Enabling Eye Tracking to Study the Use of Software Artifacts on Code Summarization

by

SHRUTHI NAGARAJ

A thesis submitted to the Faculty of Graduate and Postdoctoral Affairs in partial fulfillment of the requirements for the degree of

Masters's in Computer Science

in the

SCHOOL OF COMPUTER SCIENCE

Declaration of Authorship

I declare that this thesis titled, 'Enabling Eye Tracking to Study the Use of Software Artifacts on Code Summarization' and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Abstract

Master's in Computer Science

by SHRUTHI NAGARAJ

Application Programming Interface (API) form an important part of software development. Eye tracking is an interesting and emerging field which is gaining wide popularity. The goal of the thesis is to collect and utilize eye tracking data to better understand how software developers summarize code. When developers are presented with a set of change tasks, they use various mediums (or as we call them *information sources*) such as developer forums, bug reports, etc. to quickly understand code elements. But what information sources are more useful and effective for developers' understanding of code? To answer this question, we perform a study that measures the effects of complex information sources on summarization tasks. In this work, we conduct a controlled experiment with 12 professional and student software developers to better understand how they perform code summarization tasks. These tasks aimed at summarizing a set of APIs based on source code, Stack Overflow, Bugzilla, or a combination of them. We used iTrace, a plugin for Eclipse, to record the developers' eye movements. The summaries given by developers and their collected eye movements were analysed. We could find out which part in each information source do developers prefer to view during the change task and also which information source would they find more reliable when given a choice between different information sources. And also we could find on using which information source the developers could give best quality summaries. We hope the results of our work can help to design better automatic summarization tools in the future.

Acknowledgements

I would like to thank my supervisor Dr. Olga Baysal for all the support and guidance.

I would like to thank all the thesis defence committee members for their valuable time.

Contents

List of Figures

List of Tables

This Thesis is dedicated to my mother, father, sister, bruno and friends...

Chapter 1

Introduction

In this chapter, we first present our motivation for this work (Section 1.1), overview on eye tracking (Section 1.2) and iTrace (Section 1.3), definition and planning of the experiment (Section 1.4), outline our research questions (Section 1.5), and summarize the contributions of our work (Section 1.6). Organization of the thesis is offered in Section 1.7.

1.1 Motivation

Software developers deal with a lot of software elements in their day to day work. They deal with a number of programming languages to develop code. In order to develop good quality code they should be aware of the software code elements, their properties and their usage. And also the best way to indulge any particular code element is to bring out a good quality software program. But when a developer is asked to make certain changes to code elements, they will have to go through much information and the process might be time consuming. To help developers quickly understand the classes or methods, code summarization technique has been proposed. Code summarization is a process in which developers create summaries about an element in a software to best describe its purpose in the software program.

APIs (Application Programming Interface) are an integral part of building any software application or program. They are a set of definitions or protocols which best describe how the software components in an application will react. An API is mostly in a form of a method or class. Our objective is to investigate what strategies developers use to better understand API elements. In the past, few researchers have suggested code summarization to help developers quickly understand code elements. Majority of the

summarization techniques are built around source code and very few works on using informal documentation for the summarization of code elements. However, developers often depend on other sources of informal documentation such as developer forums (e.g., Stack Overflow), bug reports (e.g., from Bugzilla issue tracking system), etc. There is little work that studies how various information sources are leveraged by the developers for API comprehension and summarization tasks. The main focus here is to consider both formal and informal pieces of information about APIs (methods and classes) and see how they may affect developer performance in summarizing APIs. We consider the following information sources in our study: source code, information from forums such as Stack Overflow, bug reports mined from Bugzilla, and a combination of all these information sources to study how developers leverage these mediums for API summarization.

Eye tracking is an emerging field of research and science. It has been gaining a vast popularity in the current decade. While conducting study including eye tracking, there are many levels of advanced techniques that can be adopted based on the study needs. In experiments with eye tracking, the eye gazes of a participant are recorded while she performs tasks and then her gaze data is analyzed. The level of detail we can collect from eye tracking data is pretty vast. It could be a simple collection of data with just eye movements or much advanced collection of different angles of vision and its schema levels.

In our research work, the main idea of conducting the experimental study is to ask developers to perform a number of summarization tasks while collecting their gaze data to be able to track precise elements they look at using various information sources, the duration of the gazes, as well as overall tracking of the developer eye movements. Also the study mostly revolves over how eye tracking data helps to analyze the behavior of the developer while she does the tasks. The depth of information we collect analyzing the eye tracking techniques is impressive.

The entire study was designed to take place using iTrace [1] platform which is a Java Eclipse plugin. iTrace was combined with a working eye tracker to collect the eye tracking data and the output of the study. Work spaces were created for every developer who was be a part of the study and their actions during a task were recorded. In addition to capturing the eye tracking movements, we even collected the screen video captures with the help of Camtasia software [2] to triangulate our results.

iTrace is an Eclipse plugin that records developers' eye developments while they deal with change undertakings. It is one of the first platforms that makes it workable for research scientists to direct eye movements about on extensive programming frameworks. The outline is sufficiently adaptable to record eye developments on different kinds of

programming antiquities like Java code, XML, HTML, etc. iTrace is intended to help an assortment of eye trackers. The module has been effectively utilized for programming traceability undertakings.

We conducted an experiment with 12 software developers at professional, bachelors, Master's and PhD levels who were given set of tasks to summarize certain API elements using various information sources. While they perform the experiment, their eye movements were recorded and the outcome summaries were analyzed. The data was later decoded to answer a number of research questions (described in Chapter 1.5. Based on the results of our experiment, we present a number of findings that developers follow a specific behaviour or strategy in summarizing APIs.

1.2 Overview on Eye Tracking

The concept of eye tracking plays a vital role in the experiment. An eye tracker is a gadget which is designed to collect the user's eye movements during a task. Advanced eye-trackers gather this data in a subtle way without the utilization of main apparatus. The gadget itself gathers eye looks and alternatively records movements of the eye movement session. The visual concentration of the eyes on a specific area triggers mental procedures required to tackle a given assignment. Eye stare developments can be portrayed as obsessions and saccades. An obsession happens when the eye settles on a question for a specific day and age. Saccades are speedy developments that move the eyes starting with one area then onto the next. A check way is a coordinated way shaped by saccades between obsessions. The general accord in the eye following examination [1] group is that the preparing of imagined data happens amid obsessions, while, no such preparing happens amid saccades [3].

1.3 iTrace

iTrace [1] is a plugin for the Eclipse IDE which has been designed to use eye tracking feature. Eclipse plug-ins are executed in Java. The Eclipse IDE is comprised of a few view parts, known as perspectives and editors. Perspectives allude to the windows that give prompt data or control while editors allude to the windows that give control of substance that must be spared, for example, the source code editorial manager. Every view part contains a few gadgets that show content or encourage client connection (e.g., catches, content marks). iTrace empowers eye-following on subjective gadgets inside both perspectives and editors.

Eye tracker objects are associated with and read from an eye tracking gadget. The Tobii Tracker usage utilizes the Java Native Interface (JNI) to execute C++ code interfacing with the gadget's SDK. An immaculate Java approach can be taken for gadgets with which Java can straightforwardly connect. As eye gazes are collected from the gadget, iTrace makes new Gaze objects from the look area and timestamp, leaving the reaction field discharge. The Tobii eye tracker was connected to the monitor screen with the help of an adhesive strip which made it fit to the top of the screen. From the eye tracking data collected in the experiment, we wanted to look for any specific parts of the information the developers would rely on the most to understand and summarize an API element. Analyzing their eye movements on the types of information and the time length of their gazes, we were eager to look for any patterns or strategies that are usually followed [1].

1.4 Experiment Definition and Planning

The primary objective of this study is to explore the distinctive procedures practiced by the software developers while summarizing API components utilizing different software artifacts. The information source consists of formal, informal documentation and combination of all. The quality concentration is program comprehension of API components outlined utilizing diverse sorts of data. From the developers point of view, the experiment can offer insights into how developers comprehend distinctive API components utilizing code and informal documentation to enhance appropriate tool support based on our discoveries. The participants involved in the experiment are both professional and unprofessional developers with some decent experience in the software development field. The items are API components, and various kinds of data (e.g., source code, bug reports, developer forums, or the combination of all).

1.4.1 Experimental Tasks

Participants conducted tasks using an eye tracking device. They were requested to summarize a set of four API components. Each participant was given a sequence workflow in iTrace. And each sequence had four tasks. One for summarizing an API using source code, using Stack Overflow, using Bugzilla and using combination of all the three sources. The programming interface components comprise of classes, methods, factors, and so forth. When leveraging source code as provided information, participants would look for the API components utilizing the Eclipse search where they have the source code of the ventures being shown. When utilizing documentation from Stack Overflow and Bugzilla, they are provided with an entry to a link where the documentation can be accessed. For instance, a link to the Stack Overflow post. The eye tracking device records the eye

movements during each task and stores them. The data collected from the experiment will be analyzed to see the results. Before starting the experiment, participants had to fill a pre-questionnaire which outlined their familiarity with APIs, Stack Overflow and Bugzilla. The nature of the final synopses of the developer performance in summarizing APIs was assessed against an oracle summary that was developed by combining three annotators' summaries. At the end of the experiment, the participants were asked to fill in post-questionnaire form on their feedback on the tasks and experiment.

1.4.2 Requirements for the Experiment

In order for our experiment to be successful, we had to establish a reliable environment for it. This means we had to follow a number of requirements for installing, configuring and testing the iTrace platform for collecting the developers' eye movements. Though the entire experiment was planned to run in Eclipse, we took many steps and considerations to make the Tobii eye tracker be compatible with iTrace and to prepare it for collecting the eye tracking movements, as well as to save the text summaries written by participants on the task page [1].

Most critical requirements for the experiment are as follows

- Eclipse IDE for Java EE Developers [4]:
  We used the Eclipse Neon version in this study
- iTrace Plugin [1]:
  After installing Eclipse, the plugin can be downloaded from the update and install section
- Java development kit (JDK) [5]:
  The version used in the study is 8.0.11, same version of JRE kit too
- Apache IvyDE Eclipse Plugin [6]
- Java native interface (JNI) headers [7]
- Microsoft Visual Studio [8]
- Tobii EyeX eye tracker [9]
- Camtesia [2]

For building EyeX tracker JNI extension, one would need to install the following:

- MinGW32/MSYS for Windows [10]
- CMake [11]
- JDK [5] for my platform and architecture
- Tobii EyeX Engine v1.4 and the Tobii EyeX Controller Drivers [9]
- Tobii Gaze SDK v4.1 [9]

1.4.3 Participants

Initially, we planned to invite only professional developers with various work experience as study participants. However, we faced troubles recruiting professional developers for our study as the remuneration was limited to 20 CAD Starbucks gift card. Each experiment took about 45 to 50 minutes to complete and there were very few who agreed to voluntarily take part in the study. We then expanded our recruitment to consider software developers from within school who were at the Bachelor's, Master's or PhD level. We published our recruitment advertisement through the School of Science email portal and received a number of replies from students who were willing to take part in the study. A few of the PhD participants had some previous work experience. A number of professional developers who learned about the study through the world of mouth took part in the study over the weekends.

1.4.4 Data Collection

The study we conducted was a controlled experiment. The participants had to be physically present in the lab where the experiment was setup on a monitor screen fitted with an eye tracker. All the data was collected while participants performed a set of tasks. The process we followed was as follows. Once the participant arrived, she was given an Information form explaining what the experiment is about and what kind of tasks she is expected to conduct. Each participant was also given a short demo of using the iTrace plugin and the Tobii eye tracker. During the experiment, each participant had to complete four tasks for each information source: source code, Stack Overflow, Bugzilla, and a combination of all, respectively. Participant's answer (short summaries) was collected for each task and was saved under the work space for each participant. For instance, data from participant "A" was saved in work space folder "A". This way, the output data was exclusive and did not lead to any confusions. The data from eye tracker was also collected and saved in a folder under four separate folders for each of the tasks. Later, all the data was manually moved to work space folder designated for each participant to keep both summary data and eye tracker data together for the analysis.

1.5 Research Questions

Many qualitative and quantitative studies related to API elements and source code summarization have been conducted. But only a few works are available combining them with eye tracking and performing it as a controlled experiment due to high costs (such as time) associated with conducting eye tracking studies. Our research includes a quantitative analysis of the data and an explanatory survey about the use and effect of software artifacts on code summarization. The existing state-of-the art research lacks evidence on how different complex artifacts affect a developer's understanding of source code and their performance of change tasks. Therefore, in this thesis, we state and address the following *research questions*.

RQ1 : **What information sources do developers leverage when summarizing various code elements?**

When different sources of information such as source code, informal documentation from portals like Stack Overflow and Bugzilla, and a combination of all the sources are provided to software developers, do they prefer any specific source of information over other and do they follow any pattern while summarizing API elements? Do they focus on any specific elements in the information source? And what source of information is mainly considered by the developers with different experience/familarity levels? Analysis of the collected developers' gaze data can provide more information regarding these questions.

RQ2 : **How do developers perform when using different sources of informal documentation to summarize the purpose and usage of code elements?**

After developers submit their summaries, these summaries are compared with an oracle. The oracle summary was created by three different human annotators who utilized various sources to provide the most comprehensive summary or definition of an API method or class. Based on the separate summaries of the annotators, the oracle summaries was developed. The developer summaries are compared with the oracle to determine the similarity between the two. We considered the cosine similarity as a metric for similarity to assess the developer performance.

RQ3 : **What is the developer's perception on leveraging different sources of information to support code summarization?**

The first two research questions aim at analyzing the quantitative data. Yet, we also wanted to study the developers' perception on what informal documentation they find most useful for code summarization. To do so, we asked the participants to complete pre- and post-questionnaires during the experiment. Based on

their responses, we can determine what developers actually think about the tasks and whether certain information sources are more helpful to them with API comprehension tasks. How did they feel about each information source and whether they find them useful in summarizing certain APIs? Which information source did they find to be more useful? Answering these questions can help us understand developer perception on the code summarization when leveraging various complex software artifacts.

1.6 Thesis Contributions

This thesis makes the following contributions:

1. We conducted a controlled experiment with 12 software developers trying to understand how they would summarize API elements leveraging different software artifacts.
2. We leverage eye tracking technology to collect detailed developer eye movement data to better understand developer's behavior, strategies and specific area of focus in each given information source.
3. By conducting various analyses, we provide evidence on how software developers leverage formal and informal documentation during code summarization tasks.
4. With the analysis of eye tracking data we found the main area of focus in each information force.
5. We identified the gap between developer actual behaviour and focal points when they perform API comprehension tasks and their perception on the benefits of informal documentation.
6. We report the setup of the eye tracking experiment setup and execution for future replication studies that utilize an eye tracking technology.
7. Our key findings are as follows:
    - Developers mainly rely on informal information source like Stack Overflow posts during API summarization tasks (RQ1).
    - The accuracy of summarization tasks depends on the difficulty and developer familiarity with a specific API (RQ2).
    - Developers use and focus on specific elements in each information source to support their API comprehension (RQ2).

- Developers perceive Stack Overflow discussions as their preferred choice of informal documentation during API comprehension tasks (RQ3).
- There is a gap between the developers' perception about information sources and the results acquired from the eye tracking data analysis.

1.7 Organization of the Thesis

Chapter 2 presents the background and discusses the related work in the field of API summarization and eye tracking. In Chapter 3, we explain our methodology, including the use of iTrace for collecting eye gazes, different metrics in eye tracking, as well as the setup of our controlled experiment. In Chapter 4, we present the different analyses performed on the collected data. We present the results in Chapter 5. Chapter 6 discusses implications of this work for developers and researchers, and proposes future directions in extending this work. And finally, Chapter 7 concludes the thesis.

Chapter 2

Related Work

This chapter presents the background of existing qualitative and quantitative research in various areas like API summarization, iTrace, and eye tracking (Section 2.1), as well as the related work summarizing different approaches followed by researchers and practitioners to address these topics (Section 2.2).

2.1 Background

APIs form an integral form of an application or software. How would developers try to understand these APIs given different types of information is our major focus. There are not many previously suggested techniques to summarize an API. Most common approach for understanding a certain API is to examine its code. While performing such tasks many factors have to be taken into consideration such as an API definition, the software application it is a part of, developer's knowledge about the application and API, etc. Considering these factors, the main idea is to study how different developers leverage the different sources of formal and informal documentation when summarizing and understanding various API elements.

The field of eye tracking has been emerging within the past decade. But there are many challenges when it comes to collecting eye tracking data and understanding it. There are wide ranges of eye trackers from simple to complicated and advanced ones which are being used for different studies. Eye tracking also has a wide range of metrics. Research community uses various metrics for analyzing and validating eye tracking data. Current research suggests various metrics and best practices surrounding eye tracking technology to be used in the field of software engineering. Studies involving API summarization together with eye tracking have not been previously performed to the best

of our knowledge. Therefore, our study offers novelty in combining these two research areas.

iTrace plugin [1] was introduced in Eclipse to support eye trackers and collect eye tracking data. But when it comes to making it compatible to the type of eye tracker we use, there are many additional steps to be performed. Though there is much information available on iTrace, when it comes to using different types of trackers for data collection, the available documentation is sparse. It is more of a trial and error method when adopting iTrace to work with a specific eye tracker.

Next, we would be presenting different works on API summarization, iTrace, eye tracking and research works involving the combination of these.

2.2 Related Work

Researchers have been conducting eye gaze investigations of developers for quite a while. A large number of the prior studies focused on program and code comprehension [12],[13],[14].

Then on, researchers began to physically catch more point by point information on engineers' activities. Altmann, for example, broke down a ten minute interim of a specialist software engineer playing out an undertaking and utilized computational reenactment to contemplate the close term memory [15]. Maybe a standout among the most surely understood client contemplates from this class is the investigation by Ko et al. [16]. In this investigation, the authors screen caught ten engineers' work areas while they took a shot at five assignments on a toy-sized program and after that hand-coded and broke down every 70 minute session.

In an examination on designers performing more practical change tasks, Fritz et al. [17] utilized a comparable method and physically deciphered and coded the screen caught recordings of all participants. These examinations are a profitable wellspring of learning and prompted intriguing discoveries, but the cost of hand-coding an engineers' activities is high, which prompted just a predetermined number of studies giving definite bits of knowledge on a designers' conduct.

Lately, approaches have been created to consequently catch developer interaction information inside an IDE, such as Mylyn [18], [19],[20], [21]. In light of such naturally caught connection historieslogs of the code components an engineer interacted with alongside a timestamp, researchers have, for example, explored how designers work in an IDE [22],

how they explore through code [23],[24],[25], or how designers' small scale association examples may be utilized for imperfection forecast [26].

Indeed, even the Eclipse group themselves attempted a noteworthy information accumulation venture called the Usage Data Collector that, at its pinnacle, gathered information from a large number of engineers utilizing Eclipse. The programmed observing of client communications could essentially decrease the cost for certain observational examinations. These examinations are constrained to the granularity and detail of the observing methodology. In the event of client collaboration observing, the granularity is predominately the technique or class document level and itemized data, for example, the time a designer spends perusing a code component or when the engineer isn't taking a gander at the screen, is missing and makes it harder to completely comprehend the designers' trace.

In parallel to the IDE, scientists in the software advancement area have additionally begun to exploit the developing of biometric sensors. Most of this exploration concentrates on eye-tracking [27],[28], while just couple of studies have been led so far that additionally utilize different signs, for example, a fMRI to distinguish cerebrum initiation designs for little perception undertakings [29], or a mix of eye-following, EDA, and EEG sensors to quantify viewpoints, for example, errand trouble, engineers' feelings and advance [30], [31], [32].

By utilizing eye-tracking and naturally catching where a developer is looking, researchers could increase further bits of knowledge into engineers' code perception. One of the main eye-tracking examinations in program comprehension was directed by Crosby et al., who found that specialists and amateurs vary in the way they took a look at English and Pascal renditions of a calculation [33].

From that point forward, a few scientists have utilized eye-following to assess the effect of engineers' eye stare on perception for various types of portrayals and representations, for example, 3D perceptions [34], UML graphs [35],[36], plan design format [37], programming dialects [38], and identifier styles [39].

Scientists have additionally utilized eye-tracking methods to examine engineers' scan pattern for little code pieces, finding that members first read the whole code scrap to get a thought of the program [40]. Different researchers analyzed diverse procedures beginner and master engineers utilize in program cognizance and investigating [41], as well as where developers invest most energy when perusing a strategy to devise a superior technique synopsis system [42]. Finally, researchers have likewise utilized eye-following to assess its potential for identifying programming traceability joins [43],[3], [44]. These investigations are restricted to little, toy applications or single page code undertakings.

Moreover, in a significant number of these investigations, the interface between the eye stare to the components in an IDE must be done physically.

To the best of our knoweldge, our work exhibits the primary study on sensible change undertaking task that gathers and dissects both engineers' client connection and eye gaze information. Since our approach connects eye gaze information to the fundamental source code components in the IDE, we diminish the need of manual mapping and can conquer the single page code task impediment of past studies. Taking into account change tasks on a sensible measured code base with developers having the capacity to normally scroll and switch editorial manager windows.

Eye tracking research is still being advanced. Researchers contemplate eye-gaze information as for specific zones on the stimuli called Areas of Interest (AOIs) [45]. An AOI can be applicable to the right answer required from the participant playing out a task or can be superfluous. For instance, if we consider a class graph as a stimulus, an unimportant AOI can be any class or any documentation, while an important AOI could be the particular class that is pertinent to the given task. For source code, it could be any source code component, for example, a strategy call or an identifier.

Several past eye-tracking research studies concentrated on the understanding of UML class diagrams concerning plan designs[36], [46],[47], the effect of format [35], [39], and the effect of aptitude [48]. Petrusel et al. [49] concentrated on the comprehension of business process models (BPMN), while Cagiltay et al. [50] performed non-formal examinations of element relationship charts (ERD). Moreover, Sharafi et al. [51] explored the effectiveness of graphical portrayals over printed ones in demonstrating and showing programming prerequisites exhibited by TROPOS demonstrating system.

A greater part of past studies concentrated on the effect of ability on understanding and review procedures while looking at a source-code with natural text [33],[52]. The authors detailed that source-code reading is extraordinary from content reading [53] and that developers have higher obsession time and relapse rate when looking at source code over content [54]. They also revealed that amateurs spend more visual consideration on remarks than specialists, while specialists spend additional time on pertinent AOIs [33],[55]. Busjahn et al. [56] played out the primary eye-tracking study that adjusted a few linearity measurements used to investigate natural text reading to source-code perusing. Kevic et al. [57] performed the main eye tracking investigation on huge source code records in an open source framework utilizing the Eclipse module iTrace [1]. iTrace naturally maps eye developments to source code components while keeping up setting amid looking over.

A few past research works concentrated on investigating code comprehension tasks [39],[41],[38]. Turner et al. [38] examined the effect of programming dialects (C++ versus Python) and revealed that the programming dialect impacts the visual exertion spent by beginners contrasted with specialists while working with carriage lines of code.

Some past studies dissected the effect of programming representation on code comprehension techniques [58] and investigating [41], [59]. They detailed that, by helping developers to discover important AOIs and their conditions, perceptions guided members to take after more precise systems [58]. Moreover, their outcomes demonstrated that members with higher execution essentially utilize graphical portrayals in spite of the fact that they additionally utilized content from time to time [41]. developers with bring down execution performed consideration exchanging as often as possible [59].

The effect of identifier styles on code perception has additionally been explored [53],[39],[60]. No distinction with respect to precision, time, and exertion for perception tasks were accounted for[39], [60] yet expertise diminishes the effect of identifier styles [39], [60]. Fritz et al. [30] joined student widening, squint rates, electrodermal movement, and EEG (electroencephalogram) to describe and foresee difficult tasks.

All these past eye-tracking investigations utilized distinctive measurements and metrics to process visual exertion for particular tasks. No study compresses and gives a rundown of every single accessible metric along with point by point definitions and recommendations on the best way to gauge visual exertion and utilize the measurements. Furthermore, several researchers utilize a similar metric but, name it in an unexpected way. For instance, while examining the conveyance of visual consideration on various code components, all of the past examinations [33], [52] thought about each code component as an AOI and computed the totals of all fixation spans for each AOI yet called this metric in various ways: fixation time [33], [52],[41], collected fixation time [54], add up to time of fixations [61], or add up to abide time [52]. On the other hand, a similar name may allude to various measurements. Gaze time is either defined as the entirety of all fixation terms for an AOI for the entire investigation [34] or then again is characterized as the entirety of all fixation terms [52].

Hence, such ambiguities make correlation and replication of past research studies troublesome. This makes it significantly more essential to elucidate and standardize the names and meanings of eye tracking measurement metrics. In our work, we tried to convey attention to these ambiguities and prepare towards future metric standardization.

Several scientists have been working towards natural language processing and summarization [62]. Some research has concentrated mostly on the source code while summarizing the code. For instance, Murphy [63] recommended synopsis strategies to blend

the basic data of source code. Haiduc et al. [64] utilized the lexical and basic data of code and connected content recovery strategies to compress Java classes. Moreno et al. [65] produced Java class synopses utilizing the substance and duties of the classes rather than their associations with different classes. Sridhara et al. [66] produced Java strategies outlines in light of catchphrases from the most significant strategy's announcements, while McBurney et al. [67] proposed a system that breaks down how Java strategies are summoned to create documentation.

Other research has focused on summarizing documentation. For instance, Rastkar et al. [68] abridged bug reports in view of machine learning procedures. The promising consequences of their investigation inspired us to consider bug reports while summarizing code components. Furthermore, Moreno et al. [69] have investigated summarization of software release notes.

Recently, Guerrouj et al. [70] proposed a model to summarize code components in Stack Overflow. They researched the utilization of the setting that encompasses code components in Stack Overflow posts for rundown. Additionally, Treude and Robillard [71] proposed a machine learning-based approach to deal with naturally increased API documentation with experiences from Stack Overflow about a specific programming interface. We share with these works the idea that code summarization is a non-trivial task. Nonetheless, unlike previous research, we examine the problem through a controlled experiment with developers, studying the impact of four complex artifacts on code summarization.

iTrace has been recently developed as a plugin for Eclipse. Eclipse is a platform to execute wide variety of Java programs. There have been changes done to the plugin to make it more compatible to all the variety of codes. Shaffer et al. [1] have explained the architecture of iTrace, its various features and how it can support eye movements captured from different developers. They also described different usage scenarios and showed its flexible nature with eye movement developments.

Recently, Sharif et al. [72] addressed techniques and methods for using iTrace with much guideline on how researchers can conduct the studies on their own. Eye tracking investigations are conducted on a solitary screen of content and there is no support for looking over or exchanging between documents. This situation is unrealistic for engineers as they move towards bigger programming interfaces. To conquer the Eclipse module, iTrace screens developer eye developments even in the nearness of looking over and document exchanging inside an IDE. In expansion, it naturally maps the eye gaze to source code components.

JB Won et al.[73] have tried to compare the total aberrations measured by iTrace over OPD scan, which involved laser technology. iTrace when leveraged much with an advanced eye tracker, performs really well with the change tasks and the results are mostly apt. However, for high level eye tracking studies, other technologies are used, while for tasks involving code comprehension iTrace has been demonstrated to be a well suited platform.

Source code summarization is an emerging topic for producing brief summaries of the source code. Rodeghero et al. [42] offer a good explanation on this. Current synopsis strategies work by choosing a subset of the announcements and catchphrases from the code, and after that including data from those announcements and keywords in the code. The nature of the code depends intensely on the way toward choosing the subset: a superb choice would contain similar proclamations and keywords that a software engineer would pick. Lamentably, little proof exists about the announcements and watchwords that software engineers see as critical when they summarize source code.

Code summarization is an essential source for understanding inquiries regarding programming libraries and applications. Numerous use settings for code illustrations expect them to be refined to their quintessence. Ying et al. [74] led an investigation to find how code can be summarized and why. As a major aspect of the investigation, they gathered 156 sets of code cases and their synopses from 16 developers, alongside more than 26 hours of verbally process verbalization specifying the choices of the developers amid their outline exercises. A source of practices, took after by the developers to summarize code and propose experimentally and upheld theories legitimizing the utilization of particular practices. One fundamental finding was that none of the members only separated code verbatim for the codes, inspiring abstracting outline. The outcomes give a grounded premise to the improvement of code summarization synopsis and introduction innovation.

Another interesting work was performed by Sharif et al. [43] who conducted an eye tracking experiment to study software traceability tasks. The work advocates for the enlistment of eye tracking innovation in programming traceability and takes a position that the utilization of eye tracking measurements can add to a few programming traceability undertakings. They suggested that the part of eye tracking isn't just confined to an instrument for experimental studies, yet additionally could reach out to giving a foundation of another product traceability philosophy. They introduced a few situations where eye-tracking measurements could be important. The particular research headings incorporate directing observational investigations with eye-gaze measurements and imitating already detailed empirical examinations, eye-tracking empowered traceability to connect administration strategy and visual bolster.

Eye tracking movements can be collected when combining other tools too. Some of the research has been done on languages like MATLAB. Frans et al. [75] empower experimenters to gauge eye developments while at the same time executing the stimulus introduction schedules given by the Psychophysics Toolbox. Illustration programs are incorporated with the toolbox conveyance. Their work explains the usage of software link tool box and eye tracking combination.

Rayner in his investigations of eye developments [27] considered other data processing assignments, for example, music reading, writing, visual hunt, and scene observation. The real accentuation of the audit is on reading as a particular case of psychological preparing. Essential subjects examined regarding reading are a) the qualities of eye developments, b) the perceptual traverse, c) combination of data crosswise over saccades, d) eye development control, and e) singular contrasts (counting dyslexia). The essential subject of the review is that eye development information reflect minute-to-minute psychological procedures in the different tasks analyzed. In addition, they discussed hypothetical and functional works concerning the utilization of eye development information.

In our work, we utilized the state-of the-art research on eye tracking (in particular, iTrace) and code summarization. The main focus of this work is to investigate the effect of different kinds of information on code comprehension tasks by utilizing the eye tracking technology.

Chapter 3

Methodology

In this chapter, we describe the outline and details of the controlled experiment (Section 3.1 and Section 3.2), iTrace design, architecture and implementation (Section 3.3), software traceability in iTrace (Section 3.4) and advantages of iTrace (Section 3.5). We also discuss about the features of Tobii EyeX eye tracker (Section 3.6), selection of APIs for the study (Section 3.7), how we build the API oracle (Section 3.8) and use of Camtasia for screen capture (Section 3.10). We also explain the experiment workflow (Section 3.11) and challenges faced (Section 3.12).

3.1 Experiment Outline

The research study was planned to take place in different stages. Initial stage would be setting the platform for the experiment to take place by putting the technical requirements together. These requirements consist of 1) setting up and configuring iTrace, and 2) setting up a Tobii EyeX Eye tracker. Once the necessary steps are taken to make iTrace and Tobii Eye tracker compatible, we checked for data collection by running a trial session of iTrace. Once we got confirmed that the requirements were in place, work spaces were created for each participant. This was followed by recruiting the study participants. We put out advertisements about our study in the University Board and a few Slack channels accessed by software developers. Interested developers were then scheduled for participating in experiments. Participants were provided with quick demonstrations about the study and steps they would be following while performing tasks. The participants were also given written instructions to go through and the experiment was started only once the participants were comfortable about the steps.

During the experiment, the data for each participant was collected and stored in a separate folder and was later used for the analysis. The detail steps for each stage are explained in further sections.

3.2 Controlled Experiment

To answer the research questions we conducted a controlled experiment. The main goal of the experiment is to study developers' performance for a given task of API summarization using formal and informal source of information.

3.2.1 Goal and Objective

The objective of our experiment is to test and measure the impacts of four types of information sources such as source code, Stack Overflow, Bug Reports, and their combiination on code summarization [70]. The quality concentration is the performance of developers estimated regarding accuracy of their summaries with respect to the API oracle summaries. The viewpoint is better comprehension of how informal documentation (Stack Overflow, bug reports) can be utilized to summarize classes and methods and uphold official documentation with the bits of knowledge extracted from additional sources.

3.2.2 Study Participants

For the study we recruited 12 software developers. We had to undergo Carleton University's research ethics approval in order to proceed recruiting the developers and perform the study. Once we received ethics approval, we initiated the recruitment of software developers. We approached several professional developers from different local Canadian companies by reaching out to them through Slack channels and advertisements (i.e., study information letters). We also posted an advertisement in the Carleton University's Graduate office board, as well as sent out emails to all the graduate and undergraduate students of the School of Computer Science. We also used face to face advertisement by meeting with graduate students at the university lab explaining them the study goals. We also asked many fellow graduate students and acquaintances to spread the word about the study. Several developers got in touch with us, and we finalized 12 participants for the study. Software developers were asked to dedicate around one hour of their time to be able to participate in the study and as a token of appreciation a remuneration of 20 CAD Starbucks gift card was given to every developer who took part.

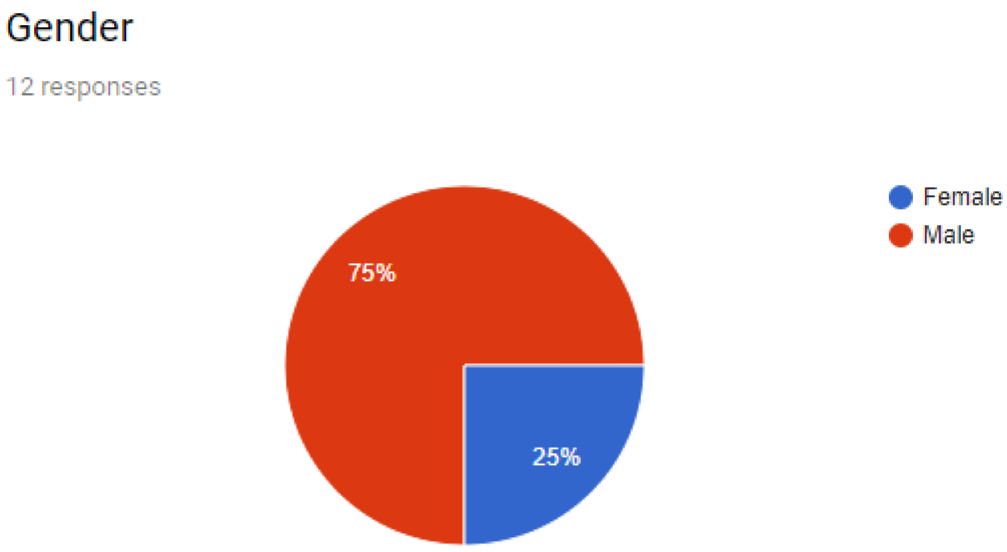

FIGURE 3.1: Participant's gender.

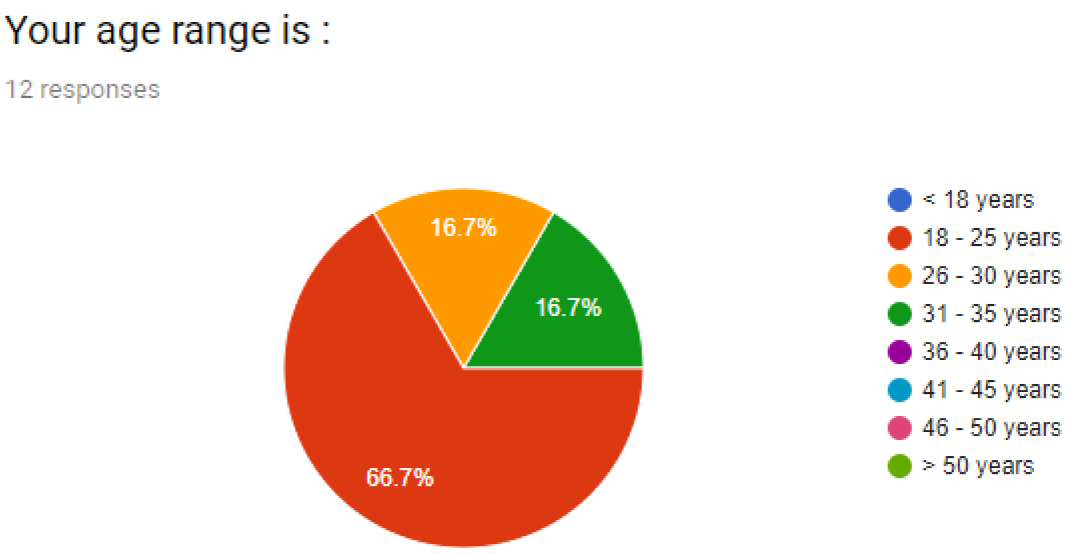

FIGURE 3.2: Participants' age category.

Participants had to answer a pre-questionnaire where they filled out demographics details and how familiar they were with different aspects of software development. Based on their responses, Figures 3.1, 3.2, 3.3, 3.4, 3.5 report statistics about the participants' gender, age, education, programming and development experience, respectively.

From the developers who agreed to take part in the study, the majority of them were male. As the ratio of male developers to female developers in the industry is pretty high, there were few female developers who took part in the study.

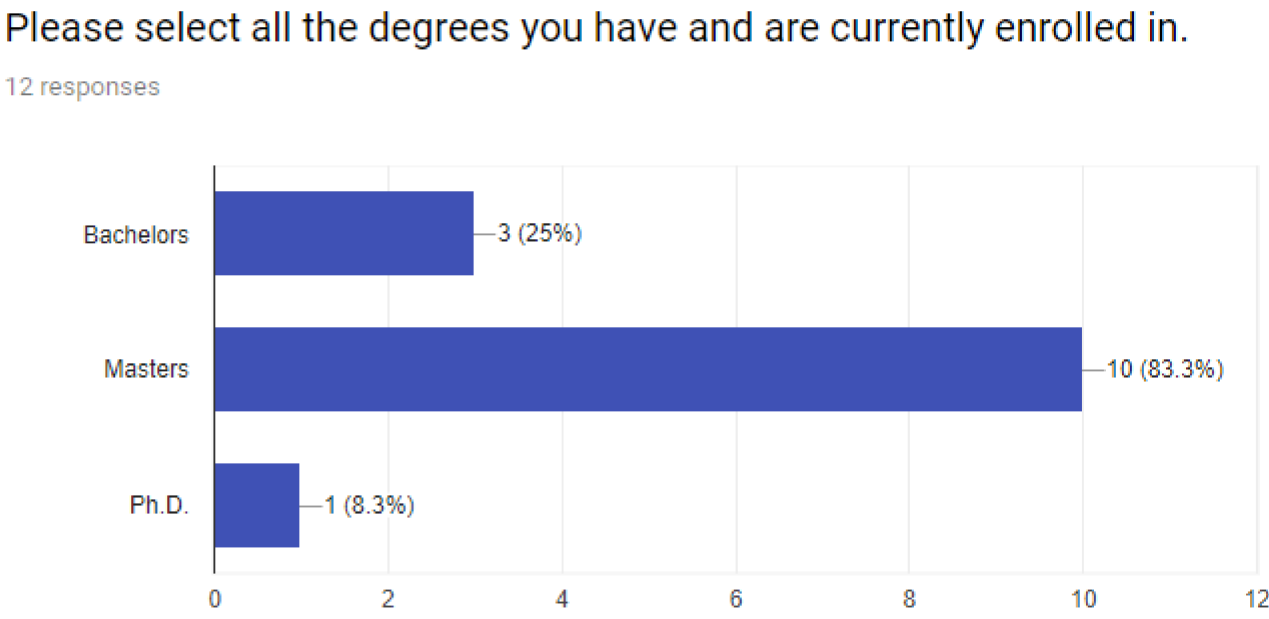

FIGURE 3.3: Participants' education.

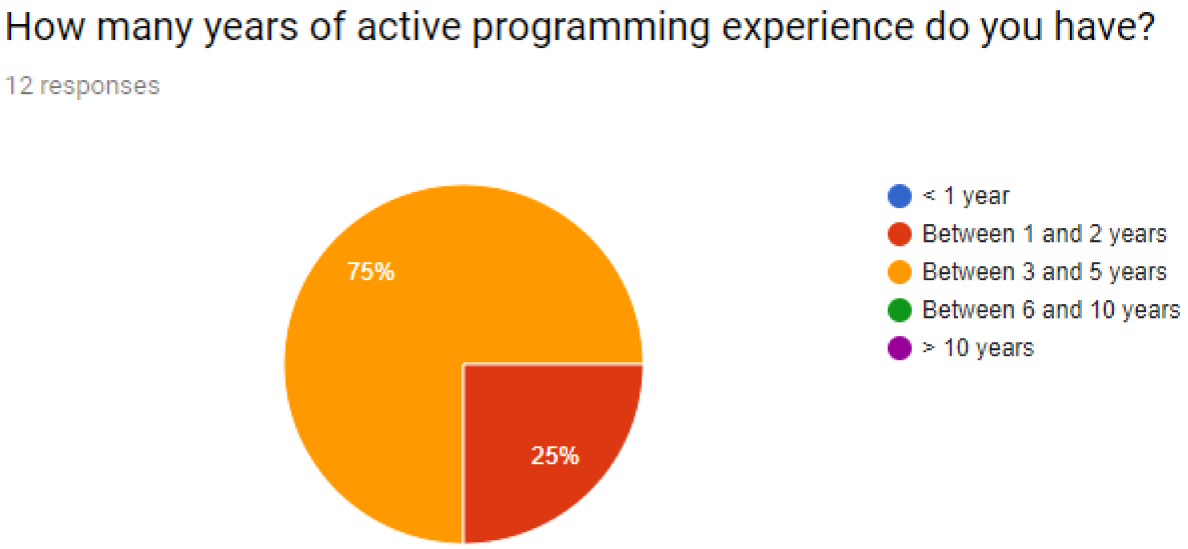

FIGURE 3.4: Participants' programming experience.

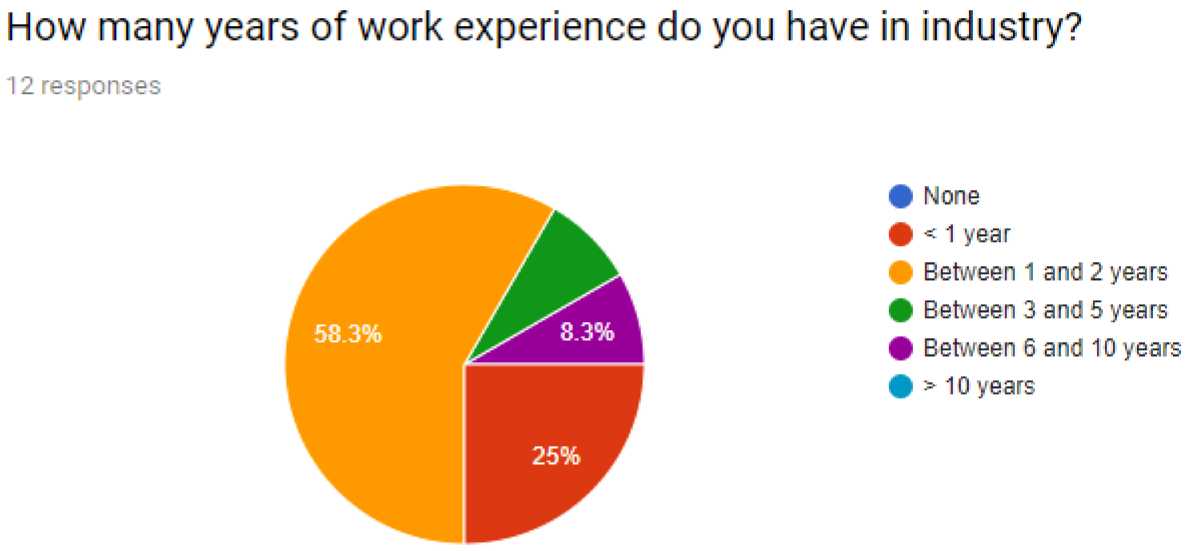

FIGURE 3.5: Participants' industry experience.

Around 66% of the participants were between 18 to 25 years of age group and with around 17% of them between 18 to 25 and 26 to 30 years old. Most of them had Master's degree with around 3 to 5 years of programming experience and around 2 years of industry experience. Based on these demographics, we were satisfied with the recruited participants and their programming experience.

3.3 iTrace Plugin Design and Implementation

In this section, we provide the description about iTrace, its architecture and features, as well as how it has been used in our study.

iTrace [1] is an Eclipse plugin which is available for downloads. It is used by researchers to conduct studies involving eye tracking techniques in software systems. In our study, iTrace collects eye movements from software developers while they work on change tasks. It is designed to support a wide variety of eye trackers. iTrace is also very flexible and user friendly and supports many software elements like Java, HTML, XML and many more. The plugin has been effectively utilized for programming traceability tasks and program comprehension studies. iTrace is additionally pertinent to different tasks, for example, code synopsis and code suggestions in view of a developer's eye developments.

Recently, the research community observed a number of studies involving eye trackers. There are wide variety of eye trackers in all price ranges. When collecting participant's eye movements for code comprehension tasks, one must decide on the granularity of the

data to be collected. Even if there is a switch between screen, scrolling or changing to any artifacts, the eye movements should be accurately collected. Keeping in mind the end goal of running sensible examinations with eye trackers in software engineering, we should have the capacity to run eye tracking investigations on big frameworks comprising of several source records and not be restricted to a solitary static view. iTrace was composed precisely for this reason. There is no requirement for the researcher to physically delineate directions to source code components as the majority of this tedious work process is handled by iTrace. The plugin runs continuously inside Eclipse, recording engineers' eye movements while they are working in the IDE.

3.3.1 iTrace Architecture

We now elaborate more on the iTrace module architecture integrated into the Eclipse user interface. The present architecture handles gazes on Java source code, documents like HTML and XML, and Eclipse UI components. iTrace offers capabilities to compose new handlers for various tasks to gather fine-grained information.

Empowering eye tracking for the IDE requires executing three major states. These are 1) catching a participant's gaze from the eye tracker, 2) figuring out which UI component the client is taking a gaze at inside the IDE, and 3) handling this data towards some practical objective. These assignments must be done in parallel to achieve best results.

To achieve this, here is a multi-threaded configuration comprising of a thread for each task that speaks with different threads by means of shared blocking queues. The design of iTrace influences utilization of a `Gaze object class` and three principle Java interfaces for working with gazes: `IGazeHandler`, `IGazeResponse`, `ISolver`. The definition of each is as follows:

- `Gaze` [1]: conveys position, time, pupil size, legitimacy, and error data for each look identified by the eye tracker.
- `IGazeHandler` [1]: explains a handler that acknowledges a look on a gadget and returns an `IGazeResponse`.
- `IGazeResponse` [1] : Describes data seen by a particular gaze on a particular gadget.
- `ISolver` [1] : Describes what to do with a particular `IGazeResponse object` record.

FIGURE 3.6: iTrace design.

Eye Tracker objects are associated with and read from an eye tracking gadget. The Tobii Tracker usage utilizes the Java Native Interface (JNI) to execute C++ code interfacing with the gadget's SDK. An unadulterated Java approach can be taken for gadgets with which Java can straightforwardly interact. As gazes are received, iTrace makes new Gaze objects from the gaze area and timestamp, leaving the reaction field hollow. iTrace gives the capacity to join IGazeHandler articles to any Eclipse gadget, which get a Gaze question and restore an IGazeResponse. A part of class diagram depicting iTrace design is shown Figure 3.6.

Every gadget compose has its own specific gaze handler and reaction. A handler's granularity relies upon its motivation. In Eclipse, the code manager window has a gaze handler that catches the substance of the line being looked at by the participant and in addition the names of method and class to which that line belongs to.

Each gadget progression is navigated to discover the internal child that contains the directions of the eye gaze. This empowers us to get fine-grained traceability joins at the

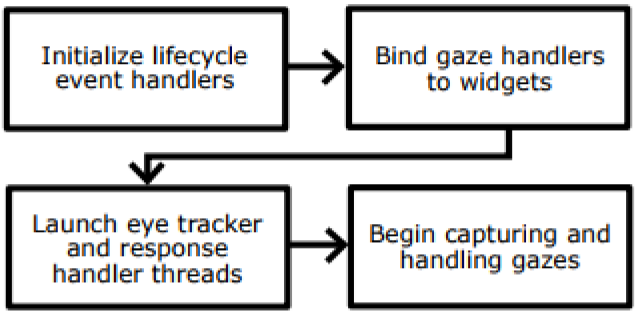

FIGURE 3.7: Plugin initialization process [1].

class and method level in view of the client's eye gaze. The GazeRepository stores gazes until the point that connections are produced. The LinkRepository stores interfaces that are created. To depict what was explained previously through a diagram, Figure 3.7 can be referred to. How the iTrace plugin is initialized and the flow chart of handling multiple threads are described in the following section. We also provide explanation about its integration with Eclipse and various other features.

3.3.2 iTrace Integration with Eclipse

After describing iTrace architecture, we now explain integrating iTrace with Eclipse. Eclipse plugins are actualized in Java. The Eclipse IDE is comprised of various parts such as views and editors [3]. Views refer to the windows that give prompt data or control such as a package explorer, while editors refer to the windows that give control of substance that must be saved, for example, the source code editor. Each view part contains a few gadgets that show content or encourage user interaction. iTrace empowers eye-tracking on self-assertive gadgets inside the two perspectives and editors.

To implement iTrace as a module inside the Eclipse there is utilization of the class - `org.eclipse.swt.widgets.Widget`. This class speaks to every UI object that is a piece of the IDE, and shows some substance being seen by the participant. It also uncovers two main methods `getData()` and `setData()` [1] which are used to tie and access gaze handler objects with the end goal that each UI component with substance of intrigue has its own particular `GazeHandler` occurrence.

The lifecycle handlers control tasks, for example, stopping look preparing when the IDE loses the focus or the participant minimizes the window or does any such change. Eye trackers are actualized in the module by executing the `IEyeTracker` interface and changing the eye tracker class. In Figure 3.8, we see the flow chart illustrating the

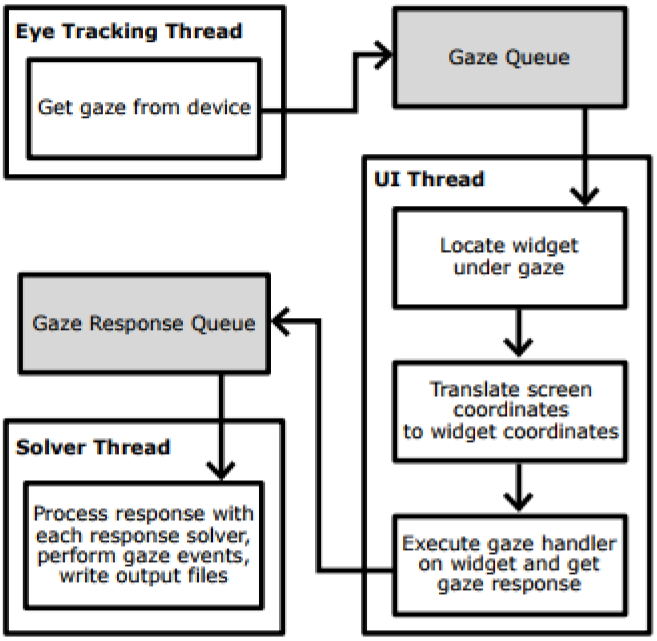

FIGURE 3.8: Handling eye gaze responses from multiple threads [1].

utilization of the Java Native Interface (JNI) to interface with the eye tracker SDK in C or C++.

3.3.3 Session Creation

In iTrace, it is possible to create a new session every time we turn on and turn off the eye tracker. This makes it possible for us to have the data separated to support further analysis. For every new session created, there is a unique session ID, session description, developer ID and developer name fields available that can be edited. Below is an example of how a newly created session looks like:

```
{
    "session_info": {
    "session_ID": "20171112T190404-0307-0500",
    "session_purpose": "Other",
    "session_descrip": "",
    "developer_username": "IDC1",
    "developer_name": "abc"
  }
}
```

```
  <?xml version="1.0" encoding="UTF-8"?>
- <itrace-records>
  - <environment>
      <screen-size height="900" width="1600"/>
    </environment>
  - <gazes>
    - <response line_base_y="590" line_base_x="309" col="0" line="23" font_height="14" line_height="22"
      path="C:\CodeSummarizationStudy\CodeSummarization_Study\_tasks\Sequence 2\T1-ALL.txt"
      nano_time="263333456729179" system_time="1510531508451" tracker_time="263333444"
      session_time="206341026" timestamp="2017-11-12T19:05:08.451-05:00"
      right_pupil_diameter="4.7362097752887246E-138" left_pupil_diameter="0.0" right_validation="0.0"
      left_validation="0.0" y="595" x="309" type="txt" name="T1-ALL.txt">
        <sces/>
      </response>
    - <response line_base_y="590" line_base_x="309" col="0" line="23" font_height="14" line_height="22"
      path="C:\CodeSummarizationStudy\CodeSummarization_Study\_tasks\Sequence 2\T1-ALL.txt"
      nano_time="263333519910241" system_time="1510531508514" tracker_time="263333504"
      session_time="269522088" timestamp="2017-11-12T19:05:08.514-05:00"
      right_pupil_diameter="4.7362097752887246E-138" left_pupil_diameter="0.0" right_validation="1.0"
      left_validation="1.0" y="593" x="310" type="txt" name="T1-ALL.txt">
        <sces/>
```


FIGURE 3.9: iTrace gaze response.

3.3.4 Calibration

Prior to each experiment with a participant, the eye tracker needs to be calibrated. The participant is asked to look at certain points on the screen, while the eye tracker uses its model to calculate the gaze data. iTrace uses a nine-point mechanism for calibration. After calibration, the user will be prompted with the result, either success or failure. Based on this, the participant may proceed with the further steps of the study.

3.3.5 Data Export

The gazes recorded by iTrace rely upon the number of samples collected every second by the eye tracker. Each movement recorded by the eye tracker is utilized to produce a gaze response object. iTrace, as of now, converts collected gaze data into JSON and XML files which are saved in the folders with session IDs. It has fields like time, pupil diameter, validation, etc. The XML file or the JSON file can be used for data analysis and to find out where the participants are looking at during the task. There is almost an entry for each second, thus, the files can be large in size. Figure 3.9 shows how an iTrace response record in XML file looks like. In the figure we can notice that different gaze information fields like path, tracker time, system time, pupil diameter, file name, etc are shown.

3.4 Generating Software Traceability

The procedure of recording eye gazes should be possible by means of two utilization situations: task based and persistent [3]. In the task based mode, the tracking starts before a task begins and is halted when the assignment session closes. A task could comprise of more than one session. For instance, when a participant is gazing during a bug settling task, she may illuminate it over different coding sessions. iTrace can monitor these sessions alongside user information. The link generation algorithm at that point produces joins based on the pattern of communication between the user and the artifacts. iTrace can likewise be utilized as a part of ceaseless mode where the eye tracker is dependably gathering eye gaze information that are being connected with. This technique for utilizing iTrace can help in creating the repository of traceability joins, while a user is reading through the framework. The behaviour gazed would then be able to be mined to recommend applicant traceability joins. iTrace can likewise be valuable in catching detail behavior that can help beginners in figuring out the framework's capacity.

3.5 iTrace Advantages

iTrace has many advantages, including the following:

- It creates links as the user is performing and keeps records of the data.
- It is financially accessible. Eclipse is free and open source. Since links are produced by eye developments that the client should perform among the advancement tasks, extra time costs are relatively nonexistent.
- iTrace is easily configurable. It gathers and stores particular data about artifacts being seen like source code, software link, class and technique names, report file names. It is a trusted source. iTrace runs as an unpretentious observer, naturally creating and refreshing connections as the product experiences changes.
- Another characteristic is that it is adaptable. It is easily usable with many types of artifacts.
- Compact and brisk. Traceability connections and gaze data are stored away in a XML file, for reusing and sharing crosswise over ventures and additionally for further investigation.
- iTrace can be considered as Esteemed. We believe that as developers see joins based on their eye stare that is produced easily through an IDE, they will appreciate and generally expect such usefulness.

3.6 Tobii EyeX Tracker

The eye tracker used in the research study is Tobii EyeX Eye Tracker [9]. It comes in a decent packaging and easy to use. Installation package is available on the Tobii online website. The device(shown in Figure 3.10) has a rectangular strip with three focus points which record the eye movements. It comes with a USB cable and a small adhesive strip to stick on to laptop or desktop.

For installing the Eye Tracker Engine, we had to connect it via the USB port and run the installation wizard. The installation files can be found on the Tobii website. There are different installation files for different versions of EyeX tracker. We installed the Tobii EyeX. The files which we used for installation were "Tobii Eye Tracking Core v2 10.0.64". There were many available versions for EyeX. We tried with 5.0.55, and 7.0.4 versions too. After finding some problems with the performance, we decided to use the most recent version.

Once the installation was successful, the Tobii EyeX folder was installed in the suggested location. There is also a desktop shortcut available to launch the eye tracker. In order to launch the eye tracker, it has to be connected to the laptop or desktop through the USB port. Once launched, there are several features which can be tried on, for instance, eye placement, calibration. There is a trial game version where you can look at an asteroid belt that detects the asteroid based on the gaze. There are three red points on the eye tracker which glow indicating the calibration was successful and the tracker is ready to collect eye movements. Figure 3.10 shows how a Tobii EyeX tracker looks like.

In addition to installing and running the software, the eye tracker needs specific SDK files to make it compatible with iTrace. The SDK files are available for download from the Tobii website. The SDK files have to be downloaded and moved to the installation directory. Once the correct SDK files are present, we can run the iTrace program without any issues.

3.6.1 Building EyeX Tracker JNI Extension

Once the EyeX Tracker is fully set up we should build the JNI extension to make it detectable from iTrace. For this we are required to use a UNIX system or MinGW32/MSYS runtime environment for Windows. And the GNU Compiler Collection for C++ is required. Since we had Windows OS, we used MinGW32.

In addition to this, one needs to install CMake and JDK tools. The SDK files that were downloaded along with the eye tracker are critical. From the SDK files we need to

FIGURE 3.10: Tobii EyeX eye tracker device.

extract all libraries in CPP to our "MinGW lib directory" and all "Includes" in CPP to the MinGW `include/` directory. We also need to make sure it is the 32bit version.

Mingw32 is a complete runtime environment for GCC to support binaries native to Windows 64- and 32-bit operating systems. It is easily available for download and can be selected based on the operating system bit version. CMake is open-source software and cross-platform. It is free for managing the build process of software using a compiler-independent method. It supports directory hierarchies and applications that rely upon numerous libraries. Once both MinGW and CMake are in place, we can proceed further. Then, we will have to follow certain steps to build the EyeX tracker as listed below.

Figure 3.11 shows how the eye tracker can be attached to the laptop or desktop screen. It can be attached to either top or bottom of the screen.

Steps for building an EyeX Tracker:

- Create a directory named "build/" within the iTrace directory.
- Open Windows command line and enter the "build/" directory.
- Run CMake and the following commands:

  `cmake .. -G ‘‘MinGW Makefiles’’ , mingw32-make`
- One may need to use CMake's -G flag to specify what type of build files to create.
- Copy the .so/.dll file created by make to the project root as `libEyeXTracker.dll` on NT systems.

Once `libEyeXtracker.dll` file has been created, the eye tracker should be detected by iTrace and is ready to be used in the study. After this we made certain trial runs to check whether the data is being collected and saved within a normal Eclipse work space.

FIGURE 3.11: Tobii EyeX on the desktop screen.

3.6.2 Advantages of Tobii EyeX

There are many advantages of Tobii EyeX when compared to other Tobii trackers.

- It is cost effective. Unlike other advanced eye trackers which are very costly.
- It is light weight and portable. As it is made of lightweight plastic, it can be carried easily from one place to another.
- It is easily scalable. The eye tracker can be connected using the USB port and can be attached to the screen using the strip behind the eye tracker.
- Tobii website has online support available for troubleshooting any problems the users run into.
- The eye tracker is user friendly. The user interface is self explanatory and not complicated.

3.7 API Selection

In the study, the participants had to summarize certain API methods and classes. Each participant is given a combination of four API elements that have to be summarized with the help of various information sources. For this task, we had to finalize a set of APIs. We wanted to choose APIs in such a way that they should not be very self explanatory and at the same time not very complex for comprehension. The API method or class must be important in its function and yet once studied they should be easy to

understand. After some research, we finalized four API families for selecting methods and classes for the study. The selected API families are as follows:

- Eclipse(T1 and T2 in Table 3.1)
- NetBeans(T7 and T8 in Table 3.1)
- Jmeter(T3 and T4 in Table 3.1)
- Tomcat(T5 and T6 in Table 3.1)

For each API family, we selected a method and a class as two different granularity levels of a task. The eight final API elements were named from T1 to T8. Each API element can be summarized using various artifacts such as a Stack Overflow, Bugzilla, source code or a combination of them. For example, T1 was further divided into T1.SO, T1.BR, T1.Code, and T1.All. In total, we had 32 instances of the API elements. Then, these API elements were further categorized. Table 3.1 shows the finalized API elements (that correspond to tasks) from the four API families.

Task ID	Projects	Version	API level	Fully qualified Name
T1	Eclipse	4.2	method	`org.eclipse.core.databinding.binding.dispose`
T2	Eclipse	4.2	class	`org.eclipse.swt.SWTError`
T3	Jmeter	3.2	method	`org.apache.jmeter.Jmeter.convertSubTree`
T4	Jmeter	3.2	class	`org.apache.jmeter.samplers.SampleResult`
T5	Tomcat	7	method	`org.apache.catalina.realm.JDBCRealm.getRoles`
T6	Tomcat	7	class	`org.apache.catalina.valves.ValveBase`
T7	Netbeans	7.4	method	`org.netbeans.api.progress.ProgressUtils.runOffEventDispatchThread`
T8	Netbeans	7.4	class	`org.openide.nodes.CHildFactory.Detachable`

Table 3.1: API element list.

From the Table 3.1, we can observe that there is a method and a class under each API family. Initially, for each API type there were around 10 to 12 API elements listed down. Each API element was well researched. We also performed a web search for each API to investigate whether enough informal documentation is available on Stack Overflow, Bugzilla, as well as formal documentation (i.e., source code), and whether developers are discussing these APIs online. As a result, we finalized the selection of two API elements (a method and a class) for each API family.

Once the APIs were finalized, we made eight main sequences to use in the experiment. Each sequence contained four different API elements for Stack Overflow discussions, bug reports, source code and combination of all. Selection of the API elements into the sequence was random generated and we made sure each API element occurred once among the eight sequences. The eight main sequences were distributed among 12 participants so that four out of the eight sequences were repeated. The generated sequences are presented in Table 3.2

The finalized set of the API elements had three levels of comprehension complexity or difficulty. Some API elements were easy to understand, while others required more time and information to thorough understand it and summarize. Each sequence was designed to include API elements of various difficulty levels, i.e., no sequence has all four easy API elements or all four difficult to comprehend API elements. Thus, each sequence included a combination of easy, moderate and difficult API elements.

Sequences	API1	API2	API3	API4
Seq1	T3-SO	T6-BR	T7-All	T2-Code
Seq2	T4-BR	T1-All	T6-SO	T7-Code
Seq3	T5-Code	T4-SO	T1-BR	T8-All
Seq4	T6-All	T3-Code	T2-BR	T1-SO
Seq5	T5-SO	T2-All	T1-Code	T8-BR
Seq6	T2-Code	T5-BR	T8-SO	T3-All
Seq7	T5-All	T8-Code	T7-BR	T2-SO
Seq8	T6-Code	T7-SO	T2-All	T3-BR

Table 3.2: Sequences

3.8 API Oracle Summary

Oracle is typically used to evaluate or verify a set of elements.to verify participant summaries from the controlled experiment. Here we defined a metric and called it 'Oracle Summary'. The summaries produced by the software developers taking part in the study, are to be verified against an oracle summaries of the APIs. For this purpose, we developed an oracle containing API summaries and used this oracle as a metric to validate developers' summaries. All eight API elements were summarized in the oracle.

Based on the information available on the web (both formal and informal) and after analyzing API usage in source code, we formed an oracle of summaries (note: three independent human annotators were involved in the process of developing an oracle). The annotators who developed the oracle summaries were three graduate research students from different universities, pursuing PhD and Master's degrees in the field of computer science and having prior experience with Java APIs.

Task ID	API name	Oracle summary
T1	`org. eclipse. core.databinding binding. dispose`	A Binding calculates a value that depends on one or more sources. A call of this method usually results in the binding stopping to observe its dependencies by unregistering its listener(s)
T3	`org.apache.jmeter Jmeter. convertSubTree`	Remove disabled elements Replace the ReplaceableController with the target subtree
T6	`org.apache.catalina.valves ValveBase`	Convenience base class for implementations of the Valve interface.
T8	`org.openide.nodes ChildFactory. Detachable`	Factory is used to create Children objects which supply child Nodes for a Node. Their usage is to write a class that extends ChildFactory. A ChildFactory can be used either to create typical Children object, or one which will be initialized on a background thread.

Table 3.3: Examples of an API oracle summary.

Each API element from the final list of APIs was looked up on the Internet by each annotator. Top three definitions or highly voted definitions of each API were shortlisted. Further, each annotator combined the three definitions to form a final summary for each API element. All the three final sets of summaries from each annotator was further combined with much discussion and thoughts among them to form one final oracle of summaries.

Java docs about APIs provide definitions and role about APIs. The idea of not utilizing Java docs instead of an oracle summary was that Java docs offer much detail and are lengthy in terms of definitions. Also, Java docs typically define an API based on formal information sources. On the other hand, our oracle summaries are precise in length and formed using both informal and formal information resources available to developers on the Internet.

This oracle summary contained comprehensive summaries for each API element and was later used to rate the quality and correctness of the summaries produced by the developers.

Table 3.3 provides several examples of the oracle summaries for some APi elements. Essentially, an oracle summary contains the most apt definition of an API.

3.9 Information Sources

For our study, we considered three information sources such as Stack Overflow discussions, Bugzilla bug reports, and source code. When collecting eye tracking data, we gathered very specific key focal points for each information sources. We now describe each information source and its key elements in detail.

3.9.1 Stack Overflow Discussions

Stack Overflow [76] is one of the famous online portals where developers, testers discuss issues and findings with respect to a problem related to any software. This portal was used as one of the informal information sources shared with the developers in the controlled experiment. Stack Overflow posts related to a particular API method or class were provided to the developers.

A typical Stack Overflow page contains a problem or an issue related to a software development context, and developers reply to the posts in the form of text comments, tag lines and suggested answers. We can categorize the entire Stack Overflow page into two main categories such as "question" and "answers". Both the main categories can be further broken down into "comment", "tag", "text", "title" and "vote" elements.

3.9.2 Bugzilla Bug Reports

Bugzilla [77] is a popular, quick, robust, and mature software defect-tracking system, or software bug-tracking system. Bug-tracking systems allow teams of developers and testers to keep track of outstanding defects, problems, issues, enhancements and other change requests of a particular product. Developers often refer to bug-tracking systems for publicly available information about software bugs and use such systems effectively.

Participants were given a search query result link to an appropriate Bugzilla repository related to an API class or method they had to summarize. A typical Bugzilla bug report page contains a bug or a defect listed about a software and information or discussion regarding the bug. Therefore, a Bugzilla page contains information about bug report's "title" and "information" that is further divided into "answer", "attachment" and "question" elements.

3.9.3 Source Code

Software developers write software programs using a programming language (e.g., Java, C++). The programming language provides a series of instructions they can use to create the program they want. All the instructions a software developer uses to build the program can be defined as source code.

A formal information source used in the controlled experiment was Java source code. Particiapants were given source code files related to the API method or class they would summarize in a task. A typical Java source code source contains the following elements such as "import statements", "if and while statements", "method", "class", "type statements", and "variables".

3.10 Using Camtasia for Screen Capturing

During our experiments, we needed a tool or software to capture the screen. We needed the recording of each session for later triangulation of our results or verification of the collected data during the analysis. We used Camtasia [2] for screen capturing. Camtasia is a screen recorder tool that records the entire screen for an entire task. It is easy to control using start, pause and end features. At the same time you can select the part of the screen you want to capture and enable audio capturing too.

During the experiments, when participants were ready to start the task in iTrace, Camtasia was started in order to capture the entire session recording. Including the calibration step, the entire task session was captured. These screen recordings were used when we performed the analysis of the data collected. Each recording can be named and saved at a specific location. Once saved, the recordings can be re-played like a video clip at any time.

3.11 Experiment Setup and Workflow

Once the requirements were all met, the iTrace was running correctly with the connected eye tracker and collecting developer's eye movements. We tried a few trial sessions and checked if all the gazes from the screen and different types of information were collected and saved. In this section, we explain the entire procedure and workflow of the experimental study.

The study took place on a monitor screen which was 23 inch wide so that the gazes are collected without any disruption. For this, we set a wide desktop screen as an extended

display and place the eye tracker on the top of the screen. The wide screen was connected to a laptop, where all the controlling of the experiment took place. The wide screen was only used by the participants during the study and all the settings such as starting and stopping of the session was controlled in the laptop by the moderator of the experiment. The wide desktop screen also had a mouse and a keyboard for the participants to use.

There were work spaces created in Eclipse which included the eight sequences of API elements. There was a new work space formed for each participant at the time of the study and named after the participant ID to avoid any confusion. For example, for participant 1 the ID is IDC1 and a work space containing a sequence (between 1 to 8) is imported to Eclipse. The works space is used by the participant during the study. Each work space has four different files with instructions. These four files are for Stack Overflow, Bugzilla, source code and a combination of all. Along with these files, there are several source code folders for Eclipse, Jmeter, Netbeans and Tomcat. These folders are used to look for the source code of an API when the developer has to summarize the API with respect to source code.

The participants opens one file at a time and follows the instructions provided for the study. Simultaneously, the eye tracker is collecting the eye movements and saving the information as a separate session. If the participant is using the task with Stack Overflow, her eye movements would be collected and saved. When she moves on to next task, say source code, a new recording of the eye movements is collected.

We created a workflow outlining a set of instructions to be used during the study. It describes the steps to be followed and serves as a manual for the moderator (i.e., the author of the thesis) so that no step is missed in the experiment and the data collection. The steps of the workflow are as follows:

- Prepare the computer for the subject session (before the subject comes in). The computer needs to have a two screen extended setup. One screen should be in front of the subject and another should be used by the moderator. Have two keyboards and two mice ready.
- Restart the computer if needed. Open Tobii EyeX eye tracker. Open Eclipse using the iTrace work space. At this point, setup a work space for the study. Create a work space copy that can be reused for each participant.
- Select iTrace in the project outline view in the above workspace and run as an Eclipse application. This will open a new version of Eclipse. Next, import from file system all the sub folders in the summarization study work space. One should see all the four projects (Eclipse, Netbeans, JMeter, TomCat) in the project outline view. Make sure the project outline looks similar. Delete any unnecessary folders.

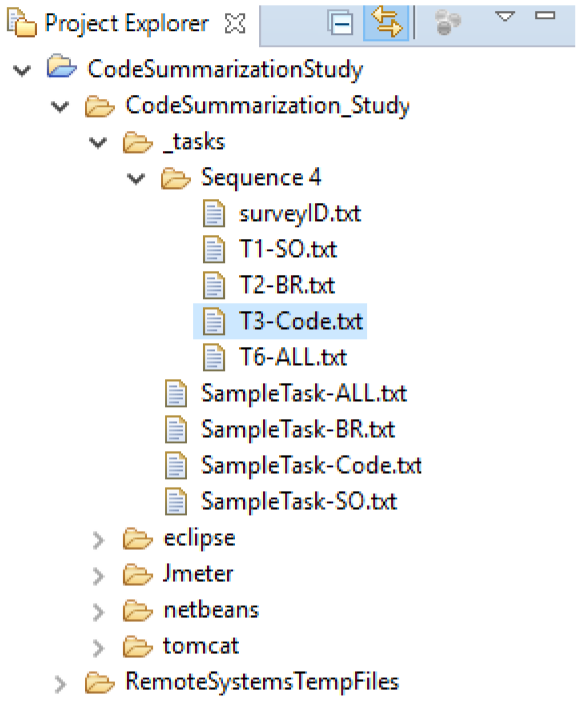

FIGURE 3.12: Workspace sequence.

- Make sure the projects is under the iTrace workspace in a folder called "runtime-eclipseApplication". At this point, set font to point 14 from the Eclipse menu.

For each participant, the following set of instructions is followed for every task:

- From file explorer, go to the folder "runtime-eclipseApplication", copy and paste and rename it to "runtime-eclipseApplication-IDC1". This allows the iTrace development workspace to point to the folder that was created in the previous step. This is where iTrace collects data when running the Eclipse application.

- Point the iTrace application to the appropriate Eclipse work space folder that was just copied and renamed. In iTrace development work space, go to Project – properties – run/debug settings – select Eclipse Application and edit – for location attribute, go to file system, browse and select "runtime-eclipseApplication-IDC1" and click "Ok". Then click "Apply" and "Ok".

  Figure 3.12 shows the files in the work space sequence every participant sees while she is performing the task.

- The environment should have the task files open at the bottom. The four task files should be open at the bottom in the order in which the participant needs to complete the tasks. Next, select iTrace in the project outline view in the above workspace and Run as an Eclipse application.

- Fully maximize Eclipse on the subject's screen. Assign an ID to a subject. Have pre-questionnaire and post-questionnaire open in a browser with populated ID.
- Have the participants sign the printed consent form and have them fill the pre-questionnaire. This does not have to be in Eclipse.
- Ask the participant to read the first page of the description of the study. The participant needs to complete the tutorial at this time.
- Go to the SampleTask-Code.txt, SampleTask-BR.txt, SampleTask-SO.txt, and SampleTask-All.txt and show the participant the task file format and let her try those out using the Eclipse browser.
- Ask the participant to copy and paste the link into the Eclipse browser. The goal of this sample task is to get the participant familiar with how the process works. She does not have to write the summary.
- Ask the subject if they have any questions about the instructions. Answers should be limited to how they are to complete their tasks, not how the data from experiment will be used because we do not want them to know our hypothesis.
- Explain to the subject briefly what is expected of them even if they claim to understand. State that it is important to read the task description they will be given at the bottom of the screen and follow the directions appropriately. Mention that only after they gather required information from code, Stack Overflow, and/or bug reports (depending on the task) to complete the summary should they finish their summarization of the API.
- Start Camtasia recording. Calibrate/re-calibrate the Tobii eye tracker.
- Create a session. Use ID number of participant as Developer ID. Make sure the JSON and XML are checked. Display crosshair to see if the tracking is reasonable. Test and set the displacement as needed. Then disable crosshair.
- Click "Start Tracking" in iTrace and begin the actual study. If observed that they are moving too much ask them politely to not move too much and assume the initial position they started with. Do not press "Escape".
- When the task is completed, click "Stop Tracking" in iTrace. Stop Camtasia. Ask them to fill out the confidence and difficulty rates and other comments.
- When the study is complete, have them fill the post-questionnaire.

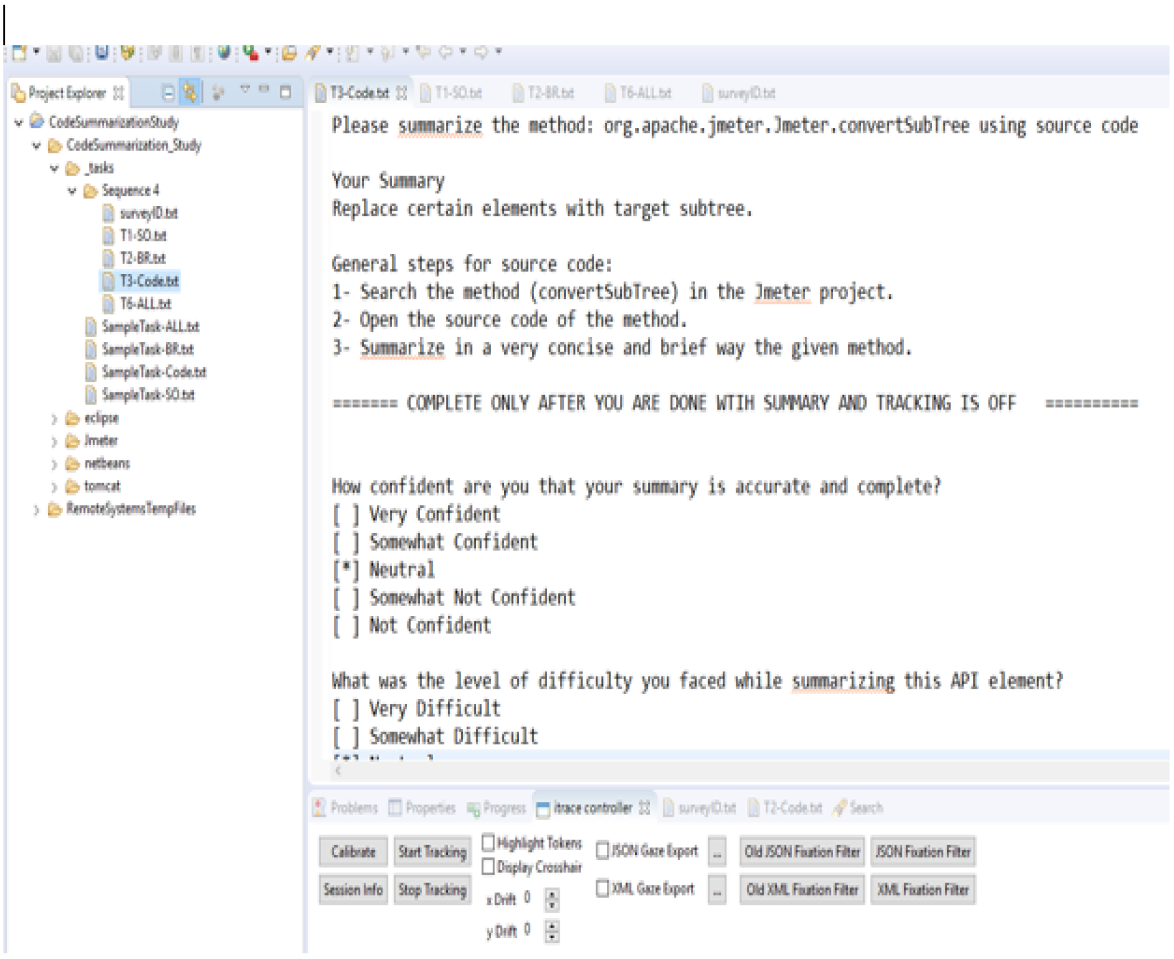

FIGURE 3.13: iTrace experiment screen.

Figure 3.13 shows how an iTrace experiment screen looks like. Participants will have to go through the instructions on the page and understand to summarize the API element given to them.

3.12 Challenges Experienced During the Experiment

During the preparation for the experiment we ran into several glitches and challenges. In this section, we discuss some of the challenges we faced.

- Ethics application clearance.

This research study involved participants and data collection. Thus, we needed to receive ethics clearance from the Research ethics board. After submitting the application, we had to wait for some time to get the clearance for the study.

- Combining iTrace with Tobii EyeX.

  iTrace and Tobii EyeX are two different independent platforms, and bringing them to work together was a challenge. Though iTrace was designed to support different eye trackers, the steps vary between every type of an eye tracker. There were no previous studies conducted using Tobii EyeX with iTrace. This was a first attempt and it took us some time to get Tobii EyeX compatible with iTrace. There had to be many SDK files suitable to the application and MinGW and CMake had to be executed correctly with proper commands. Eclipse Neon would throw many library related errors and each error had to be resolved to make the eye tracker work.

- Recruiting software developers to participate in the study.

  Recruiting software developers was time consuming. As they would usually be busy and the study had to be done at the location where the setup is ready. We had to invite developers to come to the lab and thus not many volunteered to take place in the study.

- Data collection and storing for analysis.

  While participants are performing the study, their every move would be converted into data. As a result, the end data is stored in XML and JSON files of large sizes. Each participant would have four different sets of data (SO, BZ, Code, ALL), This data needs to be saved and used again for analysis. Loading large files for the analysis takes much time.

  Figure 3.14 shows how an iTrace Eclipse window looks like before running the actual iTrace code.

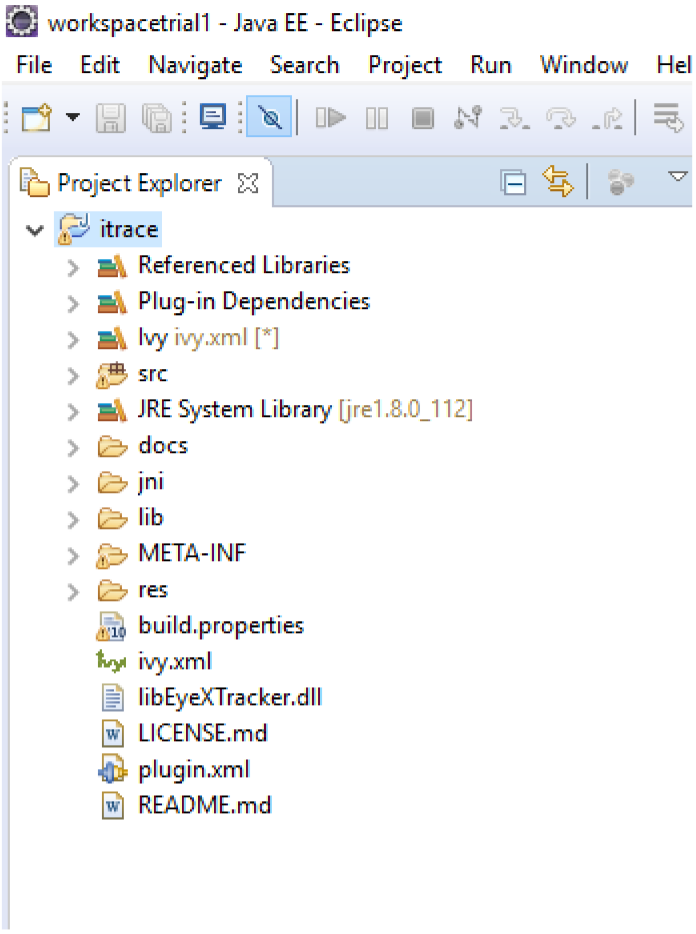

FIGURE 3.14: iTrace project files in Eclipse.

Chapter 4

Data Analysis

In this chapter, we present different analysis methods used on different types of data and how the results were achieved after analyzing all the data. We provide details on the analysis of text data (Section 4.1), eye tracking data (Section 4.2), and pre- and post- questionnaires (Section 4.3).

After conducting the controlled experiment, 12 sets of data from the participants were collected which had to be organized and analyzed. From each participant, we collected text files containing their answers of the API summaries and their eye tracking data. In total, for every participant there were four sets of text files with summaries and four sets of XML and JSON files with the eye tracking data.

To ease the process, analysis of the data is broken down into two stages. First stage is the analysis of text files from every participant. Second stage focuses on the eye tracking data analysis. First, we explain the text data analysis.

4.1 Text Data Analysis

When the developers were performing the experiment, they had to summarize the API elements with the help of information sources given to them. After understanding a particular API element, participants would write the summaries in the same iTrace Eclipse window and save it. They had to repeat this for all the four tasks given in their sequence. For example, if participant with ID "IDC1" was given Sequence 2, she had to complete tasks with Stack Overflow, Bug Reports, Source code and combination of all sources one at a time. She had to summarize an API using Stack Overflow, then bug reports, and so on. All the summaries were saved in their individual work spaces and then moved to a separate folder for just text summaries from each participant.

Task ID and type	API name	Developer summary
T1-ALL	`org.eclipse.core.databinding.Binding.dispose`	The Dispose method is called to close the active databinding object. This will delete it from memory and release the resorces
T4-BR	`org.apache.jmeter.samplers.SampleResult`	Majority issues are related to sockets. Or the class not returning right boolean.
T6-SO	`org.apache.catalina.valves.ValveBase`	The ValveBase class manages the TCP connection response.This is an abstrcct class that needs an override to the invoke method.
T7-code	`org.netbeans.api.progress.ProgressUtils.runOffEventDispatchThread`	The function is used to run a second thread and execute the run() method of the runnable object.

TABLE 4.1: IDC1 participant summaries.

Then, each summary from the software developer was compared to the oracle summary to determine the similarity between the developer summary with the oracle. All the summaries from the 12 participants were grouped based on the information type and analyzed to find out which source of information has the best developer performance. Figure 4.1 shows how a folder containing saved files from a developer looks like.

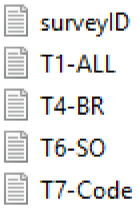

FIGURE 4.1: Task files saved.

Table 4.1 reports the summaries produced by a participant with ID "IDC1". Each summary was verified against the oracle summary and the similarity measures were recorded. To analyze the similarities between developer summaries and oracle summaries, we used *cosine similarity*. Next, we explain in detail the cosine similarity metric.

4.1.1 Cosine Similarity

Cosine similarity [78] is a measure of similarity between two non-zero vectors of an inward item space that measures the cosine of the edge between them. The cosine of 0

degree is 1, and it is under 1 for some other point in the interim (0,2π).

Here, we consider the summary as a string of vectors and pass them to the cosine algorithm. The cosine similarity value is between 0 and 1, with 1 being the exact match.

The cosine similarity algorithm was implemented using Python. By importing 'word2vec' library, the summary strings are compared considering the synonyms of words in both the strings. The arguments passed are the summaries from developer and oracle. Then the union of unique key words from both the strings are treated to calculate the cosine similarity coefficient. The outcome is a cosine similarity value.

Code for calculating the cosine similarity is presented below:
"Cosine similarity implementation in Python"

```
import math
import word2vec
from collections import Counter

def build_vector(iterable1, iterable2):
    counter1 = Counter(iterable1)
    counter2 = Counter(iterable2)
    all_items = set(counter1.keys()).union(set(counter2.keys()))
    vector1 = [counter1[k] for k in all_items]
    vector2 = [counter2[k] for k in all_items]
    return vector1, vector2

def cosim(v1, v2):
    dot_product = sum(n1 * n2 for n1, n2 in zip(v1, v2) )
    magnitude1 = math.sqrt(sum(n ** 2 for n in v1))
    magnitude2 = math.sqrt(sum(n ** 2 for n in v2))
    return dot_product / (magnitude1 * magnitude2)

l1 = "oracle summary ".split()
l2 = "developer summary".split()

v1, v2 = build_vector(l1, l2)
print(cosim(v1, v2))
```

For all summaries, we calculated 48 cosine similarity values in total and 12 cosine similarity values for each type of the information source.

The values are grouped based on the information source type and average values of cosine similarity are reported in Table 4.2.

4.1.2 R-Precision

R-precision is a slightly different metric when compared to the cosine similarity. R-precision is the exactness at the R_{th} position in the ranking of the results for a question

Information Source Type	Cosine Similarity (Mean)
Stack Overflow	0.279026491
ALL	0.271604509
Source Code	0.269840986
Bug Reports	0.262523797

TABLE 4.2: Cosine similirity (mean) values.

that has R relevant reports. Accuracy at K is precision at K which is ascertain for just K records. Reports positioned lower than K are ignored. This method of analysis was an add on and to see how different the results might be from cosine similarity.

To assess how well a developer's performance in summarizing AP elements comapred to the oracle, we calculated the R-precision, an assessment metric, broadly utilized in the field of data recovery. R-precision is figured also to precision at K, the exactness for outlines of length K. Exactness or precision at K decides out of the best K significant words returned by the developers using,

$$(|oracle \cap \text{predicted}_k|)$$

. We utilize R-precision since it has the benefit of having the capacity to deal with rundowns of variable lengths. The R-precision assesses the top R important words returned by the developers' "predicted R" where R is the length of the oracle summary. R-precision is defined as:

$$R - precision = \frac{|oracle \cap \text{predicted}_R|}{|\text{predicted}_R|} \tag{4.1}$$

R-precision also was calculated for all the sumamries to compare with the results of the cosine similarity values.

4.2 Eye Tracking Data Analysis

The eye tracking data collected during the study was saved in the form of the XML and JSON files under each participant's folder. These files had to be analyzed in order to study and determine the different fields of the eye tracking data. Each type of information (Stack Overflow, Bug Reports,source code, combination of all) had different fields collected in the eye tracking data. So each set of data from each information source had to be analyzed separately. All the files had to be analyzed and arranged based on the information source type to find the results.

gaze-responses-IDC1ALL-20171112T1904...	2017-11-12 7:14 PM	JSON File	5,906 KB
gaze-responses-IDC1ALL-20171112T1904...	2017-11-12 7:14 PM	XML Document	4,372 KB
session-info-IDC1ALL-20171112T190404-...	2017-11-12 7:05 PM	JSON File	1 KB

FIGURE 4.2: Eye tracking data.

These large in volume eye tracking files were a challenge to analyze. Also, finding the right tool or platform to analyze these was time consuming. We decided to use Tableau [79] for analyzing the eye tacking data. Tableau produces interactive data visualization for different sets of data input. Here, we took the JSON files as input and selected different schema levels for each type of data from different sources of information. Output was an organized eye tracking data which was now easy to analyze. Figure 4.2 shows how XML and JSON files are saved in the data folder.

Figure 4.3 shows Task 2 with a combination of all information sources, Figure 4.4 shows T2 with Bug Report data, Figure 4.5 with Stack Overflow data, and Figure 4.6 with source code data. Each set of data was organized and analyzed based on the schema levels suitable for the type of data.

idc5 t2 all eclipse

Name (Gazes)	Type (Gazes)	Type (Gazes.Sces)	Type (Gaze..	Part (Gazes.Bre)	Type (Gaze..	Number of Rec..	Tracker Time p..
45169 ◆ SampleResult id..	browser	Null	INFO	QUESTION	Null	74	263,643,600
59831 ◆ Copy request, response and assertion to..	browser	Null	INFO	ATTACHMENT	Null	2	7,177,875
				QUESTION	Null	20	71,770,237
61466 ◆ Adding samplers..	browser	Null	INFO	QUESTION	Null	30	107,389,113
how do I track metrics in jmeter for 'java requests' with sub results? - Stack ..	browser	Null	Null	Null	CODE	13	44,473,461
					TAG	4	13,669,733
					TEXT	154	526,559,753
java - Data Extraction from JMeter SampleResult Response - Stack Overflow	browser	Null	Null	Null	CODE	40	135,496,227
					TAG	3	10,163,652
					TEXT	141	477,399,493
JMeter custom Java sampler - Stack Overflow	browser	Null	Null	Null	CODE	2	6,809,128
					COMMENT	96	326,516,826
					TAG	36	122,337,867
					TEXT	258	878,571,702
					VOTE	4	13,617,401
performance - How to get the response of my Java Request Sampler in JMeter to reflect on all the availa..	browser	Null	Null	Null	CODE	270	1,004,569,677
					IMAGE	87	323,688,514
					TEXT	106	394,724,596
					TITLE	19	70,772,920
SampleResult.java	java	Null	Null	Null	Null	12	35,318,938
		CONDITIONAL_EXPRESSI..	Null	Null	Null	158	516,740,094
		FORSTATEMENT	Null	Null	Null	128	420,430,485
		IFSTATEMENT	Null	Null	Null	5,177	15,569,273,922
		IMPORT	Null	Null	Null	51	150,383,678
		METHOD	Null	Null	Null	16,550	46,552,630,091
		TYPE	Null	Null	Null	18,412	58,825,894,607
		VARIABLE	Null	Null	Null	3,190	10,229,857,981
		WHILESTATEMENT	Null	Null	Null	22	74,945,904
T4-ALL.txt	txt	Null	Null	Null	Null	6,036	20,889,454,276
Testing Java Classes with JMeter - Stack Overflow	browser	Null	Null	Null	CODE	15	51,390,287
					TEXT	270	928,480,101
					TITLE	16	57,647,611

Number of Records and Tracker Time per gazes broken down by Name (Gazes), Type (Gazes), Type (Gazes.Sces), Type (Gazes.Bre), Part (Gazes.Bre) and Type (Gazes.Soe).

FIGURE 4.3: T2.ALL

idc12 t2 br eclipse

Name	Type (Gaze..	Part	Type (Gazes)	Number of Records	Tracker Time
Bug 134646 Memory Leak with SWT browser widget	INFO	ANSWER	browser	893	467,404,843,794
		ATTACHMENT	browser	19	9,944,612,227
		QUESTION	browser	45	23,552,919,011
Bug 366785 org. eclipse.swt.SWTError: No more handles	INFO	ANSWER	browser	839	439,115,165,825
		ATTACHMENT	browser	3	1,570,103,572
		QUESTION	browser	30	15,701,027,298
T2-BR.txt	Null	Null	txt	3,103	1,624,409,515,389

Number of Records and Tracker Time broken down by Name, Type (Gazes.Bre), Part and Type (Gazes).

FIGURE 4.4: T2.BR

idc7 t2 so eclipse

Name	Type (Gazes)	Type (Gaze..	Number of Records	Tracker Time
java - InvocationTargetException caused by SWTError ("no more handles")? - Stack Overflow	browser	CODE	434	105,577,993,493
		TAG	2	486,539,712
		TEXT	98	23,839,501,775
		TITLE	22	5,351,728,178
		VOTE	20	4,865,180,018
java - NoClassDefFoundError org/eclipse/swt/SWTError - Stack Overflow	browser	CODE	105	25,538,828,653
		COMMENT	15	3,648,443,866
		TAG	2	486,456,691
		TEXT	64	15,566,580,670
		VOTE	5	1,216,128,790
java - SWTError: Not implemented [multiple displays] - Stack Overflow	browser	CODE	74	17,999,698,309
		COMMENT	133	32,351,343,870
		TEXT	19	4,621,545,292
		TITLE	5	1,216,182,641
		VOTE	24	5,837,759,829
T2-SO.txt	txt	Null	3,048	741,676,421,263

Number of Records and Tracker Time broken down by Name, Type (Gazes) and Type (Gazes.Soe).

FIGURE 4.5: T2.SO

t2 code idc6 eclipse

Name (Gazes)	Type (Gazes)	Type (Gazes.Sces)	Number of Records	Tracker Time per gazes
SampleResult.java	java	Null	6	1,119,616,611
		CONDITIONAL_EXPRESSI...	1	186,713,757
		IFSTATEMENT	333	62,167,615,465
		IMPORT	152	28,374,814,525
		METHOD	595	109,208,314,210
		TYPE	1,349	251,834,110,630
		VARIABLE	496	92,591,647,739
T4-Code.txt	txt	Null	1,432	267,345,015,204

Number of Records and Tracker Time per gazes broken down by Name (Gazes), Type (Gazes) and Type (Gazes.Sces).

FIGURE 4.6: T2.Code

4.3 Analysis of the Survey Data

The developers who participated in the study were asked to fill up questionnaires before and after the experiment. The pre-questionnaire was created in a way to assess developer's background, knowledge about API, software development, their experience in the software industry. While the post-questionnaire was to collect developer's feedback about the study, different information resources, best aspects in each type of task, and their confidence about the summaries they provided.

The surveys were created using Google forms with short and clear questions.

When the participants arrived to take part in the experiment, they were explained about the different steps and tasks of the study. As one of the initial steps they were given few minutes to go through the pre-questionnaire and answer the questions. The pre-questionnaire started with their ID and age, their education, background, experience as a developer and contributions to open source projects. Followed by the questions asking whether they refer to the informal documentation such as Stack Overflow and bug reports in their daily work.

After the experiment was over, the participants were asked to fill up the post-questionnaire. The post-questionnaire was more detailed asking questions about the usage of the information sources, i.e., Stack Overflow, bug reports and source code. We asked participants questions such as what information source helped them during the experiment and what part of each information source was most useful? These questions were of major focus for us. The developer's opinion about the study and the level of difficulty was also asked in the survey. Overall, the analysis of both pre- and post-questionnaires helped us to better understand developer perspective on the use and effect of various types of informal documentation. Pre-questionnaire and post-questionnaire forms can be found in the Appendix B and Appendix C, respectively.

Chapter 5

Results

In this chapter, we present the results of our quantitative and qualitative studies, and answer our research questions RQ1 (Section 5.1), RQ2 (Section 5.2) and RQ3 (Section 5.3).

5.1 RQ1: What information sources do developers leverage when summarizing various code elements?

When the developers were performing the experiment, they had to view different sources of information like Stack Overflow, Bug Reports, source code, and a combination of all sources in order to summarize the given API. We wanted to investigate what are the most viewed areas of the supporting documentation for every information source. Which means what exact area does a developer rely on to better understand an API and its usage. The findings could be used to improve automatic tools.

The eye tracking data from each session of the experiment was analyzed using Tableau. For each information type, we determined the most viewed records and calculated the percentage of view time for those records out of the overall tracker time.

During each task, the screen that the developers is viewing for each information source type is different. When it comes to Stack Overflow, they see different parts of the SO website, while using bug reports the screen presents bug information about the API. During the tasks with source code, they see code elements in a Java code with various methods, classes and if-statements. Therefore, all the eye tracking data was arranged and analyzed separately based on the information source type.

API family	Difficulty level
Eclipse	Easy
Netbeans	Moderate
Jmeter	Difficult
Tomcat	Difficult

TABLE 5.1: API family levels.

Moreover, the eye tracking records were organized based on the API family. This means that all the records involving Eclipse APIs were grouped together, similarly, for Jmeter, Netbeans and Tomcat. Each API family had its eye tracking data organized separately for each type of information source.

As shown in Table 5.1 the four API families are categorized into easy, moderate and difficult levels when presenting the results. Please note that these levels are determined based on the participants knowledge, feedback about the APIs and the API oracle.

To offer a better idea of the records collected by the eye tracker in Eclipse APIs, Figures 5.1, 5.2, 5.3 and 5.4 show the eye tracking records of the Bug Report session, Stack Overflow session, Code session and combination of all sources session, respectively.

If we take a look at Figure 5.1, in tasks using Bug Reports, the highest viewed record is highlighted and is "Info answer". Similarly, for Stack Overflow it is "Text Answer" shown in Figure 5.2. For source code session, the highest viewed records are "Method" and "Type" as shown in Figure 5.3. And for the combination of all the information resources, the highest viewed record is again "Method" and "Type" as shown in Figure 5.4.

As an example, we list all the records in details for Eclipse API. We show how much each participant spends on entire session for different information sources. And also how much time they spend on the record they have been viewing the most in the entire session. More detail are added by reporting the proportion of time the highest viewing record contributes to the entire session. Tables 5.2, 5.3, 5.4, and 5.5 report all the details about the Eclipse APIs. The results of the most viewed records for the Eclipse API family are shown in Table 5.6. Overall, for the Eclipse family of APIs the mostly viewed parts are as follows. For Bug Reports, it is "Info Answer"; for Stack Overflow, they are "Text question" and "answer"; for Source code, they are "Method" and "Type". And when given a combination of all the information sources, developers mainly refer to "Method" and "Type" parts of the source code. We can also see how much percentage the most viewed information source part takes up out of the total tracker time in each session (out of 100 percent tracker time).

Similarly, Table 5.7 shows records for the Netbeans API family. Netbeans falls under the "moderate" level category. When Netbeans API is considered, developers tend to view

IDC4 BR Eclipse T2

Name	Type (Gazes)	Type (Gaze..	Part	Number of Records	Tracker Time
Bug 366785 org.eclipse.swt. SWTError: No more handles	browser	INFO	ANSWER	781	1,207,272,237,465
			ATTACHMENT	64	98,929,091,029
			QUESTION	237	366,352,611,160
		TITLE	QUESTION	12	18,549,208,239
T2-BR.txt	txt	Null	Null	4,404	6,807,940,068,266

Number of Records and Tracker Time broken down by Name, Type (Gazes), Type (Gazes.Bre) and Part.

FIGURE 5.1: Eclipse Bug Report Record

ID	Session Time	Highest Viewed Record	Highest Record View Time	% of View Time
IDC3	18.63	Info answer	4.75	25.49
IDC11	32.14	Info question	6.92	21.53
IDC4	18.34	Info answer	6.86	36.18
IDC12	27.07	Info answer	7.79	28.77

TABLE 5.2: Eclipse BR record details.

Name	Type (Gazes)	Url	Type (Gaze..	Part	Number of Records	Tracker Time
asp.net - Automatically upload after user have selected a file using fileUploadControl - Stack Overflow	browser	https://stackoverflow.co..	COMMENT	ANSWER	45	69,540,605,015
				QUESTION	73	112,810,204,764
			TAG	QUESTION	2	3,090,688,444
			TEXT	ANSWER	15	23,180,194,166
				QUESTION	70	108,174,107,257
			TITLE	QUESTION	2	3,090,684,335
			VOTE	QUESTION	3	4,636,032,014
c# - About Disposing DataContext and DataBind() - Stack Overflow	browser	https://stackoverflow.co..	CODE	QUESTION	27	41,724,578,230
			TEXT	ANSWER	93	143,718,034,643
				QUESTION	4	6,181,410,894
			VOTE	ANSWER	9	13,908,186,223
T1-SO.txt	txt	Null	Null	Null	3,104	4,797,038,317,480

Number of Records and Tracker Time broken down by Name, Type (Gazes), Url, Type (Gazes.Soe) and Part.

FIGURE 5.2: Eclipse StackOverflow record.

"Info Answer" in Bug Reports most of the time, "Text question" and "Text answer" for Stack Overflow, "Method" and "Type" statements for both source code and combination of all resources.

We observe similar results with the "difficult" level APIs, such as Jmeter and Tomcat. Except for Bug Reports, the developers tend to view more "Info Question" part. For all the APIs, in each set of information the view records on particular parts are high. These details are shown in Tables 5.8 and 5.9.

ID	Session Time	Highest Viewed Record	Highest Record View Time	% of View Time
IDC4	16.32	Text answer	3.54	21.69
IDC12	26.45	Text answer	4.04	15.27
IDC7	12.36	Text question	4.09	33.09

TABLE 5.3: Eclipse SO record details.

IDC9 t2code eclipse

Name (Gazes)	Type (Gazes)	Type (Gazes.Sces)	Number of Records	Tracker Time per gazes
SWTError.java	eclipse.swt.win32.win32...	Null	22	7,616,255,502
		IFSTATEMENT	337	116,664,679,211
		IMPORT	40	13,847,985,077
		METHOD	2,142	675,777,374,540
		TYPE	2,604	901,501,551,595
		VARIABLE	251	86,900,195,842
T2-Code.txt	txt	Null	766	265,140,737,356

Number of Records and Tracker Time per gazes broken down by Name (Gazes), Type (Gazes) and Type (Gazes.Sces).

FIGURE 5.3: Eclipse Code record.

ID	Session Time	Highest Viewed Record	Highest Record View Time	% of View Time
IDC5	19.99	Method	3.81	19.14
IDC2	17.40	Method	3.89	22.18
IDC6	19.55	Type	4.19	21.48
IDC9	21.08	Type, Method	6.69	31.73

TABLE 5.4: Eclipse Code record details.

ID	Session Time	Highest Viewed Record	Highest Record View Time	% of View Time
IDC1	20.38	Method	3.73	18.30
IDC10	26.20	Type	7.64	29.16
IDC5	19.14	Method, Type	6.80	35.78
IDC8	16.97	Method, Type	6.18	36.57

TABLE 5.5: Eclipse ALL record details.

Type (Gazes)	Type (Gazes.Bre)	Part (Gazes.Bre)	Type (Gazes.Sces)	Type (Gazes.Soe)	Part (Gazes.Soe)	Number of Records	Tracker Time per gazes
asp.net - Automatically upload after user have selected a file using fileUploadControl - Stack Overflow, browser	Null	Null	Null	COMMENT	ANSWER	4	1,399,518,998
				TEXT	ANSWER	3	1,049,638,539
					QUESTION	21	7,347,416,301
				TITLE	QUESTION	33	11,545,868,847
				VOTE	ANSWER	36	12,595,639,042
Binding.java, java	Null	Null	Null	Null	Null	23	8,045,782,537
			IFSTATEMENT	Null	Null	422	135,384,067,716
			IMPORT	Null	Null	534	186,802,481,076
			METHOD	Null	Null	2,310	458,629,360,906
			TYPE	Null	Null	1,448	506,553,741,867
			VARIABLE	Null	Null	95	33,233,578,672
..er throws exception when called from ContentViewer. handleDispose, browser	INFO	ANSWER	Null	Null	Null	263	92,059,054,795
		ATTACHMENT	Null	Null	Null	15	5,250,553,818
		QUESTION	Null	Null	Null	105	36,753,876,657
..ableListDecorator should dispose itself if decorated list is disposed, browser	INFO	ANSWER	Null	Null	Null	14	4,900,332,384
		ATTACHMENT	Null	Null	Null	19	6,650,498,235
		QUESTION	Null	Null	Null	15	5,250,332,300
..inding if model and target observable are disposed concurrently, browser	INFO	ANSWER	Null	Null	Null	45	15,750,598,701
		ATTACHMENT	Null	Null	Null	3	1,050,030,054
		QUESTION	Null	Null	Null	99	34,651,103,273
c# - About Disposing DataContext and DataBind() - Stack Overflow, browser	Null	Null	Null	CODE	QUESTION	210	73,476,182,461
				COMMENT	QUESTION	15	5,248,363,622
				TAG	QUESTION	16	5,598,237,944
				TEXT	ANSWER	219	76,626,604,375
					QUESTION	55	19,243,774,762
				VOTE	ANSWER	42	14,695,555,274
					QUESTION	1	349,883,802
T1-ALL.txt, txt	Null	Null	Null	Null	Null	4,492	1,572,133,259,366

Number of Records and Tracker Time per gazes broken down by Name (Gazes), Type (Gazes), Type (Gazes.Bre), Part (Gazes.Bre), Type (Gazes.Sces), Type (Gazes.Soe) and Part (Gazes.Soe).

FIGURE 5.4: Eclipse Combination of all information record.

Therefore, we found that the developers view the "Info question" and "Info answer" fields in bug reports, "Text Question" and "Text Answer" fields in Stack Overflow discussions, and "Method" and "Type" statements in the source code and combination of all the sources. This tells us that in each type of information source, developers rely on certain areas and use those pieces of information more often when trying to comprehend APIs. It's likely that these particular elements of the documentation offer more detail or explanation for understanding APIs.

Maximum viewed information is same for all the sets of APIs. Hence, for APIs with any level of difficulty, developers tend to focus on certain areas for each information source, and when a combination of all the information sources are given, they spend most time on exploring the source code itself. This demonstrates that developers prefer to examine the source code more often when they have all additional information sources available. Source code remains the key source of information for developers in supporting their comprehension of the API usage.

Information	Highly viewed records	% of mean view time
BR	Info answer	27.99
SO	Text question/answer	23.35
Code	Method and Type	23.45
ALL	Method and Type	29.95

TABLE 5.6: Highly viewed records for Eclipse.

Information	Highly viewed records	% of mean view time
BR	Info answer	31.33
SO	Text question/answer	29.05
Code	Method and Type	32.61
ALL	Method and Type	27.78

TABLE 5.7: Highly viewed records for Netbeans.

Information	Highly viewed records	% of mean view time
BR	Info question	24.38
SO	Text question/answer	23.12
Code	Method and Type	33.86
ALL	Method and Type	31.04

TABLE 5.8: Highly viewed records for Jmeter.

Information	Highly viewed records	% of mean view time
BR	Info question	27.31
SO	Text answer	19.28
Code	Method and Type	30.97
ALL	Method and Type	29.49

TABLE 5.9: Highly viewed records for Tomcat.

Thus, we can conclude that eye tracking data helps us to determine the records that developers focus on and spend most of their API comprehension time for each type of information source. Also, the highest viewed records form a major percentage (around 26%) of the total session time.

The findings show the most focused subset of area in each information source. And these particular areas can be improved overall with respect to quality and content to help with a better design of automatic summarization tools.

We conducted a few statistical tests in R to check if the time duration of the highest viewed record among different information sources are different or not. We used the Kruskal-Wallis statistical test and report the results in Table 5.10. From the results, we observe that there is no statistically significant difference between the time values of different information sources (i.e., p-values are greater than 0.05).

API family	p-value
Eclipse	0.0858
Netbeans	0.06083
Jmeter	0.05857
Tomcat	0.05236

TABLE 5.10: Kruskal-Wallis test results.

5.2 RQ2: How do developers perform when using different sources of informal documentation to summarize the purpose and usage of code elements?

In order to analyze the qualities of summaries, the developer summaries were compared with the API oracle summaries. To determine the similarities between the summaries we used R-precision method for calculating the precision, as well as cosine similarity for calculating the similarity coefficient values between the oracle summary and developer summary.

The values of R-precision and cosine similarity are calculated considering slightly different aspects. R-precision is based on the word to word matching, while for cosine similarity, the summary words are considered as different vectors.

Table 5.11 reports R-precision values for different information sources. Based on the calculations, the mean precision between the summaries does not vary much except for combination of all information sources which has mean value of 20.40.

Information type	Min	Mean	Max
SO	0.00	28.24	100.00
Code	0.00	25.93	66.67
BR	0.00	24.06	55.56
ALL	0.00	20.40	55.56

TABLE 5.11: R-precision values.

Before calculating cosine similarity values, the summaries given by the participants were manually checked for spelling errors or consecutive word repetitions. Once the summaries were manually corrected, they were passed into the cosine algorithm to calculate cosine similarity values.

Similarly, the cosine similarity values are reported in Table 5.12. We can observe that SO has the highest mean value. The combination of ALL sources falls second without much difference. Mean cosine similarity values for Code and BR come next but not with a significant difference.

Information type	Min	Mean	Max
SO	0.125988157	0.279026491	0.866025403
ALL	0.140981475	0.271604509	0.696310623
Code	0.149147318	0.269840986	0.589255650
BR	0.125988157	0.262523797	0.742307488

TABLE 5.12: Cosine similarity values.

To compare similarity scores with the API complexity level, we arranged the cosine similarity values based on API family and then calculated the mean. As previously discussed, Eclipse, Netbeans, Jmeter fall into the easy, medium and difficult category of the APIs, respectively. Tables 5.13, 5.14, 5.15 present the cosine similarity values for each API family.

Information type	Cosine similarity (mean)
BR	0.312565239
Code	0.191231401
SO	0.147883164
ALL	0.125772058

TABLE 5.13: Cosine similarity (mean) values for Eclipse.

In Eclipse family, the best cosine similarity values were obtained for tasks leveraging information from bug reports and source code. This tells us that the APIs belonging to the Eclipse family have a lot of related information and discussions available online. This information has helped developers to understand the API and summarize it. There are numerous discussions about the bugs in Eclipse and online portals seems to have access to much related information.

Information type	Cosine similarity (mean)
BR	0.343088657
SO	0.313057582
ALL	0.258659129
Code	0.17121237

TABLE 5.14: Cosine similarity (mean) values for Netbeans.

For Netbeans APIs, tasks involving bug reports and Stack Overflow discussions have the best cosine similarity values explaining us that summaries produced by developers using BR and SO were more accurate when compared with the oracle. For Netbeans, there is good content of online information available which helped the participants with their understanding of the APIs. In fact, there are discussions about Netbeans APIs online posted by various developers and testers.

However, for Jmeter, the best cosine similarity values are achieved by combination of ALL information sources. This suggests us that the online information on Jmeter is

Information type	Cosine similarity (mean)
ALL	0.696310624
SO	0.444755609
Code	0.294627825
BR	0.27981551

TABLE 5.15: Cosine similarity (mean) values for Jmeter.

insufficient or limited to be helpful to the developers with their understanding of the Jmeter APIs. It seems that only recently the online portals started to document bugs for Jmeter and to put various information related to Jmeter. This might have led the participants to turn to the source code more often for summarizing the Jmeter APIs.

These findings suggest that for APIs that lack rich online documentation and are difficult to understand, developers used all possible information sources to verify whether their understanding of the API usage was correct.

5.3 RQ3: What is the developers' perception on leveraging different sources of information to support code summarization?

The participants were asked to answer a pre-questionnaire (see Appendix B) and a post-questionnaire (see Appendix C) surveys during the experiment. Based on the responses of all the 12 participants, we obtained some interesting results.

To start with, the pre-questionnaire was mainly about getting to know the participants, about their education, background and experience. The post-questionnaire was focused on the study and how participants felt about the overall experience with the experiment.

We first asked about the developer familiarity with the APIs. Based on the responses reported in Figure 5.5, Eclipse seems to be the most familiar API among our participants, followed by Netbeans, Jmeter and Tomcat. This shows that software developers mostly work with the Eclipse APIs in their work and are more familiar with Eclipse when compared to other APIs.

We next explore how developers feel about various information artifacts that were provided to them during the task. We asked them to name most useful information source when completing summarization tasks. As seen in Figure 5.6, most of the participants voted for Stack Overflow to be the best source of information to rely on, followed by the combination of all information resources. Developers felt that the Stack Overflow

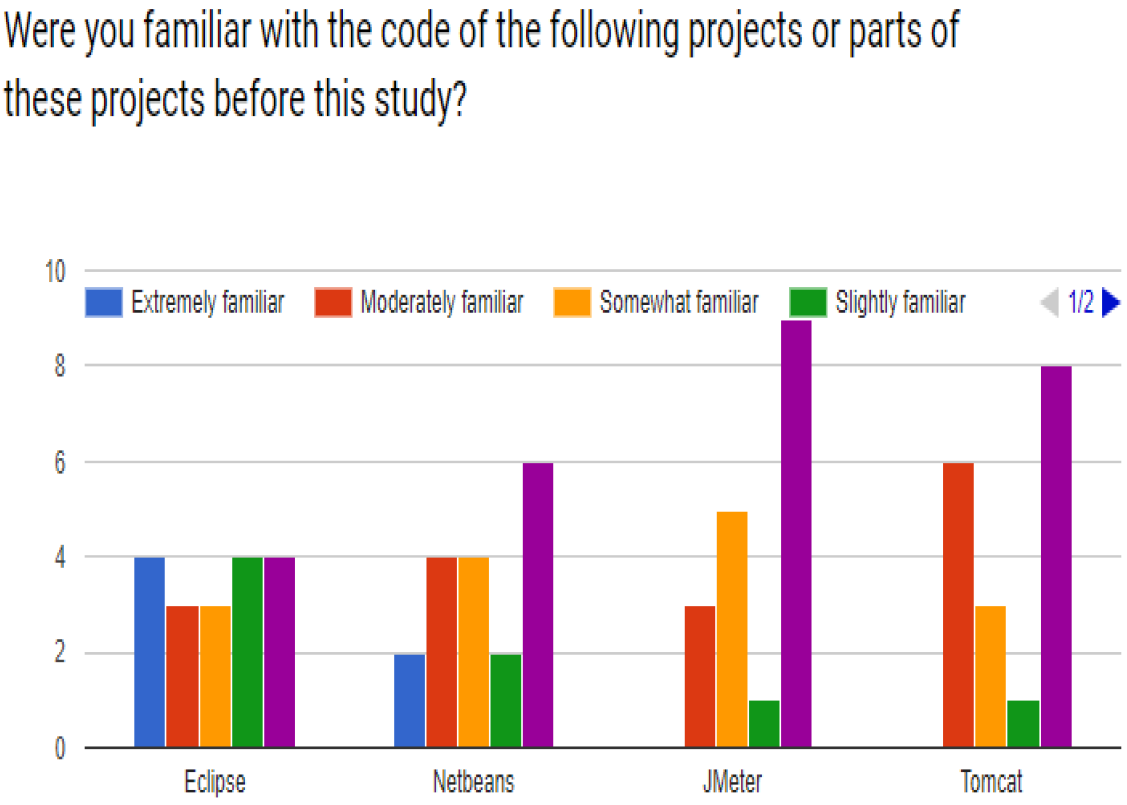

FIGURE 5.5: Familiarity with different APIs.

portal offered good information about APIs and it helped them more compared to other sources.

Although the quality of developer summaries was higher with bug reports and combination of all sources, developers actually perceive Stack Overflow to be the "go-to" place for seeking immediate help on the API-related issues and questions. Next come source code and bug reports, respectively. As per the survey, the developers consider bug reports to be the least helpful source of information.

When questioned about the time developers utilized for the experiment and whether they had sufficient time to summarize the API elements, around 33% of the participants said they used maximum time, while 50% of them were on the neutral side. A very few participants said they did not have sufficient time to complete the task. The results are shown in Figure 5.7.

We were eager to know from the developer's perspective if the entire experiment was hard or easy to participate in (we did our best to make experiments as easy and comfortable for participants). However, around 55% of the developers found the study tasks to be hard. While the rest 44% were on the neutral side. The results are shown in Figure 5.8.

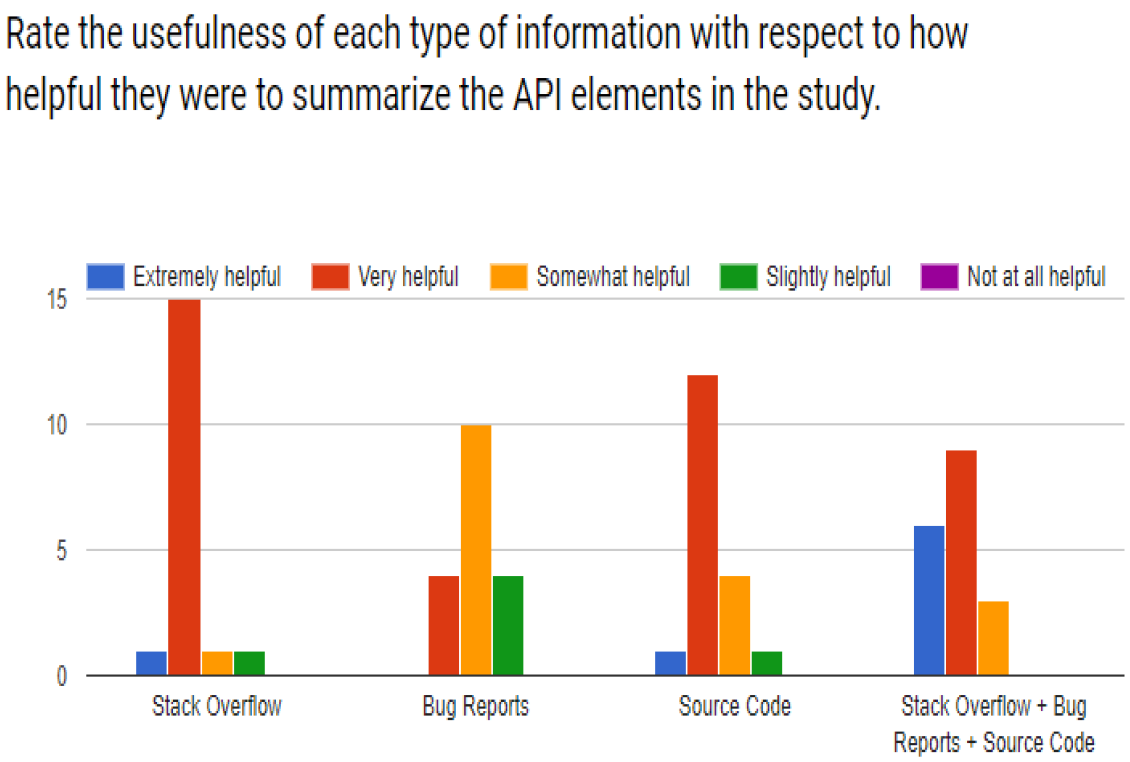

FIGURE 5.6: Usefulness of different sources of information.

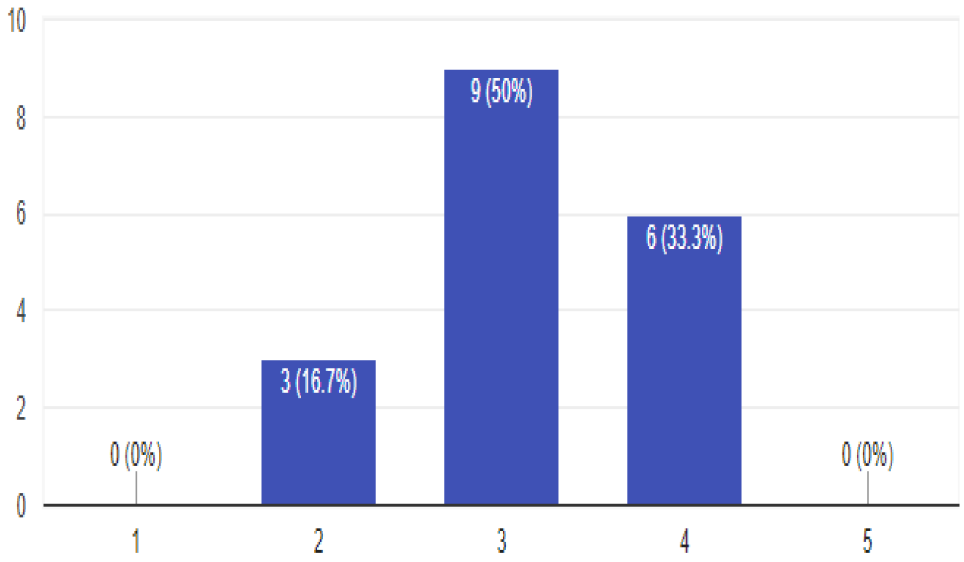

FIGURE 5.7: Satisfaction with the task completion time in the study.

To conclude, according to the developers, they perceive Stack Overflow to be the best source of informal documentation during this study, and they found Eclipse APIs to be the most familiar followed by Netbeans. Contrarily to the results of our quantitative analysis of the developer summaries and their accuracy, developers feel Stack Overflow to be a helpful source of information during summarization tasks. Most of the participants

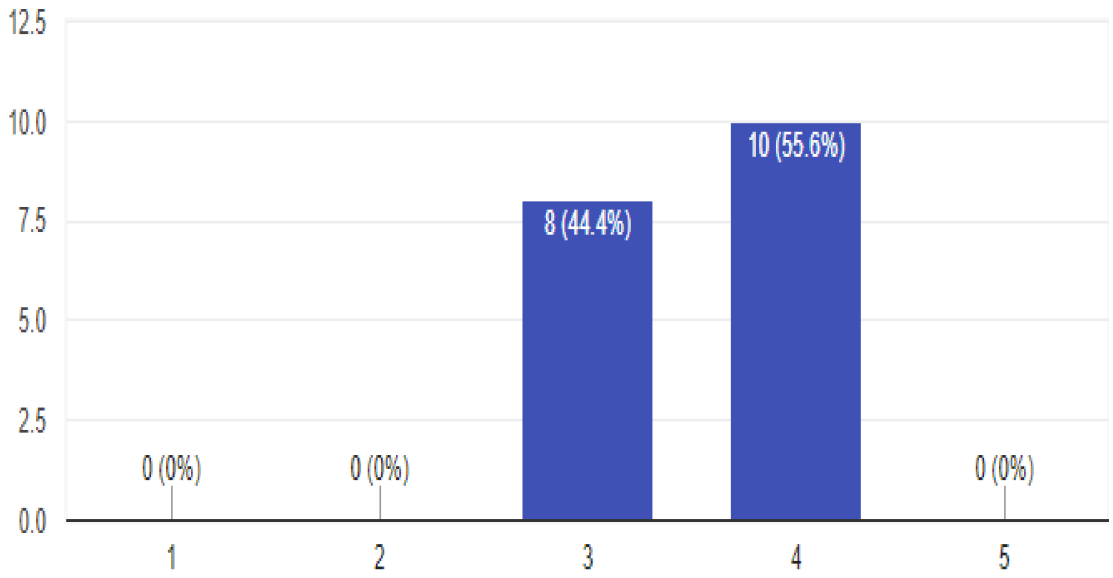

FIGURE 5.8: Difficulty level of the experiment.

found the study to be hard and time consuming.

5.4 Developer Comprehension Behavior vs. Perception

By combining the findings observed for each research question above, we can now discuss the key observations and identify potential gaps.

By analyzing the eye tracking data, we found the key areas of developers' focus for each information artifact. The most viewed parts of the information source are considered as the key comprehension elements in each information source. Here, developer efficiency is equal to the time spent on each part of the information, and we can conclude that developers rely on specific key elements in formal or informal documentation in supporting their API comprehension.

While we were able to measure the developers' accuracy of code summarization tasks, our findings suggest that the difficulty and familiarity with an API play a major role in affecting the accuracy of the summarization tasks. Informal documentation such as Stack Overflow is shown to have the best accuracy of summaries, overall.

By comparing the results of the quantitative analysis of the eye tracking data with the results of the survey data analysis, we observed a gap between developers' comprehension behavior and their perception. The results of the eye tracking data analysis show that informal documentation artifact such as bug reports and formal documentation (e.g., source code) are correlated with the higher developer performance (i.e., accuracy).

However, by analyzing the survey data, we found that developers perceive Stack Overflow to be the most useful and preferred source of information during comprehension tasks, while bug reports are seen as the least preferred source during code summarization. Thus, we can say that there is a gap between developers' eye movements and focal points when they conduct API comprehension/summarization tasks and their perception of the benefits of various information sources.

Our findings can be used to help improve the design and development of future automatic software engineering tools such as summarization tools.

Chapter 6

Discussion

In this chapter, we discuss our main findings (Section 6.1), factors affecting the quality of API summaries (Section 6.2), implications of our work for software developers (Section 6.3) and researchers (Section 6.4), as well as threats to validity (Section 6.5). We also outline a few future directions of our work (Section 6.6).

6.1 Use of Eye Tracking in Software Engineering Research

As discussed earlier, eye tracking is an emerging technology and a novel research method. Employing eye tracking as a major part of our study was challenging. By analyzing the collected eye tracking data from the experiment, we were able to determine how developers use various informal documentation sources for API comprehension tasks, and in particular what parts of informal documentation are most used by the developers. The analysis demonstrated how participants leverage each information source during comprehension task. When using bug reports, developers closely examine the question and answer contents. When reading and referring to the Stack Overflow discussions, the question's or answer's text fields are primarily used to obtain better insights into the API usage. When relying only on the source code, method and type statements are most critical for understanding APIs. But when being provided with multiple informal documentation types, developers tend to rely on the source code elements the most. This contributed to our major finding in this study based on the analysis of the collected the eye tracking data.

Considering different levels of API complexity when analyzing the eye tracking data demonstrated that developers' understanding of API usage do not change. In addition, our findings suggest that developers consider the same parts of the information source

for comprehension of the APIs of various complexity (i.e., different levels of difficulty). We were able to successfully identify the most viewed specific parts for each information source. And this identification was possible because of the eye tracking data.

6.2 Factors Affecting Quality of API Summaries

The summaries given by the participants were verified against the oracle summaries. Looking at the similarity values, we found that when the APIs are grouped together, comprehension/summarization tasks with Stack Overflow have the highest similarity with the oracle. To give a different layer to this, we organized the APIs into different families (i.e., easy, medium, difficult). We then observed that for easy and medium API families such as Eclipse, bug reports tasks has the highest similarity, while for the difficult APIs such as Jmeter, the combination of all the information sources has the highest similarity scores, on average. Therefore, we can state that for the APIs that are more difficult and hard to understand, developers must consider various sources of information to produce high quality summaries.

We can also confirm that along with the factors such as time and developer's experience, the complexity of an API is identified as an important indicator affecting the quality of summaries. Easy and popular APIs have a myriad of information available online and are easy to understand and summarize. While complex APIs might not be well documented and discussed online making their comprehension time consuming.

6.3 Implications For Software Developers

In their day to day development activities, software developers interact with various software artifacts. APIs are one of them. Our experiment highlights specific parts/areas in each artifact that the developers prefer to rely on when summarizing API elements. Developers refer to particular information source based on their familiarity with the API, as well as the API's complexity. If the API is more complex or new, and if the available documentation is limited, developers can benefit from various informal documentation types such as online developer forums where other developers worldwide provide additional context and details on the APIs.

6.4 Implications For Researchers

Several previous research studies were conducted on code summarization. But very few of them employed eye tracking technology. One such work was conducted by Kevic and Sharif [57] where eye tracking was involved while participants performed change tasks. In our work, we have used different artifacts for API summarization tasks, and our findings offer deeper insights into how developers leverage information from informal documentation than previous works. We have identified what parts in each artifact developers view the most and how developer's performance can alter between easy and difficult API families.

While we understand that eye tracking studies are very time consuming and require additional expertise in conducting experiments, we would like to encourage the rest of the research community to be more engaged with the emerging technology and conduct more studies involving eye tracking as the benefits of obtaining deep insights outweigh the costs of conducting such studies.

6.5 Threats to Validity

In our experiment, we performed various types of the data analyses. There are several threats to validity for our study. One being the time allocated for performing each task. There was no timer or a time limit set for participants during the experiment. However, the participants were told that the entire experiment is to be about one hour and thus, they might have been somewhat rushed in producing their summaries. The findings may have been different if participants have spent more time for each task, perhaps affecting the accuracy of their summaries against the oracle. Nevertheless, the summaries are brief statements, typically 2-3 sentences long, that should not require too much time to produce.

Another threat to validity is the choice of APIs. The APIs involved in the study belonged to four families and their level of complexity was different. Therefore, the time and effort with respect to summarizing various APIs may vary. More studies are needed, perhaps considering other APIs with similar levels of difficulty, to confirm our findings.

iTrace collects eye gazes only within the Eclipse editor. Thus, the entire experiment had to take place in Eclipse without switching the screen. All the web searches that developers performed happened within the Eclipse built-in browser. The study was confined to one screen which can be seen as a threat to validity.

Most software developers reported higher familiarity with Visual Studio IDE than Eclipse. And most of them felt that understanding an API was harder than understanding any other software element. The study was conducted with only 12 participants, and each developer had different software development experience. Additional studies involving more developers are recommended. Perhaps recruiting developers with a minimum 5 years of software development experience can impact the quality of the summaries.

The oracle which was built to verify the quality of API summaries can be subject to the threat of validity. Yet, the oracle was developed by three independent human annotators (i.e., graduate students) and is believed to be precise.

There is no agreement/suggestion within the research community on the set of metrics for analyzing eye tracking data. The eye tracking data is stored in the files of large size, and thus, is difficult to analyze. Offering recommendations on the best metrics or methods used for eye tracking data analysis would be beneficial for future researchers in conducting and comparing eye tracking experiments.

6.6 Future Work

The goal of our study was to understand the use of software artifacts in code summarization tasks by enabling eye tracking. We obtained some interesting findings and have explained them. There are several future directions to consider in extending our work.

Researchers who are interested in extending our study can opt for a more advanced eye tracker (note: we used a consumer-based eye tracker) to see whether additional details on developer eye gazes can be collected and studied. Using a professional eye tracker can lead to improvements in the data collection, as well as in the quality of the developer eye gazes.

The metrics for analyzing eye tracking data are not well defined the in research field. There is no one particular metric that is recommended to be reported when analyzing eye tracking data. Researchers who perform experiments in eye tracking field are free to come up with their own methods and metrics for the analysis. As a future step towards possible standardization, researchers can focus on building a standard metrics system for eye tracking, so that research studies and findings can be compared, replicated and verified.

Collected eye tracking data is very large in size resulting in a time-consuming analysis. Researchers can come up with an optimization of some kind that compresses these large-volume files to a reasonable size while preserving the quality of the data collected. There

is an XML or JSON record for every second of eye movement capturing from the eye tracker. This can be improved by having lesser records and deleting idle or repetitive records.

In our study we have used bug reports, Stack Overflow discussions and source code to be our major sources of information. One can be extended this by considering other artifacts that can improve developer comprehension.

Our study uses API comprehension tasks as a proxy for code summarization tasks. Future studies can consider different software elements and levels of code granularity (methods, classes) for designing tasks.

In summary, there are several possible improvements that can be considered in extending the current work and improving the existing body of research.

Chapter 7

Conclusions

Code summarization is an active research topic in the program comprehension area. In this work, we conduct an in-depth study of using eye tracking to investigate the effect of different software artifacts on code summarization tasks. The study was organized in a form of a controlled experiment involving 12 software developers as participants. The data collected from the experiment was both text data (i.e., developer summaries) and eye tracking data that required different types of the analysis.

Eye tracking data was analyzed to offer insights into what parts of the software artifacts participants consider and focus the most, and when they are given a choice between information sources what they would prefer for better comprehension of APIs. Similarly, the summaries provided by the participants were compared with the oracle summaries and assessed in terms of their accuracy (similarity) and quality. While evaluating the developer performance of summarizing APIs by comparing their summaries with the oracle we applied the cosine similarity metric.

We obtained some interesting findings by enabling eye tracking. We were able to identify specific focus areas for each information source (bug report, Stack Overflow discussion, and source code). We also found that when developers are given multiple information sources the source code remains the most important reference for comprehension tasks. Overall, the quality and accuracy of summaries was highest when developers leverage information from the Stack Overflow posts. However, when considering the complexity of APIs such as easy, moderate and difficult, the accuracy was dependent on the difficulty level. Therefore, we conclude that developer's performance in summarization tasks is affected by familiarity and complexity of the API.

Our findings suggest that unlike automatic tools, software developers tend to focus on a subset of information within each given software artifact. We hope that our study can

contribute to developing better automatic tools for code summarization utilizing our findings and results.

Appendix A

Recruitment Materials

Figure A.1 presents an email that we sent out for recruiting participants to the study.

Hello Graduate Students!

I am Shruthi Nagaraj, a Graduate Student from Carleton University supervised by Dr. Olga Baysal. I am currently doing my thesis on "Enabling Eye Tracking to Study the Use of Software Artifacts on Code Summarization" for which I need to conduct a short experiment. Hoping to present the findings from my research which is of help to design better automatic tools in the future.

WHAT YOU WILL BE DOING?!

- Perform a small task of API summarization in your own words which should take around 30 minutes totally.
- API summarization with the help of code and a document.
- You will be given full instructions with a demo.

WHAT I WILL BE DOING?!

- Collect data from the experiment and analyze it.
- Build my final results about the study.

Remuneration!!!

Each participant will be given a **20$ gift card from Starbucks** appreciating your participation.

This study has been reviewed and received ethics clearance through a Carleton University Research Ethics Committee.

Your participation would matter a lot and please free to contact me for any queries.

If interested please mail me at **ShruthiNagaraj@cmail.carleton.ca**

Thank you!

FIGURE A.1: Recruitment email.

Appendix B

Pre-questionnaire Survey Form

Background Questionnaire (YC)

A short background survey.

* Required

1. **ID number** *
IDC1

2. **Gender** *
Mark only one oval.
- Female
- Male
- Other: _____

3. **Your age range is :** *
Mark only one oval.
- < 18 years
- 18 - 25 years
- 26 - 30 years
- 31 - 35 years
- 36 - 40 years
- 41 - 45 years
- 46 - 50 years
- > 50 years

4. **How many years of active programming experience do you have?** *
Mark only one oval.
- < 1 year
- Between 1 and 2 years
- Between 3 and 5 years
- Between 6 and 10 years
- > 10 years

FIGURE B.1: Pre-questionnaire, page 1.

5. **What is your level of expertise in the Java programming language?** *
Mark only one oval.

- Poor
- Fair
- Good
- Very Good
- Excellent

6. **Please select all the degrees you have and are currently enrolled in.** *
Check all that apply
Check all that apply.

- [ ] Bachelors
- [ ] Masters
- [ ] Ph.D.
- [ ] Other: _____

7. **Current Positions - Select all that apply** *
Check all that apply
Check all that apply.

- [ ] I currently work in industry
- [ ] I currently work in academia
- [ ] I am currently a student
- [ ] I am currently a faculty member
- [ ] I am currently a post doc

8. **How many years of work experience do you have in industry?** *
Mark only one oval.

- None
- < 1 year
- Between 1 and 2 years
- Between 3 and 5 years
- Between 6 and 10 years
- > 10 years

9. **Do you use Stack Overflow to find solutions to your coding problems?** *
Mark only one oval.

- Yes
- No

Background Questionnaire - page 2

FIGURE B.2: Pre-questionnaire, page 2.

10. **Do you use/read bug reports to find solutions to issues while coding?** *
Mark only one oval.

- Yes
- No

11. **Have you contributed (code and/or documentation) to an open source project?** *
Mark only one oval.

- Yes
- No

12. **Which of the following IDEs are you familiar with? (By familiar we mean you are able to work in fairly well).** *
Check all that apply
Check all that apply.

- [ ] Eclipse
- [ ] Visual Studio
- [ ] Netbeans
- [ ] IntelliJ
- [ ] Other: _____

Thank you for your participation!

Powered by Google Forms

FIGURE B.3: Pre-questionnaire, page 3.

Appendix C

Post-questionnaire Survey Form

Post Questionnaire (YC)

Complete this questionnaire after you are done summarizing all four API elements

* Required

1. **ID number:** *
(Enter the ID number given to you)

2. **Were you familiar with the code of the following projects or parts of these projects before this study?** *
(By familiar, we mean you have seen or worked with this code, bug report, Stack Overflow document before and that knowledge helped you in your answers to the study.)
Mark only one oval per row.

	Extremely familiar	Moderately familiar	Somewhat familiar	Slightly familiar	Not at all familiar
Eclipse					
Netbeans					
JMeter					
Tomcat					

3. **Rate the usefulness of each type of information with respect to how helpful they were to summarize the API elements in the study.** *
Mark only one oval per row.

	Extremely helpful	Very helpful	Somewhat helpful	Slightly helpful	Not at all helpful
Stack Overflow					
Bug Reports					
Source Code					
Stack Overflow + Bug Reports + Source Code					

FIGURE C.1: Post-questionnaire, page 1.

4. **Rate the usefulness of the different types of contexts present in **source code** that helped you to summarize the API elements.** *

Mark only one oval per row.

	Extremely helpful	Very helpful	Somewhat helpful	Slightly helpful	Not at all helpful	I do not know what this is
Comments (line and block comments)						
Javadoc comments (e.g., @return, @param)						
Identifier names						
Lines of code						
General readability of the code (e.g., good indentation, structure, etc...)						

5. **Rate the usefulness of the different types of contexts present in **Stack Overflow (SO) documents** that helped you to summarize the API elements** *

Mark only one oval per row.

	Extremely helpful	Very helpful	Somewhat helpful	Slightly helpful	Not at all helpful	I do not know what this is	None given
Code examples							
Comments by users							
Stack traces							
Textual descriptions (other than code examples and stack traces)							
Votes							
Tags of questions							
User reputation/profile							
Date of a comment/answer							

FIGURE C.2: Post-questionnaire, page 2.

6. **Rate the usefulness of the different types of contexts present in **bug reports** that helped you to summarize the API elements** *

Mark only one oval per row.

	Extremely helpful	Very helpful	Somewhat helpful	Slightly helpful	Not at all helpful	I do not know what this is	None given
Code examples							
Comments by users							
Stack traces							
Proposed patch							
Test cases							
Textual description of the bug report (other than stack traces or code examples)							

7. **Overall, how difficult did you find the study?** *

Mark only one oval.

	1	2	3	4	5	
Easy						Difficult

8. **Overall, do you feel you spent sufficient time to summarize the API elements?**

Mark only one oval.

	1	2	3	4	5	
Minimum						Maximum

9. **Comments**

(Please list any comments you had on the study. We value your feedback in this research.)

Powered by Google Forms

FIGURE C.3: Post-questionnaire, page 3.

Bibliography

[1] Timothy R Shaffer, Jenna L Wise, Braden M Walters, Sebastian C Müller, Michael Falcone, and Bonita Sharif. 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*, pages 954–957. ACM, 2015.

[2] Laurie Charnigo. Lights! camera! action! producing library instruction video tutorials using camtasia studio. *Journal of Library & Information Services in Distance Learning*, 3(1):23–30, 2009.

[3] Braden Walters, Michael Falcone, Alexander Shibble, and Bonita Sharif. Towards an eye-tracking enabled ide for software traceability tasks. In *Traceability in Emerging Forms of Software Engineering (TEFSE), 2013 International Workshop on*, pages 51–54. IEEE, 2013.

[4] Michael Erdmann and Walter Waterfeld. Overview of the neon toolkit. In *Ontology Engineering in a Networked World*, pages 281–301. Springer, 2012.

[5] Clay Richardson, Donald Avondolio, et al. *Professional Java JDK*. John Wiley & Sons, 2007.

[6] Nadine Dertmann Beister, N Dertmann-R Hrestic M Kionka, M Mohr M Mühe D Murali, F Steffen S Stey E Unruh, Q Wang C Dannewitz, M Herlich, and S Weber. Opennetinf documentation design and implementation. Technical report, Technical report, University of Paderborn, 2011.

[7] Jerry D DeMaster. Java native interface code generator, May 23 2000. US Patent 6,066,181.

[8] David Kruglinski, Scot Wingo, George Shepherd, and David Kruglinski. *Programming Microsoft Visual C++*, volume 5. Microsoft press Seattle, WA, 1998.

[9] Agostino Gibaldi, Mauricio Vanegas, Peter J Bex, and Guido Maiello. Evaluation of the tobii eyex eye tracking controller and matlab toolkit for research. *Behavior research methods*, 49(3):923–946, 2017.

[10] Karin Meyer. Wombata tool for mixed model analyses in quantitative genetics by restricted maximum likelihood (reml). *Journal of Zhejiang University Science B*, 8 (11):815–821, 2007.

[11] Ken Martin and Bill Hoffman. *Mastering CMake: a cross-platform build system*. Kitware, 2010.

[12] Robert S Rist et al. Plans in programming: definition, demonstration, and development. In *Empirical studies of programmers*, pages 28–47. Ablex, Norwood, NJ, 1986.

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

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

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

[16] Andrew J Ko, Brad A Myers, Michael J Coblenz, and Htet Htet Aung. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Transactions on software engineering*, 32 (12):971–987, 2006.

[17] Thomas Fritz, David C Shepherd, Katja Kevic, Will Snipes, and Christoph Bräunlich. Developers' code context models for change tasks. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 7–18. ACM, 2014.

[18] Eric Clayberg and Dan Rubel. *eclipse Plug-ins*. Pearson Education, 2008.

[19] R Elves. Tasktop for eclipse-get more out of mylyn. *Retrieved July*, 15:2010, 2010.

[20] Mik Kersten and Gail C Murphy. Mylar: a degree-of-interest model for ides. In *Proceedings of the 4th international conference on Aspect-oriented software development*, pages 159–168. ACM, 2005.

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

[22] Gail C Murphy, Mik Kersten, and Leah Findlater. How are java software developers using the elipse ide? *IEEE software*, 23(4):76–83, 2006.

[23] Chris Parnin and Carsten Gorg. Building usage contexts during program comprehension. In *Program Comprehension, 2006. ICPC 2006. 14th IEEE International Conference on*, pages 13–22. IEEE, 2006.

[24] David Piorkowski, Scott D Fleming, Christopher Scaffidi, Liza John, Christopher Bogart, Bonnie E John, Margaret Burnett, and Rachel Bellamy. Modeling programmer navigation: A head-to-head empirical evaluation of predictive models. In *Visual Languages and Human-Centric Computing (VL/HCC), 2011 IEEE Symposium on*, pages 109–116. IEEE, 2011.

[25] Vinay Augustine, Patrick Francis, Xiao Qu, David Shepherd, Will Snipes, Christoph Bräunlich, and Thomas Fritz. A field study on fostering structural navigation with prodet. In *Proceedings of the 37th International Conference on Software Engineering-Volume 2*, pages 229–238. IEEE Press, 2015.

[26] Taek Lee, Jaechang Nam, DongGyun Han, Sunghun Kim, and Hoh Peter In. Micro interaction metrics for defect prediction. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 311–321. ACM, 2011.

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

[28] Marcel A Just and Patricia A Carpenter. A theory of reading: From eye fixations to comprehension. *Psychological review*, 87(4):329, 1980.

[29] Janet Siegmund, Christian Kästner, Sven Apel, Chris Parnin, Anja Bethmann, Thomas Leich, Gunter Saake, and André Brechmann. Understanding understanding source code with functional magnetic resonance imaging. In *Proceedings of the 36th International Conference on Software Engineering*, pages 378–389. ACM, 2014.

[30] Thomas Fritz, Andrew Begel, Sebastian C Müller, Serap Yigit-Elliott, and Manuela Züger. Using psycho-physiological measures to assess task difficulty in software development. In *Proceedings of the 36th international conference on software engineering*, pages 402–413. ACM, 2014.

[31] Sebastian C Müller and Thomas Fritz. Stuck and frustrated or in flow and happy: Sensing developers' emotions and progress. In *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, volume 1, pages 688–699. IEEE, 2015.

[32] Manuela Züger and Thomas Fritz. Interruptibility of software developers and its prediction using psycho-physiological sensors. In *Proceedings of the 33rd Annual ACM*

Conference on Human Factors in Computing Systems, pages 2981–2990. ACM, 2015.

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

[34] Bonita Sharif, Grace Jetty, Jairo Aponte, and Esteban Parra. An empirical study assessing the effect of seeit 3d on comprehension. In *Software Visualization (VISSOFT), 2013 First IEEE Working Conference on*, pages 1–10. IEEE, 2013.

[35] Shehnaaz Yusuf, Huzefa Kagdi, and Jonathan I Maletic. Assessing the comprehension of uml class diagrams via eye tracking. In *Program Comprehension, 2007. ICPC'07. 15th IEEE International Conference on*, pages 113–122. IEEE, 2007.

[36] Benoît De Smet, Lorent Lempereur, Zohreh Sharafi, Yann-Gaël Guéhéneuc, Giuliano Antoniol, and Naji Habra. Taupe: Visualizing and analyzing eye-tracking data. *Science of Computer Programming*, 79:260–278, 2014.

[37] Bonita Sharif and Jonathan I Maletic. An eye tracking study on the effects of layout in understanding the role of design patterns. In *Software Maintenance (ICSM), 2010 IEEE International Conference on*, pages 1–10. IEEE, 2010.

[38] Rachel Turner, Michael Falcone, Bonita Sharif, and Alina Lazar. An eye-tracking study assessing the comprehension of c++ and python source code. In *Proceedings of the Symposium on Eye Tracking Research and Applications*, pages 231–234. ACM, 2014.

[39] Bonita Sharif and Jonathan I Maletic. An eye tracking study on camelcase and under_score identifier styles. In *Program Comprehension (ICPC), 2010 IEEE 18th International Conference on*, pages 196–205. IEEE, 2010.

[40] Hidetake Uwano, Masahide Nakamura, Akito Monden, and Ken-ichi Matsumoto. Analyzing individual performance of source code review using reviewers' eye movement. In *Proceedings of the 2006 symposium on Eye tracking research & applications*, pages 133–140. ACM, 2006.

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

[42] Paige Rodeghero, Collin McMillan, Paul W McBurney, Nigel Bosch, and Sidney D'Mello. Improving automated source code summarization via an eye-tracking study of programmers. In *Proceedings of the 36th International Conference on Software Engineering*, pages 390–401. ACM, 2014.

[43] Bonita Sharif and Huzefa Kagdi. On the use of eye tracking in software traceability. In *Proceedings of the 6th International Workshop on Traceability in Emerging Forms of Software Engineering*, pages 67–70. ACM, 2011.

[44] Braden Walters, Timothy Shaffer, Bonita Sharif, and Huzefa Kagdi. Capturing software traceability links from developers' eye gazes. In *Proceedings of the 22nd International Conference on Program Comprehension*, pages 201–204. ACM, 2014.

[45] Zohreh Sharafi, Timothy Shaffer, Bonita Sharif, and Yann-Gaël Guéhéneuc. Eye-tracking metrics in software engineering. In *Software Engineering Conference (APSEC), 2015 Asia-Pacific*, pages 96–103. IEEE, 2015.

[46] Gerardo Cepeda Porras and Yann-Gaël Guéhéneuc. An empirical study on the efficiency of different design pattern representations in uml class diagrams. *Empirical Software Engineering*, 15(5):493–522, 2010.

[47] Sebastien Jeanmart, Yann-Gael Gueheneuc, Houari Sahraoui, and Naji Habra. Impact of the visitor pattern on program comprehension and maintenance. In *Empirical Software Engineering and Measurement, 2009. ESEM 2009. 3rd International Symposium on*, pages 69–78. IEEE, 2009.

[48] Zéphyrin Soh, Zohreh Sharafi, Bertrand Van den Plas, Gerardo Cepeda Porras, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. Professional status and expertise for uml class diagram comprehension: An empirical study. In *Program Comprehension (ICPC), 2012 IEEE 20th International Conference on*, pages 163–172. IEEE, 2012.

[49] Razvan Petrusel and Jan Mendling. Eye-tracking the factors of process model comprehension tasks. In *International Conference on Advanced Information Systems Engineering*, pages 224–239. Springer, 2013.

[50] Nergiz Ercil Cagiltay, Gul Tokdemir, Ozkan Kilic, and Damla Topalli. Performing and analyzing non-formal inspections of entity relationship diagram (erd). *Journal of Systems and Software*, 86(8):2184–2195, 2013.

[51] Zohreh Sharafi, Alessandro Marchetto, Angelo Susi, Giuliano Antoniol, and Yann-Gaal Gueheneuc. An empirical study on the efficiency of graphical vs. textual representations in requirements comprehension. In *Program Comprehension (ICPC), 2013 IEEE 21st International Conference on*, pages 33–42. IEEE, 2013.

[52] Teresa Busjahn, Roman Bednarik, and Carsten Schulte. What influences dwell time during source code reading?: analysis of element type and frequency as factors. In *Proceedings of the Symposium on Eye Tracking Research and Applications*, pages 335–338. ACM, 2014.

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

[54] Teresa Busjahn, Carsten Schulte, and Andreas 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*, pages 1–9. ACM, 2011.

[55] Martha E Crosby, Jean Scholtz, and Susan Wiedenbeck. The roles beacons play in comprehension for novice and expert programmers. In *14th Workshop of the Psychology of Programming Interest Group*, pages 58–73, 2002.

[56] Teresa Busjahn, Roman Bednarik, Andrew Begel, Martha Crosby, James H Paterson, Carsten Schulte, Bonita Sharif, and Sascha Tamm. Eye movements in code reading: Relaxing the linear order. In *Program Comprehension (ICPC), 2015 IEEE 23rd International Conference on*, pages 255–265. IEEE, 2015.

[57] Katja Kevic, Braden M Walters, Timothy R Shaffer, Bonita Sharif, David C Shepherd, and Thomas Fritz. Tracing software developers' eyes and interactions for change tasks. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 202–213. ACM, 2015.

[58] Roman Bednarik and Markku Tukiainen. An eye-tracking methodology for characterizing program comprehension processes. In *Proceedings of the 2006 symposium on Eye tracking research & applications*, pages 125–132. ACM, 2006.

[59] Prateek Hejmady and N Hari Narayanan. Visual attention patterns during program debugging with an ide. In *Proceedings of the Symposium on Eye Tracking Research and Applications*, pages 197–200. ACM, 2012.

[60] Zohreh Sharafi, Zéphyrin Soh, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. Women and mendifferent but equal: On the impact of identifier style on source code reading. In *Program Comprehension (ICPC), 2012 IEEE 20th International Conference on*, pages 27–36. IEEE, 2012.

[61] Nasir Ali, Zohreh Sharafi, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. An empirical study on the importance of source code entities for requirements traceability. *Empirical Software Engineering*, 20(2):442–478, 2015.

[62] Karen Spärck Jones. Automatic summarising: The state of the art. *Information Processing & Management*, 43(6):1449–1481, 2007.

[63] Gail Cecile Murphy. *Lightweight structural summarization as an aid to software evolution*. PhD thesis, 1996.

[64] Sonia Haiduc, Jairo Aponte, Laura Moreno, and Andrian Marcus. On the use of automated text summarization techniques for summarizing source code. In *Reverse Engineering (WCRE), 2010 17th Working Conference on*, pages 35–44. IEEE, 2010.

[65] Laura Moreno, Jairo Aponte, Giriprasad Sridhara, Andrian Marcus, Lori Pollock, and K Vijay-Shanker. Automatic generation of natural language summaries for java classes. In *Program Comprehension (ICPC), 2013 IEEE 21st International Conference on*, pages 23–32. IEEE, 2013.

[66] Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori Pollock, and K Vijay-Shanker. Towards automatically generating summary comments for java methods. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 43–52. ACM, 2010.

[67] Paul W McBurney and Collin McMillan. Automatic documentation generation via source code summarization of method context. In *Proceedings of the 22nd International Conference on Program Comprehension*, pages 279–290. ACM, 2014.

[68] Sarah Rastkar, Gail C Murphy, and Gabriel Murray. Automatic summarization of bug reports. *IEEE Transactions on Software Engineering*, 40(4):366–380, 2014.

[69] Laura Moreno, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrian Marcus, and Gerardo Canfora. Automatic generation of release notes. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 484–495. ACM, 2014.

[70] Latifa Guerrouj, David Bourque, and Peter C Rigby. Leveraging informal documentation to summarize classes and methods in context. In *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, volume 2, pages 639–642. IEEE, 2015.

[71] Christoph Treude and Martin P Robillard. Augmenting api documentation with insights from stack overflow. In *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*, pages 392–403. IEEE, 2016.

[72] Bonita Sharif and Jonathan I Maletic. itrace: Overcoming the limitations of short code examples in eye tracking experiments. In *ICSME*, page 647, 2016.

[73] Jung Bin Won, Sun Woong Kim, Eung Kweon Kim, Byung Jin Ha, and Tae-im Kim. Comparison of internal and total optical aberrations for 2 aberrometers: itrace and opd scan. *Korean Journal of Ophthalmology*, 22(4):210–213, 2008.

[74] Annie TT Ying and Martin P Robillard. Selection and presentation practices for code example summarization. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 460–471. ACM, 2014.

[75] Frans W Cornelissen, Enno M Peters, and John Palmer. The eyelink toolbox: eye tracking with matlab and the psychophysics toolbox. *Behavior Research Methods, Instruments, & Computers*, 34(4):613–617, 2002.

[76] Ashton Anderson, Daniel Huttenlocher, Jon Kleinberg, and Jure Leskovec. Discovering value from community activity on focused question answering sites: a case study of stack overflow. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 850–858. ACM, 2012.

[77] Nicolas Serrano and Ismael Ciordia. Bugzilla, itracker, and other bug trackers. *IEEE software*, 22(2):11–13, 2005.

[78] Jun Ye. Cosine similarity measures for intuitionistic fuzzy sets and their applications. *Mathematical and Computer Modelling*, 53(1-2):91–97, 2011.

[79] Daniel G Murray. *Tableau your data!: fast and easy visual analysis with tableau software*. John Wiley & Sons, 2013.