Eyes on Code: A Study on Developers' Code Navigation Strategies

Zohreh Sharafi, Ian Bertram, Michael Flanagan, and Westley Weimer

Abstract—What code navigation strategies do developers use and what mechanisms do they employ to find relevant information? Do their strategies evolve over the course of longer tasks? Answers to these questions can provide insight to educators and software tool designers to support a wide variety of programmers as they tackle increasingly-complex software systems. However, little research to date has measured developers' code navigation strategies in ecologically-valid settings, or analyzed how strategies progressed throughout a maintenance task. We propose a novel experimental design that more accurately represents the software maintenance process in terms of software complexity and IDE interactions. Using this framework, we conduct an eye-tracking study (n=36) of realistic bug-fixing tasks, dynamically and empirically identifying relevant code areas. We introduce a three-phase model to characterize developers' navigation behavior supported by statistical variations in eye movements over time. We also propose quantifiable notion of "thrashing" with the code as a navigation activity. We find that thrashing is associated with lower effectiveness. Our results confirm that the relevance of various code elements changes over time, and that our proposed three-phase model is capable of capturing these significant changes. We discuss our findings and their implications for tool designers, educators, and the research community.

Index Terms—Code navigation, Eye tracking, Human factors, Software maintenance

1 Introduction

It has been estimated that a rising 50% to 75% of the overall cost of a software system is dedicated to maintenance [1], [2]. Developers reportedly devote around 35% of that task time finding and navigating relevant areas of code to either fix bugs or add new features [3]–[5]. Improved knowledge of developers' cognitive processes of code maintenance and their navigation strategies through relevant code areas can aid in improving our current development tools and teaching methods [3], [6]–[8]. A more detailed understanding of developers' navigation behavior can also lead to significant productivity gains or reductions in the time and cost of the software life cycle [3], [4].

Several studies have investigated the code navigation strategies [9]–[11] or the areas of code developers read [12]–[14]. These studies have significantly advanced our understanding of program comprehension and bug fixing, but the majority used small code snippets (*i.e.,* few or single classes and methods). Complex environments involving many files and classes are more indicative of software engineering (SE) practice. To the best of our knowledge, only a few previous studies featured participants with access to all of the source code in the system [4], [15], [16].

In this work, we exploit three insights. First, existing techniques can be fruitfully combined to *track* eye gaze data during scrolling and editing, permitting an ambitious experimental design. Eye gaze data is an accurate representation of the underlying cognitive processes. By indicating both the targets of the participant's attention and the effort (or lack thereof), eye gaze data provide detailed insights into developers' navigation behavior [17], [18]. Second, we can evaluate the amount of time developers spend to navigate, write, and understand source code by analyzing the interaction data, such as browsing the source code of a method or inspecting an object at run-time. Third, we can *quantify* phases of the software maintenance process in terms of gaze behavior and developers' interactions with the source code in an integrated development environment (IDE), capturing time-varying activity.

Based on these insights, we conducted an experiment in which 36 participants worked for 40 minutes on bug-fixing tasks on a realistic project using the Eclipse IDE. We are examining: 1) how developers find, use, and track relevant information over time while performing realistic maintenance tasks and 2) to what extent participants' performance and outcome impact their strategies?

We develop a new model to quantify developers' navigation strategies based on areas of code that are given significant attention, as well as transitions between these areas. We divide a bug-fixing task into three phases: 1) **finding** the relevant parts of the code, 2) **learning** and understanding them, and 3) **editing** the code to fix the bug. These phases adhere to the first three categorizations presented by Sillito *et al.* [19] to organize the critical questions asked by developers during a programming change task. We use a set of eye-movement metrics in isolation to validate our proposed model. By analyzing the changes to the recorded values of these metrics, we found significant support for our three-phase model: participants showed divergent code navigation strategies across three distinct phases.

We perform a versatile multidimensional analysis of participants' navigation strategies and find that these strategies evolve over time and that we can detect these changes by quantitatively analyzing their eye movements and IDE interactions. In addition, we find notable differences in

- *Z. Sharafi, I. Bertram, M. Flanagan and W. Weimer are with the Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI, 48109.*
  E-mail: {zohrehsh,ianbtr,mflanag,weimerw}@umich.edu

Manuscript received FIXME; revised FIXME.

individual participant effectiveness. In that vein, we adapt a notion of *thrashing* — informally, "excessively switching between code elements" — as a quantifiable, critical activity in code navigation that partially accounts for individual differences. We observe that unsuccessful participants frequently shift their attention between areas of the code: thrashing is associated with lower effectiveness. We also find that thrashing behavior and the distribution of attention tended to be shared amongst the entire group of developers regardless of their experience level.

The contributions of this paper are as follows:

- A new experimental design, and an accompanying analysis method, combining current eye-tracking technology with support for both scrolling and movement in large codebases as well as edits to the code.
- A new three-phase model that characterizes and quantifies developers' navigation behavior during realistic bug-fixing tasks. We express these phases in terms of significant, measurable events.
- A detailed observations on code navigation behavior of 36 developers working on realistic bug-fixing tasks by combining eye gaze and IDE interaction data. We release our de-identified results for replication and analysis.
- A quantitative notion of measurable "thrashing" as an activity of navigation behavior that has adverse effects on performance and effectiveness.

2 Related Work

We place our work in context with respect to eye-tracking studies of code comprehension and bug fixing. We also summarize relevant studies of monitoring and analyzing developers' interaction with the system in hand while working on a maintenance task.

2.1 Eye-tracking and Code Navigation

Many early eye-tracking studies included a focus on expertise or strategies. Crosby *et al.* [20] performed the first eye-tracking study in software engineering while investigating the role of expertise on the developers' navigation strategies. In the same vein, Aschwanden *et al.* [21] looked into various visual attention patterns deployed by developers and their variances between experts and novices.

Uwano *et al.* [11] investigated the impact of scan time (time spent reading the entire code before debugging) on how developers find defects and reported that longer times help developers to find defects faster. In a partial replication, Sharif *et al.* [22] reported two different debugging strategies, deployed by experts and novices, to find defects more efficiently. Busjahn *et al.* [14] analyzed the attention distribution on code elements to detect and compare experts' and novices' code reading strategies and reported that most attention is given to identifiers, operators, keywords, and literals, in that order.

More recent studies have measured eye behavior during a richer set of software engineering tasks, including code changes and summarization. Rodeghero *et al.* [13] performed an eye-tracking study to analyze the eye movements of ten developers performing code summarization manually to improve current information retrieval algorithms that automatically summarize code. The authors further extended their work and analyzed the eye movement patterns of developers during the code summarization task and observed analogous eye-movement patterns between code and natural language reading [23]. Kevic *et al.* [4], [8] used user interaction monitoring combined with eye-tracking data to investigate developers' navigation strategies within and between methods while working on a change task. They concluded that developers spend more time on data flow areas of the methods and mostly switch their attention to the areas that are adjacent or nearby. Abid *et al.* [15] conducted an eye-tracking study similar to the work done by Rodeghero *et al.* [23]. However, the authors changed the study environment and used an actual IDE while investigating the impact of using methods of different sizes on the developers' code reading behavior.

2.2 IDE Interaction Analysis

Researchers proposed several approaches and tools to automatically capture and track developers' interaction with the code [8], [24]–[26]. Tools, such as Mylyn [27], FLUORITE [28] and Eclipse Usage Collector [29], have been used to automatically capture and store IDE interaction logs along with timestamps [8], [30], [31]. Previous work leveraged this data for different purposes. They investigate how developers use various features and windows in an IDE [24], navigate through code [25], [26], browse through software [32], distribute their time across different development activities [30], or use various refactoring tools [33]. Ahmed *et al.* [34] mined about 75,000 copy and paste (C&P) incidents captured by Eclipse Usage Data Collector extension. The results show that C&P is performed regularly by developers, and developers follow different C&P patterns than regular users.

However, previous studies are different from our work with regards to context (*e.g.*, refactoring), the level of details, and granularity of the recorded data. The majority of the previous work focus on the method level [8], [25], [30] or how various IDE features have been utilized by developers [24]. In this work, the granularity is mainly class file-level and how developers switch between various code elements in different files to fix the bug.

2.3 Summary of Related Work

To our knowledge, our approach is the first aiming at automatically splitting a development session into activity-related sub-sessions based on eye-gaze data. We propose a three-phase model to characterize developers' code navigation by analyzing the eye-movement data. The study presented in this paper has similar intent but differs from prior work in several ways. The majority of the previous work did not allow participants to change the code and used small code snippets that fit on one screen without scrolling. Similar to the work performed by Abid *et al.* and Kevic *et al.* [8], [15], we mitigate the limitations related to the study environment by allowing participants to access all the source code in the system. Also, in the majority of previous work, the size and location of the areas of interest were defined using a top-down approach assuming

a priori information about which areas might matter to participants, while we automatically detect and refine the areas of interest. Finally, in contrast to some previous work, we not only look at how developers interact with the code but also focus on determining how successful developers complete the maintenance tasks compared to the unsuccessful ones. This comparative analysis allowed us to make detailed observations about the code investigation behavior that contributes to developers' effectiveness at maintenance tasks.

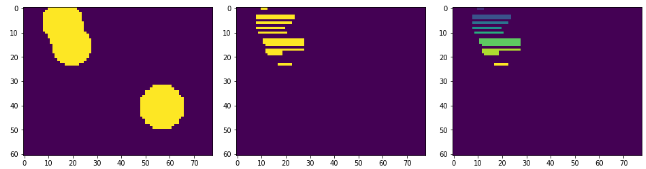

Fig. 1: Dynamic area of interest identification: a mask obtained by Gaussian smoothing and thresholding with a matrix of fixations (left), the intersection of this mask with code (center), and color-labeled areas of interest (right) in which the darkest line is over the package name, the next few lines belong to the comment preceding the class definition, and so on.

3 Analyzing Navigation Strategies

We propose a novel method for gathering and analyzing eye-tracking data in indicative bug-fixing tasks. A high level outline of our experimental and data analysis approach is as follows:

1) Collect eye gaze data while participants navigate code in a realistic manner (*i.e.,*scrolling, switching between files, editing the code and running the code using an IDE) (Section 3.1).
2) Empirically infer areas of the code that held participants' attention, and are thus significant (Section 3.2).
3) Separate the data into temporal segments representing the major phases of the bug fixing activity (Section 3.3).
4) Characterize participants' navigation strategies in terms of the distribution and transition of visual attention and IDE interactions (Section 3.4).

In the remainder of this section, we detail the individual steps in our proposed approach.

3.1 Eye Tracking with Edits and Scrolling

We aim for an empirical model to capture navigation strategies that: (1) is based on measurements from more realistic scenarios, including *scrolling and editing* larger programs; (2) accounts for how navigation behaviors *change over time*, such as from locating a relevant area to acting on it; and (3) adequately explains observed *individual variations* in outcomes.

A model of code navigation strategies expressed in terms of eye movements requires that we establish a link between gaze and code elements. For static code without scrolling or editing, that essentially reduces to a coordinate system transformation between the eye-tracking camera and screen pixels followed by a mapping from screen pixels to code elements. Allowing participants to have normal interactions with the IDE, such as scrolling, editing, and other low-level navigation actions require a more sophisticated mapping between gaze and code elements. To the best of our knowledge, no single proposed eye-tracking algorithm addresses the entirety of this problem, but we propose a combination of existing solutions to do so.

Existing eye-tracking software, such as iTrace [35], [36], supports scrolling, and some other navigation features, but do not support edits to the code. Also, iTrace plugin collects eye gaze data only within the IDE code window. Dually, editor plugins, such as FLOURITE [28], support recording low-level IDE actions, but do not incorporate eye-tracking data. We propose an experimental setup that combines these approaches, using iTrace to record gaze data and FLUORITE to detect and timestamp edits while gathering information on how developers interact with non-code elements and other features of an IDE. The result of this combination is a unified time series in which eye information and editor actions are associated with temporal offsets throughout a long-running software maintenance task.

Although we have access to gaze data at all times, we propose to abstract away gaze behavior during typing. This has the advantages of both simplifying our mathematical model and also removing certain sources of noise (*e.g.,* participants looking at the keyboard or an IDE animation occluding code elements). We propose a post-processing system that separates data into segments during which the code files were not changed by the participant, and we discard data associated with edits themselves. However, this filtering retains pre- and post-edit gaze data. Each segment is then subjected to standard post-processing, and we concatenate the resulting output into a time series dataset that spans the entire experiment.

3.2 Automatic Discovery of AOIs

The *area of interest* (AOI) formalism is widely used to describe visual stimuli. We present a method that empirically computes only those AOIs that are most relevant to general maintenance activities, producing regions that can be used by subsequent analyses. There are no universal guidelines for defining appropriate AOIs in terms of size and granularity [37]. Researchers define AOIs based on the study's research questions, hypotheses, and variables [37], [38]. In current eye -tracking research, studying coding or otherwise, it is standard to assume the locations of AOIs are known *a priori*. We claim that *a priori* assumptions about AOI locations both disregard valuable information about participant attention and treat all AOIs equally for all participants over the course of the whole maintenance task. Suppose, for example, that we wish to analyze data that was gathered from participants viewing a painting. We may divide the painting into the AOIs "foreground" and "background", and observe that the participants generally pay more attention to one or the other. While this finding may be useful and significant, this method of analysis cannot discern which objects in the foreground or background actually drew the participant's attention unless the researcher pre-annotates them. Depending on how the experiment is framed, this runs the risk of confounds, false positives or false negatives.

For example, the claim that "the foreground drew the most participant attention" may be misleading if, in fact, a small but striking piece of the foreground drew participants' gazes. In addition, if the scene changes over time, AOIs associated with earlier changes may no longer correspond to semantically-meaningful visual regions. We propose a new method that would permit empirical determinations of AOIs and would support claims that a particular area of a given stimulus is essential to its comprehension.

We formalize areas of interest in the source code files using a two-dimensional mask (or filter) on the screen (at a particular point in time) that selects relevant data. Instead of pixels, we use characters as the most basic spatial units. iTrace records the line and column numbers associated with a particular fixation. The position of a single character's cell is defined by zero-indexed line and column numbers. We produce a 2D logical array describing the intersection between the subjects' gaze and the code in the a specific source code file. Formally, each of our AOIs is a rectangular bounding box around a set of characters.

We compute an intermediate two-dimensional array of of fixation counts, correspond to the generated bounding boxes. We then apply Gaussian smoothing (smoothing parameter = 5.0) and thresholding to a matrix representing on-screen eye fixations to obtain a mask. The mask corresponds to areas in which substantial numbers of fixations occurred during a given period. This matrix-smoothing method is inspired by the approach of Caldara *et al.* [39], as implemented in their iMap4 tool [40]. We then further intersect the mask with the position of a stimulus to compute a set of areas in which the given stimulus was fixated on many times. The process is illustrated in Figure 1. The first step is generating the gaze mask. Lighter circles in the left image show eye fixations. The middle image shows relevant code elements (horizontal lines), and the right image shows their intersection (with semantic code elements labeled via colors). The darkest line is over the package name, the next few lines all belong to the comment preceding the class definition, member variables, and so on. We use this technique as a filtering method. Fixations outside of these AOIs are assumed to be located over whitespace (as in the lower-right visual fixation in Figure 1), or over areas that were not given enough attention to be significant to software engineering behavior and decisions. Such fixations are removed from further analysis.

A traditional approach may have yielded to the same set of AOIs. Our automated method eliminates the tedious effort of manually annotating the whole project and avoids considering code elements that received an insignificant amount of visual attention compared to the rest of the regions.

3.3 Partitioning Eye Behavior into Phases

We observe that a long-running bug fixing activity, such as the 20-minute bug-fixing tasks in our experiment, may feature multiple distinct *phases* with different associated behaviors and goals. A model that accounts for such phases and transitions, rather than conflating all observations, has two benefits. First, it allows us to give a more precise (phase-varying) characterization of developers' navigation behavior. Second, since we hypothesize that different developers may work at different individual rates but will use the same broad phases to solve the same task,

We propose to automatically quantify and partition processed gaze data into three phases:

- **Finding** areas in the code that are relevant to the bug.
- **Learning** about those relevant areas by exploring the code, and analyzing the code elements involved and their structural/hierarchical relationships.
- **Editing** the code to fix the bug.

These three phases are based on qualitative descriptions by Sillito *et al.* [19]. To partition the data, we define quantitative rules for distinguishing the phases from each other. Each participant begins in the first phase. A participant enters the second phase after making k consecutive fixations in any area of the code *semantically related* to the target issue. This consecutive fixation requirement is based on the immediacy assumption [18], which states that participants will attempt to interpret a stimulus as soon as they view it. Therefore, the requirement ensures that participants have found the regions related to the issue and now start to better understand it. At this phase, participants also make some "information-gathering" actions such as print-statement debugging or testing (recall that we give participants full access to the dynamic IDE).

A participant enters the third phase when the first action *to resolve the bug* is made. We assume that participants have built a sufficient understanding of any relevant areas by the time they make a first editing change.

The threshold k, the semantic relation, and the set of actions *to resolve the bug* are task-specific; we specify them for the particular debugging task in our experiment in Section 6. The Time to First Fixation, one of the most used eye-tracking metrics [41]–[43], indicates the amount of time that it takes a participant to look at a specific AOI from stimulus onset. It characterizes the participants stimulus-driven searches by capturing when they actively decide to focus on certain AOIs. Previous work used this metric for small, static stimuli consisting of a handful of AOIs. However, in our experiment, we deal with a broad set of AOIs (*i.e.,* source code entities), and we aim to find the moment that the participant sees the relevant code entity and starts investigating to understand it better. We randomly selected 15 participants and manually verified the timestamps at which they started reading the relevant method and evaluating its content. The average time for these selected participants centered around the time of their 10th fixations, so we choose the threshold $k = 10$ for our analyses. We intentionally avoid more nuanced models (*e.g.,* moving back and forth between phases). We acknowledge the risk associated with this choice (See Section 7), yet to the best of our knowledge, this paper represents the first attempt to automatically infer such phase behaviors quantitatively from long-running eye-tracking data in an indicative editing-and-scrolling setting, and we favor comprehensibility and simplicity to aid in interpreting novel results. We evaluate the goodness of fit for this model in Section 5. A pair of modules and sample script for processing output from the iTrace plugin, particularly when used in tandem with the FLUORITE plugin, along with our script for AOI

extraction and assignment are available on our website.[1]

3.4 Quantifying and Comparing Navigation Strategies

We use the information from the previous subsections, including eye gaze data aligned with edit actions and scrolling, and rules for determining phases, to analyze and compare the navigation strategies of participants. A *strategy* models gaze data, navigation, and action trends over time throughout a task: variations in gaze and IDE interaction behavior across a spectrum of phases. In the following, we structure our analysis along with two categories: quantifying navigation strategies via eye gaze data and providing a detailed patterns of IDE interactions.

3.4.1 Gaze Context

To admit indirect comparison with previous work [38], [41], [44], [45], we use scanpath metrics to quantify navigation strategies. A *scanpath* is a series of fixations or AOIs in chronological order. We use scanpaths because they encompass an entire range of eye-gaze data as one formal construct, revealing visual attention trend over time and in space. Scanpath analyses are common in eye-tracking studies [37], [38], [46]. Scan-path analyses can be done by either 1) looking at standalone metrics individually or by 2) directly computing and comparing scan paths. We chose the first approach because of the dynamic nature and large size of our experiment. In our study, participants deal with dynamic stimuli while continually interacting with them. As a result, currently available scan-path comparison algorithms such as the Levenshtein algorithm [38], Scanmatch [47], or MultiMatch [48] cannot be applied to our more modern experiment setup. As fixation-based algorithms, they all depend on the screen coordinates, which change when participants use scrolling and are undefined if participants switch to another file. Also, the AOI-based scan path comparison algorithms work based on the assumption that we have a limited number of AOIs, which is not the case while working with the larger code projects in our experiment.

We use three specific, standard fixation and saccadic metrics as the components of a scanpath for our analysis. A *fixation* is the stabilization of the eye on part of a stimulus for a period of time (typically 200–300 ms). A *saccade* is the rapid eye movements between fixation points [18], [49]. We compute the quantity of transitions between AOIs and the temporal distribution of fixations among these regions. Transition counts have been shown to capture the dynamics of visual attention [44]. We apply these traditional metrics to provide a detailed analysis of navigation strategies:

Fixation Count (FC) is the total number of fixations, indicates the number of attention shifts required to complete the task [18].

Average Fixation Duration (AFD) is the sum of the duration of all the fixations divided by the number of fixations. It reflects either difficulty in extracting information or that the stimulus is more engaging in some way [18]. Longer fixations denote greater cognitive demands [49].

1. https://web.eecs.umich.edu/~weimerw/data/navigation-behavior/

TABLE 1: Complete list of IDE interaction events.

Event Type	IDE Event	Description
Navigation	Move caret	Move cursor via the mouse
	Find	Find or find & Replace
	Open file	Open or activate a new file
Editing	Insert	Text insertion
	Delete	Text deletion
	Replace	Deletion & insertion
Inspection	Select text	Select (highlight) text
	Run	Run/Debug the application
Understanding	-	None of the above

Saccadic Length (SL) is the average Euclidean distance between consecutive fixations, in pixels. Higher SLs correspond to finding peripheral features faster [17], [18], while more complicated tasks are associated with shorter saccades (consequently higher number of fixations) [41], [44].

3.4.2 IDE Interaction Context

Beyond analyzing the viewing behavior, we investigate participants' IDE interaction to provide detailed observations on how they navigate within source code. We first describe interaction data and its properties. FLUORITE records various types of events, as shown in Table 1.

We classify interaction data according to the following groups of FLUORITE events [28]:

- Navigation events capture moving around the code, *e.g.*, by clicking on a class or method name.
- Editing events capture writing of the code by editing the text.
- Inspection events happen when developers select a text, open the debugger, or run the application.
- Understanding events are the remainder: times when developers are reading code. These events correspond directly to program comprehension [28], [50].

For each phase, per participant, we create a sequence of interaction events. Beyond Understanding events, the other events are very quick as they represent the action that is triggered by the developer. Thus, we look at the number of occurrences (frequencies) of these events while computing an estimate of developers' understanding activities.

4 Experiment Design

We recruited 36 participants and conducted an exploratory study to investigate the navigation strategies of developers for realistic maintenance tasks. Each participant worked on two bug-fixing tasks for a Java implementation of *MineSweeper* in the Eclipse IDE for up to 40 minutes. All study materials, including stimuli and de-identified responses, are available on our website 1[0].

4.1 Participants and Recruitment

The study participants were 36 volunteers. The participants were in B.Sc., M.Sc., and Ph.D. programs in the Department of Computer and Software Engineering at the authors' institution. We received the agreement from the Ethical Review Board of the authors' institution to perform and publish this study. Table 2 presents self-reported demographic data of our participants. Participants were recruited through

TABLE 2: Participant demographics.

Characteristics	Participants All (36)	Men (20)	Women (16)
Age (n (%))			
18-25	32 (88%)	18	14
25-30	4 (12%)	2	2
Class standing (n (%))			
2nd-Year	6 (17%)	3	3
3rd-Year	6 (17%)	2	4
4th-Year	8 (22%)	7	1
M.S./PhD	16 (44%)	8	8

email and were compensated with $25 voucher for their participation. We asked participants about their experience and familiarity with programming, Object-Oriented design, and IDEs. We performed standard screening, including the exclusion of participants with color-blindness, epilepsy, or seizures. Participants also completed questionnaires to gather basic information: age, gender, native and speaking languages, and educational attainment. Participants reported an average of 0.68 years (SD = 1.5) of industrial experience (*e.g.*, from working as a developer with a company). All participants were familiar with Java and had previous experience working with IDEs. Twenty-six of the 36 (72%) participants rated their IDE skills as above average, and six (16%) rated them as average. Siegmund *et al.* [51] reported that self-estimation is a reliable way to judge programming experience, especially when working with students. To reduce stereotype threat, especially its negative impact on underrepresented minorities [52], [53], participants answered questions about their coding knowledge and experience at the end of study.

4.2 Procedure

We conducted the experiment in a quiet room with an eye-tracking system; the participants were seated approximately 70 cm away from the screen in a comfortable swivel chair with arm rests. Before running the experiment, all participants signed a consent form and the experimenter verbally explained the procedure of the experiment in detail. Participants were informed that the experiment consists of one Java project and they will be fixing two bugs, one after the other. The experimenter provided no explanation to participants on the particular goal of the experiment.

Participants were given twenty minutes per task and were instructed to inform the experimenter if they finished early to stop the tracking. To mitigate any learning effects, participants received the two tasks in randomized order. Each participant used the Eclipse IDE augmented with iTrace and FLUORITE (see Section 3.1). The participants debugged *MineSweeper* source code, initially opened to a file containing a bug report as a block comment spanning the first few lines. To have better control and avoid any other factors impacting the results, no additional tools were installed and the participants were instructed to always maintain the full screen setup, not to use the debugger, and not to browse the internet.

They subsequently completed a post-questionnaire which asked them to explain (1) their approach to solve the previous tasks, and (2) their level of experience with programming in general and Java in particular. It is common to ask participants to fill out a survey regarding their experience or knowledge of software development and maintenance before the study begins. However, to mitigate the stereotype threat [53], [54], we asked questions about coding knowledge and experience at the end of study to avoid interfering with performance. In particular, women and underrepresented minorities experience the negative stereotype that they have weaker ability more strongly than others [52], [53].

4.3 Software System and Tasks

We chose *MineSweeper* as the specimen system in this experiment. *MineSweeper* is a simplistic single-player video game involving logic, mathematics and spatial layout. We use an open-source implementation of the game available on GitHub [55]. We used version 1.0 of the game, which has approximately 1.2 KLOC across 10 files. We chose Java because its wide use (reported as one of the three most popular programming languages [56]) adds to the representative power of the experiment and simplifies recruitment.

In this paper, we are interested in examining the cognitive process behind code navigation behavior in a maintenance activity. However, since we cannot directly observe how developers internally decide which code elements are relevant, and to evaluate code scanning behavior in an indicative context, we assigned participants a guiding task: a bug fixing activity that can only be completed correctly if the source code is sufficiently understood, the relevant element is found, and the defect is repaired. To present realistic bug-fixing tasks, we chose two actual bugs that already existed in the codebase. Table 3 presents information about each task. Through an informal pilot study with four other colleagues, we designed our experiment to feature two tasks of 20 minutes each (*e.g.*,, to avoid fatigue effects and meet other IRB constraints). We randomly assigned the order of the two bug-fixing tasks for each participant.

4.4 Equipment

We executed all experiments on a 64-bit Windows®10 machine with a 27″ monitor with a screen resolution of 1920x1080 pixels. We used the Tobii Pro X3-120 eye-tracker [57] which is a remote, non-intrusive device and can locate eye-gaze data in a code document at a granularity of a single line of 10pt text. The Tobii Pro X3-120 generates 120 raw samples per second. We subjected this raw data to an iTrace filter to generate fixation data. During fixations, the majority of information acquisition and processing occurs. The nature of the task, as well as the participant's characteristics, may impact the number and the duration of fixations [18]. iTrace processes gaze data offline after recording, and it supports three fixation algorithms, including basic fixations, based on a method proposed by Olsson [58], velocity-based fixations (I-VT), and dispersion-based fixations (I-DT). We use I-VT to extract fixations, which is among the most popular method currently used in the research community [59].

5 Validating Our Three-Phase Model

To validate our proposed model, we first perform a human study of two bug fixing tasks and collect viable eye move-

TABLE 3: Description of the tasks.

ID	Description	Scope of the solution
T1	When you start the *MineSweeper* program, if you click on "New Game" button, it will crash.	One class: *MineSweeperGui*. Multiple methods: *updateCheat* and *resetButtons*
T2	MineSweeper has 3 difficulty levels. The size of the board and the number of the mines are different for each level. Sometimes, we end up having a smaller number of mines in the game. For an easy game, we want to have 10 mines, but sometimes it is 9, 8, or even less.	Three classes: *MineSweeperGui*, *MineButton*, and *MineSweeperBoard*. One method: *generateMines*

ment data from 34/36 participants for task 1 and 35/36 for task 2. We mathematically partition eye behavior into phases (Section 3.3). We choose $k = 10$ as the fixation threshold for the transition to Phase 2. This value is empirically derived based on the size of the task. For our bug fixing tasks, we define semantically-relevant regions as any of 1) a function that contains the bug, 2) a function that calls the function that contains the bug, or 3) a block comment belonging to any such function. For the transition to Phase 3, we consider actions *to resolve the bug* to be any edit to the code that: 1) is not immediately deleted, and 2) is within the scope of the solution as mentioned in Table 3.

To evaluate our three-phase model, we investigate how well it captures the significant changes in participants' navigation behavior. Informally, we prefer different phases to show distinct patterns of visual attention distribution and changes. To capture the pattern of attention distribution, we statistically assess the differences between phases using Friedman test of total fixation count, average fixation duration, and saccade length throughout the task.

As shown in the first row of Table 4, we find that the number of transitions between code elements across phases differs significantly *(χ^2_r = 17.267, $p < .001$)*. In both of the tasks, the number of attention switches performed by the participants shows an increasing trend from phase 1 to phase 2 and a strong decreasing trend from phase 2 to phase 3 as shown in Figure 2.

For fixation and saccade metrics, our results also demonstrate significant changes across phases (see Table 4). This result indicate that the participants began the bug-fixing task by searching for the task-relevant source code elements. The participants explored the code more in phase 1 and phase 2 (higher number of transitions and fixations) compared to phase 3, taking less time to focus on any one location during these first two phases.

> Our quantitatively-defined three-phase model captures significant ($p < .001$) changes in participants' visual attention and navigation behavior.

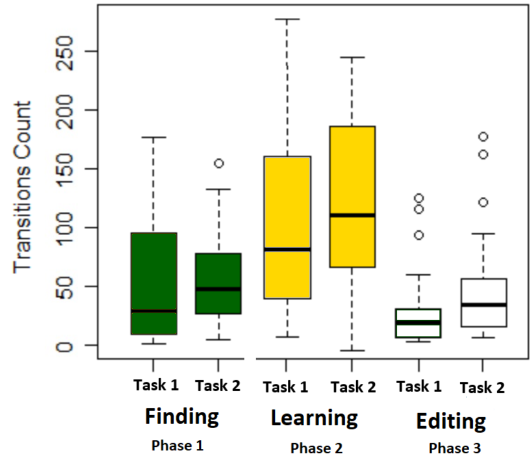

Fig. 2: Comparison of transition counts between code elements for all participants per task across phases. An increasing trend from phase 1 to phase 2 and a strong decreasing trend from phase 2 to phase 3 are observed.

6 Results and Analysis

Having validated our three-phase model, in this section, we apply our analysis methods (including phase partitioning and scanpath-based strategy characterization) to the eye-tracking and IDE interaction data collected during software maintenance tasks in which participants could freely scroll and edit files. Our study aims to provide insight into the detailed navigation behavior of developers working on realistic bug fixing tasks. Critically, while we use standalone generic eye-gaze data (*e.g.*,, attention switching, fixation and saccade merics) to validate the model, in this section we instead use other AOI-based eye-movement, thrashing, and performance metrics to analyze developers' navigation behavior.

We focus the interpretation of our results around answers to the following research questions:

RQ1. How do developers find, use, and track relevant information over time while performing realistic maintenance tasks?

RQ2. To what extent do participants' performance play a role in characterizing their navigation strategies?

RQ1 focuses on code navigation strategies. The detected AOIs identify the information that is relevant to the bug fixing tasks. In Sections 6.2 and 6.3, we measure and compare attention distribution across these AOIs while also analyzing participants' patterns of IDE interactions to quantify and capture their code navigation strategies. For **RQ2**, in Section 6.4, we perform a set of analyses to investigate the impact of participants' strategies on their outcome (measured by accuracy and thrashing rate).

6.1 Inferring Areas of Interest

Using our proposed method (see Section 3.2), we statistically determined the areas in the code that attract any significant amount of participant attention. We then characterized the code inside these areas based on the code elements they represent. We ask our participants to list and rate the relevance of the top 5 code element categories. They list relevant categories, representing the major code elements present in the *MineSweeper* project: *Method Body*, *Comment*, *Member Variable*, *Method Signature*, and *Class Signature*. We

TABLE 4: Non-parametric Friedman test statistics ($\alpha = .05$), comparing three phases. Results for each metric is at the right, with significant results (< 0.05) bolded.

	Mean (SD)				
	Finding (Phase 1)	(Learning) Phase 2	(Editing) Phase 3	χ^2_r	p
Number of Transitions	59.85 (54.20)	123.51 (82.44)	42.67 (44.70)	17.267	**<.001**
Fixation Count	224.29 (252.63)	465.64 (249.33)	158.32 (164.36)	33.091	**<.001**
Average Fixation Duration	256.27 (81.46)	274.26 (80.28)	233.44 (100.63)	8.4545	**.01**
Saccadic Length	101.77 (31.73)	90.30 (17.79)	89.24 (31.31)	12.636	**.001**

also consider the *Task Description* which is the inserted block comment in the Java files explaining the bug to be fixed.

Once the areas of interest have been mapped to code-relevant concepts, we can use visual attention on them and switches between them to quantitatively characterize participant behavior. We compared the number of transitions and the distribution of visual attention between all categories. We focus on *attention switches* (or *transitions*), which occur whenever the participant's gaze shifts from one category to a different one (*e.g.,* from *Comment* to *Class Signature*).

6.2 Understanding Phase Behavior

To determine whether a difference exists between the participants' attention distribution as they move forward in the task, going from phase 1 to phase 3, we calculate average fixation duration across AOIs and use the general align-and-rank non-parametric factorial analysis [60]. Because each code category receives different representation in the *MineSweeper* project (*e.g.,* there are more code lines than comment lines), following standard analysis practices in eye tracking [37], [38], [44], we weighted the average fixation duration in each element by the number of lines of code representing that element.

Comparing the participants' attention distribution over the three phases reveals that there is a significant interaction between these phases ($F(2, 8) = 16.88$, $p < .001$ for task 1 and $F(2, 8) = 8.4$, $p < .001$ for task 2). This indicates that participants' attention behavior changed over time as they progressed through the task.

In addition to the quantitative, statistical analysis of phase behavior, we present a qualitative explanation. Figure 3 illustrates participants' attention distribution and switching with a *radial transition graph* [61], which is a "donut" chart depicting the different code elements as segments. This visual representation is often used in eye-tracking studies; we briefly describe its main features and conclusions. Each radial border segment, identified by a unique color, corresponds to the average duration of fixations, normalized to the number of lines in the project belonging to the given category, for all participants. Edges denote transitions between code elements and are separated into outgoing and incoming transitions using two black and white anchors respectively. Edge thickness represents the total number of transitions between two code elements.

Comparing the amount of attention spent on *Method Body* and *Member Variable* shows that the importance of these AOIs changes as participants progress in the task. *Member Variable* attracts the majority of visual attention in Phase 1, but its popularity decreases in Phase 2 and Phase 3. As participants progress through the task, they increasingly pay attention to *Method Body*. Consider Phase 2 (center) and Phase 3 (right) in Figure 3. Note that the *Method Body* and *Method Signature* fixation durations are very similar for participants (orange and purple radial border segments). However, the edges (center, gray) show that Phase 3 features far fewer transitions. We also note that the edges in Phase 2 are thicker and more dense (indicating more transitions).

Fixing bugs in our Task involves predominantly changing control flow and function calls: the visualization makes this clear, showing that participants did not spend significant time focusing on *Member Variables* and *Class Signatures*. Also, as participants started to edit the code to fix the bug in Phase 3, comments receive much more visual attention (documentation likely serving to verify the behavior).

In the same vein, previous studies on developers' code navigation strategies also suggest that developers' distribution of time on various parts of a maintenance task changes throughout the task [3], [10]–[12], [14]. Uwano *et al.* [11] reported variation in code comprehension strategies, starting by a thorough scan of the complete source code (sequentially from left-to-right, top to bottom), then another set of shorts scan, and finally concentrating on certain parts of the program. Roehm *et al.* [10] reported that developers identify a starting point and filter out irrelevant code entities, and their experience profoundly modulates this. Some previous work also demonstrated distinct attention allocations within different code elements like keywords and operators [12], [14]. However, the majority of these works focused on small tasks and have been undertaken in a limited, less-realistic setup [11], [12], [14]. By capturing participants' navigation behavior on a more realistic project and task, while allowing the participants to freely interact with the IDE, we provide a detailed navigation pattern of developers, emphasizing that the relevance of code elements changes throughout the tasks.

> AOI relevance varies substantially across phases. In partial answer to RQ1, participants used different navigation and attention distribution patterns to find and track relevant information throughout bug-fixing tasks.

6.3 IDE interactions Differences

We use the collected FLUORITE data to quantify the development activities based on interaction data. Table 5 summarizes the data collected per task across phases. In both of the tasks, the frequency of *Inspection*, *Navigation*, and *Understanding* events performed by the participants show an increasing trend from phase 1 to phase 2 and a strong decreasing trend from phase 2 to phase 3. In addition, the results show an increasing number of edits from phase 1 to 2 and 3. The edits in phase 2 are information gathering edits,

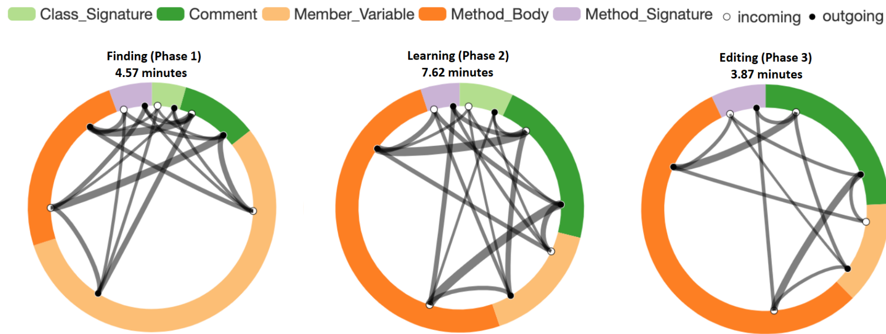

Fig. 3: Comparison of distributions of attention and transition counts between code elements for all participants per phase.

TABLE 5: Summary of IDE interaction events for Task 1 (T1) and Task 2 (T2). The number of occurrences of each event is presented.

		Interaction Events			
		Editing	**Inspection**	**Navigation**	**Understanding**
T1	**P1**	453	461	1428	2849
	P2	587	532	1565	3225
	P3	880	278	500	1990
T2	**P1**	199	390	1261	2162
	P2	663	476	1571	3646
	P3	805	216	498	1936

such as adding "print" statements. The participants enter the third phase when attempting to fix the bug. Across all tasks, *Navigation* events in phase 1 and phase 2 are three times more frequent than in phase 3.

To determine whether a difference exists between the participants' IDE interactions, across phases, we use Pearson's chi-squared test. Numerically, we find significant results for *Understanding* ($\chi_r^2 = 630.18, p < .001$), *Navigation* ($\chi_r^2 = 1081, p < .001$), *Inspection* ($\chi_r^2 = 161.35, p < .001$), and *Editing* ($\chi_r^2 = 244.51, p < .001$). Overall, our IDE interaction results further support our eye-gaze findings that developers' strategies change throughout the task.

Figure 4 gives insights on how developers behave throughout the task, across successive phases. Each color sequence is an augmented scarf-plot [62] of a participant. Each row represents a task investigation of a participant with the time axis going from left to right. Phase changes are marked over the colored sequence, while a colored vertical line represents an IDE interaction event. The participants spent the majority of their time understanding the code. For all three participants, Phase 1 and Phase 2 are composed of understanding steered by navigation. P2 occasionally does some inspections by running the code. P3 doesn't reach the Phase 3, but for P1 and P2, Phase 3 encloses editing and inspection interleaved with navigation events. Towards the end of Phase 3, P1 and P2 performs a series of inspection activities to verify their change.

> IDE interactions change significantly ($p < .001$) across phases. Furthering our investigation of RQ1, we find that participants employ various activities to gather and use pertinent information to tackle the tasks. Moreover, although the distribution varies, code understanding is still the dominant activity in each phase.

6.4 Variation in Individual Outcomes

Having observed a broad, statistically-significant similarity in participants' navigation strategies, we now consider variation in outcomes (correct bug fixes). We measure *effectiveness* (*correctness*) via manual comparison to the historical developer fix and all available tests, resulting in a categorical boolean assessment. We propose a simple quantitative *thrashing* model to analyze participants' effectiveness. We borrowed the term "thrashing" from the Systems literature. Thrashing is the result of overcommitting resources which leads to "excessive overhead and severe performance degradation or collapse" [63].

Informally, thrashing corresponds to transitioning between code elements to a disproportionately large degree. As software developers navigate throughout a complex project, they track information from multiple sources and may "swap" it in and out of working memory. Returning to a recently-viewed area to re-learn previously-acquired information has negative impacts on performance. Formally, we define the thrashing rate as the number of AOI switches per minute. We use this quantifiable thrashing metric as a lens to differentiate successful participants from the others.

We applied binomial regression and found *evidence that thrashing is a significant predictor of effectiveness* ($F = 4.7671, p = .003$). The correlation is negative: more frequent AOI switching (thrashing) *does not* help participants be more effective in our tasks. Successful participants, who fixed both issues within the allotted 20 minutes, displayed 35% less thrashing ($M = 16.02, SD = 10.19$) than the others ($M = 24.31, SD = 35.09$). Low-thrashing participants were also efficient, spending 28% less time, in a statistically significant manner ($W = 2991, p = .03$), completing phases.

We use the general align-and-rank non-parametric factorial analysis [60] to compare the successful participants with the others. The result reveals that there is a significant interaction between accuracy and attention distribution ($F(4, 8) = 4.85, p < .001$). Successful participants put on average 8% more visual effort on *Method Body*.

Figure 5 depicts the IDE interaction behavior of successful and unsuccessful participants. While the number of events may vary (e.g., between Successful and Unsuccessful participants in Phase 1), there is no statistically significant difference between the participant event distributions for these activities. The role of understanding is still dominant in all phases for all participants (around 55%). The distribution of editing, navigation, and inspection activities is

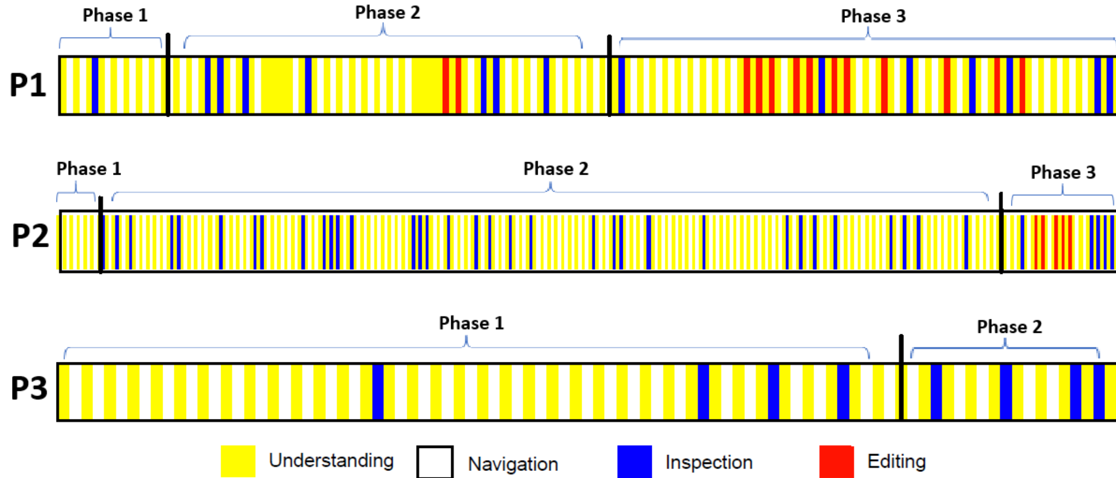

Fig. 4: Scarf-plots (history of all IDE interactions) of three example participants working on Task 1. Code understanding is a dominant activity, interleaved mainly by navigation.

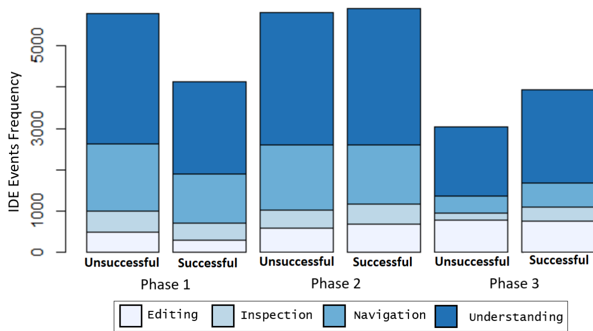

Fig. 5: Comparison of IDE interaction pattern between successful and unsuccessful participants per phase.

around 12%, 7.5%, and 23.5%, respectively. This is important because it suggests that certain aspects of the IDE activity of participants is independent of successful outcomes, and thus that tools and UI improvements can be targeted to a general distribution without requiring *a priori* knowledge of whether a software engineer is likely to be successful at a task or not. It is instead the navigation strategy (*e.g.*, frequent AOI switching) that characterizes better or worse performance.

We also evaluated the impact of experience. We examine the number of years of programming experience in addition to the level of familiarity with programming, object-oriented design, and IDEs. In our analysis, none of these values significantly interact with phases to have an effect on navigation strategies, efficiency, time, or thrashing frequency. We note that our participants are students, thus their programming experience is rather homogeneous.

Our results are in broad agreement with the work of Robillard *et al.* [30] reporting that returning to the same method repeatedly is a sign of poor performance. Research on end-user programming also report that tinkering, turning a feature "on" immediately followed by turning the feature "off", negatively impact the outcome for men [64], [65].

However, more investigation is required to explain why excessive exploration, in this context, was not effective. As with previous studies, we may be able to shed further light on thrashing behavior by considering various types of code exploration and investigating their impacts individually.

> We find that successful developers' navigation strategies are quantifiably different: more frequent AOI switching, or "thashing", is associated with worse performance ($p = .003$).

7 Threats to Validity

Several factors potentially affect the validity of our study. We recorded eye-gaze data with their timestamps and devised a model that characterizes the navigation behavior of developers. A threat to validity for our results is the accuracy of this model, which may not precisely capture participants' intent and actions. There is a possibility of participants behavior not matching our three phases. To mitigate this risk, we carefully validate our proposed model by performing a controlled experiment with 36 participants and present a quantitative evaluation of the results. Also, we corroborate our results with a detailed analysis of the IDE interactions. We record more editing and less understanding for phase 3 compared to other phases, which implies that our model captures these significant changes in developers' navigation behavior.

Our analysis is based on automatically inferring AOIs. A possible threat is that our inference approach could be inaccurate. To partially mitigate this threat, we manually analyzed a sample of the participants' data and the extracted AOIs for validity and determined them to be reasonable, acknowledging that an assessment by the authors does not give as much confidence as a full independent assessment or replication.

The iTrace plugin collected eye gaze data only within Eclipse code pane (window). Thus, we do not have any records of developers' eye movements when they use other parts of the IDE. We mitigate this risk by recording and

analyzing the IDE interaction data using FLUORITE. Moreover, since we look for the code navigation strategies, this limitation does not directly impact our analysis. Also, in this work, to better control experimental conditions, participants worked on two bugs in a small project and they were not allowed to access other software (*e.g.,* browsing the internet). Our results may not generalize to real-world software engineering tasks. Further studies investigating these effects but allowing multiple systems and a more indicative use of external tools would have greater ecological validity.

We alleviate hypothesis guessing and apprehension by not informing the participants about the precise goals of the study. However, we clearly explained the process of the study, the number of sessions, the type of tasks, and how an eye tracker works before running the experiment.

We mitigate instrument bias by using a video-based eye tracker that does not involve any heavy goggles and allows participants to move their head without changing the calibration of the camera. We calibrated the camera at the beginning of the study and between tasks. We also choose well-documented eye-tracking metrics [44] and standard statistical methods for data analysis.

The Hawthorne effect is the alteration of behavior by participants due to their awareness of being watched and evaluated. To overcome this risk, we explained to the participants that the eye tracker does not record any video or image beyond the eye itself. In addition, our experimenter sat inconspicuously away from the participants.

To reduce the stereotype threat [53], [54] and avoid interfering with participants' performance, we asked self-assessment questions about their coding knowledge and experience at the end of the study. Researchers may unintentionally influence participants to achieve specific goals or to perform in a particular manner. To mitigate this bias, we minimized the interaction between our team and participants while de-identifying the data so that it cannot be used to learn the identity of individual participants. Also, our research team contained both men and women; we conducted a set of pilot studies to help identify biased procedures or results.

All of our participants are students, and 50% of them are graduate students with good programming knowledge and expertise. As stated by Kitchenham *et al.* [66] "using students as participants is not a major issue as long as you are interested in evaluating the use of a technique by novice or non-expert software engineers. Students are the next generation of software professionals so, are relatively close to the population of interest".

We must be careful because only 36 developers participated in our study. Although this number is much higher than that of any previously reported studies [38], [45], [67], we cannot consider the population large enough to generalize the results.

Finally, we used only one system, so its quality and complexity might influence the study. We mitigate this risk by choosing an open-source project, written in a popular programming language. *MineSweeper* is not large by general software engineering standards, but is quite large by eye-tracking standards (where all but three previously-published papers focused on a single screen).

8 Conclusion

In this paper, we presented an eye-tracking experiment design for more indicative scenarios within an IDE (admitting both editing and scrolling); a statistically-significant three-phase model of code navigation behavior; and an investigation of developers interaction with IDE. Utilizing this experiment design, we investigate the variances of navigation patterns with respect to developers' individual differences using various eye-movement statistics. We selected a set of two realistic bug fixing tasks and recruited 36 participants. We consider this to be a reasonable number of tasks to validate our novel approach without causing fatigue. However, We must be careful when drawing conclusions as the reported results are based on the limited sample size in terms of systems, tasks, and the number of participants.

Our analysis shows that, in a statistically significant manner, developers spend more time searching various code elements during the beginning of a bug-fixing task, and the relevance of these code elements changes over time. Also, our multi-dimensional eye movement analysis shows that, although individuals follow different navigation strategies, participants in our study tended to broadly focus on the same set of code elements per phase. Moreover, our proposed simple thrashing model suggests that greater amounts of thrashing lead to less effective results.

We discuss how our results might (speculatively) be applied to the domains of bug fixing, code summarization, and education.

Enhancing the cost and time estimation of bug fixing: by capturing the amount of effort (visual attention), developers devote to various source code elements in the granularity of a file (*e.g.,* a class) or a set of files (*e.g.,* Java packages), we could measure the associated difficulty of fixing a type of bug. A more cognitively-grounded assessment or prediction of task difficulty could provide a better estimation of the maintenance cost and time required.

In addition, a detailed eye-movement pattern is a cost-effective way of transferring an expert's knowledge about the code to the rest of the team. In such a setting, only one or two experts need thoroughly read the code while an eye tracker is capturing their eye-movements, reducing data-gathering costs. Other developers may consult the recorded eye-movement pattern while debugging the code to find the pertinent areas faster.

Improving code summarization by locating important code regions: a critical problem with manual and automated code summarization approaches is that it is difficult to identify areas in which humans will be interested (*i.e.,*areas that should be retained when abstracting to summarize) [14], [23]. Our results demonstrate that the importance of source code elements changes over the course of the task. Since we capture the amount of attention developers devote to various code elements, we could use this data to create a more accurate models of code elements' relevance to fixing a given problem and the difficulty of fixing it.

Addressing education via navigation strategies: in this work, we demonstrate that distribution of visual attention are quantifiably different from individuals with a weaker performance. Also, we have observed that thrashing has adverse effects on effectiveness. Although code is required

to understand it, over-exploration is counter-productive. If we can identify fruitful cost-benefit tradeoffs between code exploration and thrashing, we could train developers to follow such best practices.

Although our work investigated developers' strategies in bug-fixing tasks, our experiment design may apply in the future to investigate how developers use and track relevant information over time while performing other software maintenance tasks — such as writing test cases, conducting code reviews, or implementing a new feature. Future work may also include using attitudinal data to investigate whether or not confidence after successfully or unsuccessfully resolving a first task impacts results in solving a similar task. Finally, we propose a further investigation into the effects of thrashing by exploring the types of edits developers made while performing the bug-fixing tasks.

Acknowledgment

We are very grateful to all study participants. The authors would like to thank the anonymous reviewers for their insightful comments and feedback.

References

[1] J. De Vries, C. Burki, and B. De Vries, "How to save on software maintenance costs," *Omnext, Nov*, 2014.

[2] R. S. Pressman, *Software engineering: a practitioner's approach*. Palgrave Macmillan, 2005.

[3] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung, "An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks," *IEEE Transactions on Software Engineering*, vol. 32, no. 12, pp. 971–987, Dec. 2006.

[4] K. Kevic, B. M. Walters, T. R. Shaffer, B. Sharif, D. C. Shepherd, and T. Fritz, "Tracing software developers' eyes and interactions for change tasks," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015, 2015, pp. 202–213.

[5] T. Fritz, D. C. Shepherd, K. Kevic, W. Snipes, and C. Bräunlich, "Developers' code context models for change tasks," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 7–18.

[6] M.-A. Storey, "Theories, methods and tools in program comprehension: Past, present and future," in *Proceedings of the 13th International Workshop on Program Comprehension*, ser. IWPC '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 181–191. [Online]. Available: http://dx.doi.org/10.1109/WPC.2005.38

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

[8] K. Kevic, B. Walters, T. Shaffer, B. Sharif, D. Shepherd, and T. Fritz, "Eye gaze and interaction contexts for change tasks observations and potential," *Journal of System Software*, vol. 128, no. C, pp. 252–266, Jun. 2017.

[9] T. D. LaToza, G. Venolia, and R. DeLine, "Maintaining mental models: a study of developer work habits," in *Proceedings of the 28th international conference on Software engineering*. ACM, 2006, pp. 492–501.

[10] T. Roehm, R. Tiarks, R. Koschke, and W. Maalej, "How do professional developers comprehend software?" in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12, 2012, pp. 255–265.

[11] H. Uwano, M. Nakamura, A. Monden, and K.-i. Matsumoto, "Analyzing individual performance of source code review using reviewers' eye movement," in *Proceedings of the 2006 symposium on Eye tracking research & applications*, ser. ETRA '06. ACM, 2006, pp. 133–140.

[12] N. Ali, Z. Sharafi, Y.-G. Guéhéneuc, and G. Antoniol, "An empirical study on the importance of source code entities for requirements traceability," *Empirical Software Engineering*, vol. 20, no. 2, pp. 442–478, Apr 2015. [Online]. Available: https://doi.org/10.1007/s10664-014-9315-y

[13] P. Rodeghero, C. McMillan, P. W. McBurney, N. Bosch, and S. D'Mello, "Improving automated source code summarization via an eye-tracking study of programmers," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 390–401.

[14] T. Busjahn, C. Schulte, and A. Busjahn, "Analysis of code reading to gain more insight in program comprehension," in *Proceedings of the 11th Koli Calling International Conference on Computing Education Research*, ser. Koli Calling '11, 2011, pp. 1–9.

[15] N. J. Abid, B. Sharif, N. Dragan, H. Alrasheed, and J. I. Maletic, "Developer reading behavior while summarizing java methods: Size and context matters," in *Proceedings of the 41st International Conference on Software Engineering*. IEEE Press, 2019, pp. 384–395.

[16] C. S. Peterson, N. J. Abid, C. A. Bryant, J. I. Maletic, and B. Sharif, "Factors influencing dwell time during source code reading: A large-scale replication experiment," in *Proceedings of the 11th ACM Symposium on Eye Tracking Research & Applications*, ser. ETRA '19, 2019, pp. 38:1–38:4.

[17] A. Duchowski, *Eye tracking methodology: Theory and practice*. Springer-Verlag New York Inc, 2007.

[18] M. A. Just and P. A. Carpenter, "Eye fixations and cognitive processes," *Cognitive Psychology*, vol. 8, no. 4, pp. 441–480, 1976.

[19] J. Sillito, G. C. Murphy, and K. De Volder, "Asking and answering questions during a programming change task," *IEEE Transactions on Software Engineering*, vol. 34, no. 4, pp. 434–451, 2008.

[20] M. E. Crosby and J. Stelovsky, "How do we read algorithms? a case study," *Computer*, vol. 23, no. 1, pp. 24–35, Jan. 1990.

[21] C. Aschwanden and M. Crosby, "Code scanning patterns in program comprehension," in *Proceedings of the 39th Hawaii International Conference on System Sciences*, ser. HICSS '06, 2006.

[22] B. Sharif, M. Falcone, and J. I. Maletic, "An eye-tracking study on the role of scan time in finding source code defects," in *Proceedings of the 2012 Symposium on Eye Tracking Research & Applications*, ser. ETRA'12. New York, NY, USA: ACM, 2012, pp. 381–384.

[23] P. Rodeghero and C. McMillan, "An empirical study on the patterns of eye movement during summarization tasks," in *Proceeding of 2015 International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 2015, pp. 1–10.

[24] M. Kersten and G. C. Murphy, "Using task context to improve programmer productivity," in *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. NY, NY, USA: Association for Computing Machinery, 2006, p. 1–11.

[25] C. Parnin and C. Gorg, "Building usage contexts during program comprehension," in *Proceedings of the 14th IEEE International Conference on Program Comprehension*, ser. ICPC '06. USA: IEEE Computer Society, 2006, p. 13–22.

[26] D. Piorkowski, S. D. Fleming, C. Scaffidi, L. John, C. Bogart, B. E. John, M. Burnett, and R. Bellamy, "Modeling programmer navigation: A head-to-head empirical evaluation of predictive models," in *2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 2011, pp. 109–116.

[27] https://eclipse.org/mylyn/, 2017, online; Accessed 28-03-2020.

[28] Y. Yoon and B. A. Myers, "Capturing and analyzing low-level events from the code editor," in *Proceedings of the 3rd ACM SIGPLAN workshop on Evaluation and usability of programming languages and tools*. ACM, 2011, pp. 25–30.

[29] Website, "The eclipse foundation. usage data collector results," 2009, online; Accessed 09-08-2020. [Online]. Available: http://www.eclipse.org/org/usagedata/

[30] M. P. Robillard, W. Coelho, and G. C. Murphy, "How effective developers investigate source code: an exploratory study," *IEEE Transactions on Software Engineering*, vol. 30, no. 12, pp. 889–903, Dec 2004.

[31] M. Kersten and G. C. Murphy, "Mylar: a degree-of-interest model for IDEs," in *Proceedings of the 4th international conference on Aspect-oriented software development*, 2005, pp. 159–168.

[32] J. Singer, R. Elves, and M.-A. Storey, "Navtracks: Supporting navigation in software maintenance," in *21st IEEE International Conference on Software Maintenance (ICSM'05)*. IEEE, 2005, pp. 325–334.

[33] E. Murphy-Hill, C. Parnin, and A. P. Black, "How we refactor, and how we know it," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 5–18, 2012.

[34] T. M. Ahmed, W. Shang, and A. E. Hassan, "An empirical study of the copy and paste behavior during development," in *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, 2015, pp. 99–110.

[35] B. Sharif, C. Peterson, D. Guarnera, C. Bryant, Z. Buchanan, V. Zyrianov, and J. Maletic, "Practical eye tracking with itrace," in *2019 IEEE/ACM 6th International Workshop on Eye Movements in Programming (EMIP)*. IEEE, 2019, pp. 41–42.

[36] D. T. Guarnera, C. A. Bryant, A. Mishra, J. I. Maletic, and B. Sharif, "itrace: Eye tracking infrastructure for development environments," in *Proceedings of the 2018 ACM Symposium on Eye Tracking Research & Applications*, 2018, pp. 1–3.

[37] J. H. Goldberg and J. I. Helfman, "Comparing information graphics: A critical look at eye tracking," in *Proceedings of the 3rd BEyond Time and Errors: Novel evaLuation Methods for Information Visualization Workshop*, ser. BELIV '10. New York, NY, USA: ACM, 2010, pp. 71–78. [Online]. Available: http://doi.acm.org/10.1145/2110192.2110203

[38] Z. Sharafi, Z. Soh, and Y.-G. Guéhéneuc, "A systematic literature review on the usage of eye-tracking in software engineering," *Information and Software Technology*, vol. 67, pp. 79–107, 2015.

[39] R. Caldara and S. Miellet, "imap: a novel method for statistical fixation mapping of eye movement data," *Behavior research methods*, vol. 43, no. 3, pp. 864–878, 2011.

[40] J. Lao, S. Miellet, C. Pernet, N. Sokhn, and R. Caldara, "imap4: An open source toolbox for the statistical fixation mapping of eye movement data with linear mixed modeling," *Behavior research methods*, vol. 49, no. 2, pp. 559–575, 2017.

[41] A. Poole and L. J. Ball, "Eye tracking in human-computer interaction and usability research: Current status and future," in *Prospects", Chapter in C. Ghaoui (Ed.): Encyclopedia of Human-Computer Interaction. Pennsylvania: Idea Group, Inc*, 2005.

[42] R. Matos, "Designing eye tracking experiments to measure human behavior," *Merriënboer, JJG van, and Sweller, J.(2005). Cognitive load theory and complex learning: Recent developments and future directions. Educational Psychology Review*, vol. 17, 2010.

[43] B. Farnsworth, "10 most used eye tracking metrics and terms," 2018, online; Accessed 09-08-2020. [Online]. Available: https://imotions.com/blog/7-terms-metrics-eye-tracking/

[44] Z. Sharafi, T. Shaffer, B. Sharif, and Y.-G. Guéhéneuc, "Eye-tracking metrics in software engineering," in *Proceeding of 2015 Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2015, pp. 96–103.

[45] U. Obaidellah, M. Al Haek, and P. C.-H. Cheng, "A survey on the usage of eye-tracking in computer programming," *ACM Comput. Surv.*, vol. 51, no. 1, pp. 5:1–5:58, Jan. 2018. [Online]. Available: http://doi.acm.org/10.1145/3145904

[46] H. Jarodzka, K. Holmqvist, and M. Nyström, "A vector-based, multidimensional scanpath similarity measure," in *Proceedings of the 2010 Symposium on Eye-Tracking Research & Applications*, ser. ETRA '10, 2010, pp. 211–218.

[47] F. Cristino, S. Mathot, J. Theeuwes, and I. D. Gilchrist, "Scanmatch: A novel method for comparing fixation sequences." *Behaviour Research Method*, vol. 42, pp. 692–700, 2010.

[48] R. Dewhurst, M. Nyström, H. Jarodzka, T. Foulsham, R. Johansson, and K. Holmqvist, "It depends on how you look at it: Scanpath comparison in multiple dimensions with multimatch, a vector-based approach," *Behavior research methods*, vol. 44, no. 4, pp. 1079–1100, 2012.

[49] J. H. Goldberg and X. P. Kotval, "Computer interface evaluation using eye movements: methods and constructs," in *International Journal of Industrial Ergonomics*, vol. 24, no. 6. Elsevier, 1999, pp. 631–645.

[50] R. Minelli, A. Mocci, M. Lanza, and T. Kobayashi, "Quantifying program comprehension with interaction data," in *2014 14th International Conference on Quality Software*. IEEE, 2014, pp. 276–285.

[51] J. Siegmund, C. Kästner, J. Liebig, S. Apel, and S. Hanenberg, "Measuring and modeling programming experience," *Empirical Software Engineering*, vol. 19, no. 5, pp. 1299–1334, 2014.

[52] C. M. Steele and J. Aronson, "Stereotype threat and the intellectual test performance of african americans." *Journal of personality and social psychology*, vol. 69, no. 5, p. 797, 1995.

[53] S. J. Spencer, C. M. Steele, and D. M. Quinn, "Stereotype threat and women's math performance," *Journal of experimental social psychology*, vol. 35, no. 1, pp. 4–28, 1999.

[54] J. R. Shapiro and S. L. Neuberg, "From stereotype threat to stereotype threats: Implications of a multi-threat framework for causes, moderators, mediators, consequences, and interventions," *Personality and Social Psychology Review*, vol. 11, no. 2, pp. 107–130, 2007.

[55] https://electrek.co/2018/12/01/tesla-pulled-over-cops-sleeping-drunk autopilot/, 2017, online; Accessed 23-08-2019.

[56] https://www.tiobe.com/tiobe index/, 2019, online; Accessed 23-08-2019.

[57] https://www.tobiipro.com/, 2001, online; Accessed 23-08-2019.

[58] P. Olsson, "Real-time and offline filters for eye tracking," Master's thesis, KTH, School of Electrical Engineering, 4 2007.

[59] D. D. Salvucci and J. H. Goldberg, "Identifying fixations and saccades in eye-tracking protocols," in *Proceedings of the 2000 Symposium on Eye Tracking Research Applications*, ser. ETRA '00. New York, NY, USA: Association for Computing Machinery, 2000, p. 71–78. [Online]. Available: https://doi-org.proxy.lib.umich.edu/10.1145/355017.355028

[60] J. O. Wobbrock, L. Findlater, D. Gergle, and J. J. Higgins, "The aligned rank transform for nonparametric factorial analyses using only anova procedures," in *Proceedings of the SIGCHI conference on human factors in computing systems*. ACM, 2011, pp. 143–146.

[61] T. Blascheck, M. Schweizer, F. Beck, and T. Ertl, "Visual comparison of eye movement patterns," in *Computer Graphic Forum*, vol. 36, no. 3, Jun. 2017, pp. 87–97.

[62] C.-K. Yang and C. Wacharamanotham, "Alpscarf: Augmenting scarf plots for exploring temporal gaze patterns," in *Extended Abstracts of the 2018 CHI Conference on Human Factors in Computing Systems*, ser. CHI EA '18. New York, NY, USA: ACM, 2018, pp. LBW503:1–LBW503:6. [Online]. Available: http://doi.acm.org/10.1145/3170427.3188490

[63] P. J. Denning, "Thrashing: Its causes and prevention," in *Proceedings of the December 9-11, 1968, fall joint computer conference, part I*. ACM, 1968, pp. 915–922.

[64] L. Beckwith, C. Kissinger, M. Burnett, S. Wiedenbeck, J. Lawrance, A. Blackwell, and C. Cook, "Tinkering and gender in end-user programmers' debugging," in *Proceedings of the 2006 SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '06. New York, NY, USA: ACM, 2006, pp. 231–240. [Online]. Available: http://doi.acm.org/10.1145/1124772.1124808

[65] L. Beckwith, M. Burnett, S. Wiedenbeck, C. Cook, S. Sorte, and M. Hastings, "Effectiveness of end-user debugging software features: Are there gender issues?" in *Proceedings of the 2005 SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '05. New York, NY, USA: ACM, 2005, pp. 869–878. [Online]. Available: http://doi.acm.org/10.1145/1054972.1055094

[66] B. A. Kitchenham, S. L. Pfleeger, L. M. Pickard, P. W. Jones, D. C. Hoaglin, K. E. Emam, and J. Rosenberg, "Preliminary guidelines for empirical research in software engineering," *IEEE Transactions on Software Engineering*, vol. 28, no. 8, pp. 721–734, Aug. 2002. [Online]. Available: https://doi.org/10.1109/TSE.2002.1027796

[67] J. Siegmund and J. Schumann, "Confounding parameters on program comprehension: a literature survey," *Empirical Software Engineering*, vol. 20, no. 4, pp. 1159–1192, Aug 2015. [Online]. Available: https://doi.org/10.1007/s10664-014-9318-8

Zohreh Sharafi is a senior research fellow at the University of Michigan. She received the M.S. degree in software engineering from Concordia University and the Ph.D. degree from Polytechnique Montréal. Her main research interests include program comprehension, human aspects of software development, and eye-tracking research related to software engineering. The aim is to improve current tools and processes to be more inclusive and cater to a wide range of developers.

Ian Bertram is a third-year undergraduate majoring in Computer Science at the University of Michigan. In addition to research and course work, Ian spends much of his time developing algorithms and applications for the University of Michigan's Solar Car Team.

Michael Flanagan is a third year undergraduate majoring in Computer Science at the University of Michigan. In addition to research and course work, Michael spends his time working on developing and maintaining various technical projects for his professional technology fraternity Kappa Theta Pi.

Westley Weimer received a BA degree in computer science and mathematics from Cornell University and MS and PhD degrees from the University of California, Berkeley. He is currently a full professor at the University of Michigan. His main research interests include static and dynamic analyses to improve software quality and fix defects, as well as medical imaging and human studies of programming.