Studying Developer Gaze to Empower Software Engineering Research and Practice

Bonita Sharif
Dept. of Computer Science and Information Systems
Youngstown State University
Youngstown, Ohio 44555
bsharif@ysu.edu

Benjamin Clark
Dept. of Computer Science and Information Systems
Youngstown State University
Youngstown, Ohio 44555
bjclark01@student.ysu.edu

Jonathan I. Maletic
Department of Computer Science
Kent State University
Kent, Ohio 44242
jmaletic@kent.edu

ABSTRACT

We propose a new research paradigm that leverages developer eye gaze to improve the state of the art in software engineering research and practice. We outline our vision on the use of this new paradigm for software engineering tasks such as code summarization, code recommendations, prediction, and continuous traceability. Based on this new paradigm, we foresee a new benchmark that emerges based on developer gaze. This research borrows from cognitive psychology, artificial intelligence, information retrieval, and data mining. In synergy with the above fields, new algorithms will be discovered that work with eye gaze data to help improve current IDEs thereby making developers more productive. Conducting empirical studies using an eye tracker will lead to inventing, evaluating, and applying innovative methods and tools that use developer eye gaze to support the developer in software engineering tasks. We discuss the implications and challenges of this paradigm for future software engineering research.

CCS Concepts

•Software and its engineering→ Software creation and management

Keywords

Eye tracking, summarizations, recommendations, continuous traceability, predictions, benchmarks, mining gaze data

1. INTRODUCTION

Eye trackers have been used for many decades to study how people comprehend visual stimuli [1]. Modern eye trackers implicitly collect a person's (e.g., developers) eye gaze data on the visual display (stimulus) in an unobtrusive way while they are performing a given task. This eye movement data could provide much valuable insight into comprehension strategies [2] as to how and why people arrive at a certain solution. Eye movements are essential to cognitive processes because they focus a person's visual attention to the parts of a visual stimulus that are processed by the brain. Visual attention triggers cognitive processes that are required to perform such things as comprehension. Eye movement is also a proxy for cognitive effort [1] and allows us to determine what parts of a visual stimuli are difficult to understand.

The current role of eye trackers in software engineering is mostly limited to empirical assessment [3, 4]. This restriction is understandable due to the fact that the affordability of high-quality eye-tracking equipment is not yet to the level of common computer components (the cost difference is in the order of several magnitudes). Recently, low-cost eye trackers costing a couple of hundred dollars were released for consumer use (mainly gaming). If history of technology evolution is to be our guide, it is perhaps not farfetched to say that in the foreseeable future, eye-tracking technology would become more and more affordable. Eye trackers would then be a common feature on personal computers similar to web cameras and other peripherals. It is becoming increasingly possible to do things in integrated development environments with a "blink of an eye", similar to what we do today "by word of mouth or click of a button". Capturing eye gazes would be as simple as capturing screen shots, videos or activity logs. Such eye-tracking enabled Integrated Development Environments (IDEs) would offer unique opportunities to software engineering research and practice.

The field of cognitive psychology has used eye trackers to study how we comprehend such things as words, prose, pictures, and diagrams. Computer scientists have used eye tracking devices to study how people interact with graphical user interfaces and web pages. One goal of such investigations is to learn what constitutes a good human computer interface so that we may design better ones.

The software engineering research community is now using eye trackers to study how engineers comprehend and develop software. Sharafi et al. [5] did a comprehensive systematic literature review on all eye tracking related studies (approx. 35) done in software engineering since the 1990s. Unfortunately, all of the studies use small code snippets that do not mimic real world scenarios. While fixed static text is sufficient to study how people read a sentence (or a few lines of source code), it is wholly inadequate to study how programmers attempt to comprehend an entire software system. That is, with the current technology we can only study how programmers comprehend short snippets of code. Instead, we need to study the programmer developing software in their actual work environment while using program editors and other associated integrated development tools (IDE) (e.g., Visual Studio, Xcode, or Eclipse).

DOI: http://dx.doi.org/10.1145/12345.67890

In the following sections, we describe how eye tracking can become more prevalent in developer workflow with very little effort on the part of the developer. We also outline several research applications and visions (not limited to using eye tracking just for assessment) including challenges that need to be overcome. We believe the outcome of this research to be far-reaching and directly advance the state of the art in supporting several core software engineering tasks

2. EYE TRACKING WITHIN THE IDE

The underlying basis of an eye tracker is to capture various types of eye movements that occur while a participant in our case, humans, physically gaze at an object of interest. Fixations and saccades are the two types of eye movements. 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 visualized information occurs during fixations, whereas, no such processing occurs during saccades. The visual focus of the eyes on a particular location triggers certain mental processes in order to solve a given task [1]. Modern eye trackers are accurate to 0.5 degrees (0.25 inch diameter) on the screen.

Eye trackers work by determining the *(x, y)* coordinate on a screen of where a person is looking. It is accomplished with cameras that record the subject's eye movements. This is done at a given sampling rate and a given accuracy. For a fixed stimuli (image or pdf) mapping the eye movements to the location a person is looking at is relatively straight forward geometry. Changes to the stimuli (screen), such as scrolling, present a more complicated problem. There exists very limited support of scrolling in commercial eye tracking devices however it still requires a fixed stimuli that just happens to be longer. There is no existing support in the context of a person interactively using an editor or switching between files being viewed. Basically, existing systems do not keep track of what line in which file is present on the screen (i.e., currently being viewed).

In order to deal with the above problem, we introduced *iTrace* [6]. *iTrace* is an Eclipse plugin that interfaces with an eye tracker to collect fine-grained line-level data on software artifacts the developer is viewing and interacting with. It allows for scrolling and switching between different artifacts and files during an eye tracking study. The goal is to support not just source code artifacts but also other software artifacts such as UML diagrams, stack overflow (www.stackoverflow.com) documents, bug reports, test cases, and requirements all accessible within an IDE. This will enable software engineering researchers to conduct large-scale realistic eye-tracking studies seamlessly within a software development environment. We believe the data generated by eye trackers can be used along with other data streams for added insight. There is no need for the researcher to manually map (*x,y*) coordinates to source code elements as all of this time-consuming labor-intensive process is now done automatically by *iTrace*. *iTrace* runs uninterrupted in the background within Eclipse, recording developers' eye movements while they are working. The first version of the plugin is open source under the GPL license (http://seresl.csis.ysu.edu/iTrace/). In order to use *iTrace*, a developer would need to do a quick calibration and start tracking.

3. COMPARING EYE TRACKING AND INTERACTION DATA SETS

To investigate developers' detailed behavior while performing a change task, Kevic et al. [7] conducted a study with 22 developers working on three change tasks in the JabRef open source system. This is the first study that collects both eye-tracking (using our *iTrace* prototype) and interaction data (using Mylyn) simultaneously, while developers work on realistic change tasks. The analysis shows that gaze data contains substantially more elements captured, as well as more fine-grained data, providing evidence that gaze data is in fact different and captures different aspects compared to interaction data. The analysis also shows that developers working on a realistic change task only look at very few lines within a method rather than reading the whole method as was often found in studies on single method tasks. A further investigation of the eye traces of developers within methods showed that developers chase variables flows within methods. When it comes to switches between methods, the eye traces reveal that developers only rarely follow call graph links and mostly only switch to the elements in close proximity of the method within the class. Furthermore, the fine-grained gaze context showed that developers focus only on a few methods when investigating a change task. These detailed findings provide insights and opportunities for future developer support.

4. APPLICATION SCENARIOS

We now present several scenarios that discuss how the *iTrace* environment can be used to improve and enhance various software engineering tasks.

4.1 Code Summarizations

The findings from Kevic et al. [7] demonstrate that method summarization techniques could be improved by applying some program slicing first and focusing on the lines in the method that are relevant to the current task rather than summarizing all lines in the whole method. In addition, the findings suggest that a fisheye view of code zooming on methods in close proximity and blurring out others, might have potential to focus developers' attention on the relevant parts and possibly speed up code comprehension. Studies can be conducted to compare strategies adopted by developers when summarizing code elements (methods and classes for example) using code and/or documentation. The documentation could be in the form of stack overflow documentation and/or bug reports, both of which will be supported with our infrastructure. The motivation is to help automatic tools benefit from such results and enhance their accuracy when summarizing with either code or documentation available. Researchers can compare these techniques with Natural Language Processing (NLP) [8] and Information Retrieval (IR) [9, 10] techniques. Key research questions would be: Do developers use different program comprehension strategies when summarizing code elements using different sources of information (i.e., code, documentation, or both)? What do they look at during the summarization using multiple artifacts? To what extent does using different types of information impact the summarization task (in terms of quality and time)?

The expected outcomes of this research direction would be the eye movement patterns and strategies used in both cases (using code and documentation), the quality of the answers, i.e., summaries, as well as the time to answer the tasks. The quality of the summaries can be evaluated against an oracle that can possibly built by considering all annotators answers. We can compare and contrast this realistic setting with the one from Rodeghero et al. [11] where methods were shown in isolation. The key research question to be addressed in this direction is: When given an entire open source system and asked to summarize a method, what do developers look at?

4.2 Code Recommendations

We can collect eye tracking data from developers working on real change tasks in large open source systems in order to provide contextual help when they are stuck on a task thereby tailoring code recommendations to their specific user experience. The main challenge associated with this research application is that of false positives. It could be possible that the recommendations are totally off and not what the developer wanted. One way to deal with this would be to devise an algorithm that detects stray glances and avoids them during recommendations. Another challenge is when should the recommendation be made - proactively or only when requested? The key questions to be addressed in this research agenda are: Can we provide code recommendations for developers solely based on eye gaze history? And how useful are they? When do we determine that developers need help with their development session? And how do we prompt them with recommendations? What types of elements do developers search for when they are stuck on a task?

The expected outcomes of this research area would be to create a measure of relevance/degree of interest for the eye tracking data. Once this measure is created, researchers could build extensions to the *iTrace* framework via a code recommendation module that will be based on eye movement patterns and strategies used. The open and extensible nature of *iTrace* allows for such inclusions.

4.3 Predictions Based on Eye Gaze

The collected gaze data can also be used for predictions using machine learning algorithms. Fritz et al. [12] conducted a study that found that the pupil diameter was an important feature in predicting task difficulty. The study was on small code snippets but now can be replicated using *iTrace* on larger code bases.

Besides predicting task difficulty, one can also predict developer expertise using eye gaze features such as number of fixations, fixation durations, pupil diameter, and scan path patterns. Experts and novices have different eye gaze behavior when reading code [13, 14]. Given an eye tracking developer data set, can we determine with some level of confidence if the data comes from an expert or a novice? We believe this is possible as we learn more about how developers (both experts and novices) behave while solving tasks. The more data we collect, the better our algorithms learn to model and predict expertise. The answer to such a question of predicting expertise is also of importance in coding interviews. Imagine a scenario where an interviewer would like to know based on eye movements, how skilled the interviewee actually is. Or more importantly, with eye tracking, they can see how an interviewee followed the right approach to the solution but did not quite come up with an entirely correct solution. A similar argument can be made about how a code reviewer goes about reviewing. Eye tracking developers lets us peek into these valuable insights that are otherwise lost if not tracked.

A third possibility is the prediction of developer fatigue. Blink rate and revisits/regressions can be used to determine fatigue. As *iTrace* learns more about what is normal for a developer, fatigue will be a feature that will stand out more based on the rules we provide. We can build a cognitive model that continuously learns from added gaze sessions. Emotion mining and prediction is also an interesting future possibility.

4.4 Continuous Traceability

Gotel et al. [15] gives an analysis of the requirements traceability problem. The issue of human effort and usability in traceability is of paramount importance, and needs to be thoroughly addressed for software traceability to be successfully adopted in industry and to become a common developer routine. One of the main reasons cited for this problem is the high error-prone human effort and costs required to document and maintain the traceability links over time and the high rate of false positives [16]. Therefore, traceability often downgrades to a low-priority activity with no obvious immediate benefits to the developers or managers.

We conducted a pilot study [17] on a small system (iTrust), to determine if it was possible to infer traceability links from eye gaze. In order to do this, we devised an algorithm that gives a higher weight to source code elements seen later in the session. The rationale behind this is that when a developer first starts a session, they are not sure of what they are interested in. But as the session comes to an end, they only focus their gaze on source code entities they think are related to the task. The results were promising. Software traceability researchers could work at investigating the following questions: How well can the models based on human gaze capture software traceability data (links) in large realistic open source and industrial systems? How much reduction do we see in human effort while collecting, recovering, or maintaining software traceability data (links) using the *iTrace* infrastructure in a continuous fashion?

We envision a future where continuous traceability is possible when we capture and use human gaze information to inform traceability link generation and evolution while developers work on change tasks. *iTrace* will silently observe and document the developers' eye movements while they are working on tasks and provide a novel platform that would directly support two key software traceability tasks: traceability link generation/recovery, and traceability link maintenance and evolution.

4.5 Benchmarks on Eye Gaze Data

Techniques for feature location are commonly evaluated on benchmarks formed from commits [18]. Although, commits are a reasonable source, they only capture entities that were eventually changed to fix a bug or resolve a feature. We did some preliminary work on investigating another type of benchmark based on eye tracking data. Eye tracking data provides insights into activities that go beyond entities that were changed in a commit. To establish a benchmark, we analyzed eye-gaze information for five developers from the previous study [17] required to perform bug-localization tasks on the open source subject system JabRef. We found results from the gaze tracking algorithm [17] to be more specific than the rankings from current IR methods. The gaze algorithm also exhibited the notable property of finding useful SCEs (based on an informal survey on another set of developers) given unsuccessful attempts to fix a bug by developers. This indicates that a developer might have found a starting point but not the exact solution yet. With commits, the IR techniques can only be evaluated if the solution is committed. The eye tracking benchmark can be used even if the solution is incomplete. The transparency and minimal effort required by developers makes gaze tracking as a benchmark an attractive possibility for researchers to investigate further.

5. DISCUSSION AND CONCLUSIONS

iTrace is a radical departure from existing approaches that rely on the conventional structural and semantic analysis in software artifacts. This paradigm requires some challenges to be addressed. The first challenge has to do with making sure we get accurate data from the eye tracking device. Because of the noise associated with any biometric device, we need to make sure our data is accurate and drift is manageable. We propose an automated fixation correction algorithm [19] that is a first step towards this direction. In order to tackle this problem, we need to borrow ideas from artificial intelligence and machine learning to detect drifts and offsets in the data and correct them on the fly.

An additional concern is that the proposed paradigm would generate a massive amount of eye tracking data. Such volumes could introduce scalability issues and pointless gazes. For *iTrace*, a self-feedback based learning loop can be incorporated with the goal of reducing the number of false positives. Our initial implementation handles the elimination of stray gazes quite well. The developer validation of accuracy such as in the case of traceability links could be fed back in the link recovery algorithm. According to eye-tracking vendors, a state-of-the-art eye tracker works well for approximately 98% of the population. We only expect the technology to improve to accommodate more samples as it has been in recent years. Finally, we initiated a call for standardization of visual effort metrics [20] as many researchers use different terms to mean similar things and in some cases different things. This presents problems when comparing studies across different research groups. There needs to be some community effort in standardization of visual effort metrics pertaining to studying software developers.

In this paper, we discuss an extensible infrastructure namely *iTrace* that researchers can use to build modules for specific application scenarios mentioned above. Eye trackers tell you where a person is looking but they do not tell you where they are not looking. We need to synergistically use eye tracking data with other existing techniques to our advantage to help towards the common goal of improving how software is built. The central premise in *iTrace* is to use eye-tracking equipment to implicitly collect developers' eye activity on software artifacts (that automatically map to relevant artifact elements on the fly) in an unobtrusive way while they are performing software maintenance tasks.

The anonymized eye tracking data sets can be made available to researchers and eye tracking mining challenges similar to the challenges at the Mining Software Repositories conferences can be undertaken. This engagement in the community will further improve understanding the gaze datasets. There is a lot of potential to use methods from data mining, information retrieval, artificial intelligence, and cognitive psychology to help better use the data collected on developer eye gaze. Besides software engineering research and practice, eye tracking studies can also help inform software engineering education thereby developing new guidelines (such as coding style for better readability) for educators.

6. REFERENCES

[1] K. Rayner, "Eye Movements in Reading and Information Processing: 20 Years of Research," *Psychological Bulletin,* vol. 124, pp. 372-422, 1998.

[2] E. Soloway and K. Ehrlich, "Empirical Studies of Programming Knowledge," *Software Engineering, IEEE Transactions on,* vol. SE-10, pp. 595-609, 1984.

[3] R. Turner, B. Sharif, and A. Lazar, "An Eye-tracking Study Assessing the Comprehension of C++ and Python Source Code," presented at ETRA 2014.

[4] B. Sharif, G. Jetty, J. Aponte, and E. Parra, "An Empirical Study Assessing the Effect of SeeIT 3D on Comprehension,", VISSOFT 2013, Eindhoven, Netherlands.

[5] Z. Sharafi, Z. Soh, and Y.-G. Guéhéneuc, "A Systematic Literature Review on the Usage of Eye Tracking in Software Engineering," *IST Journal,* 2015.

[6] T. Shaffer, J. Wise, B. Walters, S. Müller, M. Falcone, and B. Sharif, "iTrace: Enabling Eye Tracking on Software Artifacts Within the IDE to Support Software Engineering Tasks," in *ESEC/FSE 2015*, Bergamo, Italy, 2015, pp. 954-957.

[7] K. Kevic, B. Walters, T. Shaffer, B. Sharif, D. Shepherd, and T. Fritz, "Tracing Software Developers' Eyes and Interactions for Change Tasks," in *ESEC/FSE 2015*, Bergamo, pp. 202-213.

[8] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker, "Towards automatically generating summary comments for Java methods," presented at ASE 2010.

[9] S. Haiduc, J. Aponte, L. Moreno, and A. Marcus, "On the Use of Automated Text Summarization Techniques for Summarizing Source Code," in *WCRE* 2010, pp. 35-44.

[10] B. P. Eddy, J. A. Robinson, N. A. Kraft, and J. C. Carver, "Evaluating source code summarization techniques: Replication and expansion," in *ICPC* 2013, pp. 13-22.

[11] P. Rodeghero, C. McMillan, P. W. McBurney, N. Bosch, and S. D'Mello, "Improving Automated Source Code Summarization via an Eye-Tracking Study of Programmers," presented at ICSE 2014, Hyderabad, India.

[12] T. Fritz, A. Begel, S. Müller, S. Yigit-Elliott, and M. Züger, "Using Psycho-Physiological Measures to Assess Task Difficulty in Software Development," presented at ICSE 2014

[13] M. E. Crosby and J. Stelovsky, "How do we read algorithms? A case study," *IEEE Computer,*v.23,pp.24-35, 1990.

[14] T. Busjahn, R. Bednarik, A. Begel, M. Crosby, J. H. Paterson, C. Schulte*, et al.*, "Eye Movements in Code Reading: Relaxing the Linear Order," in *ICPC* 2015, pp. 255-265.

[15] O. C. Z. Gotel and A. C. W. Finkelstein, "An Analysis of the Requirements Traceability Problem" *ICRE* 1994,pp. 94-101.

[16] A. De Lucia, F. Fasano, R. Oliveto, and G. Tortora, "Recovering Traceability Links in Software Artefact Management Systems using Information Retrieval Methods," *TOSEM,* vol. 16, p. article no. 13, 2007.

[17] B. Walters, T. Shaffer, B. Sharif, and H. Kagdi, "Capturing Software Traceability Links from Developers' Eye Gazes," in *ICPC*, Hyderabad, India, 2014, pp. 201-204.

[18] M. Gethers, B. Dit, M. Revelle, and D. Poshyvanyk, "Feature location in source code: A taxonomy and survey," *JSME,* vol. 25, 2013.

[19] C. Palmer and B. Sharif, "Towards Automating Fixation Correction for Source Code," in *ETRA 2016*, pp. 65-68.

[20] Z. Sharafi, T. Shaffer, B. Sharif, and Y.-G. Guéhéneuc, "Eye-tracking Metrics in Software Engineering," in *APSEC*, New Delhi, India, 2015, pp. 96-103.