Tracing Software Developers' Eyes and Interactions for Change Tasks

Katja Kevic[†], Braden M. Walters[*], Timothy R. Shaffer[*],
Bonita Sharif[*], David C. Shepherd[‡], Thomas Fritz[†]

[†]University of Zurich, Switzerland
Department of Informatics
{kevic,fritz}@ifi.uzh.ch

[*]Youngstown State University, USA
Department of CS and IS
{bmwalters01,trshaffer}@student.ysu.edu
bsharif@ysu.edu

[‡]ABB Corporate Research, USA
Industrial Software Systems
david.shepherd@us.abb.com

ABSTRACT

What are software developers doing during a change task? While an answer to this question opens countless opportunities to support developers in their work, only little is known about developers' detailed navigation behavior for realistic change tasks. Most empirical studies on developers performing change tasks are limited to very small code snippets or are limited by the granularity or the detail of the data collected for the study. In our research, we try to overcome these limitations by combining user interaction monitoring with very fine granular eye-tracking data that is automatically linked to the underlying source code entities in the IDE.

In a study with 12 professional and 10 student developers working on three change tasks from an open source system, we used our approach to investigate the detailed navigation of developers for realistic change tasks. The results of our study show, amongst others, that the eye-tracking data does indeed capture different aspects than user interaction data and that developers focus on only small parts of methods that are often related by data flow. We discuss our findings and their implications for better developer tool support.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement

General Terms

Human Factors, Experimentation

Keywords

eye-tracking, gaze, change task, user study

1. INTRODUCTION

Software developers spend a majority of their time working on change tasks, such as bug fixes or feature additions [25]. In order to successfully complete these tasks, they have to read, navigate and edit the relevant pieces of code [22, 16]. Since the inception of software development, researchers have been studying how developers read and navigate code, and what kind of knowledge they acquire (e.g., [43, 5, 22]). The more we know about a developer's work, the better we are able to support her, for instance, by reducing information overload [21], improving defect prediction [24], or providing automatic navigation recommendations [13, 29].

Yet relatively few studies have been undertaken to investigate detailed navigation behavior of developers for realistic change tasks. The lack of realistic studies is due to the significant challenges and effort of acquiring the time of professional software developers to participate as well as of capturing, transcribing and coding longer sessions of developers' work on change tasks. More recently, approaches have been developed to automatically capture more data from a developer's interactions with source code elements in an integrated development environment (IDE) [2, 21]. These approaches capture source code elements mostly on the class and method level and are based on explicit user interactions with the mouse or keyboard.

Recent advances in technology afford new opportunities to collect a wide variety of more detailed information on a software developer and her work. Studies with sensors for tracking biometric features, such as eye gaze, have generated new insights on developers' work on small code tasks, such as perceptions of difficulty [15], brain activation patterns [42], the scanning patterns of code [36] or the ease of comprehending different representations of code [40, 6]. Most of these studies focus on very small code comprehension tasks with a single method or class, in particular, since they require manual linking between the gaze data collected with an eye-tracker and the source code elements a developer looked at.

While these studies provide valuable first insights, the advances in technology open up the opportunity to address further important research questions, such as, what is a developer's fine-grained navigation behavior for realistic change tasks, what is the difference in the data captured through eye-tracking and interaction logging and how can we use eye-tracking data to support developers. Answering these questions will allow us to better understand developers' comprehension of large code bases and to develop better and more fine-granular tool support for developers.

In our research, we take advantage of the opportunities that eye-tracking provides, and extend previous research by addressing some of these questions by focusing on more realistic change tasks to investigate how developers read and navigate through code while working. In particular, we are examining how eye-tracking data differs from the data captured by monitoring user interactions in an IDE, how developers' eyes move within and between methods, and how these newly gained insights can be used to better support developers in their work on change tasks. We developed an approach to automatically link eye-tracking data to the source code elements in the IDE, which combines the ease of automatically collecting data in an IDE with the finer granularity of eye-tracking data. Our approach also supports the scrolling and switching of code editor windows by developers and thus allows for change task investigations on a realistic-sized code base and is not limited to very small tasks as most previous studies are. This new approach for conducting user studies in software development provides the potential to reduce the cost of generating detailed, rich user data and valuable insights in developers' navigation behavior.

We conducted a study with 22 participants, 12 professional developers and 10 students, working on three realistic change tasks for a total of 60 minutes while automatically tracing their eye gazes and their explicit user interactions in the code editor of the Eclipse IDE. Our analysis of the gathered data shows, amongst other results, that eye-tracking captures substantially different data than a developer's navigation within the IDE, that developers only look at a few lines of a method when working on a change task and that these lines are often related to the data flow of variables within these methods. These results also provide evidence for the value of combining eye-tracking with interaction monitoring in an IDE in the future.

This paper makes the following contributions:

- Study findings based on eye-tracking and user interaction monitoring that provide insights into the detailed navigation behavior of 22 developers working on realistic change tasks.
- An approach to automatically and on-the-fly capture the fine-grained source code elements a developer looks at in an IDE while working with large files, thereby significantly improving current state-of-the-art that limits eye tracking studies to only single methods.
- A discussion on the value of the data gathered and the opportunities the data and the findings offer for better developer support.

2. RELATED WORK

Our work can be seen as an evolution of techniques to empirically study software developers working on change tasks. Therefore, we classify related work roughly along its evolution into three categories: manual capturing, user interaction monitoring, and biometric sensing of developers' work.

Manual Capturing.

Researchers have been conducting empirical studies of software developers for a very long time. Many of the earlier studies focused on capturing answers of participants after performing small tasks to investigate code comprehension and knowledge acquisition (e.g., [10, 41, 32]). Later on, researchers started to manually capture more detailed data on developers' actions. Altmann, for instance, analyzed a ten minute interval of an expert programmer performing a task and used computational simulation to study the near-term memory [5]. Perhaps one of the most well-known user studies from this category is the study by Ko et al. [22]. In this study, the authors screen captured ten developers' desktops while they worked on five tasks on a toy-sized program and then hand-coded and analyzed each 70 minute session. In a study on developers performing more realistic change tasks, Fritz et al. [16] used a similar technique and manually transcribed and coded the screen captured videos of all participants. While all of these studies are a valuable source of learning and led to interesting findings, the cost of hand-coding a developers' actions is very high, which led to only a limited number of studies providing detailed insights on a developers' behavior.

User Interaction Monitoring.

More recently, approaches have been developed to automatically capture user interaction data within an IDE, such as Mylyn [2, 20, 21]. Based on such automatically captured interaction histories—logs of the code elements a developer interacted with along with a timestamp—researchers have, for instance, investigated how developers work in an IDE [27], how they navigate through code [28, 29, 47], or how developers' micro interaction patterns might be used for defect prediction [24]. Even the Eclipse team themselves undertook a major data collection project called the Usage Data Collector that, at its peak, collected data from thousands of developers using Eclipse. Overall, the automatic monitoring of user interactions was able to significantly reduce the cost for certain empirical studies. However, these studies are limited to the granularity and detail of the monitoring approach. In case of user interaction monitoring, the granularity is predominately the method or class file level and detailed information, such as the time a developer spends reading a code element or when the developer is not looking at the screen, is missing and makes it more difficult to fully understand the developers' traces.

Biometric Sensing.

In parallel to the IDE instrumentation efforts, researchers in the software development domain have also started to take advantage of the maturing of biometric sensors. Most of this research focuses on eye-tracking [31, 19], while only few studies have been conducted so far that also use other signals, such as an fMRI to identify brain activation patterns for small comprehension tasks [42], or a combination of eye-tracking, EDA, and EEG sensors to measure aspects such as task difficulty, developers' emotions and progress, or interruptibility [15, 26, 52].

By using eye-tracking and automatically capturing where a developer is looking (eye gaze), researchers were able to gain deeper insights into developers' code comprehension. One of the first eye-tracking studies in program comprehension was conducted by Crosby et al., who found that experts and novices differ in the way they looked at English and Pascal versions of an algorithm [11]. Since then, several researchers have used eye-tracking to evaluate the impact of developers' eye gaze on comprehension for different kinds of representations and visualizations such as 3D visualizations [37], UML diagrams [51, 12], design pattern layout [39], programming languages [44], and identifier styles [40, 8]. Researchers have also used eye-tracking to investigate developers' scan patterns for very small code snippets, finding that participants first read the entire code snippet to get an idea of the program [45]. Other researchers examined different strategies novice and expert developers employ in program comprehension and debugging [7, 6], as well as where developers spend most time when reading a method to devise a better method summarization technique [34]. Finally, researchers have also used eye-tracking to evaluate its potential for detecting software traceability links [38, 48, 49]. All of these studies are limited to very small, toy applications or single page code tasks. Furthermore, in many of these studies, the link between the eye gaze (e.g. a developer looking at pixel 100,201 on the screen) to the elements in an IDE (e.g., a variable declaration in line 5 of method OpenFile) had to be done manually.

To the best of our knowledge, this paper presents the first study on realistic change task investigation that collects and analyzes both, developers' user interaction and eye gaze data. Due to the approach we developed that automatically links eye gaze data to the underlying source code elements in the IDE, we reduce the need of manual mapping and are able to overcome the single page code task limitation of previous studies, allowing for change tasks on a realistic-sized code base with developers being able to naturally scroll and switch editor windows.

3. EXPLORATORY STUDY

We conducted an exploratory study with 22 participants to investigate the detailed navigation behavior of developers for realistic change tasks. Each participant was asked to work for a total of 60 minutes on three change tasks of the open source system *JabRef* in the Eclipse IDE, while we tracked their eyes and monitored their interaction in the IDE. For the eye-tracking part, we developed a new version of our Eclipse plugin called iTrace [49], by adding automatic linking between the eye gazes captured by the eye-tracking system to the underlying fine-grained source code elements in the IDE in real-time. All study materials are available on our website [3].

3.1 Procedure

The study was conducted in two steps at two physical locations. In the first step, we conducted the study with twelve professional developers on site at ABB. We used a silent and interruption free room that was provided to us for this purpose. In the second step, we conducted the study with ten students in a university lab at Youngstown State University. We used the same procedure as outlined below at both locations.

When a participant arrived at the study location, we asked her to read and sign the consent form and fill out the background questionnaire on their previous experience with programming, Java, bug fixing and Eclipse. Then, we provided each participant a document with the study instructions and a short description of JabRef. Participants were encouraged to ask questions at this stage to make sure they understood what they were required to do during the study. The entire procedure of the study was also explained to them by a moderator. In particular, participants were told that they will be given three bug reports from the JabRef repository and the goal was to fix the bug if possible. However, we did mention that the ultimate goal was the process they used to eventually fix the bug and not the final bug fix.

For the study, participants were seated in front of a 24-inch LCD monitor. When they were ready to start, we first performed a calibration for the eye-tracker within iTrace. Before every eye-tracking study, it is necessary to calibrate the system to each participants' eyes in order to properly record gaze data. Once the system was successfully calibrated, the moderator turned on iTrace and Mylyn to start collecting both types of data while the participants worked on the change tasks. Participants were given time to work on a sample task before we started the one hour study on the three main tasks. At the end of each change task, we had a time-stamped eye gaze session of line-level data and the Mylyn task interactions saved in a file for later processing. We also asked each participant to type their answer (the class(es)/method (s)/attribute(s) where they might fix the bug) in a text file in Eclipse at the end of each change task.

For the study, each participant had Eclipse with iTrace and Mylyn plugins installed, the JabRef source code, a command prompt with instructions on how to build and run JabRef, and sample bib files to test and run JabRef. There were no additional plugins installed in Eclipse. The study was conducted on a Windows machine. Each participant was able to make any necessary edits to the JabRef code and run it. They were also able to switch back and forth between the Eclipse IDE and the JabRef application. iTrace detects when the Eclipse perspective is in focus and only then collects eye gaze data. We asked subjects not to resize the Eclipse window to maintain the same full screen setup for all subjects and not to browse the web for answers since we wanted to control for any other factors that might affect our results.

3.2 Participants

For our study, we gathered two sets of participants: twelve professional developers working at ABB Inc. that spend most of their time developing and debugging production software, and ten undergraduate and graduate computer science students from Youngstown State University. Participants were recruited through personal contacts and a recruiting email. All participants were compensated with a gift card for their participation.

All professional developers reported having more than five years of programming experience. Seven of the twelve reported having more than five years of experience programming in Java, while the other five reported having about one year of Java programming experience. Nine of the twelve professional participants also rated their bug fixing skills as above average or excellent. With respect to IDE usage, four of the twelve stated that they mainly use Visual Studio for work purposes and that they were not familiar with the Eclipse IDE, and one participant commented on mainly being a vim and command line user. Of the twelve professional developers, two were female and ten were male.

Among the ten student participants, one participant had more than five years of programming experience, five students had between three and five years programming experience, and four of them had less than two years programming experience. Three of the students reported having between three and five years of Java programming experience, while seven students had less than two years. Three of the ten students rated their bug fixing skills as above average, and seven rated them as average. All but one student stated that they were familiar with the Eclipse IDE. Of the ten students, one was female and nine male.

3.3 Subject System and Change Tasks

We chose *JabRef* as the subject system in this study. JabRef is a graphical application for managing bibliographic databases that uses the standard LaTeX bibliographic format BibTeX, and can also import and export many other formats. JabRef is an open source, Java based system available on SourceForge [1] and consists of approximately 38 KLOC spread across 311 files. The version of JabRef used in our study was 1.8.1, release date 9/16/2005.

To have realistic change tasks in our study, we took the tasks directly from the bug descriptions submitted to JabRef on Sourceforge. Information about each task is provided in Table 1. All of these change tasks represent actual JabRef tasks that were reported by someone on Sourceforge and that were eventually fixed in a later JabRef release. The tasks were randomly selected from a list of closed bug reports with varied difficulty as determined by the scope of the solution implemented in the repository. We selected a set of three change tasks to be performed by all participants. We consider this to be a reasonable number of tasks without causing fatigue in the one hour of the study. A time limit of 20 minutes was placed for each task so that participants would work on all three tasks during the one hour study. To familiarize participants with the process and the code base, each participant was also given a sample task before starting with the three main tasks for which we did not analyze the tracked data. The task order of the three tasks was randomly chosen for each participant.

3.4 iTrace

For capturing eye-tracking data and linking it to source code elements in the IDE, we developed and use a new version of our Eclipse plugin iTrace [35]. For this new version, we added the ability to automatically and on-the-fly link eye gazes to fine-grained AST source code elements, including method calls, variable declarations and other statements in the Eclipse IDE. In particular, iTrace gives us the exact source code element that was looked at with line-level granularity. Furthermore, to support a more realistic work setting, we added features to properly capture eye gazes when the developer scrolls or switches code editor windows in the IDE, or when code is edited. Eye-tracking on large files that do not completely fit on one screen is particularly challenging as none of the state-of-the-art eye-tracking software supports scrolling while maintaining context of what the person is looking at. Our new version of iTrace overcomes this limitation and supports the collection of correct eye gaze data when the developer scrolls both, horizontally and vertically as well as when she switches between different files in the same or different set of artifacts.

iTrace interfaces with an eye-tracker, a biometric sensor usually in the form of a set of cameras that sit in front of the monitor. For our study, we used the Tobii X60 eye-tracker [4] that does not require the developer to wear any gear. Tobii X60 has an on-screen accuracy of 0.5 degrees. To accommodate for this and still have line-level accuracy of the eye gaze data, we chose set the font size to 20 points for source code files within Eclipse. We ran several tests to validate the accuracy of the collected data.

After calibrating the eye-tracker through iTrace's calibration feature, the developer can start working on a task and the eye gazes are captured with the eye-tracker. iTrace processes each eye gaze captured with the eye-tracker, checks if it falls on a relevant UI widget in Eclipse and generates an eye gaze event with information on the UI in case it does. iTrace then uses XML and JSON export solvers, whose primary job is to export each gaze event and any information attached to it to XML and JSON files for later processing.

Currently, iTrace generates gaze events from gazes that fall on text and code editors in Eclipse. These events contain the pixels X and Y coordinates relative to the top-left corner of the current screen, the validation of the left and right eye as reported by the eye-tracker (i.e., if the eye was properly captured), the left and right pupil diameter, the time of the gaze as reported by the system and the eye-tracker, the line and column of the text/code viewed, the screen pixel coordinates of the top-left corner of the current line, the file viewed, and if applicable, the fully qualified names of source code entities at the gaze location. The fully qualified names are derived from the abstract syntax tree (AST) model of the underlying source code. For this study, we implemented iTrace to capture the following AST elements: classes, methods, variables, enum declarations, type declarations, method declarations, method invocations, variable declarations, any field access, and comments. These elements are captured regardless of scope, which includes anonymous classes.

3.5 Data Collection

For this study, we collected data on participants' eye traces and their interactions with the IDE simultaneously. Since we conducted our study with the Eclipse IDE, we used the Eclipse plugin Mylyn [2, 20] to monitor user interactions. For the eye-tracking data, we used our new version of the Eclipse plugin iTrace [35].

We gathered a total of 66 change task investigations from the 12 professional developers and 10 computer science students who each worked on three different change tasks. For each of these investigations, we gathered the eye-tracking data and the user interaction logs. Due to some technical difficulties, such as a participant wearing thick glasses or too many eye gazes not being valid for a task, we excluded 11 change task investigations and ended up with 55 overall: 18 subjects investigating task 2, 16 subjects investigating task 3, and 21 subjects investigating task 4. With respect to individual method investigations over all participants and tasks, we gathered a total of 688 method investigation instances.

4. STUDY RESULTS

Based on the collected logs of eye gazes (gaze context) and user interactions (interaction context) of the 22 participants we were able to make detailed observations on how developers navigate within source code. Table 2 summarizes the gaze and interaction contexts we collected and used to infer our observations from. In the following, we structure our observations along three research foci: the difference between gaze and user interaction data, developers' navigation within methods and developers' navigation between methods.

4.1 Interaction Context and Gaze Context

O1—Gaze contexts capture substantially more, and more fine-grained data. To compare the different amounts of elements within the gaze and the interaction contexts, we

Table 1: Tasks used in the study.

ID	Bug ID	Date Submitted	Title	Scope of Solution in Repository
T2	1436014	2/21/2006	No comma added to separate keywords	multiple classes: `EntryEditor, GroupDialog FieldContentSelector, JabRefFrame`
T3	1594123	11/10/2006	Failure to import big numbers	single method: `BibtexParser.parseFieldContent`
T4	1489454	5/16/2006	Acrobat Launch fails on Win98	single method: `Util.openExternalViewer`

used a paired-samples t-test[1] with pairs consisting of the gaze and the interaction context for a task and subject.

This paired-samples t-test showed that the number of different classes contained in the gaze context ($M = 4.78, SD = 3.58$) and the number of different classes contained in the interaction context ($M = 4.42, SD = 3.00$) do not differ significantly ($t(54) = 1.98, p = .053$). Nevertheless, there were more classes captured in the gaze contexts, which turned out to be internal classes or classes defined in the same file. While there is no significant difference on a class level, there is a significant difference in the amounts of methods captured. The number of different methods within the gaze contexts ($M = 12.51, SD = 11.75$) is significantly higher than the number of different methods within the interaction contexts ($M = 6.04, SD = 4.53$), $t(54) = 4.57, p < .05$. This observation on the substantial difference in the number of elements within the gaze and interaction context provides evidence that developers often look at methods that they do not select. Approaches that only analyze interaction logs, thus miss a substantial amount of information.

When analyzing the method sequences captured in the logs, the data also shows that gaze context not only captures more elements, but also more details on the actual sequences of navigation between methods. A paired-samples t-test revealed a significant difference in the number of method switches captured in gaze contexts ($M = 73.45, SD = 78.47$) and the number of method switches captured in interaction contexts ($M = 5.75, SD = 5.17$), $t(54) = 6.52, p < .05$. Table 2 summarizes the number of unique methods and the number of method switches for each context type and participant.

O2—Gaze and Interaction Contexts capture different aspects of a developer's navigation. To evaluate whether gaze and interaction contexts capture different aspects of a developer's navigation for change task investigations, we defined ranking models based on the data available in the different contexts and compared the top ranked methods. There are a variety of models that can be used to select the most important elements within a navigation sequence [29]. For our analysis, we used single-factor models to select the most important elements in each kind of context that were also suggested in previous studies [28, 29]. To rank the methods of a gaze context we used a time-based model. This model ranks methods higher for which a developer spends more time looking at. To rank the methods of an interaction context we used a frequency model, which ranks methods higher that were visited more often.

[1]According to the central limit theorem, with large samples number (>30), the distribution of the sample mean converges to a normal distribution and parametric tests can be used [14].

The comparison of the top 5 methods for each change task investigation resulted in an average agreement of 65.03% ($SD = 32.26\%$). Comparing solely the highest ranked method for each context pair results in an agreement of 27.27%. The agreement on the top 5 most important methods however is considerably lower for change task 2 ($M = 52.31\%, SD = 34.98\%$) than for change task 3 ($M = 71.88\%, SD = 27.62\%$) and for change task 4 ($M = 70.71\%, SD = 31.32\%$). While the description for change task 3 and change task 4 include concrete hints to source code elements which are possibly important for performing the change task, change task 2 required to explore the source code more exhaustively in order to find the relevant code and a possible fix. These results illustrates that gaze context, especially in form of the time of gazes, captures aspects that are not captured in the interaction context and that might be used to develop new measures of relevance. Especially, since gaze contexts also capture elements that are not in the interaction context (**O1**), the more fine-grained gaze data might provide better and more accurate measures of relevance.

4.2 Navigation Within Methods

We base the analysis of navigation within methods solely on the gaze data, since interaction contexts do not capture enough detail to analyze within method navigation.

O3—Developers only look at few lines within methods and switch often between these lines. Figure 1 depicts the lines a professional developer (middle) and a student developer (right) looked at within a certain method and over time during a change task investigation.

Across all subjects and tasks, developers only look at few lines within a method, on average 32.16% ($SD = 24.95\%$) of the lines. The lengths of methods included in this analysis thereby differed quite a lot, with an average length of 53.03 lines ($SD = 139.37$), and had a moderate influence on the number of lines looked at by a developer, Pearson's $r = .398, p = .01$.

Participants performed on average 39.95 ($SD = 100.99$) line switches within methods. The method length again influences the amount of line switches moderately, Pearson's $r = .305, p = .01$.

Further examination of the kind of lines developers actually looked at shows that developers spend most of their time within a method looking at method invocations ($M = 4081.98ms$) and variable declaration statements ($M = 1759.6$ ms), but spent surprisingly little time looking at method signatures ($M = 1090.67$). In fact, in 319 cases out of 688 method investigations analyzed, the method signature was ignored and not looked at. Our findings demonstrate that developers who are performing an entire change task involving several methods and classes, read methods differently than

Table 2: Summary of professional (pro) and student (stu) developers' average (avg) of methods and method switches captured in the gaze and interaction context, as well as the percentage of lines read within methods.

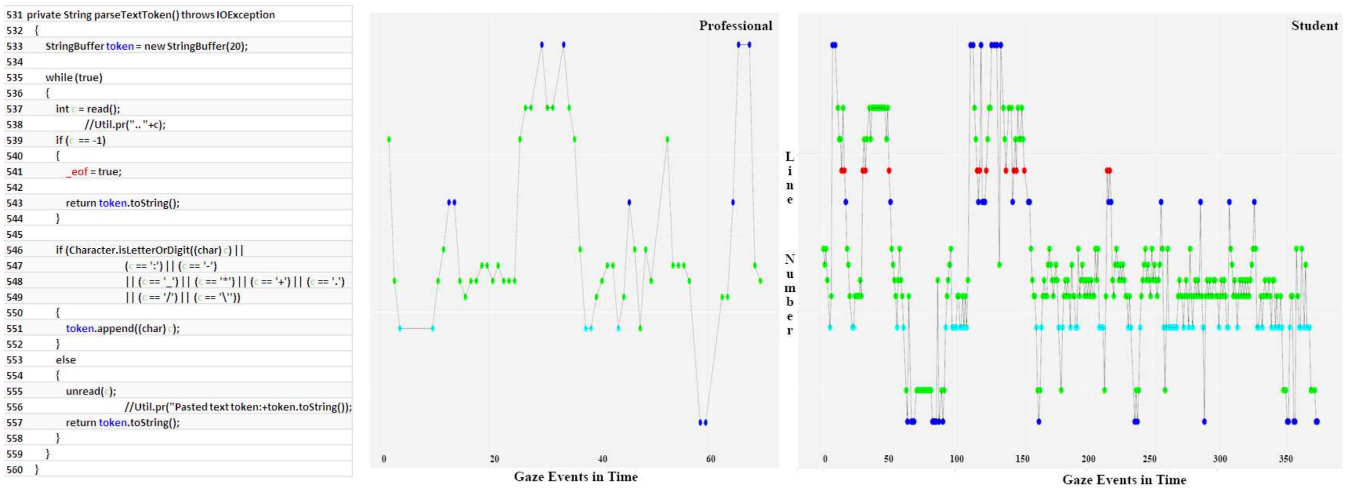
ID	avg # of method switches		average # of unique methods		avg % of lines
	gaze context	interaction context	gaze context	interaction context	read in method
P1	6.5	3	4.5	3.5	31.7%
P2	59.7	10	12	8	32.4%
P3	50	7.5	15	8	23.6%
P4	46	3.5	16.5	3.5	32.9%
P5	126	12.5	14	10.5	25.8%
P6	22.5	4.5	5.5	5.5	47.0%
P7	226	8.7	39.3	8.7	35.0%
P8	47.7	3	5.3	4	26.9%
P9	50.5	3	6.5	4	41.4%
P10	172	9	9	8	71.4%
P11	64	6.7	12.3	6	30.2%
P12	138	5	8	6	45.4%
avg pro	83.73	6.42	13.38	6.46	33.6%
S1	13.3	2	8.7	3	28.4%
S2	20	1.7	6.7	2.3	24.7%
S3	45.3	2.4	8.7	3.3	27.3%
S4	96.3	15	23.7	14.7	35.5%
S5	96	7	11.7	7.6	37.4%
S6	10.5	3.5	3	4.5	19.4%
S7	142.3	0.7	9	1.7	34.5%
S8	64	4.7	19.7	5.3	25.1%
S9	59.7	5	8.3	4.3	33.3%
S10	77	9	15	9.3	28.5%
avg stu	64.24	5.14	11.72	5.66	30.6%
total avg	**73.45**	**5.75**	**12.51**	**6.04**	**32.16%**

developers who are reading methods disconnected from any task or context, in which case the method signature might play a stronger role.

O4—Developers chase data flows within a method. To better understand how developers navigate within a method, we randomly picked six change task investigation instances from the collected gaze contexts and manually retraced the paths participants followed through a method by drawing their line switches on printouts of the methods. Closely examining these printed methods with the eye traces drawn on top, allowed us to form the hypothesis that developers often trace variables when reading a method. To further investigate this hypothesis. we selected four methods which were investigated by most participants, resulting in 40 unique method investigation instances (see Table 3). The 40 method investigation instances stem from 18 different participants and two different task. 22 of these 40 investigations stem from professional software developers, while the other 18 stem from students.

For each method, we assigned a color for each variable used within the method and colored the lines in which the variable was either defined or used in the method. We did not color lines or statements that did not include a variable. Over all four methods, we identified an average of 7.25 variable slices per method with an average of 6.2 different lines of code per slice. Then, we applied this line-to-color mapping to the sequence logs of participants who investigated these methods (see Figure 2 for an example). Within each sequence log, we ignored the lines which did not map to a slice, such as brackets or empty lines. As we are investigating if developers trace variables when reading a method we further ignored control flow statements which did not use any variable. In the event of more than one variable used in a single line, we manually checked if a color was predominantly used before or after the line was visited and decided on a color according (using the predominant color). In cases where there was no evidence of a predominant color, we picked the color of the variable that was used first in the source code line.

Our analysis revealed that developers switched between the lines of these four methods on average 178.0 (SD = 189.9) times. We then used our color coding to examine how many of these line switches are within variable slices (lines with the same color). Overall method investigation instances we



```
private String parseTextToken() throws IOException
{
    StringBuffer token = new StringBuffer(20);

    while (true)
    {
        int c = read();
                //Util.pr(".. "+c);
        if (c == -1)
        {
            _eof = true;

            return token.toString();
        }

        if (Character.isLetterOrDigit((char) c) ||
                        (c == ':') || (c == '-')
                        || (c == '_') || (c == '*') || (c == '+') || (c == '.')
                        || (c == '/') || (c == '\"))
        {
            token.append((char) c);
        }
        else
        {
            unread(c);
                        //Util.pr("Pasted text token:+token.toString());
            return token.toString();
        }
    }
}
```

Figure 1: The sequence logs mapped to line numbers and colors, with the colored source code on the left.

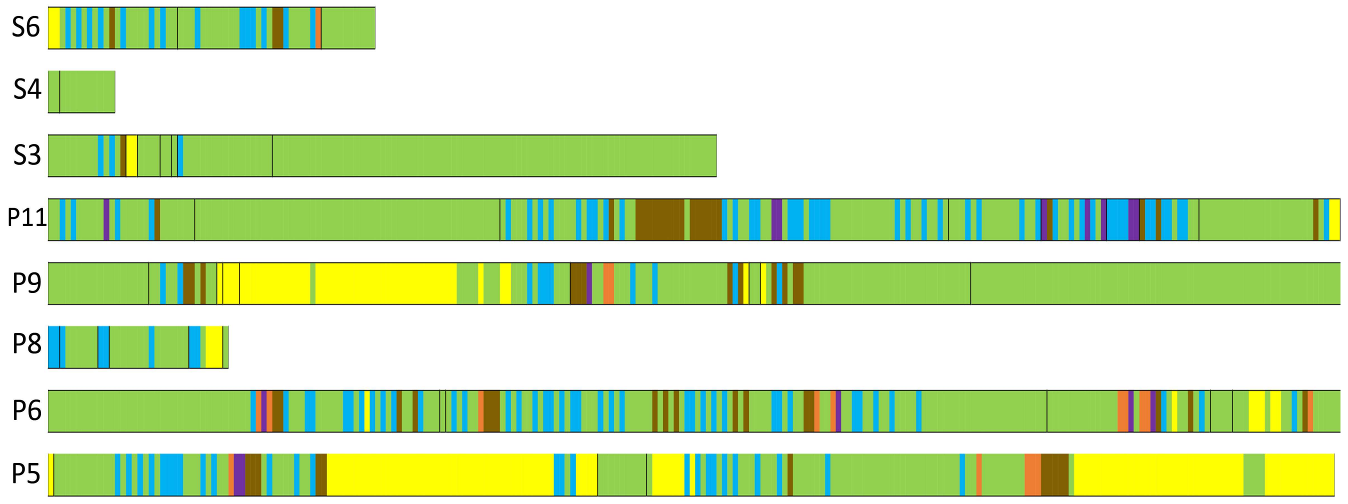


Figure 2: Colored sequence logs of 8 participants investigating method `BrowserLauncher.locateBrowser`. Each row represents a method investigation of a participant with the time axis going from left to right and an eye gaze on a line represented by a colored line within the row.

found an average of 104.2 (112.1) line switches of the 178 to be within a variable slice, supporting our hypothesis that developers are in fact following data flows when investigating a method. The long green and yellow blocks within Figure 2 further illustrate the frequency of switching within a variable slice rather than switching between different variable slices.

4.3 Navigation Between Methods

Overall, subjects switched on average 73.45 (SD = 78.48) times between methods when working on a change task. Thereby, they revisited a method on average 5.44 times.

O5—Developers frequently switch to methods in close proximity and rarely follow call relationships. To investigate the characteristics of method switches we examined whether they were motivated by call relationships or due to the close proximity of methods. We assessed for each method switch within a class and for each method switch to a different class whether the switch was motivated by following the call graph of the method. In addition, we assessed for each method switch within the same class whether the sequentially next method looked at is directly above or directly below the current method. We conducted this analysis for both contexts: the gaze context and the interaction context.

To understand if a method switch was motivated by following the call graph we memorized the method invocations within a given method and assessed if the next method in the method sequence was one of the memorized invoked methods. While we had to consider all method invocations within a given method when analyzing the interaction context, we could precisely assess at which method invocation the developer actually looked at when analyzing the gaze context. If a next method in the sequence was equal to one of the memorized invoked methods, we concluded that it is likely that the developer followed the call relationship (switch potentially motivated by call graph), although, the next method

Table 3: **Methods used in the analysis to determine the amount of within variable-slice line switches.**

method name	# investigation instances (pro,stu)	length in lines	# identified slices	avg lines per slice	avg line switches	avg line switches within slice
`BibtexParser.parseFieldContent`	12 (6,6)	92	10	4	232.6	125.7
`Util.openExternalViewer`	11 (7,4)	132	10	8.1	158.3	77.6
`BibtexParser.parseTextToken`	9 (4,5)	29	3	4	95.0	66.6
`BrowserLauncher.locateBrowser`	8 (5,3)	108	6	8.8	216.3	150.9

could have also been within spatial proximity and the call relationship not of importance for the navigation. If the next method was not contained within the memorized method invocations we concluded that the developer's navigation was motivated by other means than the call relationships. To understand if a method which was looked at next is directly above or directly below a current method, we compared the line numbers in the source file.

gaze context
We found that merely 4.05% ($SD = 6.68\%$) of all method switches were potentially motivated by following the call graph. On average, the subjects switched methods potentially motivated by the call graph more when they were investigating change task 4 ($M = 6.57\%, SD = 9.36\%$) than when they were investigating change task 2 ($M = 1.87\%, SD = 2.94\%$) and change task 3 ($M = 3.18\%, SD = 4.34\%$). A paired-samples t-test showed that developers switched methods potentially motivated by the call graph significantly more often within a class ($M = 4.44\%, SD = 7.12\%$) than between different classes ($M = 0.70\%, SD = 4.50\%$), $t(54) = 3.17, p = .003$.

At the same time, a larger amount of all method switches ended in methods which were right above or below a method ($M = 36.95\%, SD = 25.57\%$). These results suggest that the call graph of a project is not the main drive for navigation between methods, but the location of a method captures an important aspect for navigation between methods.

interaction context
We found that 22.61% ($SD = 29.09\%$) of all method switches were potentially motivated by following the call graph. Different to the results of the gaze context analysis, participants switched between methods potentially motivated by the call graph substantially more when they were investigating change task 3 ($M = 38.23\%, SD = 31.56\%$) than when they were investigating change task 2 ($M = 8.05\%, SD = 13.89\%$) and change task 4 ($M = 23.19\%, SD = 31.42\%$). On average, subjects followed considerably more call relations when they were navigating within the class ($M = 24.15\%, SD = 34.71\%$) than when they were navigating to a method implemented in another class ($M = 6.44\%, SD = 20.74\%$).

We further found that on average 69.93% ($SD = 39.01\%$) of the method switches within a class were aimed towards methods which are directly above or below a method.

Overall, these results also show that the more coarse grained interaction context indicates that developers follow structural call graphs fairly frequently (22.6%) while the more fine grained gaze context depicts a different image with only 4.1% of switches being motivated by structural call relations.

Our results on switches to methods in close proximity further support the findings of a recent head-to-head study that compared different models of a programmer's navigation [29] and that suggested to use models to approximate a developer's navigation based on the spatial proximity of methods within the source code.

O6—Developers switch significantly more to methods within the same class. A paired-samples t-test shows that developers switched significantly more between methods within the same class ($M = 65.22, SD = 73.20$) than they switched from a method to a method implemented in another class ($M = 8.24, SD = 11.95$), $t(54) = 6.07, p < .001$. While, over all three tasks, participants rarely switched to methods of different classes, the participants' method switching within the same class differs between tasks. A Wilcoxon matched pairs signed rank test indicates that participants switched significantly more between methods within classes for task 2 ($M = 103.50, SD = 106.23$) than for task 4 ($M = 36.31, SD = 39.08$), $z = -2.66, p = .008$. While it is not surprising that different tasks result in different navigation behavior of participants, this also suggests that it is important to take into account the task for support tools, such as code navigation recommendations.

4.4 Differences Based on Experience

Previous empirical studies on software developers found differences in the patterns that experienced and novice developers exhibit (e.g., [11]). To investigate such differences, we analyzed our data for differences in navigation between our professional developers and our students. In particular, we tested each statistic that contributed to the above observations and examined whether there were any statistically significant differences in gaze, respectively interaction contexts. To compare the professional developers and the students we used a Mann-Whitney test, as there are different participants in each group and the data does not meet parametric assumptions. Overall, we did not find any statistically significant difference between the two groups of participants in the amounts of unique elements on different granularity levels within the gaze context ($U = 341.0, p = .539$ on class level, $U = 363.5, p = .820$ on method level) nor the interaction context ($U = 368.0, p = .878$ on class level, $U = 286.5, p = .125$ on method level). Furthermore, there was no significant difference in the amounts of switches conducted between different elements within a class ($U = 314.5, p = .292$ for the gaze contexts and $U = 297.5, p = .174$ for the interaction contexts) nor outside of a class ($U = 337.0, p = .495$ for the gaze contexts and $U = 266.5, p = .058$ for the interaction contexts). Finally, we also could not find any significant difference in the amount of call relationships followed ($U = 325.5, p = .362$ for the gaze contexts and $U = 268.0, p = .055$ for the interaction contexts) nor if any of these two groups switched more often to methods with a high spatial proximity ($U = 367.5, p = .873$

for the gaze contexts and $U = 332.0, p = .445$ for the interaction contexts). So even though our exemplary figure (Figure 1) that depicts a sequence log for a professional and a student developer might suggest a difference in navigation behavior, our analysis did not produce any such evidence. Further analysis is needed to examine this aspect in more detail.

4.5 Threats to Validity

One threat to validity is the short time period each participant had for working on a change task. Unfortunately, we were limited by the time availability of the professional developers and therefore had to restrict the main part of the study to one hour. While the data might thus not capture full task investigations, it provides insights on investigations for multiple change tasks and thus the potential of being more generalisable.

Another threat to validity is the choice of JabRef as the subject system. JabRef is written in a single programming language and its code complexity and quality might influence the study. For instance, code with low quality and/or high complexity might result in developers spending more time to read and understand it, and thus longer eye gaze times for certain parts of the code. We tried to mitigate this risk by choosing a generally available system that is an actively used and maintained open source application and that was also used in other studies. Further studies, however, are needed to examine the effect of factors, such as code quality, to generalize the results.

In our study, JabRef had to be run through the command prompt using ANT and not directly in Eclipse. This meant that participants were not able to use breakpoints and the debugger within Eclipse and might have influenced the results. We intend to conduct further study to investigate if our findings generalize to other settings, e.g., ones in which the project can be run from within Eclipse.

iTrace collects eye gazes only within Eclipse editors. This means that we do not record eye gaze when the developer is using the command prompt or running JabRef. However, since we were interested in the navigation between the code elements within the IDE, this does not cause any problems for our analysis.

If the user opens the "Find in File" or "Search Window" within Eclipse, or a tooltip pops up when hovering over an element in the code, the eye gaze is not recorded as this overlaps a new window on top of the underlying code editor window and iTrace did not support gazes on search windows at the time of the study. To minimize the time in which eye gazes could not be recorded, we made sure to let participants know that once they were done with the find feature within Eclipse to close these windows so gaze recording can continue.

Finally, most professional developers were mainly Visual Studio users for their work, we conducted our study in Eclipse. However, all professional developers stated that they did not have problems using Eclipse during the study.

5. DISCUSSION

Tracing developers' eyes during their work on change tasks offers a variety of new insights and opportunities to support developers in their work. Especially, the study's focus on change tasks, the richness of the data, and the finer granularity of the data provide potential for new and improved tool support, such as code summarization approaches or code and artifact recommendations. In the following, we will discuss some of these opportunities.

Richness of Eye-Tracking Data and Gaze Relevance.
Our findings show that the eye-tracking data captures substantially more (***O1***) and different aspects (***O2***) of a developer's interaction with the source code. Therefore, eye-tracking data can be used complimentary to user interaction task context to further enhance existing approaches, such as task-focused UIs [21], or models for defect prediction [24]. In particular, since eye-tracking data also captures gaze times—how long a developer spends looking at a code element—more accurate models of a code element's relevance could be developed as well as models of how difficult a code element is to comprehend which might inform the necessity of refactoring it.

To examine the potential of the gaze time, we performed a small preliminary experiment to compare a gaze-based relevance model with a model based on user interaction. We focused on professional developers and were able to collect and analyze user ratings from 9 professional developers within the group of participants, also since not everyone was willing to spend additional time to participate in this part. Each developer was asked to rate the relevance of the top 5 elements ranked by gaze time as well as the top 5 ranked by degree-of-interest (DOI) from Mylyn's user interaction context [21] on a five-point Likert scale. Overall, participants rated 76% of the top 5 gaze elements relevant or very relevant and only 65% of the top 5 DOI elements as relevant or very relevant. While these results are preliminary and further studies are needed, the 17% improvement illustrates the potential of the data richness in form of the gaze time.

Finer Granularity of Data and Task Focus.
Most current tools and research approaches to support development work focus on method or class level granularity. Most prominently, editors of common IDEs, such as Visual Studio or Eclipse, display whole classes, but even the recently suggested new bubble metaphor for IDEs displays full methods [9]. Similarly, approaches to recommend relevant code elements for a task, such as Mylyn [21, 2] or wear-based filtering [13], operate on the class and method level. While the method and class level are important, our results show that developers only focus on small fractions (on average 32%) of methods that are important for the change task at hand (***O3***). These findings suggest that by identifying, highlighting and possibly filtering the parts within methods that are relevant for the task, we might be able to save developers time and effort to switch between relevant parts of code and avoid getting distracted by other irrelevant code. Since developers focus a lot on data flow within a method (***O4***) that is related to the task, we hypothesize that a task-focused program slicing approach might provide a lot of benefit to developers working on change tasks. Such an approach could take advantage of existing slicing techniques, such as static or dynamic slicing [50, 23], and identify the relevance of a slice based on its relation to the task by, for instance, using textual similarity between the slice and the task description or previously looked at code elements.

By using eye-tracking to capture a more fine-grained task context while a developer is working, we are also able to better determine what a developer is currently interested in and complement existing approaches to recommend relevant artifacts to the developer, such as Hipikat [46] or Prompter [30].

Finally, the insights from our study can also be used to inform summarization techniques to help developers comprehend the relevant parts of the code faster. Existing techniques to summarize code have mainly focused on summarizing whole methods [17, 18] rather than only summarizing the parts relevant for a given task. Similarly, the approach by Rodeghero et al. [34] focused on using eye-tracking to summarize whole methods. Our findings show that developers usually do not read or try to comprehend whole methods and rather focus on small method fractions and data flow slices for a change task. This suggests that a more task-focused summarization that first identifies relevant code within a method according to previous eye-tracking data or other slicing techniques and then summarizes these parts of the method, might help to provide more relevant summaries and aid in speeding up code comprehension.

Accuracy of Method Switches.

The eye-tracking data captured in our study shows that a lot of the switches between methods are between methods in close proximity, as well as within a class ***O5, O6***. These findings suggest that there is a common assumption among developers that nearby code is closely related. While this is not a new finding, the additional data captured through eye-tracking that is not captured by user interaction monitoring provides further evidence for this switch behavior. This finding also suggests that a fisheye view that zooms in on the current method and provides much detail on methods in close proximity but less on methods further out might support faster code comprehension for developers.

A common assumption of navigation recommendation approaches is that structural relations between elements are important in a developers' navigation [33]. While empirical studies that examined developers' navigation behavior based on user interactions have shown that developers actually follow such structural relations frequently, in particular call relations (e.g., [16]), the eye-tracking data of our study shows that developers perform many more switches that do not follow these relations and that are not captured by explicit user interaction. These findings point to the potential of eye-tracking data for improving method recommendations as well as for identifying the best times for suggesting structural navigation recommendations. However, further studies are needed to examine this possibility.

An Eye-Tracker per Developer.

As discussed, using eye-trackers in practice and installing them for each developer not just for study purposes bares a lot of potential to improve tool support, such as better task-focus, recommendations or summarization. With the advances and the price decrease in eye-tracking technology, installing eye-trackers for each developer might soon be reasonable and feasible. At the same time, there are still several challenges and questions to address to be smooth and of value to developers, in particular with respect to eye calibration, granularity level and privacy. Several eye-trackers, especially cheaper ones, currently still need a recalibration every time a developer changes position with respect to the monitor, which is too expensive for practical use. For tool integration, one has to decide on the level of granularity that is best for tracking eye gazes. While more fine-grained data might provide more potential, eye-tracking on a finer granularity level is also more susceptible to noise in the data. Finally, as with any additional data that is being tracked about an individual's behavior, finer granular data also raises more privacy concerns that should be considered before such an approach is being deployed. For instance, the pupil diameter or the pattern of eye traces might also be used to monitor the cognitive load of the developer, which could also be used in harmful ways.

6. CONCLUSION

To investigate developers' detailed behavior while performing a change task, we conducted a study with 22 developers working on three change tasks of the JabRef open source system. This is the first study that collects simultaneously both eye-tracking and interaction data while developers worked on realistic change tasks. Our analysis of the collected data shows that gaze data contains substantially more data, 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.

These detailed findings provide insights and opportunities for future developer support. For instance, the findings 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 in 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.

The approach that we developed for this study automatically links eye gazes to source code entities in the IDE and overcomes limitations of previous studies by supporting developers in their usual scrolling and switching behavior within the IDE. This approach opens up new opportunities for conducting more realistic studies and gathering rich data while reducing the cost for these studies. At the same time, the approach opens up opportunities for directly supporting developers in their work, for instance, through a new measure of relevance using gaze data. However, possible performance and especially privacy concerns have to be examined beforehand.

7. ACKNOWLEDGMENTS

The authors would like to thank the participants in the study. The authors would also like to thank Meghan Allen for her helpful feedback. This work was funded in part by an SNF grant and an ABB grant.

8. REFERENCES

[1] jabref.sourceforge.net/. Accessed: 2015-03-15.

[2] `eclipse.org/mylyn/`. Accessed: 2015-03-15.

[3] `www.csis.ysu.edu/~bsharif/itraceMylyn`. Accessed: 2015-03-15.

[4] `www.tobii.com/`. Accessed: 2015-03-15.

[5] E. M. Altmann. Near-term memory in programming: a simulation-based analysis. *International Journal of Human Computer Studies*, 54(2):189–210, 2001.

[6] R. Bednarik. Expertise-dependent visual attention strategies develop over time during debugging with multiple code representations. *International Journal of Human-Computer Studies*, 70(2):143–155, Feb. 2012.

[7] R. Bednarik and M. Tukiainen. An eye-tracking methodology for characterizing program comprehension processes. In *Proceedings of the Symposium on Eye Tracking Research & Applications*, pages 125–132. ACM, 2006.

[8] D. Binkley, M. Davis, D. Lawrie, J. Maletic, C. Morrell, and B. Sharif. The impact of identifier style on effort and comprehension. *Empirical Software Engineering Journal (invited submission)*, 18(2):219–276, 2013.

[9] A. Bragdon, R. Zeleznik, S. P. Reiss, S. Karumuri, W. Cheung, J. Kaplan, C. Coleman, F. Adeputra, and J. J. LaViola Jr. Code bubbles: a working set-based interface for code understanding and maintenance. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 2503–2512. ACM, 2010.

[10] R. Brooks. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, 18:543–554, 1983.

[11] M. E. Crosby and J. Stelovsky. How do we read algorithms? a case study. *Computer*, 23(1):24–35, Jan. 1990.

[12] B. de Smet, L. Lempereur, Z. Sharafi, Y.-G. Guéhéneuc, G. Antoniol, and N. Habra. Taupe: Visualizing and analysing eye-tracking data. *Science of Computer Programming Journal (SCP)*, 87, 2011.

[13] R. DeLine, A. Khella, M. Czerwinski, and G. Robertson. Towards understanding programs through wear-based filtering. In *Proceedings of the 2005 ACM symposium on Software visualization*, pages 183–192. ACM, 2005.

[14] A. Field. *Discovering Statistics Using SPSS*. SAGE Publications, 2005.

[15] T. Fritz, A. Begel, S. C. Müller, S. Yigit-Elliott, and M. Züger. Using psycho-physiological measures to assess task difficulty in software development. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 402–413, New York, NY, USA, 2014. ACM.

[16] T. Fritz, D. C. Sheperd, K. Kevic, W. Snipes, and C. Braeunlich. Developers' code context models for change tasks. In *22nd ACM SIGSOFT International Symposium on the Foundations of Software Engineering/ FSE 2014*, Hong Kong, China, NOV 2014. ACM.

[17] S. Haiduc, J. Aponte, and A. Marcus. Supporting program comprehension with source code summarization. In *Proceedings of the 32nd International Conference on Software Engineering*, May 2010.

[18] S. Haiduc, J. Aponte, L. Moreno, and A. Marcus. On the use of automated text summarization techniques for summarizing source code. In *Proceedings of the 17th Working Conference on Reverse Engineering (WCRE)*, pages 35–44. IEEE, 2010.

[19] M. Just and P. Carpenter. A theory of reading: From eye fixations to comprehension. *Psychological Review*, 87:329–354, 1980.

[20] M. Kersten and G. C. Murphy. Mylar: A degree-of-interest model for ides. In *Proceedings of the 4th International Conference on Aspect-oriented Software Development*, AOSD '05, pages 159–168, New York, NY, USA, 2005. ACM.

[21] M. Kersten and G. C. Murphy. Using task context to improve programmer productivity. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 1–11, 2006.

[22] A. Ko, B. Myers, M. Coblenz, and H. Aung. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Transactions on Software Engineering*, 32(12):971–987, Dec 2006.

[23] B. Korel and J. Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155–163, 1988.

[24] T. Lee, J. Nam, D. Han, S. Kim, and H. P. In. Micro interaction metrics for defect prediction. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, pages 311–321, 2011.

[25] S. C. Müller and T. Fritz. Stakeholders' information needs for artifacts and their dependencies in a real world context. In *Proceedings of the 29th IEEE International Conference on Software Maintenance (ICSM)*, pages 290–299. IEEE, 2013.

[26] S. C. Müller and T. Fritz. Stuck and frustrated or in flow and happy: Sensing developers? emotions and progress. In *Proceedings of the 37th International Conference on Software Engineering (ICSE 2015)*, 2015.

[27] G. C. Murphy, M. Kersten, and L. Findlater. How are java software developers using the eclipse ide? *Software, IEEE*, 23(4):76–83, 2006.

[28] C. Parnin and C. Gorg. Building usage contexts during program comprehension. In *Proceedings of the 14th IEEE International Conference on Program Comprehension*, pages 13–22, 2006.

[29] D. Piorkowski, S. Fleming, C. Scaffidi, L. John, C. Bogart, B. John, M. Burnett, and R. Bellamy. Modeling programmer navigation: A head-to-head empirical evaluation of predictive models. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 109–116, Sept 2011.

[30] L. Ponzanelli, G. Bavota, M. Di Penta, R. Oliveto, and M. Lanza. Mining stackoverflow to turn the ide into a self-confident programming prompter. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 102–111. ACM, 2014.

[31] K. Rayner. Eye movements in reading and information processing: 20 years of research. *Psychological bulletin*, 124(3):372, 1998.

[32] R. S. Rist. Plans in programming: Definition, demonstration, and development. In *Papers Presented at the First Workshop on Empirical Studies of Programmers on Empirical Studies of Programmers*, pages 28–47, Norwood, NJ, USA, 1986. Ablex Publishing Corp.

[33] M. P. Robillard. Automatic generation of suggestions for program investigation. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 11–20. ACM, 2005.

[34] 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. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 390–401, New York, NY, USA, 2014. ACM.

[35] T. Shaffer, J. Wise, B. 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 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2015)*, page in press. ACM, 2015.

[36] B. Sharif, M. Falcone, and J. Maletic. An eye-tracking study on the role of scan time in finding source code defects. In *Proceedings of the Symposium on Eye Tracking Research & Applications*, pages 381–384, Santa Barbara, CA, 2012. ACM.

[37] B. Sharif, G. Jetty, J. Aponte, and E. Parra. An empirical study assessing the effect of seeit 3d on comprehension. In *1st IEEE International Working Conference on Software Visualization (VISSOFT 2013)*, pages 1–10.

[38] B. Sharif and H. Kagdi. On the use of eye tracking in software traceability. In *6th ACM International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE'11)*, pages 67–70.

[39] B. Sharif and J. Maletic. An eye tracking study on the effects of layout in understanding the role of design patterns. In *26th IEEE International Conference on Software Maintenance (ICSM'10)*, pages 1–10.

[40] B. Sharif and J. I. Maletic. An eye tracking study on camelcase and under_score identifier styles. In *18th IEEE International Conference on Program Comprehension (ICPC'10)*, pages 196–205.

[41] B. Shneiderman and R. Mayer. Syntactic semantic interactions in programmer behavior: A model and experimental results. *International Journal of Computer and Information Sciences*, 18:219–238, 1979.

[42] J. Siegmund, C. Kästner, S. Apel, C. Parnin, A. Bethmann, T. Leich, G. Saake, and A. Brechmann. Understanding understanding source code with functional magnetic resonance imaging. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 378–389, New York, NY, USA, 2014. ACM.

[43] E. Soloway and K. Ehrlich. Empirical studies of programming knowledge. *IEEE Trans. Software Eng.*, 10(5):595–609, 1984.

[44] R. Turner, M. Falcone, B. Sharif, and A. Lazar. An eye-tracking study assessing the comprehension of C++ and Python source code. In *Proceedings of the Symposium on Eye Tracking Research & Applications*, pages 231–234, Safety Harbor, Florida, 2014. ACM.

[45] H. Uwano, M. Nakamura, A. Monden, and K.-i. Matsumoto. Analyzing individual performance of source code review using reviewers' eye movement. In *Proceedings of the Symposium on Eye Tracking Research & Applications*, pages 133–140, San Diego, California, 2006. ACM.

[46] D. Čubranić and G. C. Murphy. Hipikat: Recommending pertinent software development artifacts. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 408–418, 2003.

[47] V. Vinay Augustine, P. Francis, X. Qu, D. Shepherd, W. Snipes, C. Bräunlich, and T. Fritz. A field study on fostering structural navigation with prodet. In *Proceedings of the 37th International Conference on Software Engineering (ICSE SEIP 2015)*, 2015.

[48] B. Walters, M. Falcone, A. Shibble, and B. Sharif. Towards an eye-tracking enabled ide for software traceability tasks. In *7th International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE)*, pages 51–54.

[49] B. Walters, T. Shaffer, B. Sharif, and H. Kagdi. Capturing software traceability links from developers' eye gazes. In *Proceedings of the 22nd International Conference on Program Comprehension*, ICPC 2014, pages 201–204, New York, NY, USA, 2014. ACM.

[50] M. Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, pages 439–449. IEEE Press, 1981.

[51] S. Yusuf, H. Kagdi, and J. Maletic. Assessing the comprehension of uml class diagrams via eye tracking. In *Proceedings of the 15th IEEE International Conference on Program Comprehension*, pages 113–122, June 2007.

[52] M. Züger and T. Fritz. Interruptibility of software developers and its prediction using psycho-physiological sensors. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, pages 2981–2990. ACM, 2015.