Tracking Eye Movements over Source Code

Faith Chan
Advisor: Professor Sara Bahrami

Senior Project
Winter, Spring 2018

Computer Engineering Department
California Polytechnic State University, San Luis Obispo

Contents

Mapping Gaze to Source Code Elements with iTrace

Abstract

Studies on software developers' behavior guide the development of tools that facilitate source code reading and reviewing. Eye trackers have allowed researchers to study this behavior in more detail–to pinpoint where the developer is looking, or even to detect which source code element the developer is viewing. However, systems that map gaze to characteristics as specific as source code elements are often expensive, either because of the cost of compatible eye trackers or because of the cost of the required software. This project aims to use existing technology to create a lower-cost system that provides information on the source code elements that the developer views.

1. Introduction

Eye trackers have become increasingly present in studies on developers' behavior. These devices offer insight into developer behavior that previously could not easily be observed, bringing researchers closer to detecting patterns in developers' thought processes by revealing the movements they might even unconsciously make. Studies that track eye movements in source code at the method level and at the line level have already shown promising results, but information that is even more detailed may lead to a better understanding of any patterns that appear. The desire for this information has given rise to tools such as iTrace, an Eclipse plugin that maps gaze movements to source code elements.

Compatible only with expensive research-grade eye trackers at the time of development, iTrace and its features are not easily accessible to the average user. Alternatives that are compatible with lower-cost eye trackers exist, but the required software for these tools is often expensive. The cost of these systems in fact presented a roadblock in the early stages of this project, which initially proposed to use the tool in an empirical study, as well. Therefore, in its final goal to integrate iTrace with a lower-cost eye tracker, this project hopes to create a more accessible system that provides the same functionality.

With this in mind, the project has two primary objectives:

1. The system shall interface the Gazepoint GP3 eye tracker with iTrace, allowing gaze data from the GP3 to be mapped to source code elements.
2. Using the mapped data in iTrace output files, the system shall provide statistics on the source code elements that have been viewed.

According to the iTrace developers, support for additional eye trackers including the Gazepoint device, along with support for source code element mapping in different IDEs, is being developed at the time of this project. This project nevertheless aims to interface the Gazepoint system with the current iTrace release, as the tool and its ability to map eye gaze to source code elements are needed in this effort, as well as in projects that will take place before the next release.

The rest of this paper will begin with an overview of relevant background information and previous research, followed by a description of the modifications made to iTrace.

2. Background

Studies using eye trackers often map gaze movements to lines in code or to methods in code, but rarely examine gaze movements on the source code element level. Several focus on the amount of time spent on lines of code that contain a defect, for example (Uwano, 2006; Sharif, 2012). Others analyze movements between methods and files as well. Those that do examine gaze on the source code element level rely on manual efforts to map gaze data to elements in code (Kevic, 2017).

Such studies have revealed useful patterns in developer behavior. However, information about the specific source code elements examined while reading code can lead to even more detailed analysis of developer behavior (Kevic, 2017). Easier access to this information, which this project aims to provide, would be beneficial in future studies.

4. Planning and Design

This project originally consisted of two parts: the design and execution of an experiment on developer behavior in code reviews, which is described in the second part of this paper, followed by the development of a tool that would facilitate this review process based on information from that experiment.

The use of iTrace was key in the experiment, as its ability to map gaze data to source code elements provided additional detail about developer behavior that had not often been seen in previous studies. Therefore, setup for the tool began while information was being gathered for the experiment. However, investigations into iTrace revealed that none of the eye trackers accessible through Cal Poly were compatible, contrary to initial findings. Research into the trackers that were compatible, along with discussions with the original iTrace developers, revealed that those compatible trackers exceeded the budget for the project. Given this, the Gazepoint GP3 was a reasonable alternative. Because this new eye tracker was not compatible with iTrace at the time of development, however, the task of interfacing the new device with the plugin became the primary task of this project.

The added functionality–analysis of the raw mapped gaze data–became an additional requirement of the project, in order to supply a more readable output and summary of the

recorded gazes. As shown in the images below, raw output files are perhaps difficult to analyze manually. The output files for fixation-filtered data are similar. A need for a simplified version of this output led to the development of the analysis tool.

Figure 1. Example of raw data output from iTrace, presented in an XML document. The file indicates which source code elements are associated with a particular gaze location. In this case, the highlighted line shows a gaze that was mapped to an "if" statement. This frame appears in the iTrace demonstration at https://youtu.be/3OUnLCX4dXo.

Figure 2. Different view of the raw mapped gazes, output as an XML file, from iTrace. Source code elements are outlined in red.

3. Hardware and Software Components

This section describes the components used in the final system. iTrace, the Eclipse plugin, remained a large part of the project. Additional information about its functionality is outlined here. The lower-cost eye tracker used in this project is the Gazepoint GP3 eye tracker. Details on this tracker are also provided in this section.

3.1. iTrace

iTrace is an Eclipse plugin that matches source code elements–if statements, method invocations, and conditional expressions, for example–to gaze data from an attached eye tracker or to data from a mouse. This allows for examination of gaze patterns on a more specific level than is often seen in studies involving eye tracking over source code.

Additionally, iTrace records mapped data for any code window that is open in the IDE, adjusting for scrolling and other movements within the window. As a result, there is minimal disruption to the developer's normal coding experience. This is beneficial in experiment settings, as developers are less likely to exhibit different behavior due to an altered coding environment.

Mapped data for sessions recorded using iTrace can be exported to XML and JSON files, and results can be run through iTrace's fixation filters. Results from the filters are also available in XML or JSON format (Shaffer, 2015).

Further information about the plugin, as well as the plugin source code, can be found at the following links:

iTrace website: http://seresl.csis.ysu.edu/iTrace/

iTrace Eclipse plugin: https://github.com/trshaffer/iTrace

3.2. Gazepoint GP3 Eye Tracker

The Gazepoint GP3 eye tracker, which offers a sampling rate of 60 Hz, uses binocular tracking to provide eye gaze data with 0.5 to 1-degree accuracy and 0.1-degree spatial resolution (precision) ("Gazepoint Control User Manual", 2017). Although not quite as precise as the research-grade trackers currently compatible with iTrace, the GP3 has specifications that appear to be sufficient for the intended use in this project. Although the application seeks to examine eye gaze at a high degree of granularity, its aim is to map gaze to source code elements. It is therefore unnecessary to map users' gazes to exact points on the screen. Ultimately, considering its relatively low price, the GP3 is a reasonable choice for this application.

The GP3 is also a viable option for its ability to communicate to the application using Java, the programming language in which iTrace is written. Communication between the eye tracker and the application occurs via TCP/IP connection; therefore, any programming language that supports such connections is compatible with the GP3 ("Gazepoint Control User Manual", 2017). Because Java supports TCP/IP socket connections, no additional interfacing between Java and another language (via Java Native Interface or Java Native Access, for instance) is required. The GP3 is therefore a reasonable candidate not only for its price and eye tracking specifications, but also for its compatibility with iTrace.

The GP3 eye tracker is used with current software that Gazepoint provides. The software that comes with the hardware is used here and is sufficient for the project.

5. Development

This section outlines the main changes made to iTrace: the interfacing of the Gazepoint GP3 with the tool, and the addition of the analysis filter.

5.1. GazepointTracker

GazepointTracker, placed in the iTrace "trackers" package, contains functions that allow iTrace to communicate with the Gazepoint eye tracker. Using the structure iTrace provides, the GazepointTracker class implements the IEyeTracker interface, alongside the existing SystemMouseTracker and TobiiTracker classes. Some aspects of its operations are similar to those of SystemMouseTracker, while others are similar to those of TobiiTracker. Because communication with the Gazepoint tracker can take place entirely in Java, like communication with SystemMouseTracker can, tracker threads are handled in the same way for GazepointTracker as they are for SystemMouseTracker. This structure is also reasonable because the Gazepoint device does not have the main thread with which the Tobii tracker API operates. However, the main loop of the SystemMouseTracker occurs in the thread–part of a private nested class–that is started when the tracker is initialized. To keep the functionality of the tracker within the GazepointTracker class itself, the structure of the new class follows the structure of the TobiiTracker class. The Tobii tracker API appears to use a single class instance that handles its own main loop, however, so to emulate this, the Gazepoint tracker was implemented as a singleton.

5.2. XMLAnalysisFilter

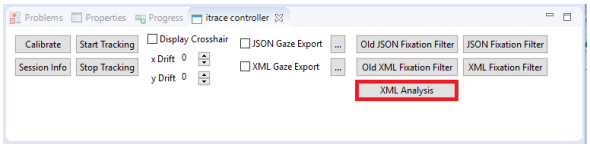

Figure 3. The added analysis option in the iTrace perspective. This perspective can be opened in the Eclipse IDE when iTrace is run from the parent Eclipse application. For additional guidance, please refer to the user manual.

This filter provides a summary of the basic mapped gaze data that is recorded when iTrace is used. iTrace filters, which implement the interface IFilter, are processed in the application's main loop with their implementations of the read(), process(), and export() functions that IFilter requires. XMLAnalysisFilter's read(), based on XMLBasicFixationFilter's read(), parses through the gaze-responses XML file and records information corresponding to each gaze in a data structure. The source code elements for the gazes in each file are parsed through in process(); they are recorded in a nested HashMap structure, allowing duration and other source code element information to be recorded only once for one source code element. For a particular method declaration, for example, duration is updated for the corresponding element in the data structure whenever that declaration is viewed, but the source code element is not added as a new element another time. In this way, XMLAnalysisFilter keeps track of the number of times the developer has viewed any particular source code element, as well as the amount of time the developer has spent viewing that element. These results are written in the XML file after the corresponding source code element. The number of times the developer has viewed the element is marked with the tag "<times_visited>," after the "</sce>" tag, and the eye gaze duration for that element is marked with the tag "<duration>." The total amount of time a file has been viewed is kept as a running total. This value is written at the end of the XML file, with the tag "<total>." The figure below shows an example of the output from the new filter.

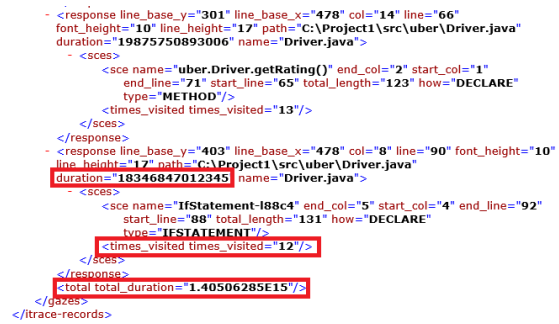

Figure 4. Example of iTrace analysis output. Duration for the source code element is noted after the "<duration>" tag, along with the source code information–line number and column, for example–relevant to that element. The number of times the element is visited is noted at the end of the line, and the total amount of time the file was viewed is written at the end of the information for each file.

For its functions' similarity to XMLBasicFixationFilter's functions, XMLAnalysisFilter is placed in the "fixations" package under the "filters" package. Filters in this package group gazes into fixations based on the similarity between gazes that are made near one another. Gazes that are similar to those close to it are recorded as one gaze with a longer duration, and gazes that are different indicate a shift in gaze to another part of the code. The new filter currently uses the mapped gaze data that is unprocessed–that is, it does not use this fixation filtering method. However, its location in the "fixations" package may also be useful for future implementations of the tool that may conduct analysis on fixation-filtered data only. This was not implemented here due to the necessity of further testing with the existing fixation filters, which still encounter a few errors.

6. Conclusions

With the modifications made to iTrace, data from the Gazepoint GP3 eye tracker can be used to map gaze movements to source code elements. The added analysis tool adequately filters the recorded gaze responses by providing information on the amount of time locations in code are

viewed, per source code element. Further testing of the system itself is required, however, before further studies can be conducted. The testing done while interfacing the eye tracker to the tool revealed some potential issues with gaze recording–gazes sometimes did not appear to be tracked as accurately as desired, for example, and issues with lagging surfaced during later testing. Additionally, changes could be made to the modifications made here, as well as to the iTrace structure that existed before this project, that could improve the integration of the new functionality into the tool.

Overall, this system has the potential for widespread use as a lower-cost alternative to other tools that map gaze movements to source code elements. With additional testing and adjustments, use of the tool in a variety of cases, including studies aiming to examine developers' gaze movements over source code in detail, is possible.

7. Future Work

Additional analysis metrics can be added to this tool based on iTrace output, and features offered with the Gazepoint eye tracker–heat mapping, for example–can also be incorporated into that analysis. To facilitate such future additions, the changes described in this paper could be more smoothly incorporated into the existing structure; this would involve adjustments not only to the new code but also to the underlying structure. Other improvements to the tool, including gaze mapping to elements in files of other formats, have been mentioned in the 2015 paper that introduces iTrace (Shaffer, 2015).

Beyond additions to the system itself, future work includes studies on developers' eye gaze patterns that make use of the tool, allowing researchers to examine developer behavior in more detail. The experiment described in the next part of this paper is one example of this. Such studies may ultimately give rise to new tools that better facilitate the processes of code reading and review because of that more specific information.

Examining Eye Gaze Movements during Code Review Tasks

Abstract

Of the many experiments that may benefit from use of the modified plugin, one experiment is considered here. This study aims to explore the ways in which programmers review code and how, if at all, any patterns in these behaviors are related to the efficiency and effectiveness of the review. Subjects' gaze movements over source code are examined on the source code element level, allowing for fine-grained analysis of developer behavior.

1. Introduction

Code reviews are an important task in software development. The outside viewpoint of the reviewers exposes errors and design considerations that authors may not have thought of during their routine review of the familiar code. This prevents unnecessary returns to the expensive testing phase of software development cycle. However, the introduction of a new developer to unfamiliar code may also be expensive during a code review; the new viewer must take time to become familiar with the source code before a thorough assessment can be made, or before an appropriate change can be proposed. Reducing the amount of effort this requires can prove beneficial not only for the reviewer, who may spend less time simply searching through the code, as well as for the original author, who may receive feedback and proceed with development more quickly.

This study attempts to address unexplored areas in research on developer behavior during code review–namely, the realm of code element-level eye tracking as applied to code review. Conclusions from this study may inspire future studies on such behavior, as well as guide the development of tools that aid in the processes of reading and reviewing code.

The rest of this paper will begin with an overview of relevant background information and previous research, followed by an outline of the empirical study.

2. Background

Previous research has demonstrated the potential of eye tracking as applied to programming tasks such as remote pair programming (D'Angelo, 2017), as well as its uses in the development of tools and teaching methods to improve the efficiency and effectiveness of common programming tasks. Although these studies report promising results, such results are prone to variation and ambiguity by nature because of the presence of many confounding variables. Further testing in these areas is therefore required to ensure repeatability. Nonetheless, results suggest that eye tracking is worth using in additional studies in myriad environments–both within the scope of computer science and beyond it.

Additionally, research done for this project indicates that eye tracking at a source code element level of granularity has not been used in studies involving the code review process. Few studies analyze gaze paths at the source code element level at all; some, such as the study on code review by Uwano et al., examine gaze patterns at line-level granularity, while others focus on method-level or class-level gaze movements. Although studies involving line-level granularity have demonstrated repeatable patterns in gaze movements during code review tasks (Uwano, 2006), gaze information about specific source code elements has been shown to be conducive to much more detailed analysis on gaze movement (Kevic, 2017). Such granularity may prove beneficial in attempts to provide more specific guidance during the code review process.

While research on eye gaze movements in code progresses, tools that aid in the code review process also continue to be developed. Bug tracking repositories and code revision systems are some of the most widely used code review tools, helping developers to track documented changes in specific files or lines of code. Use of such tools has become routine in software development process. Meanwhile, new systems to improve the efficiency of code reviews have begun to emerge; for example, one tool uses eye tracking to facilitate remote pair programming by displaying a visualization of one programmer's gaze on the partner's screen (D'Angelo, 2017). However, the benefits observed with use of visualization are not necessarily due to the presence of the visualization at all. This suggests that further testing–and further development following this–is required to evaluate, more thoroughly, the effectiveness of such eye-tracking enhanced systems in programming tasks. Still, these current code review tools do not use specific eye movement patterns to improve the review process; given the promising results from research, it seems that this more specific eye gaze data has the potential to fill the gaps in areas that current code revision tools are lacking.

3. Experiment Overview

This study aims to examine the characteristics of the eye gaze movements of developers as they perform code review tasks, and to determine any patterns in this behavior. Eye tracking technologies are used to maximize the level of detail that can be examined in pursuit of this goal, with the idea that more specific information on eye gaze movement may produce a clearer image of any patterns or characteristics of gaze paths. This image can easier be used to design tools that aid in the code review process.

Given this goal, gaze paths shall be analyzed in their entirety. With the capabilities of the eye tracking technologies used here, patterns in gaze with respect to specific source

code elements will be analyzed as well. Other measures, such as time stamps, will be used to evaluate the efficiency of the code review sessions.

The research questions to be explored during the empirical study of this project are as follows.

RQ1. Are there patterns in developers' eye gaze movements during code review tasks?

RQ2. Does eye tracking reveal new information about eye gaze movement that occurs during code review tasks?

RQ3. How can the code review process be made more efficient with a gaze-based visualization?

4. Variables and Hypotheses

4.1. Dependent Variables

4.1.1. Full Gaze Path. During analysis of the collected data, full gaze paths will be examined for patterns in eye gaze movements. This provides an overall view of participants' gaze path through the source code and suggests patterns that may then be examined on a lower level. The variable will be evaluated qualitatively, and quantitatively where additional analysis is required.

4.1.2. Visual Effort. Visual Effort will be evaluated based on the amount of time a participant's gaze remains on a specific location in code (fixations). It will also be evaluated based on the number of times that location is revisited during the review session. This measure is based on the Visual Effort variable used in the Sharif study, which examined the tendency of reviewers to perform a scan through source code during defect detection tasks (Sharif, 2012).

Visual Effort will be evaluated on a code element level of granularity.

4.1.3. Efficiency. The efficiency of a code review session will be evaluated based on the duration of the entire review of one code change example. It will also be based qualitatively on reviewers' descriptions of their experience and perceived familiarity with the source code sample.

4.1.4. Effectiveness. Changes and reviews will be evaluated for their effectiveness based on comparisons between reviewers' conclusions and those described in the published documentation for that code change.

This will be a qualitatively evaluated variable.

4.2. Hypotheses

Analysis of collected data evaluates the following null hypotheses, which are based on the research questions listed in the Experiment Overview.

H10. There are no patterns in developers' eye gaze movements during code review tasks.

H20. Source code element-level eye tracking does not reveal additional information about eye gaze movement during code review tasks.

5. Participants

Data will be collected for fourth year students studying Software Engineering at Cal Poly. All should have recent experience coding in Java and working in the Eclipse IDE.

6. Equipment and Setup

Collection of eye gaze information will be done with the Gazepoint GP3 eye tracker along with iTrace, the open source plugin that maps eye gaze to source code elements in the Eclipse IDE.

The iTrace plugin is well suited to the requirements of this study, as its ability to map eye gaze to source code elements allows for analysis of eye gaze data on a more granular level. This capability, which has not been used extensively in previous research on code reviews, adds a level of specificity to the collected data–data that may be useful in the design and development of other studies and tools to facilitate the code reading and reviewing process.

The ability of the tool to record code element-level data while the developer works normally within the Eclipse IDE–that is, the developer is able to scroll within a file and switch between files within the IDE–is also beneficial to this study. Minimal changes to the coding environment are desired when observing developer behavior, as larger changes to that environment may skew results.

6.1. Code Selection Process

Code samples used during the study are excerpts from open-source projects, primarily found using Gerrit and Bugzilla for Eclipse. In order to simplify the code acclimation process for the subjects and attempt to prevent any variation in results due to any confusion on the subjects' part, samples were chosen from only a few projects: mylyn and SWT (the Standard Widget Toolkit).

From these projects, snippets were chosen based on several considerations. For one, these snippets were adjusted to the experience level of the participants. The scope of the code samples, along with the level of documentation of those code samples, were taken into consideration as well, and the difficulty and number of files involved in the code to be reviewed were limited according to time constraints of the experiment (the study was limited such that one reviewer would have about an hour to complete the task). A variety of code sample types–bug fixes, for example, along with functionality additions–were included in order to obtain data for a wider range of code review tasks. This was also done to ensure that potential differences resulting from the type of code sample could be examined, as the focus of this project is to examine the ways in which reviewers trace through changes during a code review. Code samples with varying levels of difficulty, and those affecting different amounts of code, were chosen and tested with each reviewer for the same reason.

7. Data Collection

The experiment will be a latitudinal study, involving a larger number of participants who participate in several code review tasks over a short period of time. Two sets of data collection sessions will be conducted; for each, a separate group of participants will perform code review tasks associated with that set of sessions. The group that participates in the first set of data collection sessions will make changes to given code snippets, and then document these changes for the next group to review. The group that participates in the second set will review the changes that the first group has made to the code.

Before all review sessions are conducted, all participants will be sent briefings on their tasks–making and documenting changes to code excerpts, or reviewing changes that have previously been made–along with descriptions of the programs that they will be reviewing during the test session. Participants will be encouraged to ask questions about the programs until they feel familiar with what those programs do. They will also be given brief preliminary surveys regarding previous experience with code review.

When participants come in for the review session, they will be asked if they require any additional clarification regarding the experiment process. Before the recording session is started, the eye tracker will be calibrated to capture the participant's gaze. The participant will then be able to begin the task whenever ready, making verbal or written notes on the subject of the review. When the participant indicates that he/she has finished, the review session recording will be stopped.

Before leaving the testing session, participants will be asked to discuss observations of the code snippet changes they reviewed. Finally, they will be asked to complete surveys after completion of the testing period.

8. Analysis

For quantitative variables–Visual Effort and Efficiency–collected data will consider whether results are statistically significant. Qualitative variables will be evaluated in terms of patterns or similarities among participant responses. The inclusion of such qualitative variables allows for participants to describe their thoughts, which cannot be observed using the eye tracker. It may also provide insight into any confounding variables and additional factors to be taken into consideration when making conclusions about the study.

Further analysis, along with results and discussion, will be included after the study has been conducted.

9. Threats to Validity

Empirical studies related to individuals' programming behaviors necessarily involve confounding variables. The presence of an eye tracker during the study, for example, may affect the performance of programmers in their code reviewing–this is only one of many issues in that single vein. Additional threats to validity may be revealed during the study.

10. Conclusions

Previous research, along with observations from the development process described in the first section of this paper, demonstrates that iTrace, used with the Gazepoint eye tracker, may be useful in detailed studies on developer behavior. However, conclusions on the hypotheses presented here will be made after the study has been conducted.

Appendix

Bill of Materials

Component	Description	Cost per Unit	Quantity	Total Cost
Gazepoint GP3	Eye tracker	$695	1	$695
iTrace	Eclipse plugin	$0	1	$0
Total				$695

References

D'Angelo, S. (2017). Improving Communication Between Pair Programmers Using Shared Gaze Awareness. , 6245–6290. doi: 10.1145/3025453.3025573

Gazepoint Control User Manual. (2017).

Kevic, K. (2017). Eye Gaze and Interaction Contexts for Change Tasks - Observations and Potential. *The Journal of Systems and Software*, *128*, 252–266. doi: 10.1016/j.jss.2016.03.030

Shaffer, T. R. (2015). iTrace: Enabling Eye Tracking on Software Artifacts within the IDE to Support Software Engineering Tasks. , 954–957. doi: 10.1145/2786805.2803188

Sharif, B. (2012). An Eye-tracking Study on the Role of Scan Time in Finding Source Code Defects. , 381–384. doi: 10.1145/2168556.2168642

Uwano, H. (2006). Analyzing Individual Performance of Source Code Review Using Reviewers' Eye Movement. , 133–140. doi: 10.1145/1117309.1117357