iTraceVis: Visualizing Eye Movement Data Within Eclipse

Benjamin Clark and Bonita Sharif
Department of Computer Science and Information Systems
Youngstown State University
Youngstown, Ohio USA 44555
bjclark01@student.ysu.edu, bsharif@ysu.edu

***Abstract*—The paper presents iTraceVis, an eye tracking visualization component for iTrace, a gaze-aware Eclipse plugin. The visualization component is designed to work with data generated from iTrace after an eye tracking session. iTrace provides us with an automatic mapping of raw eye gaze on corresponding source code elements according to their hierarchy in the abstract syntax graph even in the presence of scrolling and context switching between files. This feature provides us with the ability to visualize eye tracking data in large source code files and is not just restricted to visualize only a method at a time i.e., something that fits on the screen. Due to the enormous size and richness of data collected from the eye tracker, visualizations help both the researcher and the developer to comprehend what transpired during an eye tracking session. iTraceVis currently supports four visualization views - heat map, gaze skyline, static gaze map, and dynamic gaze map. In order to determine the usefulness of the visualizations, we conduct a pilot user study with 10 senior students. We present an existing eye-tracking developer session to the study participants and ask them a series of questions while they interacted with iTraceVis and its various views in Eclipse. The results indicate that the visualizations do indeed help users understand the visualized session represented by the data, and also provide insight into where the visualizations can be improved as part of future work.**

***Index Terms*—eye-movement visualization, Eclipse, heat map, gaze plot, synchronized views**

I. Introduction

Researchers have successfully used eye trackers to better understand how people read natural language [1], compiler errors [2], source code [3], [4], comprehend diagrams [5], and process 3D visualizations [6]. Computer scientists use eye tracking devices to study how people interact with various artifacts such as graphical user interfaces and web pages with the goal of designing better ones. Eye trackers are a critical research tool in understanding how people understand visual stimuli [7]. There has been an increase in the use of eye trackers in the software engineering community since 2006 [8] mainly because eye trackers are becoming more accessible to the community both in terms of cost and practical use.

An eye tracker works by presenting stimuli such as an image or text on a computer screen and then using the data from cameras, determines the location (i.e., x,y-coordinate) the person (i.e., subject/developer) is looking at. There are a number of limitations to this technology. High quality eye trackers only work on some fixed stimuli (i.e., an image or text) that fits on the computer screen. Changes to the stimuli (screen), such as scrolling, present a very complex problem. Currently, there is no existing support (in any commercial eye tracking software) for a subject to interactively use an editor or switch between different files. Basically, existing systems do not keep track of what line in which file is present on the screen currently being viewed. While a fixed sized stimuli is sufficient to study how people read a sentence (or a few lines of source code), it is inadequate to study how programmers attempt to comprehend an entire software system. In order to solve this problem, we introduced an eye-aware Eclipse plugin namely, iTrace [9], [10]. It automatically maps eye gaze to source code elements in the Abstract Syntax Tree (AST). Using iTrace, a researcher is not limited to study how programmers comprehend short snippets of code. Instead, they can study the programmer developing software in their work environment while using program editors within associated integrated development environments (IDE) such as Eclipse. iTrace allows programmers to work naturally on software engineering tasks (e.g., reading code, writing code, fixing bugs, feature location, etc.). When iTrace is activated, eye tracking data is collected during the entire session along with the file and location data on many different software artifacts such as source code, bug reports, and requirements. At the time of this writing, iTrace supports code folding, context switching between files, and well as scrolling both horizontally and vertically.

In this paper, we present iTraceVis, a visualization component responsible for visualizing and comprehending eye tracking data generated from iTrace. In the first prototype release of the tool, we present four different views to visualize eye tracking data from a developer session. The main novelty of our tool is that it incorporates visualizing eye tracking data directly within the Integrated Development Environment (IDE) and is not restricted to small code snippets. We focus our visualizations on source code artifacts. This same concept can easily be extended to visualize eye gaze data on class diagrams, requirements, test cases, bug reports visited and so on. However, that is not the scope of this paper. A key feature of data visualizations is that the data needs to be broken down into manageable parts. Multiple different visualizations are often necessary to show all of the information that the data has to offer. Additionally, creating interactive visualization tools such as filtering the data allow the user to explore parts of

the data by cutting down on cognitive overload. We intend for researchers, educators and developers to be the main users of iTraceVis.

The rest of the paper is organized as follows. In the next two sections, we present a brief overview of eye tracking and iTrace in Sections II and III respectively. This overview is necessary as iTraceVis works with iTrace to visualize a developer's eye tracking session. Section IV presents the visualization module along with the architecture, design and views supported. Several usage scenarios are given in Section V. A preliminary study evaluating the visualization is presented in Section VI. Section VII discusses related work in visualizing eye tracking data and points out differences in our work. Finally, we conclude with a summary of the work along with avenues for future work in Section VIII.

II. Eye Tracking Overview

The underlying basis of an eye tracker is to capture various types of eye movements [11] that occur while a person physically gazes at an object of interest. Fixations and saccades are two types of eye movements [12]. A fixation is the stabilization of eyes on an object of interest for a certain period of time. Saccades are quick movements that move the eyes from one location to the next (i.e., refixates). A scan path is a directed path formed by saccades between fixations. The general consensus in the eye tracking research community is that the processing of information occurs during fixations, whereas, no such processing occurs during saccades [13]. The visual focus of the eyes on a particular location triggers certain mental processes in order to solve a given task [14]. Modern eye trackers are accurate to 0.5 degrees (0.25 inches. in diameter.) on the screen.

The eye fixation is a unit for measuring the participant's gaze. A fixation occurs when the participant focuses on an area for longer than some time threshold. Different tasks might require different lengths of time to quantify as a fixation. Eye tracking vendors recommend setting the threshold to 100 milliseconds. Sometimes a lower setting might be necessary especially in the case of fine-grained reading. The movement between fixations is a saccade, that is mainly used for navigation.

Areas of Interest (AOIs) are sections of the stimulus that the researcher is studying and concerned with. An AOI can be a variety of things, from a section of an image to a piece of text. In our case, since our stimulus is source code, our areas of interest are source code elements. For example, every line is an area of interest and in turn, parts of a source code line such as an identifier or method call are also areas of interest. Due to the richness in syntax and semantics of source code, we consider it to be a dense stimulus with hierarchical areas of interest that map directly to the abstract syntax graph.

An eye tracker generates a large amount of data per second based on its frequency. Each data point consists of many attributes including validity codes for the eyes. Depending on the eye tracker's frame rate (i.e., how many samples of data are collected per second) and what is actually tracked, the resulting data for a 20 minute session can easily go upwards of 50 Megabytes. The samples per second can range from 30 Hz. all the way up to 2000 Hz. Next, we present a brief overview of iTrace and how the data is generated and what it consists of.

III. iTrace Overview

iTrace is an eye tracking plugin for the Eclipse IDE. It performs three main tasks [9]. First, it captures the data generated by the eye tracker. Second, it finds the element of the UI that the gaze falls on. Finally, it processes this information towards some end goal. See Figure 1 for an illustration of how the data flows from the eyetracker into the output files. Trackers are represented by the *IEyeTracker* interface within iTrace, This class interfaces with the eye tracker's C++ SDK through the Java Native Interface. iTrace currently supports the trackers supported by the Tobii Analytics SDK and the Tobii EyeX SDK. Since the 2015 paper on iTrace [9], we have moved from the queue polling system for distributing the eye tracker's data throughout the system to an event based system that we consider to be more efficient.

There are four main classes involved in the processing of data: the $IEyeTracker$, $iTraceCore$, $IGazeHandler$, and the $ISolver$. The $IEyeTracker$ class receives data from the eye tracker and does some initial processing before posting an event containing the data. These events are then picked up by the core iTrace class which passes the data to the $IGazeHandler$ which in turn finds where the gaze falls within the editor widgets in Eclipse. The handlers return the processed data to the iTrace Core class which then sends the data to the to $ISolver$. iTrace saves its data to two files, a JSON file and an XML file. There is an $ISolver$ for both formats. When iTrace [9] initially debuted in 2015, it only tracked gazes within the Java editor widget. Since then support for various other widgets have been added. iTrace now collect data from the Project Explorer widget and the Browser widget on Bugzilla and Stack Overflow pages. There are $IGazeHandler$ classes for each of these. iTrace also provides cross hair functionality that draws a circle on screen following the user's gaze. This is used both to demonstrate the functionality of the eye tracker and verify the accuracy of the calibration (by asking the developer to verify if the cross hair points to where they look).

The gaze data files can be exported to JSON and/or XML formats. Each gaze response tuple from iTrace has a number of fields as shown in Table I. Every response is time-stamped and keeps track of the file viewed, the type of file, and the (x,y) position in the file that maps to the fully qualified name of the source code element. We record the fully qualified name to maintain unambiguity and scope of the data captured. The line and the column number of the source code element looked at is also recorded. Validity codes are also an important part of the data. A validity code data value of 0.0 indicates invalid data which implies that the gaze response in question needs to be discarded. The pupil size records pupil dilation in millimeters.

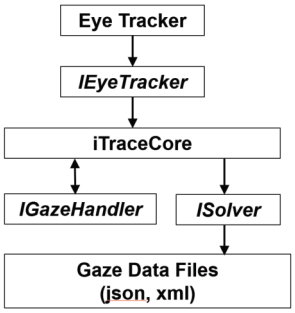

Fig. 1. A high-level view of the flow of gaze data through iTrace

After the raw gaze is collected, a fixation filter [15] is run on the raw data to provide us with fixation data. Besides the fields mentioned in Table I, a fixation also has a duration in milliseconds, telling how long a person looked at the element on the screen. The heatmap view takes into account the fixation duration to show how much longer a participant views source code elements. In this paper, we focus our iTraceVis visualizations to the file name, the x and y gaze position, the fully qualified names and the line and column numbers. The last column gives a sample of values for each of the attributes. The fully qualified name is a representation of the source code element (SCE) being viewed at the corresponding (x,y) gaze position. See Table II for an example of how a fully qualified name is represented for each gaze response. In the example, it states that the developer was looking at the for statement declaration. Note that we have intentionally left out low-level XML and JSON formatting while still presenting key aspects of the data recorded.

TABLE I
iTRACE DATA FILE FIELD DESCRIPTION

Field	Description	Example
Name of the file	This is the filename of the artifact being viewed	Quest.java
Type of the file	Is the file a requirements document or source code file. In this paper, we are only concerned with source code	java
x gaze position	The x-coordinate mapping of the eye gaze on the screen. (0,0) is on the top left.	256
y gaze position	The y-coordinate mapping of the eye gaze on the screen. (0,0) is on the top left.	499
validity codes	The codes tell us if the eye is valid or not.	1.0
pupil size	The size of the pupil in millimeters	2.6
Timestamp	Internal time stamp (an integer)	2017-04-03T10 :44:24.105-05:00
Session time	Number of nanoseconds from the beginning of a session.	2549909551
File path	Path of the file	C:/Test/src/Quest.java
Line number	Number of the line being viewed in the editor	2
Column number	Column number of the line being viewed in the editor	18
Fully qualified name	Source code entities mapped to the (x gaze position, y gaze position)	See Table II

TABLE II
AN EXAMPLE OF A FULLY QUALIFIED NAME REPRESENTING THE SOURCE CODE ENTITY (SCE) IN THE DATA FILE FIELD DESCRIPTION

Field	Description	Example
Name of the SCE	Name of the SCE from the AST	EnhancedForStatement-l5c2
Type of SCE	Type of source code entity described i.e. for loop; method; class	FOR STATEMENT
How	How the SCE is being used i.e. is it a method declaration or a method call	DECLARE
Total length	Length of SCE in characters	67
Start line	First line of the SCE	5
End line	Last line of the SCE	7
Start Column	First Column of SCE	2
End Column	Last Column of SCE	3

IV. THE VISUALIZATION MODULE

This section describes the architecture and design methodology of iTraceVis. The visualization views currently supported are also presented. Finally, a summary of all the views and the data they tap into is shown. iTraceVis is implemented in Java and is directly accessible within iTrace. The prototype implementation is available at *https://github.com/SERESLab/iTrace-Archive/tree/issue88-vis*.

A. Design Methodology

The visualizations included with iTraceVis are all two dimensional. Since the data we are using has three or more dimensions we need multiple visualizations. This means that all of our visualizations will have shortcomings. We have three guiding principles to our designs. First, we can mitigate the effect of the shortcomings by augmenting the visualizations. This means adding features such as color to the visualization to add context to one of the variables it does not natively display. Second, we create associations between the visualizations to allow the user to use them in tandem and get a better picture of the data. Finally, interactive visualizations allow the user to explore the data at their own pace to get an understanding of what happened during the session.

We will use a running example to explain and demonstrate the visualizations of iTraceVis. For the sake of brevity, this example uses data from a session lasting about 30 seconds which generated 1100 gaze responses in our data file.

B. Visualization Views

The various views of iTraceVis are shown in Figure 2. The heat map and gaze skyline view are visible along with the iTrace and iTraceVis controls. We will now discuss each of the visualization views iTraceVis currently supports.

1) Heat Map: The first visualization view is the heat map which is designed to show which tokens the participant looked at the most during the session. It highlights the tokens directly

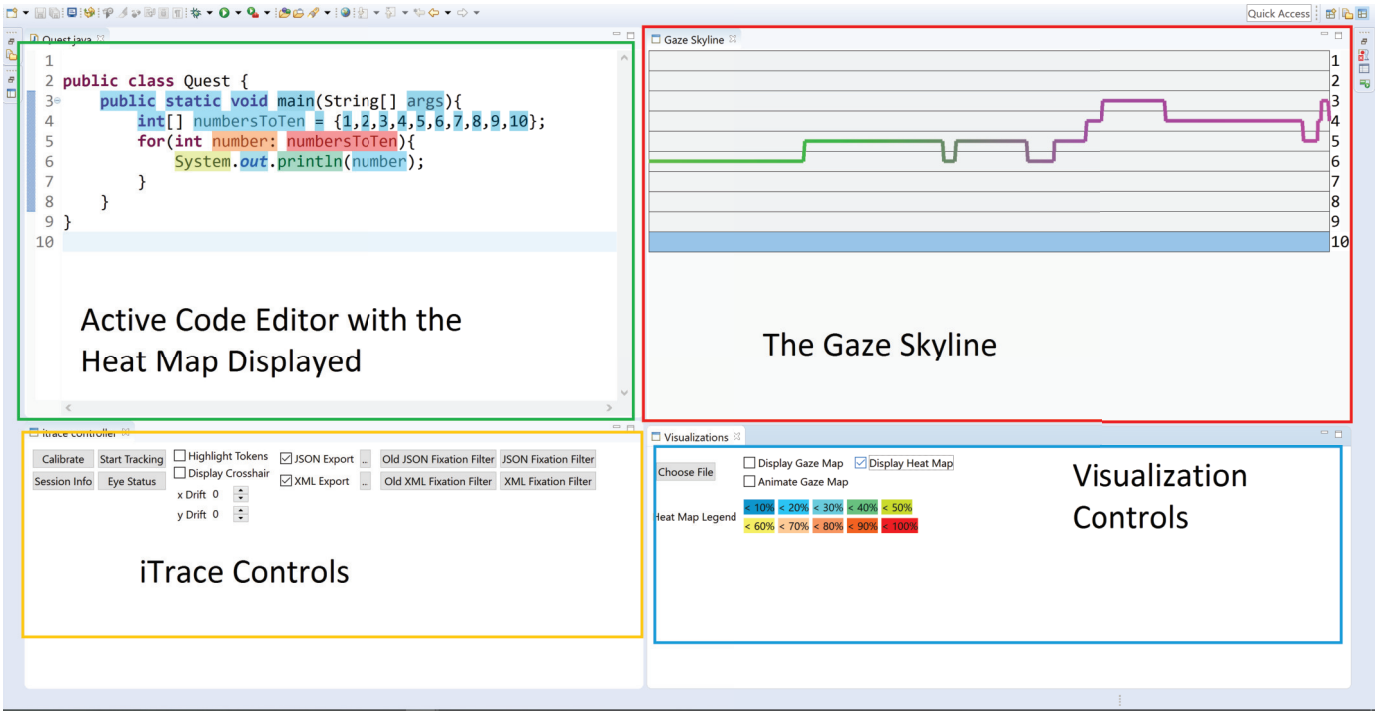

Fig. 2. A snapshot of iTraceVis enabled in Eclipse and iTrace. The active code editor is shown on the top left with the heat map displayed. The iTrace controller is shown on the bottom left used to record a developer session. The gaze skyline view is shown on the top right which is directly mapped to the active code editor. Finally, the visualization controls are visible on the lower right.

in the active editor with different colors based on how often a gaze fell onto it. The colors form a gradient changing on every tenth percentile, with red being the highest percentile, yellow being the middle percentile, and blue being the lowest. The tokens are organized into percentiles based on how many gazes fell onto them. If a token had more gazes fall on it it will be in a higher percentile. See Figure 2 for an example of how a heatmap is displayed in the active editor on a simple Java class. The legend (red indicates most viewed) is shown to the right in the visualization controls.

2) Gaze Skyline: The second visualization is the gaze skyline. It displays the line position of the developers gaze over time. It is designed to be used adjacent to the active editor. When used this way, the line section of the Gaze Skyline matches exactly to the lines in the editor. There is a one-to-one correspondence between the lines in the editor and the lines in the gaze skyline. The gaze is represented by a line drawn over the line sections with vertical movement representing changes in line position and horizontal movement representing time. See Figure 3 for an example of how a developer read the lines of code as time progressed.

The leftmost side of the gaze skyline represents the beginning of the session and the rightmost side the end of the session. This visualization is well equipped to show which line the developer was looking at during a given time window. However, it gives no information of where the developer was looking on that line. The gaze skyline is tied the active editor to function properly. Scrolling in the active editor causes the gaze skyline to scroll. The same applies to the zooming function. The selected line in the editor is also highlighted (line 4 in Figure 3) in the Gaze Skyline. This ensures that the information shown in the Gaze Skyline corresponds to the code shown in the editor. In this particular example, we can see that the developer read the main method top to bottom first but spent the most amount of time on line 4 during the middle of their session. The developer also switched between lines 4 and 5 and stayed between these two lines longer before moving to line 6 where the number is printed, and then up towards the top of the class. We also see that they did not look at the last three closing braces.

3) Static Gaze Map: The static gaze map serves as a complement to the gaze skyline. It traces the path of the gaze's line and column position onto the editor. While the gaze skyline displays information about vertical line position and time, the static gaze map shows information about the horizontal and vertical position. The static gaze map is shown in Figure 4. This visualization is convoluted and difficult to understand without modification. We can add color to it to help give context to time and make it easier to understand. The color gradient added changes over time helping the user to understand when the gaze was recorded. See Figure 5 for the improved static gaze map. Note that interactively specifying a time window to view the static gaze map with color will unclutter the display even further and provide a more focused view in time.

This augmentation of the static gaze map is an improvement

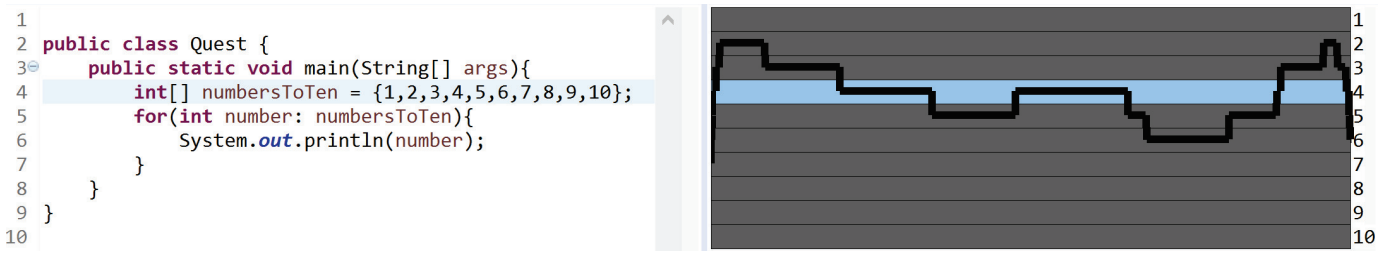

Fig. 3. Gaze skyline view showing the active editor synced to it.

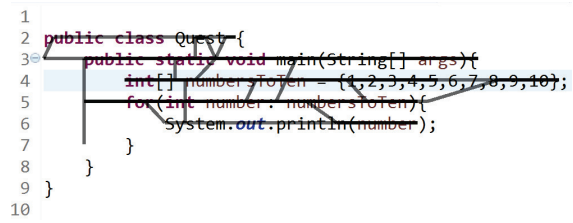

Fig. 4. Static gaze map view

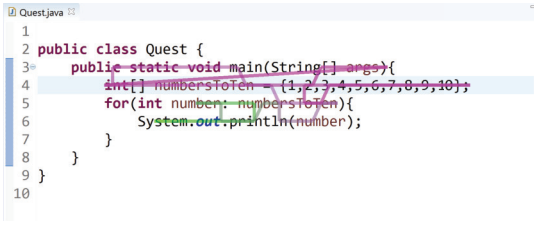

Fig. 5. Static gaze map view with color

but there is still more we can improve on. Since gaze map struggles to show the time aspect of the developer session, we can create an association to the gaze skyline which has direct representation of time. The way we accomplish this is by adding the color gradient we used with gaze map to the gaze skyline, as shown in Figure 6.

Having this association allows the user to look at a segment of the gaze map and find its color in the gaze skyline telling them when the gaze occurred during the session. Additionally, if the user looks at the gaze skyline and wants to know where the developer was looking at on a specific line they can turn to the gaze map and find the corresponding color telling them where the developer looked at. For example, in this particular developer session, we can see that the gaze is focused on lines 3, 4, and 5 at the end of the session as shown in pink in both the gaze skyline view and the gaze map view. The gradient of colors used is fine grained enough that it would need to be a very long session before colors overlap. One way to avoid this ambiguity if it arises is to view and filter the gaze data in time windows. We have planned to add time window viewing functionality to the next release of iTraceVis.

4) Dynamic Gaze Map: The dynamic gaze map like the static gaze map displays the horizontal and vertical position of the gaze directly in the editor. However, the dynamic gaze map is animated so that it only displays a sliding window of the data rather than all the data in its entirety. Showing the gaze data in a real-time playback like this is a very powerful tool for understanding the actions of the developer during the session. The dynamic gaze map is temporal in nature which will require the user to study chunks of the data at their own pace by pausing the animated dynamic gaze map as needed. It has the same color gradient that the static gaze map has allowing it to be used with the gaze skyline. There is also a vertical black bar the scrolls horizontally across the gaze skyline showing where during the session the dynamic gaze map is currently drawing in the editor. The dynamic gaze map view is aware of the fact that iTrace records eye tracking data on multiple source files as the developer navigates and reads code. The dynamic gaze map view moves between files for playback of the gaze data as and when it occurs in the data file.

C. Summary of iTraceVis Architecture

The visualization component is integrated directly into the iTrace plugin. See Figure 7 for a high level overview of iTraceVis. Instances of the heat map and the gaze map are paired to each of the open editors in Eclipse. The gaze and heat maps are created upon activation of the editor and stored in the iTraceCore class. There must be a gaze and heat map for each open editor so that the visualizations can be drawn independently allowing them to be displayed in multiple editors at once. The gaze skyline is its own Eclipse widget. It is tied to the active editor so that scrolling and zooming in the active editor causes the skyline to scroll or zoom correspondingly. This only allows for one gaze skyline to be shown at a time. The gaze skyline supports code folding but the heat map and the gaze maps do not. The visualization component is controlled from its own control panel within the Eclipse IDE. When data is loaded into iTrace it is parsed and filtered by a data parser class. A condensed version of the data is then stored in the central iTraceCore class for use by the visualizations.

D. Summary of iTraceVis Views

The most common types of visualizations for eye tracking data are the heatmap and the gaze map plot [16]–[18]. In addition, for source code, the line graph (shown as a gaze skyline view in iTraceVis) makes sense as it shows how a person reads lines of code (this is the gaze map plot but shown

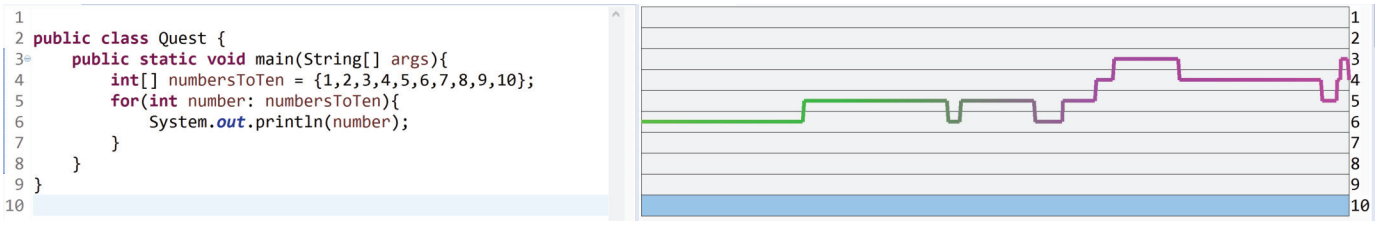

Fig. 6. Gaze skyline view and gaze map view shown together with color.

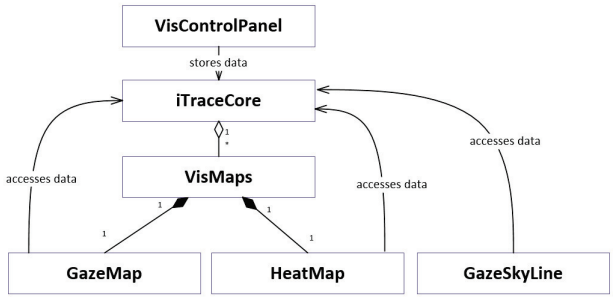

Fig. 7. High level class diagram for iTraceVis

in time). Each of these views can be shown with respect to the entire stimulus or with respect to a specific area of interest.

In summary, our visualization views when combined show all three aspects of the iTrace data file: horizontal position, vertical position, and time. The heat map shows which tokens were looked at most. The gaze skyline shows which line the developer was viewing at a certain time. It is complemented by the static gaze map which traces the horizontal and vertical position directly into the editor. The static gaze map and gaze skyline are associated through a color gradient so that the users can find when a section of the static gaze map was recorded during a session, by finding the color of that section of the static gaze map on the gaze skyline. The users can also use this association to find where on a line the developer was looking at for a section of the gaze skyline. Finally, the dynamic gaze map replays the session directly in the editor. This gives the user the ability see the session as it happened. If the developer session involves multiple files, the gaze map will move to the corresponding file and playback the gaze data in the appropriate and correct editor. See Table III for a overview of how each view works. Since the heat map shows cumulative data over time, we categorize it as aggregate in time. The gaze skyline moves sequentially through the session in time and makes use of the lines looked at. Finally, the dynamic gaze map plays back both lines looked at and what was looked at on each line. We note that the vertical position i.e., lines looked at is always used in each of the views. The horizontal position is not used in the gaze skyline view as it only deals with looking at line switches sequentially in time.

V. Usage Scenarios

We present several usage scenarios for iTraceVis. We see three types of stakeholders for iTraceVis: the researcher,

TABLE III
A SUMMARY COMPARISON OF THE FOUR VISUALIZATION VIEWS

Views \ Attribute	Horizontal Position	Vertical Position	Time
Heat Map	Yes	Yes	Aggregate
Gaze Skyline	No	Yes	Sequential
Static Gaze Map	Yes	Yes	No
Dynamic Gaze Map	Yes	Yes	Animated

the developer, and the educator. The researcher would use iTraceVis after they conduct an eye tracking study such as [19] to understand what the participants were doing during the study tasks. The heat map can show them which areas receive the most focus, The gaze maps and gaze skyline can show how the gaze path of the participant's gaze move. They can also use these visualizations to look for common patterns between their participants, and find where they differ. Researchers can also analyze the eye tracking data of developers outside of controlled studies providing insight into how the developers naturally work in the wild.

The simplest scenario for a developer to use iTraceVis is to better understand their own behavior while reading and navigating code. The developer could look to iTraceVis to see what elements of code he or she commonly overlooks. Additionally, iTraceVis could also be used to improve communication between developers. If one developer is reading a section of code to understand it and asks another developer for help, the second developer can look at iTraceVis views to see where the first developer did and did not look. This would allow the second developer to identify what the first developer might have overlooked or what caused the first developer to expend more effort in order to understand.

Educators could also use eye trackers along with these visualizations to better understand how their students learn to read code. They could look at the data from a student struggling to read and understand code to see if there are any important areas that the student did not give adequate attention. Alternatively they could see the order that their students read the lines of code to determine if the students make connections between relevant pieces of code that interact. Finally, they could see how much time students spent on particular areas of code to learn which spots are giving their students trouble.

VI. Preliminary User Study

In order to evaluate our visualization views, we conduct a small user study on the current version of iTraceVis and its

four views. The research question we are seeking to answer is: Does iTraceVis help a developer understand what transpired during another developer's eye tracking session?

A. Dataset and Participants

Before conducting this study, we collected gaze data from a developer reading two Java files from the Apache TomCat subsystem (using the Tobii X3-120 eye tracker) to use in the study with the goal of trying to understand what those files implement. The files and system were randomly chosen. The developer was asked to read the files and try to understand their functionality. These files were 47 and 112 lines long and the developer spent approximately 3 minutes viewing these files. The developer was an expert Java programmer who was not familiar with the code and did not participate in the actual study. There were 4348 gaze responses recorded in the gaze data file. The gaze data was distributed throughout the files.

For the study, we had ten senior students in Computer Science at Youngstown State University volunteer to participate. None of the participants were familiar with either the code or the data before the study. The participants were asked to use iTraceVis visualization views to answer questions about the developer's gaze behavior during the session. The study took approximately 15 minutes on average. Before they began, the participants were given a short tutorial on iTraceVis using a different dataset so they are familiar with the existing views.

B. Questionnaire

The two Java files we asked the developer to read prior to the study were the *AsyncChannelWrapperNonSecure* class and the *AsyncChannelWrapper* class from the Apache TomCat project. *AsyncChannelWrapperNonSecure* class is called by one of methods of *AsyncChannelWrapper* class. *AsyncChannelWrapperNonSecure* is an implementation of *AsyncChannelWrapper*.

The questions were designed to be answered using the visualizations provided but also required some extrapolation and comprehension of the views to respond correctly. See Table IV for the questions asked during the study.

The questions were separated into five sections. The first section was for information about the participants and asked the number of years they had been using Java. This was to help us understand how well the participants knew Java while reviewing their answers. The next two sections had questions regarding the two Java classes individually. These questions asked about the most looked at source code elements, whether certain segments of code were read, and which visualizations they used. The next section asked questions about the two files together. The last section asked the participants to rate the visualizations in terms of helpfulness, which visualization that they would use to answer a particular question, and to provide suggestions for improvements.

C. Results

The results from the user study were largely positive while still providing suggestions for improvements. Nine of the ten participants said that they would use iTraceVis to visualize eye tracking data of developers; the tenth participant answered maybe. All of the participants were able to identify the most looked at method in the first file as well as the most looked at tokens in both files. Most of the participants were able to see the places that the developer had not looked inside the most-looked-at method. However they had much more trouble identifying the most-looked-at line within that method. When asked the participants were able to tell if a comment within the second file was read, as opposed to just passed over. Only one of the participants did not correctly identify which file was looked at less than the other. When the participants were asked to identify parts of the code that were not looked at within a method some of them reported sections of code that were outside of the method but were not looked at. These answers were counted as incorrect. Figures 8 and 9 show the correct and incorrect answers for each of the 10 participants for the two Java classes *AsyncChannelWrapperNonSecure* and *AsyncChannelWrapper* respectively. The participants were also asked to rate the visualizations in terms of usefulness. The most highly rated of the visualizations was the heat map. Figures 10 and 11 show the user ratings of each individual visualization and the combined ones respectively.

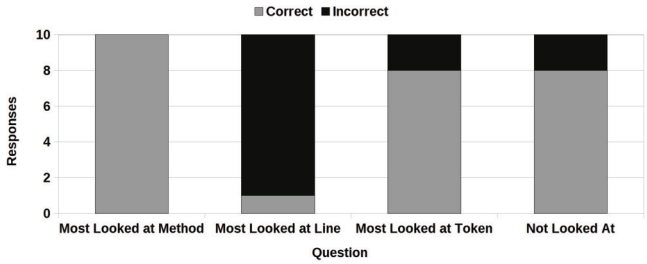

Fig. 8. Results from the questions on AsyncChannelWrapperNonSecure

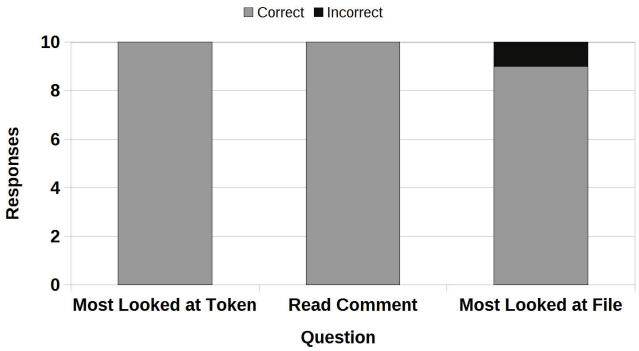

Fig. 9. Results from the questions on AsyncChannelWrapper and the session as a whole

D. Discussion

The most commonly used visualization while answering the questions was by far the heat map. This shows us that it is the most intuitive to use of the visualizations. However the

TABLE IV
QUESTIONS ASKED DURING THE USER STUDY

Question Scope	Question Text
AsyncChannelWrapperNonSecure	• What method was the most looked at? • Which line was the most looked at in the method? • In the most looked at method, which specific source code elements (tokens) did the developer look at the most? • Where (if anywhere) did the developer not look at with respect to the same method? • Which visualization(s) did you use to answer the previous questions?
AsycnChannelWrapper	• Which source code elements (tokens) were looked at the most in this class? • Did the developer read the comment on line 26 to 30? • Which visualization(s) did you use for answering the previous questions?
AsyncChannelWrapperNonSecure and AsycnChannelWrapper	• Which class was looked at longer? • Which visualization did you use to answer the previous question?
Review and Rating	• Rate each of the visualizations and their combinations in terms of helpfulness. • Which visualization would you use if you had to replay the developer session from the beginning? • Which visualization would you use if you wanted to see where a developer did and did not look in the session? • Which visualization would you use if you wanted to see where the developer was looking at a certain point in time during the session? • Which visualization would you use if you wanted to see which source code elements(tokens) took longer for the developer to read during the session? • Would you use iTraceVis to visualize eye tracking data of developers? • What features would you like to see added to iTraceVis?

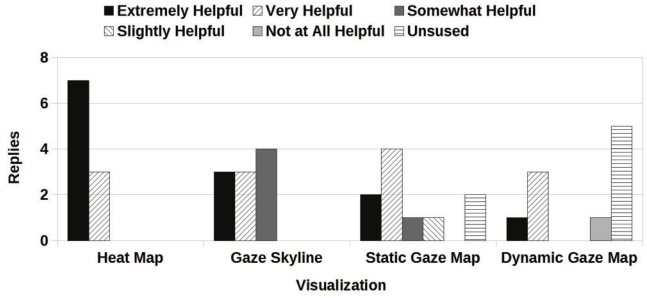

Fig. 10. User ratings of individual visualizations

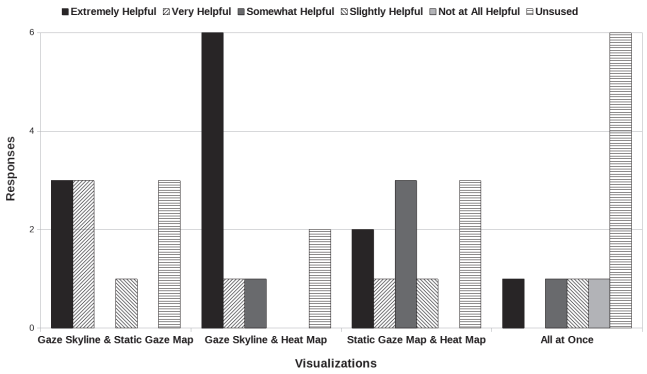

Fig. 11. User ratings of combined visualizations

participants relied too much on it, causing them to incorrectly answer questions. When the participants were searching for the most looked at line many of them chose the line with the most looked at token with the heat map. This was the incorrect line for the question asked during the study. It is possible for a line with more tokens to be looked at for longer than the line with the most looked at token. We can make adjustments to the visualizations to make this more apparent. It is useful to know that the participants favored the heat map so that we know to expand its features so that the users get the most from the most intuitive visualization. This could include the scope of the heat map. We could allow the user to select different scopes to have the heat map highlight entire lines or methods instead of the just the tokens. The idea of a dynamic heat map to complement the dynamic gaze map was suggested by one of the users, with tokens highlighting as they are visited by the gaze and becoming hotter upon repeated visits.

The dynamic gaze map was the least used visualization during the study. The questions were designed not to prescribe a particular visualization to a particular question. There were several questions that were answerable by the dynamic gaze map but the participants chose to use other visualizations. This indicates the dynamic gaze map needs to made more accessible to users for them to use it. With more developer use and adequate training this threat can be mitigated. We have received requests from researchers using iTrace for a playback

feature which is exactly what the dynamic gaze map does. We believe that the questions did not require enough time series information for the participants to utilize the dynamic gaze map. We leave this for a larger study to address.

The gaze skyline was used frequently especially for the questions that asked what the most looked at method was and which file was looked at for longer. The users suggested that the gaze skyline could also indicate which lines are highlighted in the active editor and not just the selected lines. The static gaze map was used less than expected because the participants often used the heat map to get similar information. Nevertheless, it tells us that the heat map and static gaze map can be used to fulfill the same purpose for some tasks.

E. Threats to Validity

The user study used a small number of participants - only ten. Since this was an early stage study a larger number of participants were not needed as the results from this study will help steer the future development of iTraceVis. A larger study will be conducted on usage when more features are added to the tool. The questions presented to the users in the study were designed purposely from the perspective of what can be deduced from the visualizations, but the participants were not told which visualizations to use for any question. This allowed us to see where the users used the visualizations in an unexpected way. This also lead to some visualizations not being used by some participants. The study was mainly qualitative in nature and did not cover the researcher or educator as a stakeholder. We plan on testing these usage scenarios in a future study. Both iTrace and iTraceVis do not currently support code editing in a robust way. Recording eye tracking data while code editing is possible however, the data would be hard to interpret because the semantic structure would have changed as the developer edits the code. We plan on implementing this feature in both tools as part of our future work.

VII. Related Work

There are several tools for visualizing eye tracking data on source code that were presented at the Eye Movements in Programming Workshop (*http://emipws.org/*) as a technical report [20]. The tool eyeCode provides several visualizations similar to the gaze skyline view that shows movement between lines over time and one that shows total number of fixations on a line. There are also *flow* charts that show how a developer's gaze moves between sections of code. However none of these work within the editor itself like iTraceVis and are not able to visualize more than what fits on a screen at a time.

SEQIT is a visualization tool for eye tracking data that shows how the participant moves between areas of interest (i.e. AOIs) [21]. The tool color codes each of the AOIs in the stimulus. It is not designed for visualizing source code. SEQIT represents an eye tracking session using a timeline, coloring segments of the line based on which AOI the participant was looking at during the represented time. SEQIT also provides tools to search the data for sequences of AOIs from the sessions. Multiple timelines can be shown on screen at once allowing the user to compare multiple studies. SEQIT focuses on a single visualization type compared to iTraceVis which uses several. SEQIT shows how color can be used to make a powerful yet simple visualization.

VA^2 is robust eye tracking visualization and search tool [22]. Its main visualization feature is the sequence chart, which shows how the participant moves between the various AOIs of the stimulus. This is shown in a manner similar to a line graph, with the x-axis representing time and the y-axis representing the AOIs. Additionally this tool also visualizes think-aloud data by displaying an icon in the appropriate area above the sequence chart, and interaction data using color coded dots and lines to show how the participant interacted with the stimulus. VA^2 can display multiple participants data at once and provides tool to search for AOI sequences like SEQIT. iTraceVis does not show think-aloud data, and only displays eye tracking data. The Gaze Skyline is similar to the sequence chart, but instead of displaying how the gaze moves between AOIs the Gaze Skyline shows how the gaze moves between lines of code (fine-grained AOIs). VA^2 was not designed to visualize eye tracking data on source code.

Word sized visualizations are small visualizations designed to be used with representations of other data. Beck et al. describe a wide variety of these which fall into two categories: point-based and AOI-based [23]–[25]. A wide variety of data can be visualized in these graphics, from spatial location of the gaze to AOI sequences to fixation duration. Color is often used in these visualizations to show different AOIs, to add context to time, or to indicate duration of fixations. Beck et. al. perform a study on various word sized visualizations [26]. They found that three visualizations stood out amongst the rest: the AOI timeline, the attention map, and the scan path. The AOI was the most highly rated and displayed which AOIs the participant looked at over time, with the AOIs being represented with coded colors and differing vertical positions. The scan path simply traces the path that the participant's eyes took during the session onto the visualization. The attention map is a grid based visualization with the cells being colored darker based on how much attention the corresponding section of the stimulus was given.

Blascheck et al. also developed a radial AOI timeline to complement the sequence chart from VA^2 [27]. The radial AOI timeline plots eye tracking data along with interaction data. The center of the radial timeline represents the beginning of the session and the outer rim of the visualization represents the end of the session. Each AOI is represented by a subsection of the visualization and fixations are represented as circles with the size of the circle denoting the duration. Blascheck et al. also describe circular heat map transition diagrams in their earlier work [28]. These visualizations use radial style visualization with a circle divided into segments for each AOI. These segments a color coded based on the number of fixations that fell within the corresponding AOI. Additionally the segments can vary in size based on the proportion of the total time spent in that AOI. The transitions between

these AOIs are represented be line segments with arrows representing directions. The thickness of these lines denotes the number of transitions.

Blascheck et. al. describe a visualization for eye tracking data on stimuli with nested AOIs [29]. These nested AOIs form a hierarchy that can be displayed with a tree based visualization. Eye movement between the AOIs displayed in this visualization are represented by arcs moving between them. An arc between to a AOI that is a child of another implies that the gaze also fell within the parent AOI. iTraceVis is not an AOI based visualization suite, but the hierarchy based visualization could still be applied due to the hierarchical nature of code.

Gaze Stripes is a unique visualization technique for eye tracking data [30]. While most other visualizations are AOI based, gaze stripes instead uses small segments of the stimulus's image around the gaze points and orders them sequentially. This allows gaze stripes to directly show the visualization's user what the participant was looking at. Additionally gaze stripes from multiple participants can be displayed at once allowing for easy comparison. Gaze Stripes takes a radically different approach to visualization than iTraceVis. This visualization style is not well suited for iTraceVis because it would mangle the source code from the stimulus, rendering it unintelligible. However, this type of visualization could be used to provide an overview approach outside the stimulus (i.e. active editor).

In other related eye-tracking work, Beymer et al. present WebGazeAnalyzer, a tool that records and analyzes web reading behavior [31]. They find that highlighting has increased reading time because the readers slow down to read the text but do not re-read them. There are no custom visualizations in this work (besides a timeline view and a heatmap), however the analysis performed can inform the visualizations for future iTraceVis features. Cheng et al. study gase reading behavior [32] by using annotations such as gray shading, borders, and lines to indicate reading speed, re-reading frequency and transitions respectively. These solutions are extremely adhoc, do not extend well into source code and also do not provide a working tool that can be reused. Other adhoc visualizations using AOIs and fixations are shown in [33].

Minelli et. al. created DFlow [34] a recording and visualization tool for developer interaction with an IDE. It records events such as navigations, edits, understanding, and inspections during a session. DFlow does not record or visualize eye tracking data. Unlike iTraceVis, DFlow's visualizations are not drawn directly into the editor or tied to it. DFLow-Web [35] a web based visualization tool using nodes and arcs to represent developer behavior. iTraceVis focuses more on working within the IDE's editors than DFlow, but only works with eye tracking data in its current state. The authors also present a collection of development stories in [36], that are informed by their visualizations. DFlow provides good reference for visualizations when iTrace expands into visualization other interactions in addition to eye tracking data. We envision building a program analysis layer on top of iTraceVis that can provide many of the novel visualizations that DFlow provides.

Seesoft was developed to visualize source code using thin rows of pixels representing each line of the code [37]. These rows are colored to represent some aspect of the the source code, namely how recently the code was changed. When visualizing eye tracking data it is important to represent the stimulus as well (for iTraceVis, source code is the stimulus). Seesoft provides an excellent example of how to visualize source code in an intuitive and informative way.

To the best of our knowledge, iTraceVis is the first tool to incorporate visualization of developers' eye tracking data on large source code files including scrolling and context switching between files, all within the working environment of the developer.

VIII. Conclusions and Future Work

The paper presents iTraceVis, a visualization module, for an existing gaze-aware Eclipse plugin namely iTrace. To our knowledge, this is one of the first visualizations of eye tracking data that ties in seamlessly with the IDE and works on any number of files regardless of their length. It is not restricted to visualizing just one method that fits on the screen. We leverage our existing iTrace infrastructure that provides us with the mapping of developer eye gaze on source code elements at a fine-grained level down to the word. In the first phase of our tool release, we support four basic views: heat map, gaze skyline, static gaze map, and the dynamic gaze map. The study we conducted was a feasibility assessment to give a general idea of what developers considered useful in the visualization views to help us improve the tool. The feedback we received was overwhelmingly positive.

As part of future work, we will continue to add features to iTraceVis and improve the current visualizations provided by it. The heat map view is currently scoped to highlight only tokens. We plan to add multiple scopes to choose from such as lines of code, methods, if statements or loops. Controls for the dynamic gaze map will be created allowing the user to pause, adjust the speed, step forward and back, and jump around in the data. The gaze skyline will be modified to better show where the gaze falls outside of the active editor's file. We also plan on adding layers to the visualization to show metrics [38] related to the eye gaze data. In addition, we will add search functionality for the user to be able to display only certain parts of the gaze data in any view. The current visualizations in iTraceVis focus on horizontal position, vertical position, and time. However there are other aspects of the data that we have not shown through visualizations, such as pupil dilation or saccade amplitude. Support for visualizing multiple developer sessions is also a planned. Finally, we plan on running a larger user study to determine the usefulness of iTraceVis on eye gaze data generated from bug fixing developer sessions.

Acknowledgment

This material is based upon work supported by the National Science Foundation under Grant Number (CCF 1553573).

REFERENCES

[1] K. Holmqvist, J. Holsanova, M. Barthelson, and D. Lundqvist, *Reading or scanning? A study of newspaper and net paper reading.*, ser. The mind's eye: cognitive and applied aspects of eye movement research. Elsevier, 2003, pp. 657–670.

[2] T. Barik, J. Smith, K. Lubick, E. Holmes, J. Feng, E. R. Murphy-Hill, and C. Parnin, "Do developers read compiler error messages?" in *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, 2017, pp. 575–585.

[3] B. Sharif and J. I. Maletic, "An eye tracking study on camelcase and under_score identifier styles," in *2010 IEEE 18th International Conference on Program Comprehension*, June 2010, pp. 196–205.

[4] L. Yenigalla, V. Sinha, B. Sharif, and M. E. Crosby, "How novices read source code in introductory courses on programming: An eye-tracking experiment," in *Foundations of Augmented Cognition: Neuroergonomics and Operational Neuroscience - 10th International Conference, AC 2016, Held as Part of HCI International 2016, Toronto, ON, Canada, July 17-22, 2016, Proceedings, Part II*, 2016, pp. 120–131.

[5] B. Sharif and J. I. Maletic, "An eye tracking study on the effects of layout in understanding the role of design patterns," in *2010 IEEE International Conference on Software Maintenance*, Sept 2010, pp. 1–10.

[6] B. Sharif, G. Jetty, J. Aponte, and E. Parra, "An empirical study assessing the effect of seeit 3d on comprehension," in *2013 First IEEE Working Conference on Software Visualization (VISSOFT)*, Sept 2013, pp. 1–10.

[7] K. Rayner, "Eye movements in reading and information processing: 20 years of research." *Psychological bulletin*, vol. 124, no. 3, p. 372, 1998.

[8] Z. Sharafi, Z. Soh, and Y.-G. Guéhéneuc, "A systematic literature review on the usage of eye-tracking in software engineering," *Information and Software Technology (IST)*, 2015.

[9] T. R. Shaffer, J. L. Wise, B. M. Walters, S. C. Müller, M. Falcone, and B. Sharif, "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*, ser. ESEC/FSE 2015. New York, NY, USA: ACM, 2015, pp. 954–957.

[10] B. Sharif, T. Shaffer, J. L. Wise, and J. I. Maletic, "Tracking developers' eyes in the IDE," *IEEE Software*, vol. 33, no. 3, pp. 105–108, 2016.

[11] K. Rayner, "Eye movements in reading and information processing," *Psychological Bulletin*, vol. 85, no. 3, pp. 618–660, 1978.

[12] M. A. Just and P. A. Carpenter, "A theory of reading: from eye fixations to comprehension." *Psychological review*, vol. 87, no. 4, p. 329, 1980.

[13] A. Duchowski, *Eye tracking methodology: Theory and practice*. Springer-Verlag New York Inc, 2007.

[14] M. A. Just and P. A. Carpenter, "A theory of reading: from eye fixations to comprehension." *Psychological review*, vol. 87, no. 4, p. 329, 1980.

[15] P. Olsson, "Real-time and offline filters for eye tracking," *Masters Thesis, KTH, School of Electrical Engineering*, 2007.

[16] T. Blascheck, K. Kurzhals, M. Raschke, M. Burch, D. Weiskopf, and T. Ertl, "Visualization of eye tracking data: A taxonomy and survey," *Computer Graphics Forum*, pp. n/a–n/a, 2017.

[17] O. Spakov and D. Miniotas, "Visualization of eye gaze data using heat maps," *Electronics and Electrical Engineering*, pp. 55–58, 2007.

[18] T. Blascheck, K. Kurzhals, M. Raschke, M. Burch, D. Weiskopf, and T. Ertl, "State of-the-art of visualization for eye tracking data," *EuroVis STAR*, pp. 63–82, 2014 2014.

[19] K. Kevic, B. M. Walters, T. R. Shaffer, B. Sharif, T. Fritz, and D. C. Shepherd, "Tracing software developers eyes and interactions for change tasks," *Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 2015.

[20] S. Tamm, R. Bednarik, T. Busjahn, C. Schulte, H. Vrzakova, and L. E. Budde, "Eye movements in programming education: Spring academy 2017 - technical report tr-b-17-02," Freie Universität Berlin, Department of Mathematics and Computer Science, Berlin, Germany, 2017.

[21] M. M. Wu and T. Munzner, "SEQIT: Visualizing Sequences of Interest in Eye Tracking Data," *Proc. IEEE Conference on Information Visualization (InfoVis)*, 2015.

[22] T. Blascheck, M. John, K. Kurzhals, S. Koch, and T. Ertl, "Va 2: A visual analytics approach for evaluating visual analytics applications," *IEEE transactions on visualization and computer graphics*, vol. 22, no. 1, pp. 61–70, 2016.

[23] F. Beck, T. Blascheck, T. Ertl, and D. Weiskopf, "Exploring word-sized graphics for visualizing eye tracking data within transcribed experiment recordings," 2015.

[24] T. Blascheck, F. Beck, S. Baltes, T. Ertl, and D. Weiskopf, "Visual analysis and coding of data-rich user behavior," in *IEEE Conference on Visual Analytics Science and Technology (VAST)*. IEEE, 2016, pp. 141–150.

[25] F. Beck, T. Blascheck, T. Ertl, and D. Weiskopf, *Word-Sized Eye-Tracking Visualizations*. Cham: Springer International Publishing, 2017, pp. 113–128.

[26] F. Beck, Y. Acurana, T. Blascheck, R. Netzel, and D. Weiskopf, "An expert evaluation of word-sized visualizations for analyzing eye movement data," in *IEEE Second Workshop on Eye Tracking and Visualization (ETVIS)*. IEEE, 2016, pp. 50–54.

[27] T. Blascheck, M. John, S. Koch, L. Bruder, and T. Ertl, "Triangulating user behavior using eye movement, interaction, and think aloud data," in *Proceedings of the Ninth Biennial ACM Symposium on Eye Tracking Research & Applications*, ser. ETRA '16. New York, NY, USA: ACM, 2016, pp. 175–182.

[28] T. Blascheck, M. Raschke, and T. Ertl, "Circular heat map transition diagram," in *Proceedings of the 2013 Conference on Eye Tracking South Africa*, ser. ETSA '13. New York, NY, USA: ACM, 2013, pp. 58–61.

[29] T. Blascheck, K. Kurzhals, M. Raschke, S. Strohmaier, D. Weiskopf, and T. Ertl, "Aoi hierarchies for visual exploration of fixation sequences," in *Proceedings of the Ninth Biennial ACM Symposium on Eye Tracking Research & Applications*, ser. ETRA '16. New York, NY, USA: ACM, 2016, pp. 111–118.

[30] K. Kurzhals, M. Hlawatsch, F. Heimerl, M. Burch, T. Ertl, and D. Weiskopf, "Gaze stripes: Image-based visualization of eye tracking data," *IEEE transactions on visualization and computer graphics*, vol. 22, no. 1, pp. 1005–1014, 2016.

[31] D. Beymer and D. M. Russell, "Webgazeanalyzer: a system for capturing and analyzing web reading behavior using eye gaze," in *Extended Abstracts Proceedings of the 2005 Conference on Human Factors in Computing Systems, CHI 2005, Portland, Oregon, USA, April 2-7, 2005*, 2005, pp. 1913–1916.

[32] S. Cheng, Z. Sun, L. Sun, K. Yee, and A. K. Dey, "Gaze-based annotations for reading comprehension," in *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems, CHI 2015, Seoul, Republic of Korea, April 18-23, 2015*, 2015, pp. 1569–1572.

[33] S. Cheng, ""third eye": designing eye gaze visualizations for online shopping social recommendations," in *Computer Supported Cooperative Work, CSCW 2013, San Antonio, TX, USA, February 23-27, 2013, Companion Volume*, 2013, pp. 125–128.

[34] R. Minelli and M. Lanza, "Visualizing the workflow of developers," in *Software Visualization (VISSOFT), 2013 First IEEE Working Conference on*. IEEE, 2013, pp. 1–4.

[35] R. Minelli, A. Mocci, M. Lanza, and L. Baracchi, "Visualizing developer interactions," in *Software Visualization (VISSOFT), 2014 Second IEEE Working Conference on*. IEEE, 2014, pp. 147–156.

[36] R. Minelli, L. Baracchi, A. Mocci, and M. Lanza, "Visual storytelling of development sessions," in *2014 IEEE International Conference on Software Maintenance and Evolution*, Sept 2014, pp. 416–420.

[37] S. Eick, J. L. Steffen, and E. E. Sumner, "Seesoft-a tool for visualizing line oriented software statistics," *IEEE Transactions on Software Engineering*, vol. 18, no. 11, pp. 957–968, 1992.

[38] Z. Sharafi, T. Shaffer, B. Sharif, and Y. Guéhéneuc, "Eye-tracking metrics in software engineering," in *2015 Asia-Pacific Software Engineering Conference, APSEC 2015, New Delhi, India, December 1-4, 2015*, 2015, pp. 96–103.