# Eye gaze and interaction contexts for change tasks – Observations and potential

Kevic, Katja ; Walters, B M ; Shaffer, T R ; Sharif, Bonita ; Shepherd, D C ; Fritz, Thomas

DOI: https://doi.org/10.1016/j.jss.2016.03.030

# Eye Gaze and Interaction Contexts for Change Tasks — Observations and Potential

K. Kevic[a,*], B. M. Walters[b,*], T. R. Shaffer[b,*], B. Sharif[b,*], D. C. Shepherd[c,*], T. Fritz[a,*]

[a] *University of Zurich, Department of Informatics, Switzerland*
[b] *Youngstown State University, Department of CS and IS, USA*
[c] *ABB Corporate Research, Industrial Software Systems, USA*

**Abstract**

The more we know about software developers' detailed navigation behavior for change tasks, the better we are able to provide effective tool support. Currently, most empirical studies on developers performing change tasks are, however, limited to very small code snippets or limited by the granularity and detail of the data collected on developer's navigation behavior. In our research, we extend this work by combining user interaction monitoring to gather interaction context-the code elements a developer selects and edits -with eye-tracking to gather more detailed and fine-granular gaze context-code elements a developer looked at. In a study with 12 professional and 10 student developers we gathered interaction and gaze contexts from participants working on three change tasks of an open source system. Based on an analysis of the data we found, amongst other results, that gaze context captures different aspects than interaction context and that developers only read small portions of code elements. We further explore the potential of the more detailed and fine-granular data by examining the use of the captured change task context to predict perceived task difficulty and to provide better and more fine-grained navigation recommendations. We discuss our findings and their implications for better tool support.

*Keywords:* eye-tracking, interactions, change task, empirical study, context

## 1. Introduction

Developers spend a significant amount of their time searching, navigating and reading source code to find and modify the elements relevant to a change task at hand [1]. During this code exploration a developer gradually builds up an implicit change task context that consists of all the explored source code elements and relations. While these task contexts often stay implicit [2], there has been a shift towards automatically capturing task context based on a developer's interactions with the code elements in an integrated development environment (IDE) [3, 4, 5]. The more we know about task contexts and the code a developer explores for a change task, the better we are able to

*Corresponding author
*Email address:* kevic@ifi.uzh.ch (K. Kevic )

develop effective tool support for a variety of programming activities, such as proactive navigation recommendation (e.g., [6, 7]) or task resumption support (e.g., [3]).

Recent advances in technology, such as eye-tracking devices, afford new opportunities to collect more detailed information on a developer and her work. Studies using eye-tracking sensors and other biometric sensors have generated new insights on, for instance, brain activation patterns [8], developers' perceptions of difficulty [9], and the ease of comprehending different representations of code [10, 11]. Yet, these studies predominantly focus on small code snippets of the size of source code methods and do not capture contexts of real-world change tasks. Additionally they fail to leverage established methods of collecting interaction data, such as instrumenting the IDE and automatically mapping x,y coordinates back to source code elements, and thus often produce data that is difficult to analyze.

In our research, we extend previous work by taking advantage of eye-tracking technology and examining developers' fine-grained code exploration behavior for realistic change tasks. By using an eye-tracker and capturing a developer's eye gazes on line and statement level, we are able to gather much deeper insight into a developer's code exploration behavior than existing techniques that operate on file or method granularity. This type of information is particularly valuable since developers spend a considerable amount of their time reading individual source code methods [1]. In addition to the analysis of a developer's fine-grained navigation behavior, using an eye-tracking approach also enables us to answer questions on the difference in the data captured through eye-tracking and interaction logging and how such fine-grained data can be used to better support developers.

To investigate the fine-grained navigation behavior for realistic change tasks, we conducted a study with 12 professional and 10 student developers. In this study we used interaction monitoring in combination with eye-tracking and simultaneously captured all code elements a developer selected or edited—*interaction context*—and all code elements a developer looked at—*gaze context*—while they were working on three change tasks from an open source software system. While the interaction context includes source code elements on method-level or higher granularity, the gaze context was captured on statement and line-level.

Based on the analysis of the detailed code exploration traces of developers, we made observations on developers' task contexts from different perspectives. Our analysis on the within method navigation behavior revealed that developers only read few lines of a method and that these lines are generally connected through data flow. Our analysis on the navigation behavior between methods revealed that developers frequently switch to methods in close proximity or within the same class and that they only focus on few methods thoroughly. We further found that developers either use a skimming or a seeking strategy to explore the source code for a change task and that developers who solved a change task successfully read more methods thoroughly. In our analysis we also investigated the differences between these two kinds of contexts and found that the gaze context not only captures more and more fine-grained source code elements, but also different aspects about the developers' navigation.

We further explore the potential application of this new fine-granular data source in two scenarios: line-level navigation recommendation and the prediction of task difficulty. In an empirical analysis of the data captured in our study, we found that out of four models based on data flow, proximity, recency and frequency, the proximity-model works

best and allows to predict the next line visited by a developer in 73.6% of the cases. These results can be used to inform, for example, navigation tool support for summarizing methods or highlighting parts therefore in the context of a change task. For the prediction of task difficulty, we conducted a second empirical analysis that focused on predicting the difficulty of the current change task based on specific characteristics of a developer's navigation behavior, such as the number of line switches or the number of switches between methods. We found that gaze context can be used to more accurately predict task difficulty, and that for both, interaction and gaze context, a developer's navigations to methods right above or underneath a current method can be used to predict task difficulty best.

The contributions of this paper are summarized as follows:

- 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.
- Analysis of different strategies developers use for code exploration with respect to successful and unsuccessful results
- Potential uses of the fine-grained eye tracking data for navigation recommendations and predicting task difficulty.
- A discussion on the value of the data gathered and the opportunities the data and the findings offer for better developer support.

In this paper, we extend our previous work [12] by analyzing and investigating the different strategies developers employ for code exploration during change tasks as well as an analysis of successful versus unsuccessful code exploration (see Section 4.4). In addition, we also go a step further and look at potential uses of the fine-grained data in two scenarios, one on line-level navigation recommendation and one on the prediction of task difficulty (see Section 5).

The paper is organized as follows. First, we provide an overview of related work (Section 2), before we describe the exploratory study and how we collected the data (Section 3). In Section 4, we present the results of our study in a form of observations, and in Section 5 we explore the potential use of the fine-grained data in two different scenarios. Section 6 lists threats to validity and Section 7 discusses our observations and future ideas before we conclude in Section 8.

## 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., [13, 14, 15]). 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 [16]. Perhaps one of the most well-known user studies from this category is the study by Ko et al. [17]. 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. [18] 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 [5, 19, 3]. 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 [20], how they navigate through code [21, 22, 23], 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 [25, 26], 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 [8], or a combination of eye-tracking, EDA, and EEG sensors to measure aspects such as task difficulty, developers' emotions and progress, or interruptibility [9, 27, 28].

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 [29]. 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 [30], UML diagrams [31, 32], design pattern layout [33], programming languages [34], and identifier

styles [10, 35]. 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 [36]. Other researchers examined different strategies novice and expert developers employ in program comprehension and debugging [37, 11], as well as where developers spend most time when reading a method to devise a better method summarization technique [38]. Finally, researchers have also used eye-tracking to evaluate its potential for detecting software traceability links [39, 40, 41]. 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 [41], 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 [42].

### *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 acquired 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.

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`, `BibtexParser`, `parseFieldContent` |
| 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` |

### *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 [43] 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. We chose an earlier release of JabRef to ensure that there was a sufficient number of resolved change tasks available for us to choose study tasks from and that had change sets associated with them.

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 only criteria for selecting tasks was that they had to address a change in the source code of the system and, for instance, not in the configuration files. We randomly selected tasks from a list of closed bug reports that fulfilled this criteria and that also varied in difficulty as determined by the scope of the solution implemented in the repository.

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 [44]. 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 [45] 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 to 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 [5, 19] that monitors a user's interactions with code elements in the IDE, in particular *selects* and *edits* of classes, methods and fields. For the eye-tracking data, we used our new version of the Eclipse plugin iTrace [44]. In our analyses, we only considered edit and selection interaction events and eye gazes on java files and did not analyze captured data on other file types, such as html or xml files.

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 change task T2, 16 subjects investigating change task T3, and 21 subjects investigating change task T4. These 55 change task investigations comprise totally 119,618 single eye gazes and 3524 single interactions with methods, classes and fields. 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 gaze contexts and interaction contexts of the 22 participants we were able to make detailed observations on how developers navigate within source code and build up their contexts. Table 2 presents some descriptive statistics over the gathered interaction and gaze context averaged over all participants and change tasks, and Table 3 presents data on the gaze and interaction context per participant. The data presented in these tables already highlights the often big differences between the elements and events captured in the different kinds of context—interaction and gaze—as well as the very fractional reading of methods for developers' change task investigations.

In the following, we will further discuss this data in more detail and in the context of the concrete observations we made. We structure our *observations* along four research foci: the difference between gaze and user interaction data, developers' navigation within methods, developers' navigation between methods and developer-specific navigation characteristics and start each paragraph with the observation we made. Since almost all participants used the maximum time of twenty minutes for the change task investigations, we did not perform any analysis of the data with respect to task completion time.

Table 2: Descriptive statistics of the analyzed interaction and gaze contexts gathered for the change tasks, averaged over all participants and tasks (± denotes the standard deviation).

| Variable | Description | Interaction | Gaze |
|---|---|---|---|
| $Num_{UMe}$ | Number of unique methods which were selected, edited, or looked at | 6.0 (±4.5) | 12.5 (±11.8) |
| $Num_{MeSw}$ | Number of times a developer selected or looked at a different method | 5.8 (±5.2) | 73.5 (±78.5) |
| $Rat_{SwInVsAll}$ | Ratio between the number of switches to a method within the same class and the number of method switches | 54.4% (±33.8) | 88.3% (±14.1) |
| $Rat_{SwOutVsAll}$ | Ratio between the number of switches to a method in a different class and the total number of method switches | 45.6% (±33.8) | 11.7% (±14.1) |
| $Rat_{ProximitySw}$ | Ratio between the number of switches to a method right above or underneath and the total number of method switches within a class | 69.9% (±39.0) | 37.0% (±25.6) |
| $Dwell_{Me}$ | Time spend reading a method | | 0.3min (±0.5) |
| $Num_{LineSw}$ | Number of line switches within a method | | 40.0 (±101.0) |
| $Perc_{LinesLooked}$ | Percentage of lines which were looked at within a method | | 32.2 (±25.0) |
| $Num_{Me>HfLi}$ | Number of methods for which a developer looked at more than half of the lines | | 7.2 (±10.1) |
| $Num_{UMe>HfLi}$ | Number of unique methods for which a developer looked at more than half of the lines | | 2.7 (±2.9) |
| $Tsp_{ToMe>HfLi}$ | Time span from start of change task investigation to first method for which a developer looked at more than half of the lines | | 2.9min (±2.9) |
| $Num_{Me>AvgDw}$ | Number of methods a developer spent more than $Dwell_{Me}$ (average time reading a method) | | 13.6 (±16.0) |
| $Num_{UMe>AvgDw}$ | Number of unique methods a developer spent more than $Dwell_{Me}$ | | 5.0 (±4.8) |
| $Tsp_{ToMe>AvgDw}$ | Time span from start of change task investigation to first method for which a developer looked at for more than $Dwell_{Me}$ | | 2.1min (±2.1) |

Table 3: Summary of professional (pro) and student (stu) developers' average (avg) of methods and method switches captured in the gaze and interaction context over all three change tasks, as well as the percentage of lines read within methods.

| ID | avg # of method switches | | average # of unique methods | | avg % of lines |
|---|---|---|---|---|---|
| | gaze context | inter. context | gaze context | inter. context | looked at |
| 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%** |

### 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 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 (Mean ($M$) = 4.78, Standard Deviation ($SD$) = 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 3 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 [22]. 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 [21, 22]. 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.

We compared for each change task investigation the top 5 methods resulting from the frequency model and from the time-based model. We then analyzed for how many

[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 [46].

methods the interaction and the gaze context agree and found that in 65.03% ($SD$ = 32.26%) of the recommended methods this was the case. 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 T2 ($M = 52.31\%, SD = 34.98\%$) than for change task T3 ($M = 71.88\%, SD = 27.62\%$) and for change task T4 ($M = 70.71\%, SD = 31.32\%$). While the description for change task T3 and change task T4 include concrete hints to source code elements which are possibly important for performing the change task, change task T2 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. Within Method Navigation*

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 of a randomly chosen professional developer (middle) and a randomly chosen 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 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

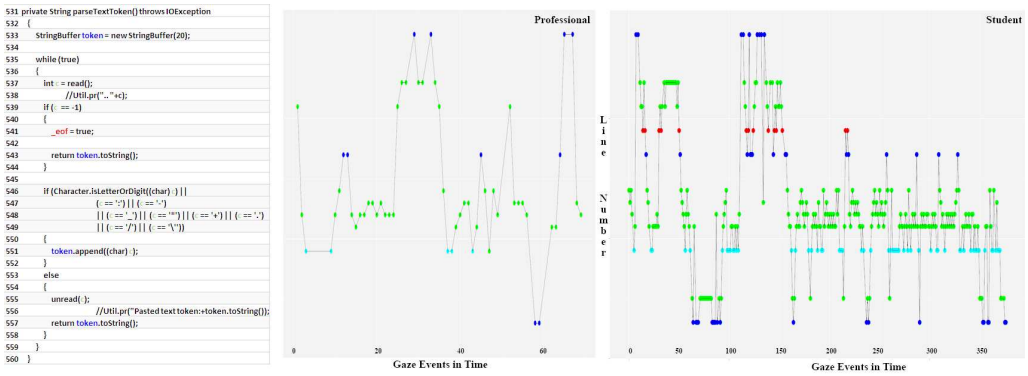


Figure 1: The sequence logs mapped to line numbers and colors, with the colored source code on the left.

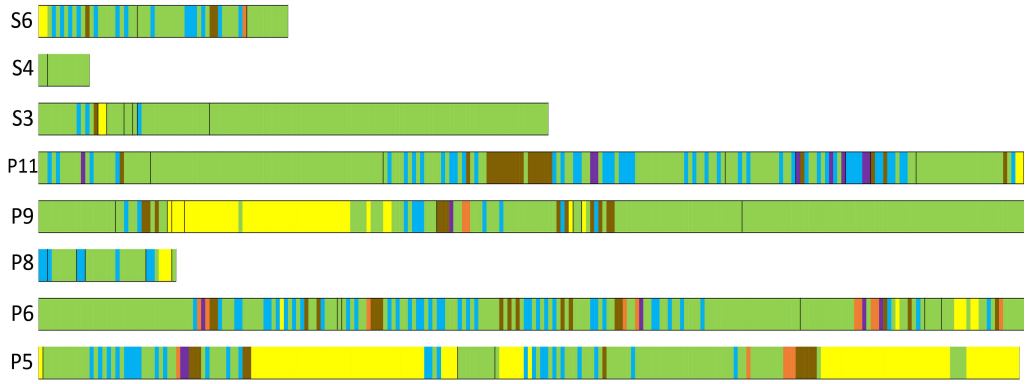


Figure 2: Colored sequence logs of eight participants investigating method `BrowserLauncher.locateBrowser`. Each row represents a method investigation of a participant with the time axis going from left to right. Eye gazes on lines that talk about the same variable are colored the same. For instance, S3 exclusively gazed at lines containing the same variable for more than the second half of the method investigation, and thus more than the second half of the bar for S3 is colored green.

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 observation that developers often trace variables when reading a method. To investigate this observation further, we selected four methods which were investigated by most participants, resulting in 40 unique method investigation instances.

The 40 method investigation instances stem from 18 different participants and two different tasks. 22 of these 40 investigations stem from professional software developers, while the other 18 stem from students.

For each method, we assigned a color to each variable used within the method. Colors were chosen randomly. We then colored the lines in which a variable was either defined or

used in the method. We did not color lines or statements that did not include a variable. In case more than one variable was used in a single line, we manually checked if a color was predominantly used before or after the line was visited and used 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. 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, i.e., an average of 6.2 lines in a single method referred to the same variable.

In a next step, we took participants' eye gaze logs—the sequentially ordered eye gaze events including line numbers—for these four methods, filtered the gaze events that did not map to a variable slice, such as brackets and empty lines, and then colored each log entry with the colour of the variable it referred to. Figure 2 illustrates some of these color-coded eye gaze logs from participants' method investigations with the sequence of events going from left to right.

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 a variable slice, i.e., lines that refer to the same variable and have the same color. Over all method investigation instances we found an average of 104.2 (112.1) line switches of the 178 to be within a variable slice, supporting our observation that developers are in fact following data flows when investigating a method. The long green and yellow blocks within Figure 2 provide further visual evidence on the high frequency of participants switching between lines within a variable slice (same color) rather than switching between different variable slices (different consecutive colors).

### *4.3. Between Method Navigation*

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 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 T4 ($M = 6.57\%, SD = 9.36\%$) than when they were investigating change task T2 ($M = 1.87\%, SD = 2.94\%$) and change task T3 ($M = 3.18\%, SD = 4.34\%$). 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 T3 ($M = 38.23\%, SD = 31.56\%$) than when they were investigating change task T2 ($M = 8.05\%, SD = 13.89\%$) and change task T4 ($M = 23.19\%, SD = 31.42\%$). 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 picture with only 4.1% of the switches being motivated by structural call relations. To understand whether these switches to methods in close proximity were intentional or mainly present an inadvertent glimpse to a neighbouring method, we examined how many lines of the neighbouring method a developer looked at. We found that in 30.20% of the method switches to a method in close proximity were rather an inadvertent glimpse with the developer only looking at a single line of the method, while in 36.45% of the cases the developer read the nearby method more carefully, i.e., she read more than half of the lines of the method. This indicates that a big part of the switches to proximate methods serves a purpose and is

not necessarily caused by inadvertently wandering around.

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 [22] 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.***

Applying a paired-samples t-test on the gaze context 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 change task T2 ($M = 103.50, SD = 106.23$) than for change task T4 ($M = 36.31, SD = 39.08$), $z = -2.66, p = .008$. 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.

***O7—Developers only read few of the explored methods more thoroughly during a change task investigation.***

While developers read parts of several methods for each task, they only read very few of these methods more thoroughly. In only 24.54% ($SD = 19.36$) of the unique methods that developers explored, they spend time to read more than half of the lines of the method see Table 3). Professional developers thereby read on average more methods more thoroughly ($M = 27.17\%$, $SD = 16.18\%$) than student developers ($M = 22.18\%$, $SD = 21.83\%$), although this difference is not statistically significant.

For all change tasks, there is also only little overlap amongst the developers with respect to the more thoroughly explored methods, i.e., different developers explored different methods more thoroughly. For instance, while there was one method that 16 of the 21 participants that worked on task T4 explored more thoroughly, for all other methods that were explored more thoroughly for this task, it was only an average of 3.38 of the 21 participants doing so. For task T2 it was even just an average of 1.48 developers of the 18 working on this task that explored the same method more thoroughly.

To see whether a method's complexity might have an influence on the number of developers reading a method more thoroughly, we looked at McCabe Cyclomatic Complexity. Our analysis showed that, for instance for task T4, the complexity scores of the methods that developers read more thoroughly do not correlate with the numbers of developers who focused on a more thoroughly read method (Pearson's $r = .077, p = .768$). Further investigations on what the reason might be for methods being read more carefully is planned for future work.

### *4.4. Developer-Specific Context Characteristics*

Studies showed that the source code elements captured in task contexts are highly developer-specific [18]. To train an individual navigation recommender tool, a rather

large history of interaction data is needed, which is often unavailable. Hence, we aim to explore whether we can identify different groups of developers that explore source code in a similar way, such that recommendation tools might be adapted to groups rather than individuals. Further, we explore how the context of developers who successfully solved a change task differs from the contexts of developers which have not had enough time to complete the change task. Finally, we also look into how developers with a rich programming experience build up context compared to developers with less programming experience.

***O8—Developers either use a skimming strategy or a seeking strategy to explore source code for a change task.***

To investigate whether our data includes different groups of developers, which explore source code in a similar way, we conducted a cluster analysis on the gathered gaze contexts. We used an agglomerative hierarchical clustering algorithm, as we followed a more exploratory approach and did not want to decide on the number of clusters beforehand. We used the log-likelihood as distance measure. In this analysis we focused on the gathered gaze contexts, as the data collected through interactions is too coarse-grained to detect specific code exploration strategies.

We obtained two clusters from our analysis. These clusters have a silhouette measure of cohesion and separation of 0.5, which denotes a good cluster quality (a silhouette measure below 0.2 denotes poor cluster quality, while a silhouette measure of 0.5 and higher denotes a good cluster quality). The two clusters have different sizes. The first cluster comprises 34.5% of the data points, while the second cluster comprises 65.5% of the data points. The variables which influence the classification are the ratio of switches to a method right above or below, the average percentage of lines which were looked at within methods, and the number of methods on which developers spent more than their average method investigation time.

Based on the values for these variables, we interpret one cluster as "seeking" the source code and the second cluster as "skimming" the source code. Code exploration instances belonging to the "seeking" cluster are characterized by less switches to proximate methods ($M = 0.29$), more lines being read within a method ($M = 0.37$), and by having considerably more focus points than the code exploration instances in the "skimming" cluster, with an average of 16.97 methods. Skimming the source code on the other hand is characterised by more switches between proximate methods ($M = 0.67$), reading less lines within methods ($M = 0.22$), while focusing on average at far less methods than seekers with an average of 7.26 methods. Overall, the analysis suggests that developers seek source code more often (36 of the task investigations were classified into this group) than they skim the source code (19 of the task investigation sessions were classified into this group).

Looking at these clusters, we recognized that the change task itself has a high impact on a developer's code exploration behavior, i.e. whether the developer seeks or skims the source code. We discovered that all developers, except for one, were seeking the source code when investigating change task T3 (see Table 1). Change task T3 is the only change task included in our analysis which has a stacktrace. As the stacktrace offered more code specific information about the change task to the developers, almost all of them applied a seeking-strategy. These results empirically show that the information given in change tasks can influence the way how developers build contexts. This further

confirms the survey results by Bettenburg et al. [47] that showed stacktraces in change tasks are among the most important information fragments for solving a change task.

17 developers investigated both of the remaining change tasks T2 and T4. Considering only the change task investigations for these two change tasks, our clustering method could assign 65% of these 17 developers to one specific category. 6 developers were identified as seekers, while 5 developers were identified as skimmers. The remaining 6 developers applied each time another strategy for these two change tasks.

***O9—Developers who solved a change task successfully read more lines within methods, switched more between these lines, and focused on more methods.***

How does a context leading to a successful change look like? While each developer's context is very individualized, we explore whether contexts which allowed for a successful change have commonalities. Exploring commonalities of successful or faster changes might inform new tool support, which directs developers to adopt more efficient navigation behavior.

Based on the changes made by the developers and the short descriptions of their solutions, we manually assessed whether the study participants successfully solved the given change tasks. Over all 55 change task investigations we determined that 12 change tasks were solved successfully. Each of the three change task types is among the successfully solved ones, although change task T3 was most often (6 times) solved successfully. Since almost each change task investigation referring to change task T3 was classified as a seeking-session (see **O8**), most of the successful change task investigation are classified as seeking-sessions (75%). We ran a Mann-Whitney U test to compare the different variables related to the gathered gaze contexts and the gathered interaction context (see Table 2) for successfully solved change tasks and unsuccessfully solved change tasks. For the gaze context our analysis suggests that developers who solved a change task successfully switched significantly more between lines when reading a method ($U = 119, p = .005$). The amount of lines they looked at on average when reading a method and the number of methods they focused on are also considerably different, although not significant ($U = 165, p = .058$, respectively $U = 163, p = 0.052$). However, since change task T3 had a stacktrace included, we ran the same analysis on the dataset excluding change task T3. The results suggest the same parameters to be decisive. Developers who successfully solved change task T2 and T4 switched on average more between lines of a method ($U = 33, p = .008$), focused on more methods ($U = 57.5, p = .012$) and read more different lines within a method ($U = 49, p = .52$). Since too few change task investigations for change task T2 and T4 were solved successfully (2 for change task T2 and 4 for change task T4), we cannot conclude whether the choice of strategy has an impact on the task outcome. We plan to investigate whether a particular investigation strategy has an impact on the task outcome in future studies.

When analyzing the interaction contexts with respect to successful and unsuccessful changes, we did not find any significant differences.

***O10—There were no significant differences in contexts built by professional developers and student developers in our study.***

Previous empirical studies on software developers found differences in the patterns that experienced and novice developers exhibit (e.g., [29]). 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.

## 5. Approaches

In this section we demonstrate the potential of fine-grained task context on the basis of two approaches.

### *5.1. Fine-Grained Navigation Recommendation*

When developers explore source code, they navigate extensively between methods. To support developers during the time-consuming source code navigation, different approaches have emerged (e.g., [48, 49]). These approaches are based on an underlying model which imitates developers' navigation steps and points the developers directly to interesting places to go to next. A head-to-head study by Piorkowski et al. [22] compared a variety of underlying models, and found that recently visited methods and methods which are in close proximity are most likely to be visited next.

Using the gathered gaze contexts, we transferred these approaches to the much finer-granular line navigation within methods. Specifically, we evaluated four models based on our observations (see Section 4) and previous research to predict the next source code line a developer will visit. Based on observation **O3** which states that developers only look at a few lines within a method and switch often between these lines, we formulated a recency- and frequency-based model. Based on the observation **O4** which states that developers chase data flows within methods we formulated a data flow-based model. Finally, inspired by our observation **O5** which states that developers frequently switch between methods in close proximity, we also included a proximity-based model in this experiment. In summary, we included the following within method navigation models in our experiment:

- *Data flow-based model*: ranks the source code lines within a method higher that include a variable occurring in the current line of focus.
- *Proximity-based model*: ranks the source code lines higher, which are in close proximity to the current line of focus. In case of an uneven number of recommendations, this model favors the lower part of the method, as more methods are read strictly from top to bottom (64.77%).
- *Recency-based model*: ranks the source code lines higher that were looked at more recently.
- *Frequency-based model*: ranks the source code lines higher that were looked at more frequently.

Similarly to the evaluation strategy applied by Piorkowski et al. [22], we calculated each model's average accuracy for the top-N results. In this analysis we tested results using an N value which ranges from 1 to 10. We calculated the hit ratio for each method investigated by each developer by comparing the top N results produced by each model with each next line visited by the developer. The hit ratios averaged over all subjects and methods investigated are depicted in Figure 3 and can be summarized in the following *finding*:

***F1—The proximity-based model has the highest hit ratio over all prediction models for each*** $N \leq 10$***.***

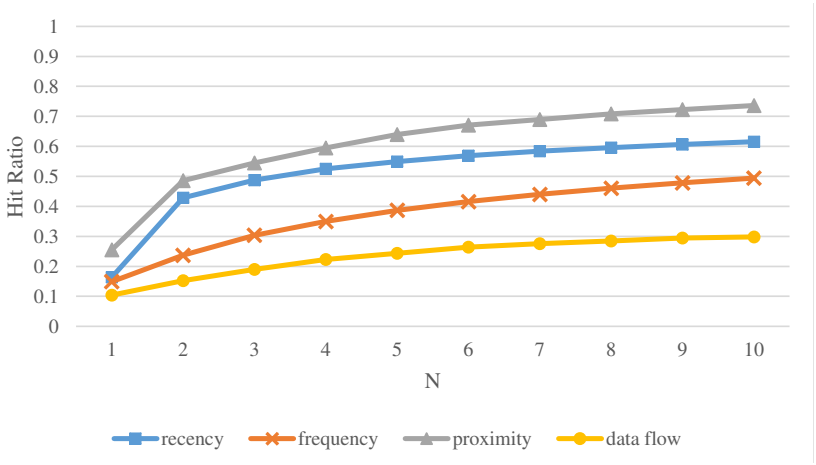


Figure 3: The averaged hit ratios over all subjects and methods investigated.

Overall, the proximity-based model converges most to the developers' navigation within methods. However, the average length of the methods included in this analysis differs quite a lot ($M = 53.03$, $SD = 139.37$). Thus, in the case of a short method (method length <= N) the proximity-based model simply recommended all the lines of a method and hence caused a high accuracy. The relatively high accuracy of the recency-

and frequency-based models confirms our observation **O3** that states that developers switch a lot between lines they already visited.

Due the comparatively low accuracy of the data flow-based model that does not provide strong support for **O4**, we conducted a follow up analysis to investigate possible causes. For this analysis, we grouped the methods investigated into "focused" and "skimmed" methods as we did when exploring **O7**. Methods of which more than half of the lines were looked at are defined as "focused", while the remaining ones are defined as "skimmed" methods. We then again calculated the accuracy of each model for just the "focused" methods. The results are depicted in Figure 4 and can be summarized in the following *finding*:

***F2—The hit ratios of the data flow-based model are substantially higher for "focused" methods compared to all explored methods.***

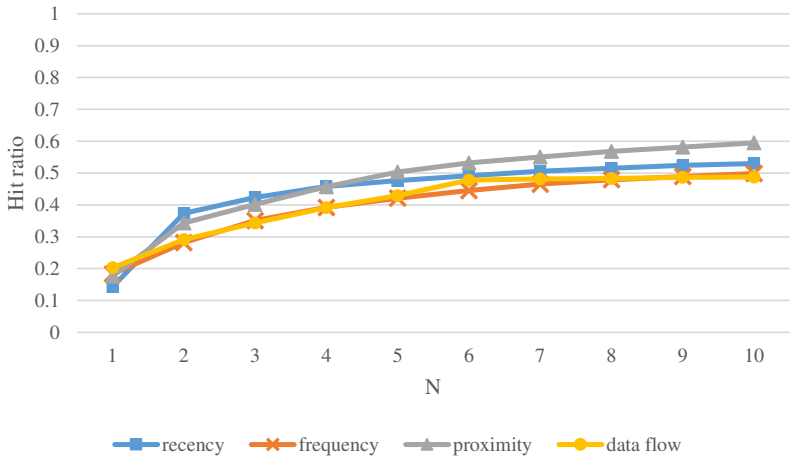


Figure 4: Average hit ratios of each prediction model overall focused methods.

When looking at the top recommendations of each model applied to the group of focused methods ($N = 1$), the data flow-model outperforms the remaining models with a hit ratio of 0.2. Overall, the different models do not differ as much for the focused methods as they do when applying the line prediction models to all methods (see Figure 3). In particular, the proximity-based model's hit ratios are considerably lower.

These results further show that it is highly important to understand which methods developers focus on when investigating a change task and that an automatic detection thereof could be of high value for tool support.

### *5.2. Predicting Task Difficulty*

Captured task contexts can be used to support different software engineering steps and aspects, for example to enable a task-focused development environment [3, 50], to support code navigation [48], or to localize reported bugs in the source code [51]. While many of these approaches are based on the source code elements within the captured contexts,

we additionally investigate the sequence of gaze and navigation steps. In particular, we use the gathered interaction and gaze contexts to predict perceived task difficulty of a developer when working on a change task. Knowing whether a developer experiences difficulties when working on a change task might inform approaches for prioritizing change task reviews or even for better interruption management.

For this analysis, we used the study participants' ranking of the change tasks into one of three categories of perceived difficulty and applied a stepwise multinomial logistic regression to predict membership in one of these categories. We used a stepwise multinomial logistic regression [46] on the variables gathered for the interaction and gaze contexts (see Table 2). We think that these parameters represent a good starting point for our analysis and we are also not aware of any previous research in the area that already identified variables for this kind of prediction in this context. Of the 55 change task investigations, participants ranked 8 (14.5%) as "easy", 26 (47.3%) as "average", and 21 (38.2%) as "difficult". Each task difficulty category includes change task investigation instances for all three kinds of change tasks of our study (see Table 1). Also, only change task investigations with the "easy" and "average" difficulty were performed successfully, providing further support for the validity of the rankings.

The final model of both kinds of context—interaction and gaze—allowed us to predict the perceived task difficulty significantly better than with the baseline model, i.e. a model that omits all variables and only uses the constant (see Table 4). The final prediction model based solely on gaze context, which captures substantially more, and more fine-grained data than interaction context (**O1**), allows a higher decrease in unexplained variance from the baseline model to the final model ($\chi^2(4) = 20.44, p < .001$), than the final prediction model based on the interaction context ($\chi^2(2) = 7.57, p = .023$). For the interaction context as well as for the gaze context the switch ratio to methods in close proximity has a significant effect on predicting the perceived task difficulty. For the interaction context, the switch ratio to methods in close proximity is significant with a p-value of $p = .023$ and decreases the amount of unexplained variance by 7.57. For the gaze context, the time to first focus ($\chi^2(2) = 8.88, p = .012$) and the switch ratio to methods in close proximity ($\chi^2(2) = 11.56, p = .003$) significantly helps to predict the difficulty level.

The parameter estimates that allow to compare two categories with each other (e.g., how the parameters compare for "easy" to "difficult") are summarized in Table 5.

**gaze context**

The parameter estimates presented in Table 5 indicate that for tasks that are perceived as "easy", developers skim the source code longer before a method was read thoroughly, i.e. read more than half of its lines, and that they looked less frequently to methods in close proximity than for tasks that are perceived "average" or "difficult".

**interaction context**

The parameter estimates presented in Table 5 paint a different picture for the navigation behavior when it is used to predict perceived difficulty. In particular, the parameters indicate that for tasks that are perceived as "average" developers are more likely to *select* methods in close proximity more often than for tasks that are perceived as "difficult".

Table 4: Results of applying a multinomial logistic regression to parameters of the gaze and the interaction context (2 Log-Likelihood measures how much unexplained variability there is, $\chi^2$ = chi-square, df = degrees of freedom, Sig. = Statistical significance).

| | 2 Log-Likelihood | $\chi^2$ | df | Sig. |
|---|---|---|---|---|
| interaction context | | | | |
| baseline model | 45.06 | - | - | - |
| final model | 37.48 | 7.57 | 2 | .023 |
| *proximate switches ratio* | 37.48 | 7.57 | 2 | .023 |
| gaze context | | | | |
| baseline model | 110.26 | - | - | - |
| final model | 89.81 | 20.44 | 4 | $< .001$ |
| *time to first focus* | 101.37 | 8.88 | 2 | .012 |
| *proximate switches ratio* | 89.81 | 11.56 | 2 | .003 |

Table 5: Parameter estimates of applying a multinomial logistic regression to the gaze and the interaction context (B = coefficient, Std.Error = Standard Error, df = degrees of freedom, Sig. = Statistical significance).

| | | B | Std.Error | df | Sig. |
|---|---|---|---|---|---|
| gaze context | "average" vs. "easy" | | | | |
| | Intercept | .05 | .78 | 1 | .95 |
| | proximate switches ratio | 8.13 | 3.63 | 1 | .025 |
| | time to first focused method (time-based) | -.106 | .041 | 1 | .01 |
| | "difficult" vs. "easy" | | | | |
| | Intercept | -.56 | .86 | 1 | .518 |
| | proximate switches ratio | 9.37 | 3.73 | 1 | .012 |
| | time to first focused method (time-based) | -.15 | .05 | 1 | .005 |
| interaction context | "easy" vs. "difficult" | | | | |
| | Intercept | -1.62 | .83 | 1 | .05 |
| | proximate switches ratio | 1.06 | 1.07 | 1 | .32 |
| | "average" vs. "difficult" | | | | |
| | Intercept | -1.36 | .72 | 1 | .06 |
| | proximate switches ratio | 2.24 | .86 | 1 | .012 |

***F3—The more often a developer looks at methods in close proximity and also the less time it takes until a developer explores a first method thoroughly, the more likely the developer perceives the task as difficult.***

## 6. 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. Furthermore, as participants were investigating three change tasks in the same source code, there might be a learning effect which threatens the internal validity of this study. We counteract this learning effect by applying a counterbalance measure design.

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. This restriction might has influenced the way the developers explored the source code and specifically the relatively low call relationships which were observed between the explored methods might be influenced. Further, it is imaginable that generally more lines within a method might be read when using a debugger. 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.

## 7. 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.

### *7.1. 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 [3], 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 [3] 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.

### *7.2. Finer Granularity of Data and Task Focus*

Most current research focuses on how developer build up context on class or method level. 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 [50]. Similarly, approaches to recommend relevant code elements for a task, such as Mylyn [3, 5] or wear-based filtering [48], display the change task context on class and method level. While the method and class level are important, our results show that developers build up their context by focusing only on small fractions (on average 32%) of methods (***O3***). Hence, exploring the fine-grained fragments of a change task context might enable to inform new approaches to identify and highlight the parts which are relevant for the current task.

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 [52, 53], 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 [54] or Prompter [55].

Our results also suggest that task contexts can be used to assume a developer's perceived difficulty. Since no change task was successfully solved with a high perceived difficulty, this information could be used to inform approaches which prioritize change tasks to be assigned to developers. Furthermore, if we could recognize when a developer is having difficulty, adequate approaches to help could be provided, such as suggesting expert-developers for that place in the source code.

Furthermore, the fine-grained eye-tracking data also enables to recognize when developers are skimming source code (***O8***). This might inform approaches which depict summaries of source code elements first and if they want to focus on a specific element the view zooms in and hides remaining irrelevant source code, such that developers are not distracted by proximate source code.

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 [56, 57] rather than only summarizing the parts relevant for a given task. Similarly, the approach by Rodeghero et al. [38] 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.

### *7.3. 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 [6]. 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., [18]), 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.

While developers switch relatively often between methods, they only focus on few methods ***O7***. Exploring further how these methods can automatically be distinguished

from the remaining methods, might improve approaches to summarize task contexts which can help resume work faster, be shared with colleagues, or be mined again for further research ideas.

### *7.4. Eye-Tracking for Each 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, summarization, or even recognizing the perceived difficulty. 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. In our study, we recalibrated twice during each session to make sure that we captured eye gazes on the correct source code lines. Further, we also asked the study participants to not make very large head movements (small head movements are natural and taken care of by the eye tracker's headbox). 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.

## 8. 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. Furthermore, the fine-grained gaze context showed that developers focus only on a few methods when investigating a change task.

These detailed findings provide insights and opportunities for future developer support. For instance, our approach for fine-granular navigation recommendations or our approach to recognize the perceived task difficulty demonstrate the potential of capturing

gaze contexts. The findings demonstrate further 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.

## Acknowledgement

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 the SNF grant DARINO (200021_ 150050) and an ABB grant.

## References

[1] A. J. Ko, B. A. Myers, M. J. Coblenz, H. H. Aung, An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks, IEEE Transactions on Software Engineering 32 (2006) 971–987. doi:http://doi.ieeecomputersociety.org/10.1109/TSE.2006.116.

[2] T. D. LaToza, G. Venolia, R. DeLine, Maintaining mental models: a study of developer work habits, in: ICSE '06: Proceedings of the 28th international conference on Software engineering, ACM, New York, NY, USA, 2006, p. 492501.
URL http://research.microsoft.com/apps/pubs/default.aspx?id=74240

[3] M. Kersten, G. C. Murphy, Using task context to improve programmer productivity, in: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering, 2006, pp. 1–11.

[4] R. Robbes, M. Lanza, Spyware, in: Software Engineering, 2008. ICSE '08. ACM/IEEE 30th International Conference on, 2008, pp. 847–850. doi:10.1145/1368088.1368219.

[5] eclipse.org/mylyn/, accessed: 2015-03-15.

[6] M. P. Robillard, Automatic generation of suggestions for program investigation, in: ACM SIGSOFT Software Engineering Notes, Vol. 30, ACM, 2005, pp. 11–20.

[7] D. Piorkowski, S. Fleming, C. Scaffidi, C. Bogart, M. Burnett, B. John, R. Bellamy, C. Swart, Reactive information foraging: An empirical investigation of theory-based recommender systems for programmers, in: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI '12, ACM, New York, NY, USA, 2012, pp. 1471–1480. doi:10.1145/2207676.2208608.
URL http://doi.acm.org/10.1145/2207676.2208608

[8] J. Siegmund, C. Kästner, S. Apel, C. Parnin, A. Bethmann, T. Leich, G. Saake, A. Brechmann, Understanding understanding source code with functional magnetic resonance imaging, in: Proceedings of the 36th International Conference on Software Engineering, ICSE 2014, ACM, New York, NY, USA, 2014, pp. 378–389. doi:10.1145/2568225.2568252.

[9] T. Fritz, A. Begel, S. C. Müller, S. Yigit-Elliott, 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, ACM, New York, NY, USA, 2014, pp. 402–413.

[10] B. Sharif, J. I. Maletic, An eye tracking study on camelcase and under_score identifier styles, in: 18th IEEE International Conference on Program Comprehension (ICPC'10), 2010, pp. 196–205.

[11] R. Bednarik, Expertise-dependent visual attention strategies develop over time during debugging with multiple code representations, International Journal of Human-Computer Studies 70 (2) (2012) 143–155. doi:10.1016/j.ijhcs.2011.09.003.

[12] K. Kevic, B. M. Walters, T. R. Shaffer, B. Sharif, D. C. Shepherd, T. Fritz, 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, 2015, pp. 202–213. doi:10.1145/2786805.2786864.
URL http://doi.acm.org/10.1145/2786805.2786864

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

[14] B. Shneiderman, R. Mayer, Syntactic/semantic interactions in programmer behavior: A model and experimental results, International Journal of Parallel Programming.

[15] 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, Ablex Publishing Corp., Norwood, NJ, USA, 1986, pp. 28–47.
URL http://dl.acm.org/citation.cfm?id=21842.28884

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

[17] A. Ko, B. Myers, M. Coblenz, 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) (2006) 971–987. doi:10.1109/TSE.2006.116.

[18] T. Fritz, D. C. Sheperd, K. Kevic, W. Snipes, C. Braeunlich, Developers' code context models for change tasks, in: 22nd ACM SIGSOFT International Symposium on the Foundations of Software Engineering/ FSE 2014, ACM, Hong Kong, China, 2014, pp. 7–18.

[19] M. Kersten, 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, ACM, New York, NY, USA, 2005, pp. 159–168. doi:10.1145/1052898.1052912.
URL http://doi.acm.org/10.1145/1052898.1052912

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

[21] C. Parnin, C. Gorg, Building usage contexts during program comprehension, in: Proceedings of the 14th IEEE International Conference on Program Comprehension, 2006, pp. 13–22. doi:10.1109/ICPC.2006.14.

[22] D. Piorkowski, S. Fleming, C. Scaffidi, L. John, C. Bogart, B. John, M. Burnett, 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, 2011, pp. 109–116. doi:10.1109/VLHCC.2011.6070387.

[23] V. Vinay Augustine, P. Francis, X. Qu, D. Shepherd, W. Snipes, C. Bräunlich, 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, pp. 229–238.

[24] T. Lee, J. Nam, D. Han, S. Kim, 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, 2011, pp. 311–321. doi:10.1145/2025113.2025156.
URL http://doi.acm.org/10.1145/2025113.2025156

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

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

[27] S. C. Müller, 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, pp. 688–699.

[28] M. Züger, 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, ACM, 2015, pp. 2981–2990.

[29] M. E. Crosby, J. Stelovsky, How do we read algorithms? a case study, Computer 23 (1) (1990) 24–35.
URL http://dl.acm.org/citation.cfm?id=77577.77580

[30] B. Sharif, G. Jetty, J. Aponte, E. Parra, An empirical study assessing the effect of seeit 3d on comprehension, in: 1st IEEE International Working Conference on Software Visualization (VISSOFT

2013), pp. 1–10.

[31] S. Yusuf, H. Kagdi, J. Maletic, Assessing the comprehension of uml class diagrams via eye tracking, in: Proceedings of the 15th IEEE International Conference on Program Comprehension, 2007, pp. 113–122. doi:10.1109/ICPC.2007.10.

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

[33] B. Sharif, 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), pp. 1–10.

[34] R. Turner, M. Falcone, B. Sharif, 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, ACM, Safety Harbor, Florida, 2014, pp. 231–234.

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

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

[37] R. Bednarik, M. Tukiainen, An eye-tracking methodology for characterizing program comprehension processes, in: Proceedings of the Symposium on Eye Tracking Research & Applications, ACM, 2006, pp. 125–132.

[38] P. Rodeghero, C. McMillan, P. W. McBurney, N. Bosch, 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, ACM, New York, NY, USA, 2014, pp. 390–401. doi:10.1145/2568225.2568247.
URL http://doi.acm.org/10.1145/2568225.2568247

[39] B. Sharif, 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), pp. 67–70.

[40] B. Walters, M. Falcone, A. Shibble, 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), pp. 51–54.

[41] B. Walters, T. Shaffer, B. Sharif, H. Kagdi, Capturing software traceability links from developers' eye gazes, in: Proceedings of the 22nd International Conference on Program Comprehension, ICPC 2014, ACM, New York, NY, USA, 2014, pp. 201–204. doi:10.1145/2597008.2597795.
URL http://doi.acm.org/10.1145/2597008.2597795

[42] www.csis.ysu.edu/~bsharif/itraceMylyn, accessed: 2015-03-15.

[43] jabref.sourceforge.net/, accessed: 2015-03-15.

[44] T. R. Shaffer, J. Wise, B. M. Walters, S. Müller, M. Falcone, 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), ACM, 2015, p. in press.

[45] www.tobii.com/, accessed: 2015-03-15.

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

[47] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, T. Zimmermann, What makes a good bug report?, in: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, SIGSOFT '08/FSE-16, ACM, New York, NY, USA, 2008, pp. 308–318. doi:10.1145/1453101.1453146.
URL http://doi.acm.org/10.1145/1453101.1453146

[48] R. DeLine, A. Khella, M. Czerwinski, G. Robertson, Towards understanding programs through wear-based filtering, in: Proceedings of the 2005 ACM symposium on Software visualization, ACM, 2005, pp. 183–192.

[49] T. Zimmermann, A. Zeller, P. Weissgerber, S. Diehl, Mining version histories to guide software changes, Software Engineering, IEEE Transactions on 31 (6) (2005) 429–445. doi:10.1109/TSE.2005.72.

[50] A. Bragdon, R. Zeleznik, S. P. Reiss, S. Karumuri, W. Cheung, J. Kaplan, C. Coleman, F. Adeputra, 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, ACM, 2010, pp. 2503–2512.

[51] K. Kevic, T. Fritz, A dictionary to translate change tasks to source code, in: Proceedings of the 11th Working Conference on Mining Software Repositories, MSR 2014, ACM, New York, NY, USA, 2014, pp. 320–323. doi:10.1145/2597073.2597095.
URL http://doi.acm.org/10.1145/2597073.2597095

[52] M. Weiser, Program slicing, in: Proceedings of the 5th international conference on Software engineering, IEEE Press, 1981, pp. 439–449.

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

[54] D. Čubranić, G. C. Murphy, Hipikat: Recommending pertinent software development artifacts, in: Proceedings of the 25th International Conference on Software Engineering, ICSE '03, IEEE Computer Society, Washington, DC, USA, 2003, pp. 408–418.
URL http://dl.acm.org/citation.cfm?id=776816.776866

[55] L. Ponzanelli, G. Bavota, M. Di Penta, R. Oliveto, 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, ACM, 2014, pp. 102–111.

[56] S. Haiduc, J. Aponte, A. Marcus, Supporting program comprehension with source code summarization, in: Proceedings of the 32nd International Conference on Software Engineering, 2010, pp. 223–226.

[57] S. Haiduc, J. Aponte, L. Moreno, 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), IEEE, 2010, pp. 35–44.