected, the fixations are flagged to be processed for correction. In the second stage of the algorithm, fixation clusters are scored based on the ratio of fixations in the cluster contained within an AOI out of the total number of fixations in the cluster. If a cluster has no points within an AOI, the cluster is moved until at least one fixation is in an AOI. Once a cluster touches at least one AOI, the Hill Climber algorithm is run on the cluster. The validation of this approach uses manually corrected fixations and reports an average accuracy of 89.78% among all the trials, with 90% of the corrections matching manual corrections. This approach primarily focuses on the vertical drift to correct fixation to the appropriate line. When token or “sub-line”

level AOIs are considered, the effectiveness becomes 59.47%. In the next section, we present our dataset and study methodology.

3 Data Set and Methodology

The Distributed Collection of Eye Movement Data in Programming dataset (hereby referred to as the EMIP dataset) released in 2019 [Bednarik et al. 2020] is the primary data source for this study. At the time of writing, this dataset is the most extensive public collection of eye-tracking data on source code stimulus, with 216 participants from 11 different institutions in 9 countries. Study participants are presented with two source code samples for program comprehension tasks written in Java, Scala, or Python, depending on their experience with a language. From the EMIP dataset package, we use the raw gaze data for each trial, stimulus images, and the textual version of the Java source code. This data was processed using the fixation algorithm from the EMIP toolkit [Al Madi et al. 2021] to generate fixations.

3.1 Privacy and Ethics

The eye tracking data used in this paper was collected and published by prior researchers in accordance with their ethics and institutional review board processes. The dataset was released publicly to be used by the eye tracking community. All data is provided de-identified to preserve confidentiality.

3.2 Preprocessing

Before we can start to address the research questions, we need to preprocess the EMIP dataset. This procedure is explained next. To use the stimulus effectively with the gaze data recorded from the eye tracker, areas of interest (AOIs) must be identified as potential locations for fixation events. The EMIP dataset package provides jpeg images of each stimulus shown to a participant and files with line and token bounding box positions in the stimulus directory. Each bounding box represents the x and y coordinates for two points on a stimulus image, and we can identify a rectangular region using the two points. Using the provided AOI bounding boxes, line bounding boxes do not contain leading white space, which is a part of each indented line, and token boundaries share a portion of any separating white space between characters. Token identification in this manner results in different types of tokens being grouped. Current research has not demonstrated an optimal AOI granularity level for source code tokens. To that end, the finest level of syntactic granularity supporting the broadest possible analysis is used in this work. Tokens are separated by white space and complex names when the dot operator ('.') accesses class attributes or methods. This granularity level makes it possible to reconstruct any higher-level syntactic constructs in the code. In previous work [Lohmeier 2015], a common approach to determining the AOI a fixation would hit is based on the center of the token. However, this introduces a bias when the tokens are all different sizes. Consider the example in Figure 1. There are two words: *dissertation* and *PhD* representing token AOIs. For each AOI, a red line splits the middle of each token, showing the center point along the vertical axis. Subsequently, a blue line represents the midpoint between the center of each token. Using these AOIs, any fixation that falls to the left of the blue line would be associated with the word *dissertation*, while any fixation right of the blue line would be associated with *PhD*. These regions show that while some fixations can fall on the "ion" in the word *dissertation*, they will still be associated with the smaller AOI of *PhD*.

While this may seem innocuous, the eye tracker used for the EMIP data collection reports an accuracy of <0.4° and a precision of ≈ 0.03° of visual angle. The human eye only has between 2° and 5° viewing angles with clear visual acuity [Duchowski 2017] within the eye's fovea region. Using the visual angle formula [Duchowski 2017], with participants seated 50 cm from the eye tracker [Bednarik et al. 2020] and the textual characters having an area of

Fig. 1. Demonstration of how using distances to center points for AOIs of varying sizes can result in mismatching. The two red lines in the image show the center of each word, and the blue line represents the exact midpoint between the centers of each word. All fixations that fall to the right of the blue line will still be associated with shorter fixation AOI *PhD* while still including characters from the longer word *dissertation*, introducing a bias to the smaller AOI.

≈3.8 cm (or 11 x 13 pixels), each character is ≈4°of visual angle. This means that the tracker can misreport actual gaze locations by a distance of around one character. Figure 1 demonstrates this bias of a few characters is significant enough to make a difference in fixation assignment to tokens. For this study, AOIs are assigned to every character in each stimulus file to limit the impact on token assignment and provide a more equitable comparison of closeness between tokens. Generation of the character-based bounding boxes is bootstrapped with Google's Tesseract OCR engine [Smith 2007] to locate the characters in the image, and then the bounding boxes are manually expanded.

Source code syntactic category information is obtained using srcML [Collard et al. 2011, 2013] on the Java source code stimulus (presently, the only language of the three that srcML supports). srcML is a robust and highly scalable infrastructure that transforms source code into an XML representation. The XML tags wrap around all of the tokens within the original source code and provide syntactic information. For example, an if statement would be surrounded by an <if> tag, the condition of the if statement would be further nested within a <condition> tag, and so on. Additionally, srcML preserves the entirety of the original code, including all comments and whitespace, and also provides starting and ending line and column information per syntactic tag, allowing us to identify not only what token or character was being viewed, but also what that token means within the context of source code. Along with the syntactic information generated by srcML, the *–position* argument to srcML provides all tokens' starting and ending line and column information. Combining the syntactic information, line and column position data, source code text, and the character AOI bounding boxes, a mapping document is created to merge this content together. See Figure 2 for an example of the srcML representation.

```
if(n < 10) {
    is_valid = true;
}

// ------------------------------------------------------------------------

<if_stmt><if>if<condition>(<expr><name>n</name> <operator>&lt;</operator> <literal
type="number">10</literal></expr>)</condition> <block>{<block_content>
    <expr_stmt><expr><name>is_valid</name> <operator>=</operator> <literal
type="boolean">true</literal></expr>;</expr_stmt>
</block_content>}</block></if></if_stmt>
```

Fig. 2. An example of Java code (top) and its srcML representation (below). The code consists of a single if statement which contains an assignment. Each tag wraps the tokens from the original code. Position information (line X column), within the file, can also be added to each tag via the tool.

3.3 Token Classification

To answer RQ1, we first need to determine token categories. Previous work by Busjahn et al. [Busjahn et al. 2014] categorizes Java source code tokens as presented in the first column of Table 1. The identifier category is the broadest and is defined as "sequences of letters and digits that denote [names of] variables, methods, etc." [Busjahn et al. 2014].

Implicitly expressed by this statement, type names are also included in this category as long as they are not reserved keywords in Java (e.g., int, double, etc.). Keywords are words reserved for the Java language and cannot be used in identifiers. In the Rectangle and Vehicle stimulus, the keywords used are *this*, *class*, *public*, *private*, *static*, *void*, *int*, *float*, *double*, *return*, *new*, *if*, and *else*. Literals are any value represented as a string in quotes like *"Audi"* from the Vehicle code example or constant numeric values like *10* used in Rectangle. There are other literal values, but strings and numeric values are the only types used in the code examples. Separators are explicitly defined in [Busjahn et al. 2014] as parenthesis (*( )*), curly braces (*{ }*), brackets (*[ ]*), period (.) comma (,), and semi-colon (;). Operators are described as "one or two characters," with an example using the addition (+) and increment operators (++). Given the operator definition and the explicit list of separators, the characters classified as operators present in the Rectangle and Vehicle code are =, >, +, -, and *. The last token category was added for this work and is simply whitespace, which serves as indentation and horizontal separation between tokens in lines of code. Only the space characters are considered in the whitespace category despite newline characters appearing in the document. Since source code lines contain either a curly brace or semicolon as a visible ending character, “empty” lines only have a new line and no content to read by a participant. For those reasons, newline characters are ignored as AOI target candidates.

The Busjahn categories are a solid start for this work and provide a baseline for categorizing the tokens. Using these categories from their 2011 work, they found that the identifier category dominated total dwell time (time spent fixating on a given token) at 53% [Busjahn et al. 2011]. Follow-up work in 2014 found that when normalized for token length, dwell time was nearly an even split with all token categories between ≈20% and ≈26% except for separators at ≈8%. This finding leads to an issue with this categorization of tokens when using token types from syntactic contexts for fixation correction. Given the similarity of dwell time over the smaller categories, it is likely worth expanding the token categories to cover more granularity.

Table 1. Source code token categories from Busjahn’s work alongside the token categories from srcML used by *srcGaze*. Similar token categories have been bolded.

Busjahn Token Categories	**srcML Token Categories**
identifier, **keyword**, **literal**, **operator**, specifier, **separator**, expression, **whitespace**	class-name, class-specifier, constructor-name, constructor-specifier, complex-name-expression, function-call, function-name, function-type-specifier, **keyword**, **literal**, name-declaration, name-expression, **operator**, **separator**, type-name, type-specifier, **whitespace**

In a recent study [Aljehane et al. 2021], srcML was used for token identification, revealing nine categories for tokens viewed by novice and expert developers. These categories include identifiers, method signatures, keywords, variable names, variable types, names in if statements, operators in if statements, names in else statements, and arguments. Notably, these categories tend to lose coverage when the hierarchical context is considered, such as observation of tokens specifically within control structures like if statements and else clauses. This, combined with the findings of Busjahn et al., underscores the need for more granularity in the token categories and expanded categories with less hierarchical context. Such an expansion could potentially enhance the accuracy of fixation correction, a vital goal of *srcGaze*. The second column of Table 1 presents the srcML token categories for the tokens based on the Rectangle and Vehicle source code stimulus.

Additionally, work from 2015 finds that method signatures are read frequently, especially by novice developers [Rodeghero et al. 2015]. The inclusion of specific token categories in *srcGaze* for elements of functions and constructors, which are more precise than the Busjahn categories, is a crucial step in improving fixation corrections. Even with the additional whitespace between the lines in the code stimulus, the textual tokens remain in close proximity. The incorporation of categories with enhanced granularity means the difference between correcting a gaze to a function name or a variable name within that function. This distinction is of utmost importance for the resolution of fixation corrections in eye-tracking studies.

The *srcGaze* categories are also more closely related to the programming language's syntax. For example, the period character is the "dot operator" or member-access operator in the Java programming language and not a separator as in prose text or the Busjahn categories. This is used to access members of a class, such as the x and y member variables for Rectangle or *producer* and *topSpeed* from Vehicle and also call functions from the method such as *width* from Rectangle or *accelerate* from Vehicle. In addition to the dot operator, when parenthesis controls the order of operations, they are considered operators acting on the result of an expression.

4 Golden Set

We construct and use a golden set for the following reasons. Fixations can vary significantly based on the algorithm, the parameters used, and the correction methods applied. This limits the ability to perform a one-to-one comparison of efficacy while working with the data. The purpose of the golden set is to use manual human fixation corrections as a benchmark for success. These corrections are used as a proxy for *ground truth* to compare against the automated approach provided by *srcGaze*. The rationale is to develop an approach that will correct data automatically and similar to a human reviewer. A golden set of fixation corrections is a set of changes agreed upon by a set of reviewers.

4.1 Generating the Golden Set

Multiple manual validators for each task/trial are needed to construct a golden set of fixations to reduce personal bias in the fixation correction process. We use a sample of 12,605 total fixations, over 47 tasks, from 28 EMIP trials as a subset to validate manually. Nine data validators were recruited from five different academic institutions to participate. Three of the data reviewers have performed eye-tracking research before this work. All validators have a firm understanding of source code, with two being undergraduates, four being graduate students, and three being instructors or professors in computer science fields of study. All validation participants are voluntary, and no compensation or rewards are provided to bias their work.

Each validator uses a custom-built fixation correction tool (see Figures 3 and 4) that presents the stimulus overlaid with the fixation to be corrected, two prior fixations, two subsequent fixations, numbering to determine fixation order, and saccadic movement lines to connect the fixations. The tool supports showing the nearest character AOI and the original fixation location to help decide where to position a fixation. Participants only need to click on the stimulus image where they believe the fixation should go, and the offset is recorded. Documentation with one example of fixation correction and instructions for using the tool are provided, but no additional hints or guidance to avoid potential bias. Figure 3 is a simple screenshot of the entire UI showing the fixation correction tool that annotators used. Figure 4 shows the stimulus with labels for what the visual elements are.

Two different validators manually correct each dataset, and fixation corrections from each pair of reviewers are assessed to ensure agreement to construct the golden set. Agreement in this context does not mean that participants must select the same AOI. Since the correction is done manually, and the AOIs are generated for each character in the

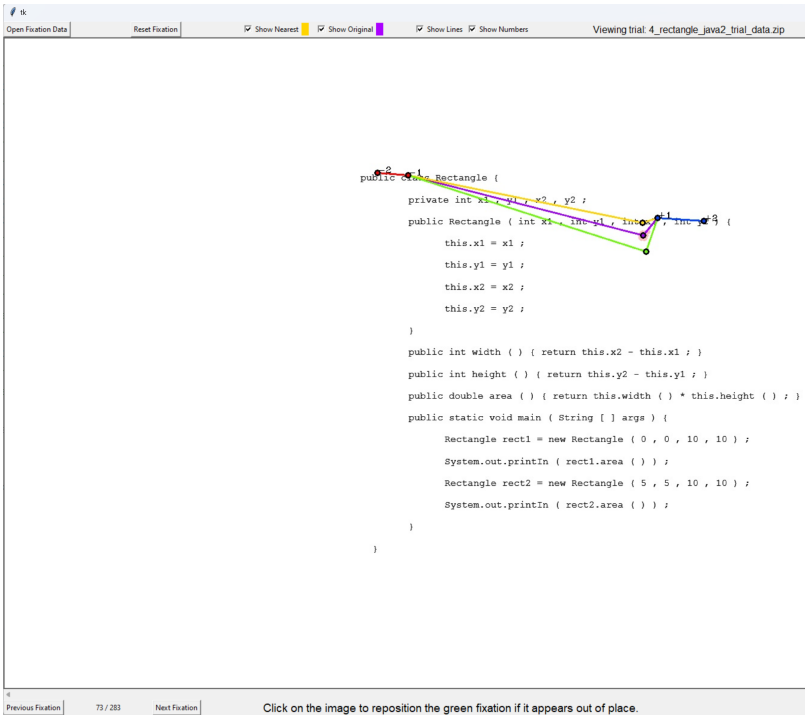

Fig. 3. Eye movement data represented within the visualization tool used for manual fixation correction. Light red dots are the raw gaze positions used to calculate a fixation. The red, green, purple, yellow, and blue dots are all fixations, with the lines between the dots showing saccadic motions representing the scan paths for the previous (red dots) and subsequent (blue dots) two fixations to the current fixation (the single green dot). The order of prior and subsequent fixations is shown using a +/- 1 and 2. The green dot is the current fixation that can be repositioned by clicking on the stimulus image. The green dot defaults to the initial location of the fixation calculated by the SMI tracker's dispersion algorithm. If the current fixation needs to be moved, the purple dot always represents the original location of the current fixation. The yellow dot can assist reviewers in showing the character token nearest to the original location of the current fixation. This feature is useful when, to the human eye, there may be two equally "close" token characters.

stimulus, it is reasonable to assume that misclicks within a token are possible. Additionally, the specific letter of a token is only partially representative of the content. Instead, participants must choose a character within the same token to agree. This is the only safe option with only two reviewers for each trial, as evaluating automated fixation corrections on data that does not have a consensus between human reviewers will be inconclusive. The results from the manual validation are processed automatically using a Python script for agreement, and subsequent golden set files are created for fixation data from each task. This data is used to help establish the syntactic-based correction heuristics for *srcGaze*, which is described next.

4.2 Golden Set Token Frequency - RQ1 Results

Utilizing the token categories described in Section 3.3 on the golden set of manually corrected fixations described above in Section 4.1, we can answer **RQ1: What are the syntactic types of the tokens fixated on most frequently?** The srcML-based categories occurring in the golden set are presented in Table 2. The *complex-name-expression* category includes all use of the *this* keyword in expressions using the dot operator (.) to access a class member. Considering the

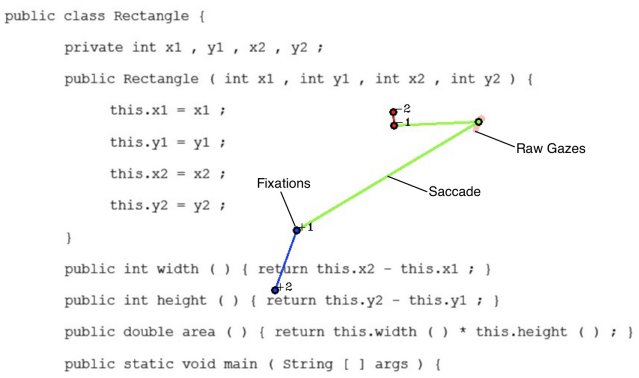

Fig. 4. A close-up of the fixation data overlaid on the image stimulus in the visualization tool to provide movement context during manual fixation correction. Light red dots are the raw gaze positions used to calculate a fixation. The red, green, and blue dots are all fixations, while the lines between the dots are saccadic motions representing the scan paths. The green dot is the fixation to be corrected and defaults to the location calculated by the SMI tracker's dispersion algorithm. The red dots and blue dots provide context representing the previous and subsequent two fixations with respect to the current fixation (the green dot). The order of prior and subsequent fixations is shown using a +/- 1 and 2. Users can click a location on the image stimulus to reposition the fixation, and the green scan path lines are redrawn to show the impact of the change.

Vehicle and Rectangle stimulus, this would include many expressions in the code. Variables not included in this category are names of the class members used in the *complex-name-expression*, variables in parameter lists, and declarations. Variables in parameter lists and declarations are classified by the *name-declaration* category, indicating the intent to create and use a new variable in the program. A point of interest is that the second most frequently fixated token category are tokens in the *name-declaration* category despite representing only ≈2% of all the tokens in the stimulus and ≈5% of the total characters. The *type-name* category indicates types defined by a programmer or by the Java programming language (e.g., primitive types like int, float, double, etc.) used for all program variables.

The following two categories, *function-call* and *function-name*, represent expressions that call a function (e.g., width or accelerate from Rectangle and Vehicle) and the name of the function in a method signature, respectively. These categories represent a divergent point in the *srcGaze* categories. The *function-call* category has nearly 10% of the fixations, while *function-name* drops to half that at about 5%. The assumption for this sudden drop is that for program comprehension activities, *function-call* represents that last category in the stimulus used repeatedly throughout the code. In the case of a function's name (remembering that methods and functions here are synonymous), prior research [Rodeghero et al. 2015] identifies that developers pay attention to the signature of a function. This finding aligns with that as once a developer identifies the name of a function and relates that to an associated purpose, the calls to the function being the context where it is used take precedence. Leveraging the benefits of our multi-granular token syntax information from srcML, we can search for all of our tokens that are enclosed in *function-signature* srcML element and confirm this with 1,138 fixations (≈10%) focused on that aspect of the code. While we view this finding in line with current research, it is worth noting that the code stimuli are short and only require the knowledge of at most two well-named functions. In contrast, a more extensive application may need more attention to this information due to the number of functions used in the code and the complexity of their interactions.

The trend of lower fixation counts and duration falls moving toward the end of the data in Table 2, but does illustrate a few general trends. Starting with the *function-name* category, only four categories, *function-name*, *name-expression*, *constructor-name*, and *class-name* are elements of the code that may represent content named by a developer. The remaining nine categories are all content defined by the Java programming language. The lower fixation counts can be the result of either this information being irrelevant to the task (likely with specifier categories) or familiarity with the tokens due to their prevalence within the Java language, like with the *keyword* and *operator* categories. The syntactic information collected from the golden set of fixations will serve as the heuristic component for the *srcGaze* approach.

Table 2. *srcGaze* token category occurrences in the manually corrected fixation golden set. This table shows similar results to the Busjahn categories for whitespace and "separator" like tokens have the lowest representation in the golden set. Looking at the token representation percentages, the table shows that the distribution is more balanced than with the Busjahn categories to have the intended effect of a finer granularity for the heuristic in *srcGaze*. The optimized weights are groupings that consider whether developers name the tokens, the frequency at which tokens in the token categories appear, the relative impact on the semantic behavior of the program, and token familiarity. These 30%, 15%, and 0% weights are based on the highest token occurrences in the groupings (*complex-name-expression* at ≈30% and *name-declaration* at ≈15%). The 0% weights are for tokens that are the most infrequent, familiar, or have the least impact on program semantics.

***srcGaze* Token Category**	**Fixation Count**	**Token %**	**Fixation Duration**	**Duration %**	**Optimized Weights**
complex-name-expression	3,804	30.18%	424,544	30.20%	30.00%
name-declaration	1,761	13.97%	198,492	14.12%	15.00%
type-name	1,730	13.72%	193,336	13.75%	15.00%
function-call	1,279	10.15%	137,140	9.76%	30.00%
function-name	660	5.24%	73,744	5.25%	30.00%
literal	647	5.13%	71,884	5.11%	30.00%
name-expression	454	3.60%	49,456	3.52%	30.00%
keyword	447	3.55%	51,696	3.68%	15.00%
operator	444	3.52%	49,564	3.53%	15.00%
function-type-specifier	369	2.93%	40,876	2.91%	0.00%
constructor-name	309	2.45%	34,476	2.45%	15.00%
class-name	216	1.71%	25,368	1.80%	15.00%
separator	142	1.13%	15,344	1.09%	0.00%
constructor-specifier	140	1.11%	15,644	1.11%	0.00%
type-specifier	104	0.83%	12,580	0.89%	0.00%
class-specifier	95	0.75%	11,144	0.79%	0.00%
whitespace	4	0.03%	424	0.03%	0.00%

5 srcGaze: Fixation Correction Algorithm

Recall that automated fixation correction approaches commonly utilize brute force [Lohmeier 2015; Nüssli 2011; Palmer and Sharif 2016] to repeatedly shift fixation positions either solely by x and y offsets or additional parameters for axis offsets [Lohmeier 2015]. These approaches aim to ultimately move the data such that they approach or enter a target AOI. While *srcGaze* also must locate target AOIs, compared to other methods, it does not use trial and error over x/y coordinates of an arbitrary range. Instead, *srcGaze* utilizes preconstructed mapping data to narrow the search space to the distance from the fixation's original position to the center of each AOI. The distance between the two points is a Euclidean distance calculation where distance (d) is equal to $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$. The initial implementation of

srcGaze performs this calculation from each fixation to the center of all character AOIs within the source code stimulus. The shortest distance is determined to be a candidate for the corrected AOI position. However, not all distances are treated equally.

As Table 2 shows, certain token types are more likely than others to interest programmers reading the source code for comprehension. To that end, each distance calculation is adjusted based on a token category weight. Weights are represented as a percentage of the overall distance. This percentage is removed from the original distance calculation. A simple example is to assume that a token category is weighted at 0.50 or 50%. Suppose the distance from the fixation to the center of an AOI with that token category is 100 pixels. In that case, the distance becomes $100 - (100 * .50)$ or 50 pixels instead, prioritizing this token over other tokens that may be closer but with a smaller weight. Algorithm 1 shows the pseudocode for *srcGaze*. Please refer to the supplementary package for the complete implementation of the algorithm. *srcGaze* iterates over all fixations (line 8) and each character AOIs present in the stimulus (line 9). The distance between each fixation and AOI is calculated using Euclidean distance (line 10). The weight of a given token offsets the distance from fixation to an AOI (lines 11 and 12). All AOIs will be checked as potential candidates to find the closest weighted match, the current "best" shortest distance, and the AOI mapping at that distance (lines 14-16). The result of this function is the AOI mapping with the closest weighted distance to the fixation (*nearest_aoi*). Runtimes or algorithmic performance of approaches to fixation correction utilizing an exhaustive search space, or brute force, are problematic for scalability as the size of the data increases. srcGaze has a runtime of $O(n * m)$ where n is the number of fixations and m is the number of AOIs, while brute force approaches run in $O(n^2)$ or $O(n^m)$. When considering small stimuli examples, the number of fixations and corrections needed will likely be smaller depending on the difficulty of the example. We plan to apply this approach to correct data from large-scale studies on code in open-source or industrial domains. Studies of this scale require extended periods of data collection, increasing the number of fixations and spanning the data collection across multiple files. At scale, the efficiency that the srcGaze approach provides is beneficial for timely data processing results. To demonstrate the performance of the approach, we provide an empirical runtime using the 44,184 fixations in the dataset where the algorithms can correct all the fixations in 6 seconds (see Section 6).

6 Results for RQ2 and RQ3

Evaluating *srcGaze*'s performance in terms of both its correctness when repositioning fixations and computational efficiency at this task helps answer RQ2 and RQ3:

- RQ2: Are heuristics effective for enhancing fixation event correction?
- RQ3: Is using a heuristic approach at scale feasible?

To answer RQ2, fixation corrections performed by *srcGaze* are compared against the golden set created by human reviewers. For this first test, *srcGaze* is configured to use the percentages in Table 2 based on the token target preferences in the golden set. For additional context, the golden set is compared to the results from the SMI fixation algorithm without correction and correction that only considers the closest token as the best AOI target. Table 3, with no correction performed on the fixation, performance is roughly 30% agreement with the golden set. Naive correction, which considers the closest token only without context, roughly doubles agreement with the golden set to nearly 66% agreement with the golden set. Applying *srcGaze* with syntactic context to the correction increases agreement to ≈72%. To put *srcGaze*'s results into perspective, the best approach for fixation correction at the token level was 59.47% agreement with a golden set by Palmer et al. However, even though a golden set was also used in that work, Palmer et al.'s approach intended to

Algorithm 1 ***srcGaze*** **Algorithm**

```
procedure FIXATION_CORRECTION
    uncorrected_fixations ← List of fixations from any event detection algorithm
    candidate_aois ← List of AOIs with token syntactic context
    syntactic_weights ← Collection of all weights used for fixation repositioning
    shortest_aoi_distance ← None
    nearest_aoi ← None

    for each fixation in uncorrected_fixations do
        for each aoi in candidate_aois do
            distance ← find_distance(fixation, aoi)
            weight ← get_token_weight(syntactic_weights, aoi)
            distance ← distance − (distance ∗ weight)

            if nearest_aoi == None OR shortest_aoi_distance > distance then
                shortest_aoi_distance ← distance
                nearest_aoi ← aoi
    return nearest_aoi
```

target fixation correction at line-based granularity, not sub-line or token-based level. The 59.47% agreement reported by Palmer et al. was to disclose at what granularity level their approach was most successful. Other than the agreement percentage, no additional details regarding token level correction are provided to provide a one-to-one comparison with srcGaze. [Palmer and Sharif 2016]. Additionally, the average and median distance required to correct a fixation using *srcGaze* is only 13 pixels and ≈10 pixels, respectively, from its original position. Recalling that a character in on the EMIP stimulus is roughly 11x13 pixels, meaning that *srcGaze* movements are not drastic and stay within a one to two-character distance depending on the direction of movement.

Table 3. Performance difference between the golden set and no fixation correction, nearest token correction, and *srcGaze* heuristic correction. Performance is worst without any correction in that only 32% of the fixation targets agree with the golden set. Correcting fixations to the nearest token increases agreement to nearly 65%, while using *srcGaze* with token heuristics for correction agreement rises to nearly 71%.

Correction Methods	Mean Correct	Median Correct	Total Correct
No Correction	31.24%	28.67%	31.84%
Naive Nearest AOI	64.36%	65.33%	65.77%
srcGaze	70.79%	71.12%	71.86%

After these results, further optimizations to the category weights were considered. Examination of the categories in Table 2 reveals that tokens named by programmers tend to get more fixations than keywords, operators, or other consistent syntactic tokens regularly used in the language. Additionally, some of the syntactic tokens provided by the language are shown to be of moderate interest, specifically operators and keywords. With this in mind, the last column of Table 2 shows new "optimized" category weights for *srcGaze*. The weights are based on the token category distributions from the corrected fixations in the golden set, and intuition-based groupings.

The first grouping is tokens named by developers, used often in the code, and impact program semantics. This grouping is given a weight based on the token category *complex-name-expression* with ≈30% of the corrected tokens

being from that category. Other tokens in this category are *literal*, *name-expression*, *function-call*, and *function-name* which meet that criteria. The next grouping is based on the second highest percentage which was *name-declaration* at ≈14%. This value was rounded up to 15% to have "even" weight values. A developer still names tokens in this group, which have semantic meaning in the code but appear less frequently or are familiar tokens from the language (e.g., keywords and operators). This group includes *class-name*, *constructor-name*, *keyword*, *operator*, and *type-name*. The remaining tokens were given 0% as they have the lowest correction rates, have little to no impact on program semantics, or occur infrequently. These are *class-specifier*, *constructor-specifier*, *function-type-specifier*, *separator*, *type-specifier*, and *whitespace*. With these new weights, *srcGaze*'s agreement increased to 72.94%. While this is a small increase, it illustrates that the original token weights and categories primarily represent the tokens expected to be manually viewed by researchers correcting fixation data on source code. Additionally, it shows room for improvement within the category weights and heuristic measures, providing a source of future work. With these findings, we can answer RQ2 that heuristics effectively enhance fixation event correction.

RQ3 is concerned with the computational efficiency of *srcGaze*, so it supports fixation correction of a more extensive study at scale. From a runtime performance perspective, this initial prototype performs a linear search over all AOI positions to find the shortest weighted distance candidate. Finding the closest fixation target requires checking each AOI and determining the distance to a fixation. This operation is performed once for each fixation, meaning this approach has a worst-case runtime complexity of $O(n * m)$ where m is the number of fixations and n is the number of AOIs. This is a significant improvement compared to the quadratic ($O(n^2)$) or exponential ($O(n^m)$) runtime of the brute force methods proposed in prior research. To further illustrate the efficiency of this approach, real-world measurements show performing fixation corrections on 44,184 fixations generated via the SMI dispersion fixation filter from all 125 trial tasks in the EMIP dataset takes only 6 seconds. This efficient performance is without any optimizations by limiting the AOI checks to lines closest to the fixation or any early stoppage of the algorithm when the AOI distances consistently increase over multiple lines (i.e., no other closer candidates exist). The performance demonstrated in both theoretical and practical runtimes for *srcGaze* confidently shows that the *srcGaze* approach is, in fact, efficient enough to be used on large datasets.

7 Threats to Validity

One limitation of this work is that the EMIP code stimuli are small single-file code examples. Since the EMIP stimulus is designed for a broad range of participant experience from novice to expert, some common language features such as control structures (while and for loops) are not used at all, and the occurrence of other programming constructs used may not be representative of the frequency at which they occur in more realistic software applications. This can impact the heuristics values used for *srcGaze*, be tuned too closely for the EMIP stimulus, and be insufficient to correct fixation data when larger-scale open or closed-source projects are used as stimulus. Given this limitation, all programming languages follow grammar rules. As such, regardless of the size of a source code application, tokens will still be used in similar ways. Additionally, some token categories could be consolidated, such as the *complex-name-expression* and *name-expression categories*, as they may better mimic the *function-call* and provide a more consistent categorization. Revisions to the categories and comparisons of the correction performance with other studies using different stimuli are planned for future work.

It is also important to note that fixations on tokens in the source code may depend on the type of task [Mansoor et al. 2024] under observation. Comprehending what a program can do and searching an application to locate sources of bugs or defects will undoubtedly overlap in the kinds of tokens viewed. Still, the degree of the fixations for these token

types may differ. The limitations mentioned above concerning category representation in the stimulus are partially addressed with configurable parameters, the same method utilized by the state-of-the-art event detection and correction algorithms. This allows for tuning of the heuristic values based on the token categories if later work finds more optimal or task-specific heuristic values for correction. While *srcGaze*'s improved runtime performance compared to brute force methods from state-of-the-art approaches holds, the performance concerning token-level fixation corrections may be impacted by the close association with the EMIP dataset used for the heuristics. While the quantity of data used for *srcGaze*'s heuristic preferences is likely adequate for similar use cases, and the distance for adjustments is well within reasonable margins, follow-up work with a study using a large-scale application is planned to tune the approach further. At this scale, a golden set may not be feasible, so crowd-sourcing methods will be investigated based on sampling the fixations and presenting participants with a few options of possible tokens rather than correcting the data themselves.

8 Conclusions and Future Work

The current state-of-the-art eye-tracking research on source code does gaze event correction as a post-processing phase to fixation identification with event detection algorithms. Most approaches to event correction are limited to brute force and manual correction methods. Researchers often use these methods with textual prose and not source code. This gap in support for gaze event correction on source code as an eye-tracking stimulus inspired the development of *srcGaze*, a syntactic-aware gaze event correction heuristic algorithm, which is the primary contribution of this work. Using a manually corrected set of over 12,000 fixations, *srcGaze* demonstrates exceptional improvements with a rate of nearly 73% agreement with a golden set of manual corrections. *srcGaze* also scales better than previous automated approaches that utilize brute force methods with quadratic ($O(n^2)$) or exponential ($O(n^m)$) runtime complexities as it supports a linear runtime ($O(n * m)$) based on the number of potential AOI targets (n) and the number of fixations to correct (m). This superior computational performance allows *srcGaze* to correct 44,184 fixations in only 6 seconds. *srcGaze*'s improved fixation correction accuracy will save countless person-hours for data correction and facilitate more extensive studies and rapid eye-tracking research analysis in the program comprehension community.

Extensions to the *srcGaze* heuristics are planned to incorporate movement trends in saccadic activity between fixations. Additionally, blended heuristic stages operating at multiple syntactic granularity levels could increase *srcGaze*'s awareness of movement patterns within source code structures such as loops, conditionals, etc. Concerning the heuristic categories, evaluation of a Bayesian model approach to token category weights will be evaluated along with new studies integrating larger, project-scale eye-tracking studies that fully represent typical programming language features (e.g., control structures like loops). Heuristics pulled from these new studies will improve the robustness of the heuristic categories and demonstrate *srcGaze*'s effectiveness when used on stimuli other than the EMIP dataset. Additionally, performing eye-tracking research exploring dynamic software development activities such as debugging and fixing source code issues or software evolution when developers create new features or re-architect the software using refactorings. While many studies focus on program comprehension for reading static source code, the support for interacting with the source code during eye-tracking with source code is minimal. Yet, these types of activities, especially for maintenance, can account for up to 80% of software development costs [Microsystems 1997]. These activities will likely have different viewing patterns that could impact the heuristics of *srcGaze*. Exploring these activities can help improve *srcGaze*'s performance and fill current research gaps.

References

Naser Al Madi, Drew Guarnera, Bonita Sharif, and Jonathan Maletic. 2021. EMIP Toolkit: A Python Library for Customized Post-processing of the Eye Movements in Programming Dataset. In *ACM Symposium on Eye Tracking Research and Applications (ETRA '21 Short Papers)*. Association for Computing Machinery, New York, NY, USA, 1–6. https://doi.org/10.1145/3448018.3457425

Naser Al Madi, Brett Torra, Yixin Li, and Najam Tariq. 2025. Combining automation and expertise: A semi-automated approach to correcting eye-tracking data in reading tasks. *Behavior Research Methods* 57, 2 (24 Jan 2025), 72. https://doi.org/10.3758/s13428-025-02597-3

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* (Virtual Event, Germany) *(ETRA '21 Short Papers)*. Association for Computing Machinery, New York, NY, USA, Article 30, 6 pages. https://doi.org/10.1145/3448018.3457424

Richard Andersson, Linnea Larsson, Kenneth Holmqvist, Martin Stridh, and Marcus Nyström. 2017. One algorithm to rule them all? An evaluation and discussion of ten eye movement event-detection algorithms. *Behavior Research Methods* 49, 2 (April 2017), 616–637. https://doi.org/10.3758/s13428-016-0738-9

Roman Bednarik, Teresa Busjahn, Agostino Gibaldi, Alireza Ahadi, Maria Bielikova, Martha Crosby, Kai Essig, Fabian Fagerholm, Ahmad Jbara, Raymond Lister, Pavel Orlov, James Paterson, Bonita Sharif, Teemu Sirkiä, Jan Stelovsky, Jozef Tvarozek, Hana Vrzakova, and Ian van der Linde. 2020. EMIP: The eye movements in programming dataset. *Science of Computer Programming* 198 (Oct. 2020), 102520. https://doi.org/10.1016/j.scico.2020.102520

Dave Binkley, Marcia Davis, Dawn Lawrie, Jonathan I. Maletic, Christopher Morrell, and Bonita Sharif. 2013. The Impact of Identifier Style on Effort and Comprehension. *Empirical Software Engineering* 18, 2 (April 2013), 219–276. https://doi.org/10.1007/s10664-012-9201-4

Teresa Busjahn. 2021. *Empirical analysis of eye movements during code reading: evaluation and development of methods.* Ph. D. Dissertation. Paderborn. https://nbn-resolving.org/urn:nbn:de:hbz:466:2-38777 Tag der Verteidigung: 04.03.2021.

T. Busjahn, R. Bednarik, A. Begel, M. Crosby, J. H. Paterson, C. Schulte, B. Sharif, and S. Tamm. 2015. Eye Movements in Code Reading: Relaxing the Linear Order. In *2015 IEEE 23rd International Conference on Program Comprehension*. IEEE, Florence, Italy, 255–265. https://doi.org/10.1109/ICPC.2015.36

Teresa Busjahn, Roman Bednarik, and Carsten Schulte. 2014. What Influences Dwell Time During Source Code Reading?: Analysis of Element Type and Frequency As Factors. In *Symposium on Eye Tracking Research and Applications (ETRA '14)*. ACM, Safety Harbor, Florida, USA, 335–338. https://doi.org/10.1145/2578153.2578211 event-place: Safety Harbor, Florida.

Teresa Busjahn, Carsten Schulte, and Andreas Busjahn. 2011. Analysis of Code Reading to Gain More Insight in Program Comprehension. In *11th Koli Calling International Conference on Computing Education Research (Koli Calling '11)*. ACM, Koli, Finland, 1–9. https://doi.org/10.1145/2094131.2094133 event-place: Koli, Finland.

M. L. Collard, M. J. Decker, and J. I. Maletic. 2011. Lightweight Transformation and Fact Extraction with the srcML Toolkit. In *2011 IEEE 11th International Working Conference on Source Code Analysis and Manipulation*. 173–184. https://doi.org/10.1109/SCAM.2011.19

M. L. Collard, M. J. Decker, and J. I. Maletic. 2013. srcML: An Infrastructure for the Exploration, Analysis, and Manipulation of Source Code: A Tool Demonstration. In *2013 IEEE International Conference on Software Maintenance*. 516–519. https://doi.org/10.1109/ICSM.2013.85

Andrew T. Duchowski. 2017. *Eye Tracking Methodology: Theory and Practice* (3rd ed.). Springer Publishing Company, Incorporated.

Anna Maria Feit, Shane Williams, Arturo Toledo, Ann Paradiso, Harish Kulkarni, Shaun Kane, and Meredith Ringel Morris. 2017. Toward Everyday Gaze Input: Accuracy and Precision of Eye Tracking and Implications for Design. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems* (Denver, Colorado, USA) *(CHI '17)*. Association for Computing Machinery, New York, NY, USA, 1118–1130. https://doi.org/10.1145/3025453.3025599

Michael C. Frank, Edward Vul, and Rebecca Saxe. 2012. Measuring the Development of Social Attention Using Free-Viewing. *Infancy* 17, 4 (2012), 355–375. https://doi.org/10.1111/j.1532-7078.2011.00086.x

Kenneth Holmqvist, Marcus Nyström, and Fiona Mulvey. 2012. Eye tracker data quality: what it is and how to measure it. In *Proceedings of the Symposium on Eye Tracking Research and Applications* (Santa Barbara, California) *(ETRA '12)*. Association for Computing Machinery, New York, NY, USA, 45–52. https://doi.org/10.1145/2168556.2168563

Anthony J. Hornof and Tim Halverson. 2002. Cleaning up systematic error in eye-tracking data by using required fixation locations. *Behavior Research Methods, Instruments, & Computers: A Journal of the Psychonomic Society, Inc* 34, 4 (Nov. 2002), 592–604. https://doi.org/10.3758/bf03195487

Aulikki Hyrskykari. 2006. Utilizing eye movements: Overcoming inaccuracy while tracking the focus of attention during reading. *Computers in Human Behavior* 22, 4 (2006), 657–671. https://doi.org/10.1016/j.chb.2005.12.013 Attention aware systems.

Andrew J. Ko, Brad A. Myers, Michael J. Coblenz, and Htet Htet Aung. 2006. An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks. *IEEE Transactions on Software Engineering* 32, 12 (Dec. 2006), 971–987. https://doi.org/10.1109/TSE.2006.116

Bo Liu, Qi-Chao Zhao, Yuan-Yuan Ren, Qing-Ju Wang, and Xue-Lian Zheng. 2018. An elaborate algorithm for automatic processing of eye movement data and identifying fixations in eye-tracking experiments. *Advances in Mechanical Engineering* 10, 5 (May 2018), 1687814018773678. https://doi.org/10.1177/1687814018773678

Sebastian Lohmeier. 2015. *Experimental Evaluation and Modelling of the Comprehension of Indirect Anaphors in a Programming Language.* Master's thesis. Technische Universiteit, Berlin, Germany. http://www.monochromata.de/master_thesis/ma1.0.pdf

Niloofar Mansoor, Cole S. Peterson, Michael D. Dodd, and Bonita Sharif. 2024. Assessing the Effect of Programming Language and Task Type on Eye Movements of Computer Science Students. *ACM Trans. Comput. Educ.* 24, 1, Article 2 (Jan. 2024), 38 pages. https://doi.org/10.1145/3632530

Sun Microsystems. 1997. Java Code Conventions. https://www.oracle.com/technetwork/java/codeconventions-150003.pdf. [Accessed 22-10-2024].

Mishra, Abhijit, Michael Carl, and Pushpak Bhattacharya. 2012. A heuristic-based approach for systematic error correction of gaze data for reading. In *First Workshop on Eye-tracking and Natural Language Processing*. Mumbai, India, 71–80. http://www.aclweb.org/anthology/W12-4906

Diederick C. Niehorster, Tim H. W. Cornelissen, Kenneth Holmqvist, Ignace T. C. Hooge, and Roy S. Hessels. 2018. What to expect from your remote eye-tracker when participants are unrestrained. *Behavior Research Methods* 50, 1 (01 Feb 2018), 213–227. https://doi.org/10.3758/s13428-017-0863-0

Marcus Nyström, Richard Andersson, Kenneth Holmqvist, and Joost van de Weijer. 2013. The influence of calibration method and eye physiology on eyetracking data quality. *Behavior Research Methods* 45, 1 (March 2013), 272–288. https://doi.org/10.3758/s13428-012-0247-4

Marc-Antoine Nüssli. 2011. *Dual Eye-Tracking Methods for the Study of Remote Collaborative Problem Solving*. Ph. D. Thesis. Ecole Polytechnique Federale De Lausanne, Lausanne, Switzerland. https://infoscience.epfl.ch/record/169609/files/EPFL_TH5232.pdf;

Unaizah Obaidellah, Mohammed Al Haek, and Peter C.-H. Cheng. 2018. A Survey on the Usage of Eye-Tracking in Computer Programming. *Comput. Surveys* 51, 1 (April 2018), 5:1–5:58. https://doi.org/10.1145/3145904

Anneli Olsen. 2012. *The Tobii I-VT Fixation Filter*. Technical Report. 21 pages. http://www.vinis.co.kr/ivt_filter.pdf

Pontus Olsson. 2007. *Real-time and Offline Filters for Eye Tracking*. Master's thesis. KTH Electrical Engineering, Stockholm, Sweden. https://pdfs.semanticscholar.org/4167/7844556582adc68a5a14dbb1cea0b28d9016.pdf

Christopher Palmer and Bonita Sharif. 2016. Towards automating fixation correction for source code. In *Proceedings of the Ninth Biennial ACM Symposium on Eye Tracking Research & Applications* (Charleston, South Carolina) *(ETRA '16)*. Association for Computing Machinery, New York, NY, USA, 65–68. https://doi.org/10.1145/2857491.2857544

P. Rodeghero, C. Liu, P. W. McBurney, and C. McMillan. 2015. An Eye-Tracking Study of Java Programmers and Application to Source Code Summarization. *IEEE Transactions on Software Engineering* 41, 11 (Nov. 2015), 1038–1054. https://doi.org/10.1109/TSE.2015.2442238

Dario D. Salvucci and Joseph H. Goldberg. 2000. Identifying Fixations and Saccades in Eye-tracking Protocols. In *2000 Symposium on Eye Tracking Research & Applications (ETRA '00)*. ACM, Palm Beach Gardens, Florida, USA, 71–78. https://doi.org/10.1145/355017.355028 event-place: Palm Beach Gardens, Florida, USA.

Zohreh Sharafi, Zéphyrin Soh, and Yann-Gaël Guéhéneuc. 2015. A systematic literature review on the usage of eye-tracking in software engineering. *Information and Software Technology* 67 (Nov. 2015), 79–107. https://doi.org/10.1016/j.infsof.2015.06.008

R. Smith. 2007. An Overview of the Tesseract OCR Engine. In *Ninth International Conference on Document Analysis and Recognition (ICDAR 2007)*, Vol. 2. 629–633. https://doi.org/10.1109/ICDAR.2007.4376991

Andrew C. Trapp, Wen Liu, and Soussan Djamasbi. 2019. Identifying Fixations in Gaze Data via Inner Density and Optimization. *INFORMS Journal on Computing* (April 2019). https://doi.org/10.1287/ijoc.2018.0859

Raimondas Zemblys, Diederick C. Niehorster, Oleg Komogortsev, and Kenneth Holmqvist. 2018. Using machine learning to detect events in eye-tracking data. *Behavior Research Methods* 50, 1 (Feb. 2018), 160–181. https://doi.org/10.3758/s13428-017-0860-3

Yunfeng Zhang and Anthony J. Hornof. 2011. Mode-of-disparities error correction of eye-tracking data. *Behavior Research Methods* 43, 3 (01 Sep 2011), 834–842. https://doi.org/10.3758/s13428-011-0073-0