Using Developer Eye Movements to Externalize the Mental Model Used in Code Summarization Tasks

Nahla J. Abid
Taibah University
Department of Computer Science
Madinah, Kingdom of Saudi Arabia
nabd@taibahu.edu.sa

Jonathan I. Maletic
Kent State University
Department of Computer Science
Kent. Ohio, USA 44240
jmaletic@kent.edu

Bonita Sharif
University of Nebraska - Lincoln
Dept. of Computer Science and Eng.
Lincoln, Nebraska, USA 68588
bsharif@unl.edu

ABSTRACT

Eye movements of developers are used to speculate the mental cognition model (i.e., bottom-up or top-down) applied during program comprehension tasks. The cognition models examine how programmers understand source code by describing the temporary information structures in the programmer's short term memory. The two types of models that we are interested in are top-down and bottom-up. The top-down model is normally applied as-needed (i.e., the domain of the system is familiar). The bottom-up model is typically applied when a developer is not familiar with the domain or the source code. An eye-tracking study of 18 developers reading and summarizing Java methods is used as our dataset for analyzing the mental cognition model. The developers provide a written summary for methods assigned to them. In total, 63 methods are used from five different systems. The results indicate that on average, experts and novices read the methods more closely (using the bottom-up mental model) than bouncing around (using top-down). However, on average novices spend longer gaze time performing bottom-up (66s.) compared to experts (43s.)

CCS CONCEPTS

• **Human-centered computing** → **Empirical studies in HCI**;
• **Software and its engineering** → *General programming languages*;

KEYWORDS

program comprehension, eye tracking study, code summarization

ACM Reference Format:
Nahla J. Abid, Jonathan I. Maletic, and Bonita Sharif. 2019. Using Developer Eye Movements to Externalize the Mental Model Used in Code Summarization Tasks. In *2019 Symposium on Eye Tracking Research and Applications (ETRA '19), June 25–28, 2019, Denver, CO, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3314111.3319834

1 INTRODUCTION

Program comprehension is an essential task during software maintenance for enabling successful evolution of programs [von Mayrhauser and Vans 1995]. For several years, researchers have attempted to understand how programmers comprehend programs during maintenance [Brooks 1983; Burkhardt et al. 1997; Letovsky 1987; Pennington 1987; Shneiderman and Mayer 1979; Soloway and Ehrlich 1984; Storey 2005; von Mayrhauser and Vans 1995, 1997]. The program comprehension process uses existing knowledge to obtain new knowledge and build a mental model of the program that is under consideration [von Mayrhauser and Vans 1995]. A mental model defines a developer's mental or conceptual representation of the source code to being understood [Storey 2005; von Mayrhauser and Vans 1995]. Code cognition models examine how programmers understand code by describing the cognitive processes and temporary information structures in the programmer's mind [von Mayrhauser and Vans 1995].

This work studies the activities and behaviors that a software developer performs to support the process of source code comprehension. We accomplish this via the analysis of an eye-tracking study data set [Abid et al. 2019] done on 18 developers who were tasked with summarizing Java methods. We analyze the developers' behavior from two directions. *First*, we introduce a mechanism to predict the mental model [von Mayrhauser and Vans 1995] developers apply during the comprehension process. We distinguish between two models: top-down and bottom-up. Top-down is typically applied when a developer has a hypothesis about the code under consideration and wants to confirm it by examining a set of locations. Alternatively, bottom-up is used when a developer has no understanding of the code, nor any hypothesis about the code, and must closely examine (i.e., read) the code.

Second, we adopt an approach used by [Rodeghero and McMillan 2015] to analyze eye-movement patterns. These patterns consist of the order of words or sections that developers follow when they read. Examples of reading patterns are reading top-to-bottom or bottom-to-top [Rodeghero and McMillan 2015]. They concluded that unlike natural English text (that is typically read top-to-bottom in Western cultures), developers tend to read source code in a non-linear manner and often jump around in the file. A similar conclusion was reached independently by Busjahn et al. [Busjahn et al. 2015]. Here, we examine the same three types of reading patterns namely: top-to-bottom vs. bottom-to-top, skimming vs. thorough, and disorderly vs sectionally. The contributions of this work are as follows.

- Predicting the mental model of developers from their gaze data during a program comprehension task. We distinguish between two types of mental models from existing program comprehension literature: top-down and bottom-up.
- Identifying three phases (preparing, reading, and writing) in the

ETRA '19, June 25–28, 2019, Denver, CO, USA

ACM ISBN 978-1-4503-6709-7/19/06…$15.00
https://doi.org/10.1145/3314111.3319834

eye tracking data while developers comprehend source code for the purpose of summarizing methods.
- Qualitatively analyzing three reading patterns (top-to-bottom versus bottom-to-top, skimming versus thorough, and disorderly versus sectionally) found in source code in experts and novices.

2 RESEARCH QUESTIONS

In this paper, we address the following research questions.

- RQ 1. Can eye-tracking data be used to predict a participant's conceptual model [top-down vs. bottom-up]?
- RQ 2. To what degree do [experts / novices] prefer to read from [top-to-bottom vs. bottom to top], [skim source code vs. read thoroughly], and [read sectionally vs. disorderly]?

The results from RQ 1 would help in externalizing the mental model used (which is hard to do verbally) by using eye tracking data. We would not have to rely on asking the developer to think aloud their thought processes which has its own set of drawbacks such as misreporting or forgetting to report entirely. Such an approach would be quite useful during interviewing for a programming position where the interviewer is interested in understanding the program comprehension skills of their potential employee including how they traverse through the problem solving process. It could also help an educator learn how novices comprehend concepts while they learn programming and advanced data structures. The results from RQ 2 could give us insight into how experts and novices differ in their reading strategies. Such insights could help in devising evidence-based programming languages and tools to help developers become more productive in their day to day tasks.

3 RELATED WORK

The first part of the related work deals with the main domain of program comprehension which is a sub-field of software engineering. This is followed by the literature on the use of eye tracking by program comprehension researchers.

3.1 Program Comprehension

One of the earliest cognition models is the Shneiderman and Mayer model that differentiates between two types of memories. First, is short-term memory that records the internal semantic representation of the program via chunking mechanism. The second type of memory is the long-term memory that contains the knowledge base with semantic and syntactic knowledge. The knowledge base in long-term memory helps build the short-term memory during the comprehension process [Shneiderman and Mayer 1979].

Brooks hypothesizes that program comprehension is reconstruction knowledge about the domain of the program [Brooks 1983]. In such an approach, programmers understand a completed program in a top-down manner. The process starts with a hypothesis about the general goal of the program. This initial hypothesis is then confirmed or refined by forming additional hypotheses. On the other hand, Soloway and Ehrlich observe that top-down understanding is used when a developer is familiar with code [Soloway and Ehrlich 1984]. They observe that expert programmers use beacons (familiar features such as three lines used in a swap) and rules of programming discourse to decompose goals and plans into lower-level plans. They note that de-localized plans complicate program comprehension. Data flow - transformations applied to data objects Computer Program Text Abstractions Goal hierarchy - functional achievements of program Control flow - sequences of program actions and the passage of control between them The Pennington model observes that programmers first develop a control-flow abstraction of the program which captures the sequence of operations in the program [Pennington 1987]. This model is referred to as the program model. Control flow is the sequences of actions performed by the program. Then, the situation model (bottom-up), uses knowledge about data-flow abstractions and functional abstractions (i.e.,transformations applied to data objects). The situation model is complete once the program goal is reached.

Letovsky proposes the knowledge-based model that depends on pre-programming expertise, problem domain, knowledge, rules of discourse plans and goals [Letovsky 1987]. Finally, the latest cognition is the integrated metamodel that is found by [von Mayrhauser and Vans 1995]. The metamodel is built based on the previous models. The model consists of four major components. The first three components deal with creating mental representations at various levels of abstraction and the fourth component describes the knowledge base needed to create the mental representations.

The top-down model is usually applied as-needed (i.e., the programming language or code is familiar). It integrates domain knowledge as a starting point for formulating hypotheses. The program model may be invoked when the code and application is completely unfamiliar. This is done in examining the control-flow of a program. The situation model describes data-flow and functional abstractions in the program. It may be developed after a partial program model is formed using systematic or opportunistic strategies. The knowledge base consists of information needed to build these three cognitive models. It represents the programmer's current knowledge and is used to store new knowledge [von Mayrhauser and Vans 1995].

A number of user studies are conducted to examine what types of mental models are formed by both novice and expert programmers during comprehension [Burkhardt et al. 1997; von Mayrhauser and Vans 1997]. Von Mayrhauser and Vans perform a user study included four developers doing maintenance tasks (e.g., fixing a bug) [von Mayrhauser and Vans 1997]. They execute a think-aloud experiment while audio and video taping developers. They conclude that a developer switches between top-down, bottom-up, and program model in order to build the mental model. With little experience on the domain, the comprehension is done at a lower level by reading the source code closely (bottom-up) to build a higher-level abstraction. Furthermore, programmers with domain expertise look for specific information such as statement execution order and definition as well as variable use.

Burkhardt et al. conduct a user study to evaluate a cognitive model of object-oriented (OO) program understanding [Burkhardt et al. 1997]. In particular, they want to draw a distinction between the program model and the situation model in OO program understanding. 30 experts and 21 novices are asked to perform one of two tasks: designing a variation of the library problem or writing comments for existing code. The mental model is judged based on a post-questionnaire. Burkhardt et al. conclude that the situation model is more fully developed than the program model, even in an early phase of comprehension [Burkhardt et al. 1997]. This contrasts

with the results of Pennington for procedural programmers [Pennington 1987]. She showed that the program model developed more fully in the early stages of comprehension, whereas the situation model emerged later. They explain the difference as the OO paradigm, with its emphasis on objects and relationships of objects, may facilitate the construction of a situation model earlier in program comprehension. In this paper, we analyze the comprehension process one step further by studying if the reading behavior captured by eye-tracking data can predict the mental model of developer during a comprehension task (i.e., summarizing a method).

3.2 Eye Tracking in Code Comprehension

Eye-tracking technology has been used to study how developers read [Crosby and Stelovsky 1990; Rodeghero et al. 2014] and review [Sharif et al. 2012; Uwano et al. 2006] source code. Crosby et al. [Crosby and Stelovsky 1990] concluded that subjects needed numerous fixations in most areas of a binary algorithm than did subjects in studies using pseudocode-like text [Crosby and Stelovsky 1990]. Uwano et al. found that the longer a reviewer read through the code, the more efficiently the reviewer could find the defect [Uwano et al. 2006]. This correlation was later confirmed by Sharif et al. [Sharif et al. 2012]. Moreover, experts tend to focus on beacons more, while novices may read lines more broadly [Sharif et al. 2012]. This result was later statistically reproduced by Busjahn et al. [Busjahn et al. 2015].

Bednarik and Tukiainen observed that low-experience programmers repeatedly fixated on the same sections, while experienced programmers target the output of the code, such as evaluation expressions [Bednarik and Tukiainen 2006, 2008]. Kevic et al. conducted a study on three bug fix tasks. They found that developers focus on small parts of methods that are often related to data flow [Kevic et al. 2015]. Eye-tracking studies have also been done to gauge the effectiveness of identifier style [Binkley et al. 2013; Sharif and Maletic 2010b] and class diagram layout on comprehension [Sharif 2011; Sharif and Maletic 2010a].

Rodeghero et al. conducted a study of programmers during summarization [Rodeghero et al. 2014]. They concluded that programmers consider method signatures as the most important section of code followed by invocation terms then control flow terms. Term-based summarization was enhanced when the locations of terms is considered in and information retrieval technique namely, VSM used to generate keyterms for a summary. Rodeghero et al. also analyzed the same data set to study the eye movement patterns (the order of words or sections that people follow when they read). In particular, they looked at three types of reading patterns namely: top-to-bottom vs bottom-to-top, skimming vs thorough, and disorderly vs sectionally. They found that programmers prefer to read code by jumping between sections rather than reading code one section at a time [Rodeghero and McMillan 2015]. In this paper, the same reading patterns analyzed by Rodeghero et al. [Rodeghero and McMillan 2015] are also examined as part of the analysis on a new dataset.

4 EYE TRACKING STUDY OVERVIEW

We now describe the process of designing and performing the eye-tracking study for the method summarization tasks. In this paper we analyze data collected from an eye-tracking study of 18 developers summarizing methods [Abid et al. 2019]. The tasks, processed eye-tracking data, and the statistical analysis will be made available through the replication package at http://seresl.unl.edu/ETRA2019.

Table 1: Java Systems Used in the Study

System version	Domain	Total methods	Selected methods
ArgoUML 0.30.2	UML diagramming tool	14,635	15
MegaMek 0.36.0	Computer game	12,490	15
Siena 1.0.0	Database library	4,116	12
sweetHome3d4.1	Interior design application	6,084	12
aTunes 3.1.0	Audio player	9,579	9
Total		**46,904**	**63**

4.1 Study Tasks

The study consists of a total of 63 methods chosen from five open source Java systems randomly selected from different domains (see Table 1). While the selection of methods chosen to be summarized is random, two conditions were maintained: trivial methods such as setters and getters were not part of the sample and the largest method was restricted to 80 lines of code (LOC). This was done in the prior study to avoid excessive fatigue during tasks and to complete the study in under an hour. McCabe's cyclomatic complexity is computed for all 63 methods. The complexity of the methods ranges between 1 to 28 with average = 6, SD = 5, and Median = 5). A total of 63 methods were selected from the systems shown in Table 1. On average, participants summarized 15 of the 23 methods provided to them via a random sampling.

The participants were asked to read the assigned methods and write a summary for the method. They were also told that they could navigate the codebase if they needed to. The study was conducted inside the Eclipse integrated development environment (IDE) using iTrace [Guarnera et al. 2018; Shaffer et al. 2015]. iTrace connects to an eye tracker and maps eye gaze data to semantic elements on the fly even in the presence of file scrolling and file switching.

4.2 Participants

There were 18 participants in this study of which 5 were experts and 13 were novices. Two of the experts were from industry with the other three experts were senior PhD students with 5+ years of experience working in open source software. Novices were mainly undergraduate students at a local university and had between one to five years of programming experience.

4.3 Eye Tracking Apparatus and Measures

The Tobii X60 eye tracker was used for the data collection. As in [Rodeghero et al. 2014], two types of eye-movement data [Rayner 1998] are used: number of fixations and their durations (gaze time). Since we compare in RQ2 our results to [Rodeghero et al. 2014], we use the same eye tracking measures. Gaze time is the total number of milliseconds spent on a region of interest (ROI) such as a method call or identifier. A fixation filter [Olsson 2007] is set to count fixations that are more than 100 milliseconds (same as Rodeghero). We use number of fixations and number of visits interchangeably.

4.4 Study Procedure

After signing the informed consent form, the participants were first given a short description of study procedure including types of questions asked. They were then given examples of three method summaries so they were familiar with how to answer the question. These examples were different from the ones they were asked to summarize. We did this to make sure the participants understood what they were expected to do during the summarization task. They were encouraged to use their own words to summarize the methods rather than just narrating what each line in the method was doing. After a quick calibration the data was captured using iTrace [Guarnera et al. 2018]. The participants wrote their summary in a text file. The study took between 45-90 minutes for 15 method summarizations.We recalibrated the participant after every 5 tasks or as needed to be sure there was no issue with drift. We double checked to make sure calibration is still valid before moving to the next task.

5 STUDY RESULTS

The data was checked for validity before analysis. Some invalid samples include writing a narration of the code word for word (e.g.,"The method has a while statement to check if values are false") had to be discarded. Other cases are because participants were unable to locate or unable to understand the assigned method. In order to catch such cases, three individual researchers reviewed 257 summaries written to judge their suitability for further analysis. We recorded an inter-rater reliability of Cohen's kappa (κ) is 0.92. In total, we discarded 44 summaries from the total and kept the remainder 213 (correct) summaries for our analysis. Correct summaries were determined by three researchers via manual inspection.

5.1 RQ1: Determining the Mental Model

According to the integrated metamodel [von Mayrhauser and Vans 1995], during source-code comprehension, developers apply one of the two comprehension models: top-down or bottom-up. Top-down is used when a developer has a hypothesis and wants to confirm it by examining several locations. Conversely, bottom-up is used when a developer has less understanding and needs to closely read and form a hypothesis. During the comprehension process, a developer mentally chunks or groups each set of statements into higher level abstractions. These abstractions (chunks) are aggregated further until a high-level understanding of the program is obtained [Storey 2005; von Mayrhauser and Vans 1995]. These models are mostly theoretical and there are only a few user studies using traditional techniques (e.g., think aloud) to understand what mental model people perform during maintenance or debugging tasks [von Mayrhauser and Vans 1997] as it is a non-trivial problem. Here, we use eye-tracking data to determine which model is used during the summarization task.

To determine the type of mental model a developer performs, we first divide each method into chunks/sections. Identifying source code sections (or code snippet) is explored by Chatterjee et al. [Chatterjee et al. 2017]. They concluded that code snippets embedded in different document types ranges between 10 to 13 LOC. This suggest that they are code examples for single and specific actions.

Here, we identify code snippets or chunks within source code

```
1  public void run() {
2      int port = 0;
3      try {                                              Chunk 1
4          port = Integer.parseInt(hostPort.getText());
5          ServerSocket serverSocket = new ServerSocket(port);
6          Socket s = serverSocket.accept();
7          serverSocket.close();
8
9          System.out.println("Accepted peer connection.");   Chunk 2
10
11         conn = ConnectionFactory.getInstance().createServerConnection(s, 0);
12
13         conn.addConnectionListener(connectionListener);
14
15         board = new Board();
16         panConnect.setEnabled(false);                  Chunk 3
17         panXmit.setEnabled(true);
18
19     } catch (Throwable err) {                          Chunk 4
20         err.printStackTrace();
21     }
22 }
```

Figure 1: The four chunks of the method `run` from class `PacketTool`. This method belongs to the `megamek` system.

Table 2: The average time of reading bottom-up or top-down by experts and novices.

Level of expertise	Average of	Inside the method		Outside the method	
		Bottom-up	Top-down	Bottom-up	Top-down
Experts	Actual duration	43913ms	37351ms	1200ms	2684ms
	Percentage of duration	49%	48%	1%	2%
Novices	Actual duration	66780ms	59763ms	5060ms	6780ms
	Percentage of duration	48%	45%	2%	4%

files [Wang et al. 2014]. A chunk is set of continuous statements that contains various levels of text-structure abstractions. Through manual analysis of the source code and how source code authors tend to organize the code, we group each set of lines that form one high level action to identify a chunk. The chunk ranges between 2 LOC to 10 LOC. Figure 1 presents an example of a method that is divided into four chunks. The first chunk creates and prepares the `serverSocket` object. The second chunk establishes the connection. The third creates the `board` object. Finally, the last is the catch block. Another example is shown in Figure 2. Based on how the code is written (spacing between lines), this method is divided in to three chunks. The first chunk is composed of 10 lines of creating two variables and an array of objects. The second chunk is a loop. The last chunk consists of two `void` calls that carries out the final (usually the main) action of the method. We determine the mental model by comparing two contiguous eye-tracking fixation records.

- If a developer starts reading a chunk and the next line read belongs to the same chunk, then we conclude that the developer is reading closely and performing bottom-up comprehension.
- When the developer switches to a different chunk, we conclude that the developer is performing top-down comprehension.

Furthermore, there are three main locations on the screen that a participant can look at, namely: the text file to write the summary, inside the assigned method to be summarized, and outside the assigned method being summarized (which includes other methods perhaps being used by the method being summarized). In our analysis, we detected four reading behaviors namely; top-down inside the method, bottom-up inside the method, top-down outside the method and bottom-up outside the method.

```
1 public void deploy(int id, Coords c, int nFacing, int elevation,
2                    List<Entity> loadedUnits, boolean assaultDrop) {
3       int packetCount = 6 + loadedUnits.size();                    Chunk 1
4       int index = 0;
5       Object[] data = new Object[packetCount];
6       data[index++] = new Integer(id);
7       data[index++] = c;
8       data[index++] = new Integer(nFacing);
9       data[index++] = new Integer(elevation);
10      data[index++] = new Integer(loadedUnits.size());
11      data[index++] = new Boolean(assaultDrop);
12
13      for (Entity ent : loadedUnits) {                             Chunk 2
14          data[index++] = new Integer(ent.getId());
15      }
16
17      send(new Packet(Packet.COMMAND_ENTITY_DEPLOY, data)); Chunk 3
18      flushConn();
19  }
```

Figure 2: The three chunks of the method `deploy` from class `Client`. This method belongs to the `megamek` system.

Table 3: Wilcoxon Test of experts and novices that compares top-down and bottom-up reading. n is number of samples. A sample is a result from one developer and one method. T is the sum of ranks. A hypothesis is rejected if and only if the standard normal distributed |Z| >= 1.96 and p <.05

	H	Metric	Mental model	n	T	Z	p
Experts	H1	Fixations	Top-down	30	937	-1.07	0.28
			Bottom-up	36	1274		
Novices	H2	Fixations	Top-down	75	5164	-0.47	0.63
			Bottom-up	65	4706		

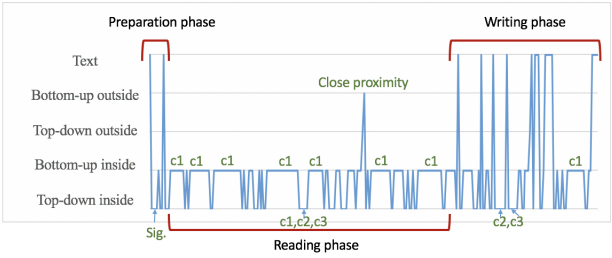

Figure 3: The timeline of the expert (E1) summarizing the method `deploy` in Figure 2. c1 and c2 are short for chunk1 and chunk2.

As presented in Table 2, for experts and novices, bottom-up is applied more than top-down. To examine the significance of this difference, we use the Wilcoxon test [Hollander and Wolfe 1973] to compare the percentage of time spent performing each model. We present the two hypotheses as follows:

H_n: For [experts / novices], the difference between the computed metric for reading top-down and bottom-up is not statistically significant.

From Table 3, we cannot reject hypothesis H_1 (experts) and H_2 (novices) which indicates that the difference between top-down and bottom-up is not statistically significant. Therefore, we conclude that experts and novices apply bottom-up more that top-down but the difference is not statistically different.

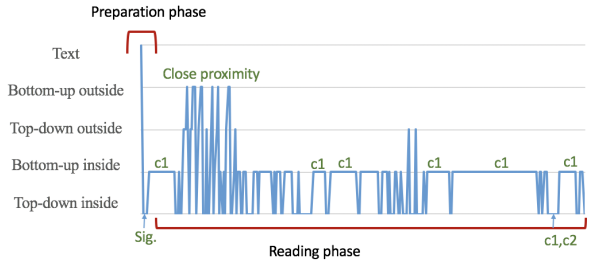

Figure 4: The timeline of the novice (N1) summarizing the method `deploy` in Figure 2. c1, c2, and c3 are short for chunk1, chunk2, and chunk3.

Timeline Analysis of Developers' Mental Models: Considering the three locations that a participant has access to and the type of mental model applied, we generate the timeline of each participant. We distinguish between three phases: 1) preparation phase; 2) reading phase; and 3) writing phase. Figure 3 shows the timeline of one expert (E1). The x-axis reflects the changes overtime and y-axis reflects the location that the expert read and the mental model applied. The straight lines indicate that certain reading is done more than others i.e., the developer spent a lot of time in bottom-up-inside indicating that he/she stayed reading bottom up inside the method being summarized for quite some time. The time line is also fragmented according to the level of jumping between sections and their locations. In the first four steps in Figure 3, the expert (E1) switches behaviors or locations. The order of locations of the first four steps are: text, top-down inside, bottom-up inside, and top-down inside and finally back to text. This switching behavior is an indicator that developer is getting ready to start, we call this the *preparation phase*. Once this behavior is followed by four or more stable (e.g., participant reads inside the method only) behaviors, we conclude that the participant started a new phase which in this case is the *reading phase*. The middle section is the reading phase; the participant switches between bottom-up and top-down inside or outside the method. The final stage is the writing phase when the participant starts switching between the text file (to write the summary) and the source code base. The first three lines of the text file contains the task number, the name of the assigned method, and the path of the file containing the assigned method. This is followed by a section that asks the participant to write the summary. It is possible that the participant's gaze falls on the method name or the path in the text file during the performing of the task. Therefore, to detect the writing phase, we check the line number of the text that a participant read. The writing phase starts if the participant starts looking at line number four or higher in the text file. We are able to determine gaze on line number because iTrace [Guarnera et al. 2018] gives us this mapping automatically since it maps gaze to specific lines/tokens not just in the code but also in other files in the IDE such as text files in this case.

Figure 4 presents the timeline of one novice (N1). Notice that, the writing phase is not shown in this diagram although this participant did write the summary. Most of the participants followed the sequence displayed in Figure 4. After locating the method, a participant starts by bouncing between the text file, inside and outside the

method before actually reading the method. This preparation phase is detected on 54% of novices' samples and 53% of experts' samples. All experts and novices perform the reading phase. Finally, the writing phase is detected on 15% of novices samples and 17% of experts' samples since in many cases the participants moved their eyes from the screen to look at the keyboard which the eye tracker did not capture. However, we know from the text summary files that were generated and the external video recorded that the summaries were indeed being written. During the writing phase participants switch between the text file and the source code to refine their summary. Additionally, there are two participants who start writing their summaries earlier than most in several tasks. Then, they read the source code for some time to refine the summary.

Table 4 and Table 5 presents the average time spent in each phase by experts and novices, respectively. On average, an expert spends about 19% of their time preparing for the task. They usually collect the information they need about the class and data members during this phase. Novices spend about 16% of their time during the preparation stage. Although the percentage of the time is not different between experts and novices, the duration spend by novices is higher. On average, novices spend about 16s. (16,500ms.) in the preparation phase while expert spend about 12s. (12,500ms.). Similarly, novices spend longer time in reading and writing phases, as they need longer time to understand and describe source code [Abid et al. 2019].

Qualitative Analysis: We now present some additional insight into the reading behavior of experts and novices by manually analyzing what specific line they looked at in the chunks during the task. We do this analysis on four samples. The timeline of the expert (E1) presented in Figure 3 is during reading the method `deployin` Figure 2. In Figure 3, after locating the task from the text file, the expert reads the method in a sequential manner starting from the method signature. The straight lines of bottom-up-in are when the expert is reading chunk 1. The intermediate jump to bottom-up-out is made to read lines within close proximity to the method. In this case, it is the first three lines of the method immediately below. Chunk 2 and 3 are mostly read in top-down-in fashion due to their short sizes. However, it is worth mentioning that the expert (E1) is focusing on reading lines 13, 14, 17, and 18 while writing the summary. The summary written by this expert is "Transmit game unit object data positioning over the network and flush the connection".

The timeline of the novice (N1) presented in Figure 4 is during reading of the method `deploy)` in Figure 2. In Figure 4, after reading the task in the text file, the novice (N1) read the signature of the method. Similar to the expert (E1) explained above, most of the bottom-up-in are spent on chunk 1. Additionally, this novice (N1) made 36 jumps to read the signature of the method compared to 10 jumps made by the expert (E1). All outside method locations read were methods in close proximity. The summary written by this novice is "The method deploys different entities to a specified spot in the coordinate system. It organizes the data for the entities before the data is sent via a packet into an appropriate format".

Figure 5 presents the timeline of one expert (E2) summarizing the method `run` in Figure 1. According to the collected data, this expert (E2) started reading the method from the middle at line 11 followed by line 15. Then, the expert jumped about 200 lines up to read `private Board board = null;` which is the declaration of the

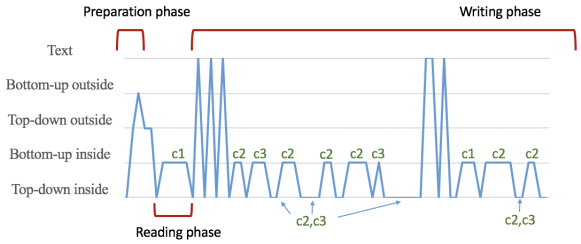

Figure 5: The timeline of the expert (E2) summarizing the method `run` in Figure 1. c1, c2, and c3 are short for chunk1, chunk2, and chunk3.

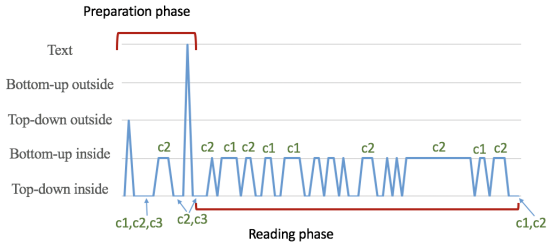

Figure 6: The timeline of the novice (N2) summarizing the method `run` in Figure 1. c1, c2, and c3 are short for chunk1, chunk2, and chunk3.

data member `board`. This jump indicates that the expert (E2) used the search feature that was available during the experiment. Then, a series of bottom-up-in readings are observed in the following order: chunk1, chunk2, chunk3, and chunk2. Next, top-down is observed between chunk 3 and chunk 2 followed by bottom-up-in: chunk2 and chunk3. The final long top-down is between chunk1 and chunk 3. Note that the expert (E2) starts writing the summary earlier on in the session. The summary written is "Creates a connection to host port through a socket and sets up a board".

Figure 6 demonstrates the timeline of one novice (N2) summarizing the method in Figure 1. Notice that the writing phase is not detected. The novice (N2) starts from the middle of the method at line 13. Then, the novice (N2) performs a series of bottom-up-in and top-down-in behaviors in all chunks. The longest bottom-up-in is made in chunk 2. The summary written is "The port number is found from the hostPort and a server is started based on that number. After notifying the user, the server makes a connection and creates a new board". For both the expert (E2) and novice (N2)summarizing the method *run* in Figure1, none of the reading records belongs to the method signature. This is a reasonable behavior as limited high level information can be obtained from this signature as it is a method with no parameters and returns void. It is possible that because the line is so short (`public void run()`) it causes developers to skip it since they saw it has limited information in their peripheral vision. This is the same case that can be made for seeing no fixations occurring on opening and closing braces.

Table 4: Average duration and percentage of duration spent in each phase by experts.

Phase Name	Number of non-zero cases	Average Number of steps	Average Percentage of steps	Average Duration in ms	Average Percentage of duration
Preparation	37/69	17	14%	12500ms	19%
Reading	69/69	86	86%	75500ms	85%
Writing	12/69	40	23%	38000ms	22%

Table 5: Average duration and percentage of duration spent in each phase by novices.

Phase Name	Number of non-zero cases	Average Number of steps	Average Percentage of steps	Average Duration in ms	Average Percentage of duration
Preparation	79/144	18	18%	16500ms	16%
Reading	144/144	114	82%	122140ms	83%
Writing	29/144	83	39%	77700ms	38%

5.2 RQ2: Reading Patterns

We examine three types of reading patterns namely: top-to-bottom versus bottom-to-top, skimming versus thorough, and disorderly versus sectionally studied by [Rodeghero and McMillan 2015]. To allow for a fair comparison, the same approach is used to compute the three metrics in both works.

5.2.1 Top-to-bottom vs. Bottom-to-top. We observed that experts and novices tend to start at the beginning of a method (not necessarily the signature) in about 75% of the cases. However, developers do not examine each line in the code. Instead, a developer may read code in the following order line 2, line 5, back to line 4, then to line 6. The overall pattern of the previous example is top-to-bottom. To examine if a developer is reading top-to-bottom or bottom-to-top, we compute the two metrics for each data sample. For each sample, we keep track of every time a developer changes line by moving up or down. Then, to determine the significance, we compare the percentage that the developers read from top-to-bottom or bottom-to-top using the Wilcoxon test [Hollander and Wolfe 1973] (paired sample) and propose the following two hypotheses:

H_n: For [experts / novices], the difference between the computed metric for reading top-to-bottom and bottom-to-top is not statistically significant.

We reject a hypothesis if |Z|> 1.96 and p <= .05 (Table 6). In this case, we reject two hypotheses (H_1 and H_2). This indicates that both novices and experts tend to read top-to-bottom more often than bottom-to-top.

5.2.2 Skimming vs. Thorough. Rodeghero et al. use 1000ms to be the cut point to define the *reading thoroughly* pattern. Furthermore, we observe that developers spend 1000ms to 10,000ms when they focus on a word or a line. Therefore, we also use 1000ms to be our cut point to define reading thoroughly in order to facilitate comparison to previous work. On average, experts and novices read thoroughly 10% and 11% of the time, respectively. To measure the significance of the result, we compare the average number of cases when a developer read thoroughly (spending over 1000ms) vs. skimming the code using the Wilcoxon test. We propose the following two hypotheses:

H_n: For [experts / novices], the difference between the computed metric for reading thoroughly and skimming the code is not statistically significant.

Table 6: Wilcoxon test results of experts and novices comparing the three reading patterns.

	H	Metric	Reading pattern	n	T	Z	p
Experts	H1	Fix.	Top-to-bottom	69	2335	-6.74	<.0001*
			Bottom-to-top	69	80		
Novices	H2	Fix.	Top-to-bottom	144	10270	-10.1	<.0001*
			Bottom-to-top	144	170		
Experts	H3	Fix.	Skim	69	2346	-7.22	<.0001*
			Thorough	69	0		
Novices	H4	Fix.	Skim	144	1	-10.4	<.0001*
			Thorough	144	10439		
Experts	H5	Fix.	Sectionally	69	1242	-1.6	0.1
			Disorderly	69	773		
Novices	H6	Fix.	Sectionally	144	7667	-5.89	<.0001*
			Disorderly	144	2062		

From Table 6, we reject hypotheses H_3 and H_4. This strongly suggests that both novices and experts tend to skim the code and focus on only a few lines.

5.2.3 Disorderly vs. Sectionally. Reading disorderly means that a developer jumps between terms/lines while reading the source code. On the other hand, reading sectionally means that the developer inspects through sections surrounding terms/lines. Similar to [Rodeghero and McMillan 2015], we define a section as a set of three contiguous lines. Then, we compute the percentage in which a developer changes the section to the percentage in which the developer stays in the same section. Finally, the percentages are compared using the Wilcoxon test for the following hypotheses:

H_n: For [experts / novices], difference between the computed metric for reading disorderly and sectionally is not statistically significant.

From Table 6, we reject hypothesis H_6 but not H_5 which indicates that experts tend to use a combination of disorderly and sectional techniques when reading code which confirms the result from the Rodeghero study [Rodeghero and McMillan 2015]. On the other hand, novices read the code more sectionally. Therefore, it could be that reading sectionally maybe a sign of less experience.

5.2.4 Comparing Findings to Prior Work. We now compare findings for RQ2 with prior work [Rodeghero and McMillan 2015]. Rodeghero and McMillan conclude that developers tend to read top-to-bottom 49% of the time while in our case developers read top-to-bottom about 59% of the time. Furthermore, they conclude that developers jump between sections 75% of the time while our results suggest that developers use both jumping and reading within a section. One way to explain the above differences is that in prior work

Table 7: Reading patterns of experts between this study and [Rodeghero and McMillan 2015]. (✓ indicate results reproduced).

Reading Pattern	Prior Work [Rodeghero and McMillan 2015]	This Study
Top-to-bottom vs. Bottom-to-top.	49% Top-to-bottom Not significant	75% Top-to-bottom Top-to-bottom is performed significantly more than bottom-to-top.
Skimming vs. Through.	90% skimming Skimming is performed significantly more than reading thoroughly.✓	90% skimming Skimming is performed significantly more than reading thoroughly. ✓
Disorderly vs. Sectionally.	25% sectionally Sectionally is performed significantly more than reading disorderly.	47% sectionally Not significant

developers see a fixed screen where the entire code is displayed (no scrolling is allowed). With this limited size, the memory lost is minimal and developers can return to the previous location easily. In our study, methods are large in size and developers need to scroll up and down during the summarization task. When they scroll some of the code become invisible. Although our study environment makes the task a bit harder than the one in [Rodeghero and McMillan 2015], our study environment simulates the real world environment. Finally, similar to prior work, we conclude that developers tend to skim the code more frequently and read thoroughly 10% of the time. Refer to Table 7 for a comparison.

5.3 Threats to Validity

When a developer writes a summary for a method from a class, he/she may build some knowledge about the class. This might affect the time and the effort to understand other methods from the same class. To mitigate this, each developer was asked to summarize methods from different unrelated classes. To avoid fatigue, we kept the number of methods to be summarized to 15 with the goal of having the study completed in about an hour. To reduce the overhead in browsing many systems, we limit the number of systems for each developer to three. As developers mostly rely on comments to understand methods [Crosby and Stelovsky 1990], we remove all comments from the source code (similar to the Rodeghero study). This was necessary as the goal of the study is to examine source code statements that developers focus on when they write their *own* summaries. Developers have different IDE environment preferences that might affect their performance. We kept the default bare-bones Eclipse syntax highlighting preferences for all participants. In addition, code folding was not allowed to reduce any confounding effects. None of the participants complained about this setting or the highlighting used. We define chunks based on how the source code was written by the original authors and according to prior literature [Wang et al. 2014]. It is possible that developers have a different chunking approach which may affect the result. Individual differences and abilities may be a possible factor however, this was out of scope of this paper. We did not address them in our analysis. It is possible that we will get different results with a larger number of experts. The study is performed by 18 participants with 5 experts. This limited number of experts may affect the validity of the result. We use appropriate statistical tests to match our data assumptions.

6 DISCUSSION

We analyze and predict the mental model of a participant by dividing the source code to predefined chunks (each chunk presents a high level operation) based on how the code is written. The chunk size ranges between 2-10 LOCs. Neither top-down or bottom-up are used more significantly than the other. This is expected as top-down is used more if the programmer is familiar with the code [von Mayrhauser and Vans 1995] and none of our developers were familiar with open source systems used in the study. We believe that this result opens the door to study the mental model of developers using eye-tracking data instead of using traditional methods (e.g., interview). The mental models applied by a developer depends on their level of expertise of the domain of the presented source code. When the sectionally and disorderly reading patterns were examined, we found that novices read sectionally (with section size equal to three) more heavily than disorderly. This means that novices read source code more in a line-by-line fashion (also shown by [Busjahn et al. 2015]), a behavior also reflected in their summaries. Novices usually write a few words describing the steps of the method being summarized. On the other hand, experts perform both patterns. With respect to RQ 1, we find this study to be the first attempt to use eye gaze as a predictor of externalizing the mental model of developers. With respect to RQ2, we find that experts perform 90% skimming in reading and top-to-bottom reading is performed significantly more (75% of the time) than bottom-to-top.

7 CONCLUSIONS AND FUTURE WORK

The paper presents an analysis of using eye-tracking data to understand the cognitive process during source code summarization. We examine two types of mental models that developers perform during source-code comprehension. On average, experts and novices read the assigned method more closely (using bottom-up mental model for comprehension) than bouncing (using top-down) around. This is expected as all developers in the study do not have domain knowledge about the code used in the study. However, on average, novices have longer gaze time preforming bottom-up compare to experts. We propose a graphical representation that demonstrates a developer's timeline while reading and summarizing a method. As part of future work, we plan to conduct a follow up study of developers reading two sets of methods - one from a domain that the developer is familiar with and the other from an unfamiliar domain.

ACKNOWLEDGMENTS

We thank all the participants who performed this study. This work is supported in part by grants from the National Science Foundation under grant numbers CCF 18-55756 and CCF 15-53573.

REFERENCES

Nahla J. Abid, Bonita Sharif, Natalia Dragan, Hend Alrasheed, and Jonathan I. Maletic. 2019. Developer Reading Behavior while Summarizing Java Methods : Size and Context Matters. In *Proceedings of the 41th International Conference on Software Engineering (ICSE 2019)*.

Roman Bednarik and Markku Tukiainen. 2006. An Eye-tracking Methodology for Characterizing Program Comprehension Processes. In *Proceedings of the 2006 Symposium on Eye Tracking Research &Amp; Applications (ETRA '06)*. 125–132.

Roman Bednarik and Markku Tukiainen. 2008. Temporal Eye-tracking Data: Evolution of Debugging Strategies with Multiple Representations. In *Proceedings of the 2008 Symposium on Eye Tracking Research & Applications (ETRA '08)*. 99–102.

Dave Binkley, Marcia Davis, Dawn Lawrie, Jonathan I. Maletic, Christopher Morrell, and Bonita Sharif. 2013. The impact of identifier style on effort and comprehension. *Empirical Software Engineering* 18, 2 (01 Apr 2013), 219–276.

Ruven Brooks. 1983. Towards a theory of the comprehension of computer programs. *International journal of man-machine studies* 18, 6 (1983), 543–554.

Jean-Marie Burkhardt, Françoise Détienne, and Susan Wiedenbeck. 1997. Mental representations constructed by experts and novices in object-oriented program comprehension. In *Human-Computer Interaction INTERACTâĂŹ97*. Springer, 339–346.

Teresa Busjahn, Roman Bednarik, Andrew Begel, Martha Crosby, James H. Paterson, Carsten Schulte, Bonita Sharif, and Sascha Tamm. 2015. Eye Movements in Code Reading: Relaxing the Linear Order. In *2015 IEEE 23rd International Conference on Program Comprehension*. 255–265.

Preetha Chatterjee, Manziba Akanda Nishi, Kostadin Damevski, Vinay Augustine, Lori Pollock, and Nicholas A Kraft. 2017. What information about code snippets is available in different software-related documents? an exploratory study. In *Software Analysis, Evolution and Reengineering (SANER), 2017 IEEE 24th International Conference on*. IEEE, 382–386.

Martha E. Crosby and Jan Stelovsky. 1990. How do we read algorithms? A case study. *Computer* 23, 1 (1990), 25–35.

Drew T. Guarnera, Corey A. Bryant, Ashwin Mishra, Jonathan I. Maletic, and Bonita Sharif. 2018. iTrace: Eye Tracking Infrastructure for Development Environments. In *Proceedings of the 2018 ACM Symposium on Eye Tracking Research & Applications (ETRA '18)*. Article 105, 3 pages.

Myles Hollander and Douglas A. Wolfe. 1973. *Nonparametric Statistical Methods*. John Wiley and Sons, New York.

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

Stanley Letovsky. 1987. Cognitive processes in program comprehension. *Journal of Systems and software* 7, 4 (1987), 325–339.

Pontus Olsson. 2007. Real-time and Offline Filters for Eye Tracking. (2007), 42 pages.

Nancy Pennington. 1987. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive psychology* 19, 3 (1987), 295–341.

Keith Rayner. 1998. Eye movements in reading and information processing: 20 years of research. 124, 3 (00 1998), 372–422.

Paige Rodeghero and Collin McMillan. 2015. An Empirical Study on the Patterns of Eye Movement during Summarization Tasks. In *2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, Vol. 00. 1–10.

Paige Rodeghero, Collin McMillan, Paul W. McBurney, Nigel Bosch, and Sidney D'Mello. 2014. Improving Automated Source Code Summarization via an Eye-tracking Study of Programmers. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. 390–401.

Timothy R. Shaffer, Jenna L. Wise, Braden M. Walters, Sebastian C. Müller, Michael Falcone, and Bonita Sharif. 2015. iTrace: Enabling Eye Tracking on Software Artifacts Within the IDE to Support Software Engineering Tasks. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*. 954–957.

Bonita Sharif. 2011. Empirical assessment of UML class diagram layouts based on architectural importance. In *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, Vol. 00. 544–549.

Bonita Sharif, Michael Falcone, and Jonathan I. Maletic. 2012. An Eye-tracking Study on the Role of Scan Time in Finding Source Code Defects. In *Proceedings of the Symposium on Eye Tracking Research and Applications (ETRA '12)*. 381–384.

Bonita Sharif and Jonathan I. Maletic. 2010a. The Effects of Layout on Detecting the Role of Design Patterns. In *2010 23rd IEEE Conference on Software Engineering Education and Training*. 41–48.

Bonita Sharif and Jonathan I. Maletic. 2010b. An Eye Tracking Study on camelCase and under score Identifier Styles. In *2010 18th IEEE International Conference on Program Comprehension*. 196–205.

Ben Shneiderman and Richard Mayer. 1979. Syntactic/semantic interactions in programmer behavior: A model and experimental results. *International Journal of Computer & Information Sciences* 8, 3 (1979), 219–238.

Elliot Soloway and Kate Ehrlich. 1984. Empirical studies of programming knowledge. *IEEE Transactions on software engineering* 5 (1984), 595–609.

Margaret-Anne Storey. 2005. Theories, methods and tools in program comprehension: Past, present and future. In *Program Comprehension, 2005. IWPC 2005. Proceedings. 13th International Workshop on*. IEEE, 181–191.

Hidetake Uwano, Masahide Nakamura, Akito Monden, and Ken ichi Matsumoto. 2006. Analyzing Individual Performance of Source Code Review Using Reviewers' Eye Movement. In *Proceedings of the 2006 Symposium on Eye Tracking Research &Amp; Applications (ETRA '06)*. 133–140.

Armeliese von Mayrhauser and A. Marie Vans. 1995. Program Comprehension During Software Maintenance and Evolution. *Computer* 28 (08 1995), 44–55. https://doi.org/10.1109/2.402076

Anneliese von Mayrhauser and A. Marie Vans. 1997. Program understanding behavior during debugging of large scale software. In *Papers presented at the seventh workshop on Empirical studies of programmers*. ACM, 157–179.

Xiaoran Wang, Lori L. Pollock, and K. Vijay-Shanker. 2014. Automatic Segmentation of Method Code into Meaningful Blocks: Design and Evaluation. *Journal of Software: Evolution and Process* 26, 1 (2014), 27–49. https://doi.org/10.1002/smr.1581