Eye Movements in Code Review

Andrew Begel
Microsoft Research
Redmond, WA, USA
andrew.begel@microsoft.com

Hana Vrzakova
University of Eastern Finland
Joensuu, Finland
hanav@uef.fi

ABSTRACT

In order to ensure sufficient quality, software engineers conduct code reviews to read over one another's code looking for errors that should be fixed before committing to their source code repositories. Many kinds of errors are spotted, from simple spelling mistakes and syntax errors, to architectural flaws that may span several files. However, we know little about how software developers read code when looking for defects. What kinds of code trigger engineers to check more deeply into suspected defects? How long do they take to verify whether a defect is really there? We conducted a study of 35 software engineers performing 40 code reviews while capturing their gaze with an eye tracker. We classified each code defect the developers found and captured the patterns of eye gazes used to deliberate about each one. We report how long it took to confirm defect suspicions for each type of defect and the fraction of time spent skimming the code vs. carefully reading it. This work provides a starting point for automating code reviews that could help engineers spend more time focusing on the difficult task of defect confirmation rather than the tedious task of defect discovery.

CCS CONCEPTS

• **Software and its engineering** → *Software verification and validation*;

KEYWORDS

Code review, Eye tracking

ACM Reference Format:
Andrew Begel and Hana Vrzakova. 2018. Eye Movements in Code Review. In *EMIS '18: Symposium on Eye Movements in Programming, June 14–17, 2018, Warsaw, Poland*. ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3216723.3216727

1 INTRODUCTION

Large-scale software development activities often include some kind of code review, in which developers read through diffs looking for bugs, performance issues, security problems, etc. that may have a negative impact on the product. They report the issues to the code's author, who must address the concerns before the code is allowed into the code repository.

EMIS '18, June 14–17, 2018, Warsaw, Poland

ACM ISBN 978-1-4503-5792-0/18/06.
https://doi.org/10.1145/3216723.3216727

Code review practices have long been studied by software engineering researchers [Bacchelli and Bird 2013; Bird et al. 2015; Fagan 1999] and eye-tracking has been helpful for studying source code comprehension (for recent literature reviews, see [Obaidellah et al. 2018; Sharafi et al. 2015]). The reading patterns of programmers have been linked to expertise [Busjahn et al. 2015], task workload [Fritz et al. 2014a], and frustration with the code [Müller and Fritz 2015]. Several researchers have focused on the cognitive processes that engineers experience when reading through code for some kind of review, rather than general program comprehension [Fritz et al. 2014b; Kevic et al. 2015; Rodeghero et al. 2014; Sharif et al. 2012; Uwano et al. 2006]. Eye tracking can help us discover exactly how engineers scan through source code looking for suspicious patterns of code, and identify which patterns trigger them to investigate further to validate their suspicions and find an underlying defect. Understanding the kinds of code make developers suspicious could one day enable us to automate the defect scanning process, relieving a tedious part of code review.

We watched 40 professional software developers perform code reviews as part of their daily work. We had each of them do their reviews with an eye tracker attached to their computer, so we could watch where they looked as they conducted the review looking for defects. We classified five kinds of code elements that triggered review comments and measured how long it took to deliberate over them. Finally, our gaze analysis enabled us to identify how fast developers read code, skimming quickly when looking for defects, carefully reading when validating a suspected defect.

Our contributions include a classification of the code elements that trigger deliberation periods, along with how long it takes to verify one's suspicions, and an early analysis of the differences between code skimming and careful reading during code reviews.

2 BACKGROUND

Code reviews began as a way for software engineers to rigorously check over one another's code to look for defects at a development stage early enough to minimize the cost of fixing them [Boehm 1976]. Software engineers have come up with many different processes for looking through the code, including defect-based reading, perspective-based reading, and checklist-based walkthroughs [Ciolkowski et al. 2002]. Code reviews provide advantages beyond the simple search for code defects; programmers reviewing the code learn about one other's projects, increasing team awareness, spreading knowledge about the code, and facilitating discussions of best practices and alternative solutions. Bidirectional knowledge transfer has been ranked as the second-most important benefit of the contemporary code review [Bacchelli and Bird 2013]. However, a lack of familiarity with the source code and time constraints can inadvertently lead to a shallow and superficial code review without any direct benefits to the code.

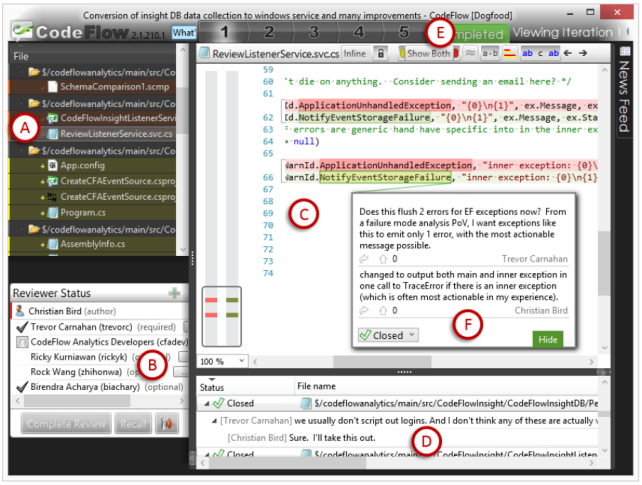

Figure 1: A layout of CodeFlow. (C) presents the main window with the source code. (F) illustrates the pop-up window for a reviewer's comment.

In contemporary code reviews, one developer evaluates another's code in a lightweight and informal way, facilitated by dedicated code review software. Typical software supports review scheduling and reviewer assignment, and synchronizes interactions to enable reviewers to write comments and sign off on the review when they are satisfied with the code changes [Bacchelli and Bird 2013]. When authors finish a set of code modifications, they use the software to identify and choose colleagues to review their code. Those colleagues then receive an email notification to open the tool and begin their review. As they work, they can right-click on a piece of code to leave a comment about that code that will be seen by the author and any subsequent reviewers. They can also leave a comment about the entire set of changes. When done, they click a button to send the comments back to the author, along with their approval or rejection of the change. The author then revises the code and sends it back out to be reviewed again. When all reviewers have approved the changes, the author is permitted to commit it into the source code repository.

Uwano *et al.* studied graduate students conducting code reviews on small, single-screen, C programs provided by the authors [Uwano et al. 2006]. They discovered that their gaze patterns followed a common scanpath, first reading code top to bottom, and then rereading a few parts in more depth. Students who scanned the *entire* codebase spent less time spotting the seeded bugs than those who looked at less code. Our experiment is similar, except our participants were professionals conducting reviews of many files of code as part of their regular work day. Our report includes more details about the kinds of code that trigger deliberations, but shows similar findings between skimming and focused reading.

Sharif *et al.* replicated and validated Uwano *et al.*'s study, finding that a longer initial scan correlated with quicker defect detection [Sharif et al. 2012]. In addition, they hypothesize that experts use top-down comprehension to focus on likely problem candidates rather than scanning all over the code.

There have been a number of studies looking at how developers find the right places to edit code while performing a change task [Fritz et al. 2014b; Kevic 2016]. This well-studied process is similar to code review, however, code review is simpler because developers use the supplied diff regions to identify where to start reading. Kevic *et al.* built on Fritz *et al.*'s work focusing on high-level navigation observations by using eye tracking to explore navigation at a finer level of granularity [Kevic et al. 2015]. Kevic *et al.* reported on a statistical analysis of reading patterns within and between methods. Developers followed data flow relationships within methods, but used text-layout order between methods. Kevic *et al.* compares their work to Rodeghero *et al.* [Rodeghero and McMillan 2015] who also used eye tracking, but for validating function summarization tools. In both cases, they found that developers focused on small, highly relevant portions of methods rather than carefully reading the entire text. In our study of code review, we find that developers quickly spot suspicious code elements, but only explore additional code in order to validate their suspicions.

3 METHODOLOGY

In this section, we describe our study methodology.

3.1 Participation

We recruited software developers from teams responsible for building customer-focused, shipping products at a large software company in the USA. Recruits were identified through daily searches through a company-wide database of every queued-up code review. We emailed employees who had been asked to review code in the last two weeks to see if they would allow us to go to their offices, instrument their machine with an eye tracker, and watch them conduct the code review.

Forty developers signed up. For their participation, we gave them USD$8 in coupons for the cafeteria. Five early sessions had to be discarded due to recording problems. In the end, we retained 35 valid code review sessions. Only one of 35 participants was a woman. On average, participants were 34 years old (SD = 5 yrs).

3.2 Procedure

Participants were consented and given a questionnaire which explored the participants' familiarity and experiences with professional code reviews. Next, eye trackers were installed on the participants' computers and run through the calibration process. Participants then conducted their previously assigned code review, often reading through 10-15 files of code changes. Five of them had enough time left in the hour-long session to perform a second review, enabling us to record 40 code reviews.

3.3 Apparatus

Each participant used CodeFlow, shown in Figure 1 [Bacchelli and Bird 2013] to access their assigned code review and read through the changes. To record the participants' gaze patterns, we installed a Tobii EyeX portable, 60Hz eye tracker [Tobii Technology AB 2018]. After calibrating it to each user at the beginning of each review, the eye tracker produces a set of (x, y) coordinates indicating where each eye is looking on the screen. We built an extension to CodeFlow to process the eye tracking data. We recorded window positions, window sizes, scroll positions, the mouse position in screen and window coordinates, the filename and path of the code

under review, and any review comments that were written. We also recorded the code review session using the Camtasia Studio 7.0 [TechSmith 2018] screen capture software.

3.4 Analysis

The stream of eye gaze positions was very noisy, often appearing in a spray paint-like pattern around the eye's actual position. After removing invalid data points (e.g. (0,0) coordinates or eyes "lost" data points), we averaged the location of the two eyes together, and applied a 10-sample median filter to smooth the signal. Our extension translates the gaze position into an editor character location to identify the code element at which the user is looking. This goal is both simpler and higher-level than typical eye tracking analyses, so instead of applying a fixation filter to the gaze positions (e.g. Tobii's I-VT algorithm) to stabilize the signal, we wrote our own filter. We apply a "majority bounding box" to the data stream. Each word in the editor is drawn inside of its own bounding box. For the most recent 10 median eye gaze positions that coincide with words in the document, the filter identifies the word whose bounding box contains the majority of those 10 points. During testing, we learned that in order to make the word identification routine accurate enough, we had to modify CodeFlow to display text using the Consolas font at 17 points.

We also assessed the subject's progress with the eye tracker. We converted the raw eye gaze data into the words (i.e. tokens) that the person read while doing the code review. From this, we computed a reading rate in tokens per second. We also normalized the value by the length of the token (since it takes longer to read a longer word), and produced a metric of characters read per second. This is similar to the intent of prior works which investigated the reviewers' reading speeds [Sharif et al. 2012; Uwano et al. 2006].

3.5 Threats to Validity

All experiments are subject to internal and external threats to validity. While we believe our knowledge of CS enables us to intuit what engineers are thinking from their eye tracking data, we cannot be sure of our interpretation. However, we were able to triangulate and refine our conclusions with the participants' own code review comments in which they clearly stated (for the code author) the defect or issue they found with the code. Second, we may have affected our participants' review performance due to our presence in their offices and the setup time for the eye tracker. Finally, we believe that our study generalizes more easily than prior work because our participants performed reviews on the actual shipping code they used at work. Prior work employed code examples created by the experimenters, which were both short and unfamiliar to the study participants [Sharif et al. 2012; Uwano et al. 2006].

4 RESULTS

Which defects in code trigger a code reviewer's decision to write a comment about it? How long does it take to find them? How fast do reviewers read through the code when looking for a defect vs. when deliberating about it? In this section, we answer these questions from our data and analyses.

Table 1: The average deliberation time spent by engineers triggered by each category of code before writing a code comment. We include the median because of the skew caused by a small number of defects requiring extensive deliberation.

Comment Target	Median (m:s)	Avg (m:s)
Single Code Element	00:15	00:19
Several Code Elements	00:20	35:16
Separated Code	01:29	02:14
Inconsistent Code	01:12	02:32
Meta Comment	00:23	00:32

4.1 Comment Triggers

To classify the code elements that trigger reviewers to comment on the code, we analyzed a random sample of 78 code review comments out of the 260 total we saw in the 35 code review videos. For each comment, we watched the video of the screen capture from the time that the person first noticed the part of the code that they eventually commented on until the time they wrote the comment. Aided by eye tracking playback, we were able to follow the parts of the code the person looked at and relate it to the nature of the comment. Figure 2 shows participant P20's gaze path while discovering, validating, and commenting on a single code issue.

We now present a list of code elements that triggered our participants to write a code review comment. The data we present is not meant to be exhaustive, especially because each participant's review was necessarily different from the others. However, to our knowledge, this kind of list has never before been presented at this low level of detail. Below, source code elements are followed by a number in parentheses which represents the number of code review comments related to that element.

Many code review comments started out with the reviewer looking at *individual code elements* inside a method, e.g. a constant (2),

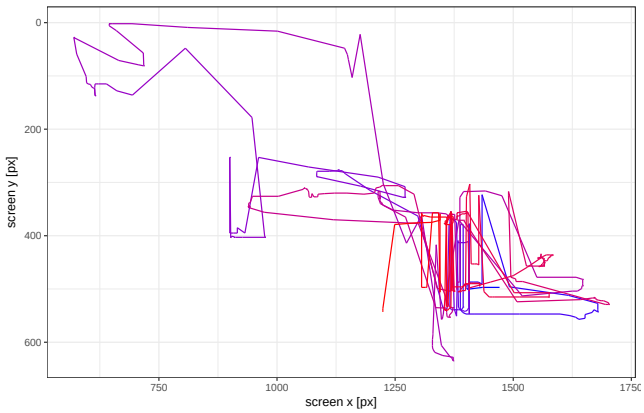

Figure 2: Gaze path for participant P20's first code review comment. The blue path occurs when P20 is triggered by the code element. The purple path is the deliberation period in which P20 validates their suspicion about the defect. The red path happens just before P20 writes the review comment.

identifier (6), method call (3), line of code (2). Simply viewing identifiers could lead directly to a code review comment, such as noticing a misspelled identifier (2), outdated identifier (1), excessively long identifier (2), or unnecessary zero-argument method call (1). One time, the code's author inadvertently checked a secret key constant into the code repository (1). Other times, there were problems associated with method signatures' doc comments, such as those that were incomplete (2) or which had incorrect capitalization (3).

Another class of review comments were triggered after the reviewer looked at *several code elements* within the class, e.g. one field (2), all fields in the class (4), a loop block with a change contained inside (1), an entire method (3), or the entire class (1). Sometimes the type of the field was wrong (2), the reviewer was confused by similarly named identifiers (1), or the reviewer noticed that a variable name was shadowed by one from an outer scope (1). Reviewers also noticed when two contiguous statements could be merged (1), or that two non-contiguous statements were related and should be moved closer together (2). Reviewers looked at a method definition and a call to that method (1), or looked at several calls to the same method (1), or focused on a change that was related to a method and then read the entire method (2).

Reviewers were puzzled by *differences between two or more related, but separated, pieces of code*, for example between a doc comment and a method name (2), or between the doc comment for a class and the file name (1). They also noticed opportunities for relating code in several files, for example, relating data in one class to validation code in another (1), refactoring common code into a common method (2), or asking to move a field to another class (1). Sometimes the request was simpler, for example, to clean up messy code (2), or to tidy inconsistent formatting (1).

Reviewers looked for *inconsistent* code changes, e.g. finding when a change in one place should have been applied in several others (4), or looking for similar code in other places but not finding any (1). They also made high-level comments about how the author wrote some code, for example, verifying that the author implemented the code idiom properly (1), theorizing why a particular code idiom was used in the codebase (1), admonishing the author not to use a particular technology in the code (1), or suggesting a way to improve the performance of the code (1).

Finally, there were several *meta comments*, such as when the reviewer summarized the entire review in a single review comment (1), converted a TODO code comment into a review comment (1), suggested adding something else to the code (1), or more commonly, simply reacted to a code review comment made by a prior reviewer (14). Twice, apropos of nothing, the reviewer created a comment, i.e. they saw something, and without waiting or looking at anything else, they wrote a fairly involved review comment. We believe the reviewer had been thinking about the code on their own time and wrote up their thoughts as soon as they realized they were looking at the right file in the code.

For each comment, we measured the *deliberation time*, which is the amount of time from the initial trigger until the reviewer started writing the review comment. We find a skewed range of times from 1 second all the way to 11 minutes and 31 seconds (Mean=74 sec, Median=23 sec). As you can see in Table 1, this seems to be mainly due to the related statements, separated code, and inconsistent code review categories, which had some long deliberation times and some nested deliberations, i.e. the person started looking at one of the code elements, then digressed to another code review comment, and then returned to the original. The deliberation times for single code elements and meta comments were almost always very short.

4.2 Skimming vs. Reading

Code review is predominantly a skimming process in which the reviewer initially looks through the code quite fast. When a particular part of the code catches the reviewer's eye, he/she will slow down to read it more carefully. On average, participants in the study read 27 words per second, but the distribution of this value was highly skewed (Median=66, SD=39). To account for the fact that it takes more time to read longer tokens, we computed the reading rate per character as well. Participants read an average of 238 characters per second (Median=731, SD=89). Most tokens were read quickly, but there was a large skew to the data, in which participants read some words very slowly.

We analyzed the difference in speeds by binarizing the reading rate into skimming vs. careful reading. *Careful reading* is defined as reading two standard deviations slower than the mean rate per person. We found that most participants skimmed through the code until they were triggered to slow down when they spotted something suspicious. We saw two cases of participants who did no careful reading prior to writing up a code review comment. In these cases, we believe the participants had prior knowledge of the code changes they were reviewing (perhaps from team meetings) and did not need to read the code carefully in order to decide what to write in the comment.

5 CONCLUSION

Now that traditional code inspections have been replaced by tool-based code reviews, software engineers review almost all of one another's code before it is checked into their source code repository. The commonplace use of code review tools and the affordability of eye trackers enables us to carefully observe the code review process at a fine temporal granularity.

In this study, we found which code elements trigger engineers to be suspicious during code reviews. From the eye tracking playback, we learned how engineers confirm or reject their defect suspicions and ultimately decide whether to leave a code review comment for the author. Our data confirms prior work showing that code review is primarily a code scanning process. We saw engineers skimming through large amounts of code before occasionally slowing down to read smaller portions of the code more carefully. Our results contribute to our field's understanding of code review, and offer opportunities to improve upon it in the future.

REFERENCES

Alberto Bacchelli and Christian Bird. 2013. Expectations, Outcomes, and Challenges of Modern Code Review. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*. IEEE Press, Piscataway, NJ, USA, 712–721.

Christian Bird, Trevor Carnahan, and Michaela Greiler. 2015. Lessons Learned from Building and Deploying a Code Review Analytics Platform. In *Proceedings of the 12th Working Conference on Mining Software Repositories (MSR '15)*. IEEE Press, Piscataway, NJ, USA, 191–201. http://dl.acm.org/citation.cfm?id=2820518.2820542

B. W. Boehm. 1976. Software Engineering. *IEEE Trans. Comput.* C-25, 12 (Dec 1976), 1226–1241. https://doi.org/10.1109/TC.1976.1674590

Teresa Busjahn, Roman Bednarik, Andrew Begel, Martha Crosby, James H. Paterson, Carsten Schulte, Bonita Sharif, and Sascha Tamm. 2015. Eye Movements in Code

Reading: Relaxing the Linear Order. In *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension (ICPC '15)*. IEEE Press, Piscataway, NJ, USA, 255–265. http://dl.acm.org/citation.cfm?id=2820282.2820320

Marcus Ciolkowski, Oliver Laitenberger, Dieter Rombach, Forrest Shull, and Dewayne Perry. 2002. Software Inspections, Reviews & Walkthroughs. In *Proceedings of the 24th International Conference on Software Engineering (ICSE '02)*. ACM, New York, NY, USA, 641–642. https://doi.org/10.1145/581339.581422

Michael E Fagan. 1999. Design and code inspections to reduce errors in program development. *IBM Systems Journal* 38, 2/3 (1999), 258.

Thomas Fritz, Andrew Begel, Sebastian C Müller, Serap Yigit-Elliott, and Manuela Züger. 2014a. Using Psycho-physiological Measures to Assess Task Difficulty in Software Development. *Proceedings of the 36th International Conference on Software Engineering* (2014), 402–413.

Thomas Fritz, David C. Shepherd, Katja Kevic, Will Snipes, and Christoph Bräunlich. 2014b. Developers' Code Context Models for Change Tasks. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. ACM, New York, NY, USA, 7–18. https://doi.org/10.1145/2635868.2635905

Katja Kevic. 2016. Recognizing Relevant Code Elements During Change Task Navigation. In *Proceedings of the 38th International Conference on Software Engineering Companion (ICSE '16)*. ACM, New York, NY, USA, 851–854. https://doi.org/10.1145/2889160.2889270

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

Sebastian C. Müller and Thomas Fritz. 2015. Stuck and frustrated or in flow and happy: Sensing developers' emotions and progress. *Proceedings - International Conference on Software Engineering* 1 (2015), 688–699.

Unaizah Obaidellah, Mohammed Al Haek, and Peter C-H Cheng. 2018. A Survey on the Usage of Eye-Tracking in Computer Programming. *ACM Computing Surveys (CSUR)* 51, 1 (2018), 5.

Paige Rodeghero and Collin McMillan. 2015. An Empirical Study on the Patterns of Eye Movement during Summarization Tasks. *2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)* (2015), 1–10.

Paige Rodeghero, Collin McMillan, Paul W. McBurney, Nigel Bosch, and Sidney D'Mello. 2014. Improving Automated Source Code Summarization via an Eye-tracking Study of Programmers. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. ACM, New York, NY, USA, 390–401. https://doi.org/10.1145/2568225.2568247

Zohreh Sharafi, Zephyrin Soh, and Yann Gael Gueheneuc. 2015. A systematic literature review on the usage of eye-tracking in software engineering. *Information and Software Technology* 67 (2015), 79–107.

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

TechSmith. 2018. Screen Recording and Video Editing Software | Camtasia | TechSmith. https://www.techsmith.com/video-editor.html

Tobii Technology AB. 2018. Tobii EyeX Specification. http://tobiigaming.com/product/tobii-eyex/.

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