

EyeIDEA: Paving the Way Towards an Augmented Eye-Tracking IDE

Master's Thesis

Arjan Langerak

EyeIDEA: Paving the Way Towards an Augmented Eye-Tracking IDE

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

EMBEDDED SYSTEMS

by

Arjan Langerak
born in Dordrecht, the Netherlands



Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

EyeIDEA: Paving the Way Towards an Augmented Eye-Tracking IDE

Author: Arjan Langerak
Student id: 4211235
Email: a.c.langerak@student.tudelft.nl

Abstract

Since their invention, the keyboard and mouse are the most used input devices that software developers use to interact with source code. However, these devices have their IDE interaction issues as developers need to spend a significant amount of time learning how to use them efficiently and effectively. To tackle some of these issues, we used an eye-tracker to provide an alternative input method. With eye-tracking, it is possible to infer where developers are putting their attention to that can be utilized to improve the IDE user experience and productivity. Therefore, we present EyeIDEA, an experimental plugin for IntelliJ IDEA that integrates eye-tracking to provide source code navigation, debugging interactions, and fine-grained information about user habits based on eye-tracking. A user study was conducted with TU Delft students to investigate the perceived usefulness of our eye-tracking IDE. Based on their reactions, we found that the gaze-based interactions feel quick and natural. Moreover, there is a strong preference for eye-tracking interactions that only require a single code element or button as input. Additionally, users perceived that an eye-tracking IDE could substitute a mouse, especially when suffering from health-related issues or when performing light programming work. However, eye-tracking also brings new challenges, including integrating eye-tracking with keyboard/mouse input and accounting for accuracy and precision issues that influence the overall usability. A short video demonstration of our tool is available at <https://www.youtube.com/watch?v=ShvIX04rcr4>.

Thesis Committee:

Chair: Dr. Ir. W. P. Brinkman, Faculty EEMCS, TU Delft
University supervisor: Dr. M. Aniche, Faculty EEMCS, TU Delft

Preface

The year 2020 marked the beginnings of a new decade and what an *interesting* start it became. Empty roads and streets, face masks were the latest fashion trend, and working from home became the new standard as we all had to adapt to the coronavirus pandemic. New challenges popped up as staying focused and motivated became more difficult. However, thanks to all the people closest to me, I was able to complete my thesis successfully. Let me show you my gratitude with lots of love and appreciation.

It also marks the end of my nine-year journey as a student at the Delft University of Technology. Throughout this journey, I met lots of people and made friends with several of them. I want to take the opportunity here to thank them all as they helped and supported me. Without them, my life as a student would not have been the same. I would also like to thank my parents for their support throughout all my years at TU Delft and letting me stay over during the weekend when I moved out, especially during the corona epidemic.

During my Masters's Thesis, I was supervised by Maurício Finavaro Aniche for over a year. During this time, Maurício gave me the freedom to choose my own research topic and supported me through this endeavor. Your supportive feedback and enthusiasm helped me in several directions to complete this study. Without it, I doubt if I had chosen to pursue the topic of this thesis. Finally, I also would like to give a shout-out to Minaksie Ramsoekh for her help in the paperwork and reserving rooms for the user study as the TU Delft was under lockdown.

Arjan Langerak
Delft, the Netherlands
April 1, 2021

Contents

Preface	iii
Contents	v
List of Figures	vii
1 Introduction	1
2 Related Work	3
2.1 IDE Usage and Associated Challenges	3
2.2 Eye-tracking for Software Development Tasks	7
3 EyeIDEA	13
3.1 Architecture Overview	13
3.2 Plugin layout	14
3.3 Connecting to an Eye-Tracker	15
3.4 Eye-tracker Setup	16
3.5 Preprocessing Gaze Data	22
3.6 IDE GUI Mapping	24
3.7 Data Export	30
3.8 Executing Gaze-based Actions	32
3.9 Creating highlights on the screen	36
4 Research Design	41
4.1 Research Questions	41
4.2 Methodology	44
4.3 Data Collection and Analysis Procedure	50
4.4 Participants	55
4.5 Pilot study	55
5 Results	57

CONTENTS

5.1	Participants statistics	57
5.2	RQ1: IDE Usage	58
5.3	RQ2: EyeIDEA Perceptions	69
5.4	RQ3: Eye-tracking IDE Prospects	76
6	Discussion	83
6.1	Research Questions	83
6.2	Implications	85
6.3	Threats to Validity	85
7	Conclusions and Future Work	89
	Bibliography	91
A	Glossary	101
B	Interview Questionnaires	103
B.1	Demographic Questionnaire	103
B.2	SUS Questionnaire	104
B.3	Interview Questionnaire	104
B.4	Training Videos	108
C	Overview Card EyeIDEA	111
D	Invitation Poster for the User study	113
E	Survey invitation form	116

List of Figures

3.1	Overview of the architecture and their roles.	13
3.2	The layout of the EyeIDEA plugin inside IntelliJ IDEA.	14
3.3	Debugging Modes button panel with the two different tabs.	15
3.4	Instances of different eye-trackers.	16
3.5	Life cycle of the calibration process.	17
3.6	A calibration plot that illustrates the measurements taken.	18
3.7	The different components of the calibration result screen.	18
3.8	Illustration of degrees of visual angle.	19
3.9	Two different methods to calculate the precision.	21
3.10	The additional components of the validation result screen.	21
3.11	The construction of the gaze cursor.	22
3.12	Examples of filtering x-coordinates with different kernels.	24
3.13	Overview of mapping architecture.	25
3.14	Overview of IntelliJ IDEA ui	25
3.15	Overview of the Debugger Tool Window.	26
3.16	Overview of the Editor. 1) Opened files tabs 2) Code text box 3) Gutter . . .	27
3.17	An example output of the text mapper. The green blob represents the current gaze.	28
3.18	Example output of the Java mapper. The green blob represents the current gaze.	29
3.19	Example popup that uses a List to show items.	29
3.20	Example of a button(visual object) that is represented as a Part.	30
3.21	ER diagram of the database.	31
3.22	A snippet of the logging obtained with Logback.	32
3.23	The two different popups	33
3.24	The relations between each mode and how they revert back to the control mode. Note, it is always possible to select Mode 2, 3, 4, and 8 from any other mode if this is not already the currently active mode.	35
3.25	The three different Gaze Button states.	36
3.26	Overview of the process in creating highlights on the screen.	37
3.27	Example highlighting of the source code.	37
3.28	Two different examples of visual objects that are highlighted.	38

LIST OF FIGURES

4.1	Overview of the study procedure.	45
4.2	Overview of the comparative study procedure.	46
5.1	Individual plots of the top three transitions found in Table 5.4.	64
5.2	Individual plots of transitions between the DebugTool and Editor where the participants spend at least 500ms on the respectable window before transitioning.	65
5.3	Navigation and DebugTool transitions to the Editor when eye-tracking is available.	65
5.4	Individual plots of transitions between the RunTool and Editor where the participants spend at least 500ms on the respectable window before transitioning.	66
5.5	Individual plots of the training time results.	73
5.6	The total amount of faults made during the training session.	75
5.7	A mock-up of the ideas to change the current layout instead of the current fixed window at the top of the screen.	82

Chapter 1

Introduction

Software developers majorly use Integrated Development Environments (IDEs) to develop and maintain source code [35]. These programs incorporate all the necessary software tools that developers typically use during software development activities into a single environment, aiming to increase their productivity. Common activities that take place are browsing source code, navigation, and debugging [4, 47, 45].

However, developers need to spend a significant amount of time learning how to use the IDE. In a study about the usability of IDEs, Kline et al. [37] found that developers only use a small subset of the offered functionalities because it is too difficult to learn them all. Furthermore, several studies show that developers face significant overhead in navigating and debugging source code despite these functionalities. Minelli et al. [45] found that 18% of the time is spent on the user interface to read and browse source code. Additionally, Piorkowski et al. [54] found that 50% of the time is spent on foraging information during debugging.

An emerging Software Development field is the investigation of the usage of an eye-tracker to tackle these issues. With eye-tracking, it is possible to infer what developers are putting their attention to [18, 72, 57, 34] and also opens up new possibilities to improve the productivity of developers by analyzing and supporting their thoughts [67, 21, 3] and by adding interactions based on their gaze [22, 56, 60].

Hejmady et al. used eye-tracking to investigate how developers are using an IDE during debugging [29]. They used a novel mining technique to analyze the visual patterns of developers. This technique helped them in finding out that experienced developers used the IDE very differently than novice developers. Bednarik et al. also found similar patterns during their study [7].

To avoid the labor-intensive mapping of gaze to source code that typically happens in this type of research, a couple of studies created and published their tools that solves this issue. A promising tool that automates the mapping is EyeCode [28] but this only works for static images of the source code. A different approach was taken by Shaffer et al. [59] who integrates the eye-tracker into the Eclipse IDE such that the mapping from gaze to source code works on large code snippets and also offers the ability to investigate the behavior of developers outside a lab setting.

Other studies investigate gaze-based input methods to interact with the IDE to increase

productivity, mostly through improving navigation. These solutions are deployed as either a standalone IDE [22] or as plugins for IntelliJ [60] or Brackets.io [56]. These studies have shown that gaze input is a promising input technique to navigate source code.

For our work, we created a plugin for IntelliJ ¹, EyeIDEA, that incorporates the eye-tracking into the IDE. This plugin incorporates gaze-based actions such as navigating source code, perform debugging actions such as placing breakpoints, evaluate code, and stepping in and out of functions. To that end, the plugin correlates the gaze to the source code and other user interface elements such as the code editor, the debugging tool, buttons, and menus. These mappings are exposed such that we can use them to investigate how developers are using the IDE and the impact of the gaze input on their behavior.

We conducted a user study among TU Delft students to investigate the perceptions of our augmented eye-tracking IDE. In this study, the students performed two debugging sessions. During the first session, the students were not allowed to use gaze input. It was only allowed in the second session. Furthermore, they were free to use the IDE in their preferred way of debugging such that all of their actions were voluntary. After these sessions, we interviewed the students about their opinions of using eye-tracking as an interaction method.

Our main findings from the study are:

- Participants preferred to use the gaze-based action for navigation and setting breakpoints but preferred the mouse or keyboard for interacting with the debugger.
- Gaze input is perceived to be better suited for actions that only require a single selection, rather than rapidly repeated selections.
- Half of the participants found it difficult to switch between the gaze-based input and the mouse and keyboard input.
- The use of eye-tracking feels natural, but the inaccuracies contribute to usability issues, and a high mental load reduces the willingness to use gaze-based actions.
- On average, the implemented gaze-based actions are easy to learn, and participants made only a couple of mistakes.
- An eye-tracking IDE is perceived as useful for light programming work and in situations when a developer cannot use the mouse because of health issues or traveling.

The remainder of this paper is structured as follows: Chapter 2 describes the usage of eye-tracking within software research to explore how developers are programming and possible methods to use eye-tracking as an input mechanism. In Chapter 3, we present the architecture, the integration of EyeIDEA within IntelliJ, and how the interactions work. We discuss our research questions and the design of the user study in Chapter 4. The results of the user study are described in Chapter 5 and in Chapter 6, we discuss the outcome of the results and relate it to the research questions. Lastly, Chapter 7 concludes this paper and provides future work.

¹<https://www.jetbrains.com/idea/>

Chapter 2

Related Work

An upcoming trend is to investigate developers' behaviors with eye-tracking to improve their productivity. However, only a few of these studies incorporated eye-tracking into software development tools. Instead, most of them choose to use screen captures or images to correlate the gaze input to the code snippets or other Areas of Interest (AOI). Before integrating eye-tracking into an IDE, it is vital to know how developers use an IDE to avoid usability issues.

This chapter discusses studies about IDE usage and associated issues, research into using eye-tracking to aid in software development, and studies that have incorporated eye-tracking into software development tools. These tools help to map the gaze to source code automatically or to provide alternative navigation methods.

2.1 IDE Usage and Associated Challenges

One of the primary ways to develop software nowadays is with an IDE. Consequently, it has attracted many researchers to study IDE usage in order to improve them. Since an IDE offers a range of different tools, we focus on navigation and debugging within an IDE because these are part of our empirical analysis into an augmented eye-tracking IDE. Additionally, we discuss which role an IDE plays in the day-to-day activities of developers.

Day-to-Day Usage of IDEs Several studies investigated the usage of IDEs by instrumenting the IDE to obtain the executed actions and window events. Murphy et al. [47] investigated the usage of Eclipse by looking at the top used actions and opened windows. The opened windows gave an indication about which tools were active. Minelli et al. [45] collected all actions, window events, keyboard and mouse events, and time spent outside the Pharo IDE for the Smalltalk programming language. Then, they analyzed this data by creating “sprees”, a sequence of events, and turn these into understanding, navigation, editing, and GUI interaction activities [44]. Amann et al. [4] followed a similar approach in the investigation of Visual Studio. Additionally, they also analyzed the top used actions and tool usage.

2. RELATED WORK

These studies show that there are some overlapping behaviors in the usage of these IDEs. For instance, both Eclipse and Visual Studio studies show that the debugger is the second and third most frequently used tool. There are also some crucial differences between the usage of these IDEs. For example, Minelli et al. found that users spend on average 17% of the time configuring the Pharo IDE but only 3.5% of their time when configuring Visual Studio. Amann et al. hypothesized that this could originate from the different GUI concepts and that these indicators could be meaningful when designing IDEs. A similar issue is also addressed by Kline et al. [37]. They noted that developers found it difficult to learn and work with IDEs with poor toolbar and window organizations.

Navigation inside IDEs Another research field focuses on understanding how software developers navigate source code and how efficient they are in finding relevant code. Minelli et al. [46] found that developers performed between 1.5 and 19 times more navigation actions than the ideal case in the Pharo IDE. They argued that the ideal navigation is unfeasible as the relationships between code snippets are often hidden. Additionally, navigation actions are also part of constructing a mental model of a software system. However, the efficiency gap does suggest the need for better navigation methods.

This usability issue is also observed in Visual Studio by DeLine et al. [16], primarily with “re-finding code”, i.e., the developer already visited a code snippet but forgot the name of it or which Editor tab to click to find the code snippet. Most of the participants even closed all tabs at some point to start over again.

Pilzer et al. [53] tried to reduce this issue by predicting which opened windows are relevant. Before they created their detection tool, they conducted a study in which window events, user input data, and eye-tracking information was collected. Their results were, similar to DeLine et al. [16], that the amount of opened windows grow over time and are only closed to reorganize their workspace. Based on the results, they developed a tool that predicted which windows are irrelevant and achieved an 88.3% accuracy in predicting which windows were relevant. Eight out of twelve participants perceived the tool as useful. This result suggests that it can help in finding relevant windows and perhaps also tabs.

Ko et al. [38] found that on average, 27% of the navigation actions returned to code snippets that they recently visited in the Eclipse IDE. Additionally, 42% of the navigation actions were indirect, i.e., scrolling and visual searching through the displayed code. Some developers tried to reduce this overhead by placing bookmarks but had difficulties recalling which bookmark to choose. A suggestion to fix these issues was to provide better cues in the IDE to guide searches such that it leads to a reduction of the needed navigation steps.

One of these cues could be created by using past navigation action. Singh et al. [62] looked into various models to predict software developers’ navigation actions. The two main prediction methods are click-based or view-based operationalization. The click-based method means that a developer navigates the source code by clicking with the mouse on a tab or the code to go to the implementation, declaration, or usage. However, the limitation is that this method fails to record scrolling through the code, and clicking does not necessarily mean that a developer has its attention on that method. According to the researchers, these limitations were shown by Kevic et al. [36] that used gaze recordings from the iTrace tool to compare click-based navigation with eye-tracking. By contrast, the view-based method re-

lies on what is visible in the Editor, specifically in the middle of it. This method is somewhat similar to eye-tracking but with the limitation that a developers' attention is not necessarily in the middle of the screen when navigating. Their results showed that click-based navigation approximated the developers' navigation intentions while the view-based method did this poorly. Additionally, the best navigation predictor was to rank more-recently visited methods higher than those that are less-recently visited. That is in line with prior research that developers revisit the same code often.

Debugging Usage inside IDEs A number of studies have investigated developers' habits during debugging inside an IDE. In one of these studies, Petrillo et al. [52] investigated how developers use the debugger inside Eclipse. For this purpose, they created the Swarm Debug Infrastructure (SDI) that collects all debugger events and shows these on a separate dashboard. A preliminary test showed that SDI is effective in collecting and showing this data. However, they argued that they could not claim generalizations on the debugging habits they found during the study because that was not the goal.

Another method taken by Afzal et al. [2] was to mine the event data set of Visual Studio produced by Proksch et al.[55] for the MSR 2018 Mining Challenge. One of their findings was that developers start using the debugger early during debugging as in 80% of the cases, the debugger was used in under 13 minutes while the average debugging time was 45 minutes. One of their hypothesis was that developers tend to use the debugger on difficult-to-find bugs. These bugs require more time to debug, causing an increase in the total debugging time.

Hejmady et al. [29] used eye-tracking to investigate which code representations offered by the jGrasp IDE gets the attention of developers. They created a labeling scheme based on the gaze duration and the representations to investigate the visual patterns. The resulting labeling was mined with the Sequential Pattern Mining (SPAM) algorithm that finds frequent sequences. These were used to investigate how developers are switching between the different representations over time. This approach allowed them to investigate temporal debugging behavior. For example, they found that experienced programmers switched more often between the code and the output. This effect would not be found without taking time into account.

2. RELATED WORK

Topic	Authors	Study
Day-To-Day IDE Usage	Murphy et al. [47]	Monitoring Java developers using the Eclipse IDE in their normal software development activities.
	Minelli et al. [45, 44]	In depth analysis of how SmallTalk developers spend their inside the Pharo IDE.
	Amann et al. [4, 55]	Mining a previously collected Visual Studio IDE interaction dataset of C# developers to investigate the usage of this IDE .
	Kline et al. [37]	Empirical studies about usability issues of various IDEs for Java and C# developers.
IDE Navigation	DeLine et al. [16]	Observing how C# developers are using Visual Studio IDE during editing of unfamiliar code.
	Pilzer et al. [53]	Detect relevant windows during software development with eye-tracking to aid in navigation.
	Ko et al. [38]	Investigation of which strategies Java developers deploy to gather information in understanding unfamiliar code by using the Eclipse IDE.
	Singh et al. [62]	Testing the performance of predictive models of software development navigation.
	Kevic et al. [36]	Investigating navigation habits of developers in Eclipse IDE during a change task using IDE interactions and eye-tracking.
IDE Debugging Usage	Petrillo et al. [52]	Collect and visualize debugging activities inside Eclipse IDE.
	Afzal et al. [2, 55]	Exploring a Visual Studio IDE interaction dataset of C# developers into their debugging behavior.
	Hejmady et al. [29]	Investigating the visual attention patterns of Java developers using the JGRASP IDE during debugging.

Table 2.1: Summary of related studies about IDE Usage and Challenges grouped by research topic.

2.2 Eye-tracking for Software Development Tasks

In order to figure out how to deploy eye-tracking in the Software Development field, research is needed into how gaze information can be leveraged to aid developers during software development tasks. These investigations range from improving existing tools and methods such as code summarization to creating new tools that improve productivity by predicting if a developer is facing problems when writing code. The following paragraphs highlight some studies that argue that their proposed method could be useable in an IDE and studies that deployed and tested those tools in an IDE to see whether it is practical to have them.

Future Development of Eye-tracking Tools Some studies investigated the use of gaze information to create models and test how well they performed compared to similar studies.

Rodeghero et al. [57] investigated how developers read source code when summarizing it and how this information can be turned into an automatic code summarization tool. Their approach assigned weights to keywords inside a code snippet based on the reading process of developers. Their findings indicate that keywords are read differently by developers, especially if sections were harder to understand. Additionally, almost four out of five extracted keywords matched the selected keywords of the developers. Therefore, an eye-tracking-based method can find keywords that are important to create a summary.

A broader set of studies by Fritz et al. [21] looked into whether bio-metric sensors such as eye-trackers can be used to predict aspects such as code difficulty, coding progression, and interruptability, that relate to productivity. All of their studies included a multitude of different sensors and found that a combination of those sensors performed better as each sensor compensated for the disadvantages of other sensors. During their studies, eye-tracking was most commonly used as a pointing device to indicate which code snippets are investigated by the developer. The other sensors were used to predict various aspects such as the current emotion or mental load of a developer.

In conclusion, they found that biometric sensors have the potential to predict the mentioned aspects. These aspects can be used to create tools that are based on individual developers' behaviors instead of behavior based on all developers. Additionally, the output of these sensors can potentially be used for real-time analysis, compared to most metrics that rely on post hoc analysis.

Integrated Innovative Eye-tracking Tools into an IDE Investigating whether gaze information can predict developers' habits does not reveal if these tools are useful during software tasks. Therefore, studies created new development tools to test if developers find them useful.

One of these studies done by D'Angelo et al. [15] investigated if sharing gaze information could help developers during pair programming. In particular, they wanted to know how the incorporated non-verbal cues affected communication about code locations. This tool visualized which five lines of code were in the developers' field of vision. The visualization used two colors, yellow and green, to indicate if the developers are looking at

2. RELATED WORK

the same code at the same time. After their user experiment, they interviewed the developers and found that they preferred to use these visualizations instead of communicating about line numbers. They also adapted very quickly to this system. After a while, some developers did not explicitly notice the visualizations.

As mentioned in the previous paragraph, biosensors have the potential to be used to predict code difficulty. That is exactly what Hijazi et al. [30] tried to accomplish with their tool TellBack. This tool uses a heart rate monitor with an eye-tracker to detect the mental load of the developer. Additionally, it provides the specific code elements that the developer is struggling with.

Each code snippet under the gaze is labeled as either “difficult” or “not difficult”. It depends on the extracted features obtained by the heart rate sensor and eye-tracker if the code snippet is “difficult” or not. Their experimental evaluation showed that it achieved an accuracy of 83% in predicting the difficulty of a specific code element. However, further research is needed to see if this still holds in an industrial setting.

2.2.1 Gaze Mapping to Source Code

Work has been done to reduce the effort needed to use eye-tracking to investigate how developers read and use source code. At the time of writing this thesis, the current analytical eye-tracking software offered by companies is limited. In particular, the software makes a recording of the screen, and researchers have to label the locations on the screen that matches the source code or other objects. This process makes it labor-intensive, particularly if the contents shown on the screen can change at every moment. Therefore, some studies have created and published tools that automatically map the source code shown on the screen with the current gaze. The following paragraphs highlight these tools and the usability of these tools.

EyeCode This experiment by Hansen et al [28]. investigated code comprehension differences based on experience. They created a tool that automatically maps the gaze to different AOI’s such as code blocks, code lines, and individual code elements. Additionally, it included a statistical library to obtain metrics for the fixations and saccades, such as the fixation duration for specific code lines. However, this tool used a single static image as an input to create these AOI’s. Therefore, the experiment used small snippets of Python code that were under twenty lines of code. Additionally, they used unmodifiable code snippets because the task was to predict the output of the program. A possible side effect of this simplification was that none of the eye-tracking metrics correlated with programming experience. Other more complex experiments did find a correlation with programming experience [58, 7, 40].

However, they could moderately predict the reading behavior obtained from the eye-tracking metrics, such as which keywords are important. This result is in line with previous experiments that used manual analysis.

iTrace This plugin for the Eclipse IDE is developed by Shaffer et al. [59] with maps the gaze of a developer to the code shown in the Editor of Eclipse. They noticed that eye-

tracking is becoming more popular to study code development. However, there was no tool available that automatically maps the gaze to code and could be used on large code snippets where the code was allowed to be modified. Therefore, they created a plugin within the Eclipse IDE to map the gaze coordinates to code elements. After an experiment, the code elements can be exported and filtered with a fixation filter.

A revised version [24] changed the architecture such that it is less dependent on Eclipse to make it easier to support other IDEs such as Visual Studio IDE. This should make it easier to perform experiments in IDEs that are used for specific programming languages. As previously found by Amann et al. [4], there are significant differences in how IDEs are used. So, it is important to be able to study these differences.

iTrace have been used successfully to conduct experiments about software traceability [72], code comprehension [58, 32] and classifying the expertise of developers [11, 12] which means that researchers find this tool useful.

2.2.2 Eye-tracking for IDE navigation

A couple of attempts have been made to use eye-tracking as a navigation tool inside an IDE. Each attempt investigated various methods to deal with the inaccuracies of eye-tracking and how to integrate the eye-tracker with pre-existing input methods such as mouse and keyboard. The following paragraphs list each of these attempts and their findings, if applicable.

EyeDE Glücker et al [22] created a prototype IDE with eye-tracking interactions. Specifically, their work focused on hands-free navigation of source code. They implemented the interactions by using dwell-based mechanisms that activate a certain action. For instance, they implemented some contextual menus that can be brought up by looking at a specific button above the currently focused code element. A specific design choice was to avoid bright colors that could introduce unwanted distractions.

The evaluation results were that the participants described it as interesting and intuitive but had some difficulties with dwelling, especially when the gaze data was fluctuating. This issue made it difficult to activate a button. Their proposed solution was to add a key-press-based alternative to skip dwelling.

EyeNav A plugin for the text editor Brackets.io, developed by Radevski et al. [56] with the purpose to explore eye-tracking based code editing actions such as code scrolling and selection. They used keyboard shortcuts to change between the different actions because their focus was to use eye-tracking in combination with the keyboard, rather than to replace it completely. Unfortunately, there is no empirical study that has tested this plugin.

CodeGazer This study investigated how gaze input can be used for navigation actions and how it performs compared to a mouse or keyboard [60]. Shakil et al. developed a plugin for IntelliJ IDEA that adds a selection-based interaction system based on Actigaze [42] to support the navigation actions. This system uses highlights to indicate interact-able elements on the screen that can be selected by buttons that have the same color as these highlights. To

2. RELATED WORK

test their tool, they conducted two different user studies. The first study focused on comparing the performance and accuracy of their tool against the mouse and keyboard. They found that the gaze is, on average slower than a mouse but comparable to the keyboard. However, the performance is comparable to the mouse for actions for which one would typically have to move the mouse and press the mouse button. For instance, when navigating code, a user moves the mouse over a specific code snippet and then presses the mouse button along with a keyboard combination to trigger the navigation action.

The second study focused on assessing user preferences. During this study, the participants were free to choose any input technique during every navigation step. They found that most participants used gaze input for their primary input technique. The most popular action was to jump directly to the definition or usage of a specific code element whereas scrolling was perceived as the most difficult feature to use. Overall, the participants found CodeGazer intuitive and easy to use. They indicated that it would be useful for general code exploration but not for heavy development work.

Topic	Authors	Study
Future Development of Eye-tracking Tools	Rodeghero et al. [57]	Estimating important Java source code keywords by using Eye-tracking for source code summarization.
	Fritz et al. [21]	Investigating the use of biometric sensors to develop prediction models for code difficulty, progression and interruptibility of a developer.
Integrated Innovative Eye-tracking Tools into an IDE	D'Angelo et al. [15]	Using eye-tracking to visualize which code lines in the Visual Studio IDE are in the vision of developers in order to help with communication during pair programming.
	Hijazi et al. [30]	Measure code difficulty with eye-tracking and heart rate sensor and locate the relevant code inside the editor of the Eclipse IDE.
Gaze Mapping to Source Code	Hansen et al. [28]	Automatically mapping gaze patterns to static images of Python source code to investigate code comprehension.
	Shaffer et al. [59, 24]	Research project to incorporate automatic mapping of gaze patterns to source code within Eclipse IDE and Visual Studio IDE.
Eye-tracking for IDE Navigation	Glücker et al. [22]	Prototype build IDE with gaze enabled navigation actions that makes use of contextual menu's and buttons.
	Radevski et al. [56]	Using gaze input for code edition and navigation actions in Brackets.io IDE that uses the gaze to integrate it with keyboard usage.
	Shakil et al. [60]	Investigate the usage of gaze enabled navigation actions within IntelliJ IDEA by using a select and confirm scheme.

Table 2.2: Summary of related studies about using eye-tracking for software development tasks grouped by research topic.

Chapter 3

EyeIDEA

This chapter describes the plugin design, the integration of the eye-tracker within IntelliJ IDEA, and the gaze-based interaction mechanism. The current implementation supports gaze analysis of Java and Text Files and detects if a gaze is inside an Area Of Interest (AOI) such as the Project View or Editor. Additionally, a complete analysis of the visual objects within the debugger is also available. The plugin is designed to be flexible and modular, such that it can be expanded and modified to support other research interests.

3.1 Architecture Overview

The architecture of EyeIDEA consists out of five layers, each with its own roles.

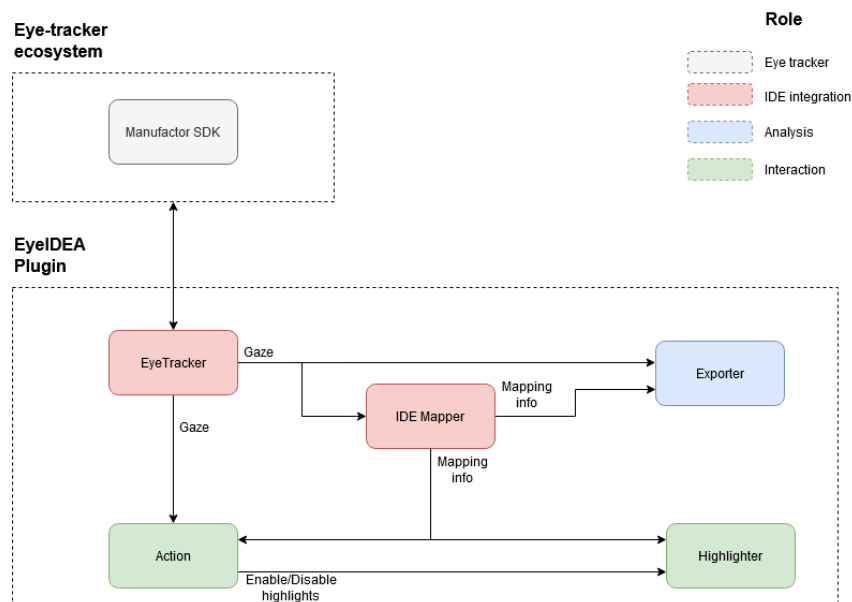


Figure 3.1: Overview of the architecture and their roles.

3. EYEIDEA

Each of these layers communicates to each other through a messaging system. This system ensures that additional layers can use the information that is already available to expand the current system.

The *EyeTracker* and *IDE Mapper* are labeled as IDE Integration because these layers are responsible for connecting to the eye-tracker ecosystem and integrate this into the IDE. Then, the *Exporter* takes this mapping information and saves it to an external information source such as a database system.

The mapping information is also used to provide gaze-based interactions through the use of the *Action* and *Highlighter* layer. The *Highlighter* adds and removes transparent color boxes that are used to highlight specific visual objects on the screen. These highlights are added or removed by the *Action* layer. This layer uses highlights to indicate that specific objects can be selected when a specific action is activated. These actions are activated with gaze-activated buttons that are added to the IDE.

3.2 Plugin layout

The plugin layout of EyeIDEA adds four different panels, three of which provide eye-tracking interactions by using gaze buttons (Section 3.8), and the remaining panel is to interact with the eye-tracker. This is illustrated in Figure 3.2.

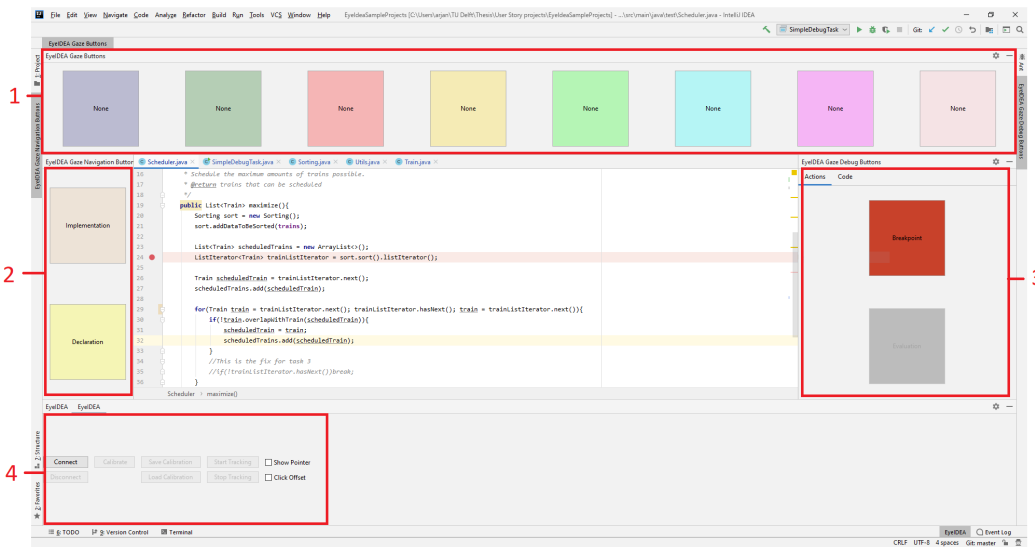


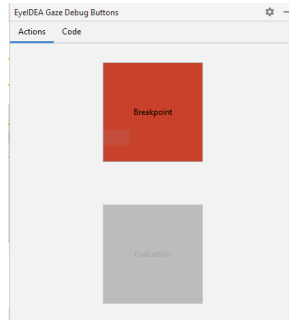
Figure 3.2: The layout of the EyeIDEA plugin inside IntelliJ IDEA.

Panel 1 is the *Gaze action button panel* and is located on top. It consists out of eight gaze buttons. Each of these buttons has a distinct color that corresponds with the used colors from the highlighting feature described in Section 3.9. The purpose of these buttons is to perform the action that is associated with them. This process is explained in Section 3.8.

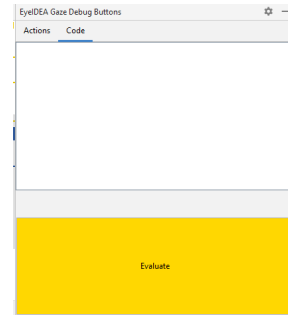
Panel 2 is the *Navigation Modes panel* and is located on the left side. It consist out of two gaze buttons. These buttons are used to select the *implementation* or *declaration* modes

which are described in Section 3.8.1.

Panel 3 is the *Debugging Modes button panel* and is located on the right side. It consists out of two different tabs. The first tab, *Actions*, consists out of two gaze buttons that are used to select the breakpoint and evaluation modes. This tab is shown in Fig. 3.3a.



(a) The Action tab.



(b) The Code tab.

Figure 3.3: Debugging Modes button panel with the two different tabs.

The second tab, *Code*, consists out of a text area and a gaze button. This tab is only used by the conditional breakpoint and evaluation window modes. The text area is used to insert code, while the function of the button depends on which of the two aforementioned modes is currently active. This will be explained in Section 3.8.1.

Panel 4 is the *Control panel* and is located at the bottom. It consists out of mouse-controlled buttons to interface with the eye-tracker. These buttons are responsible for connecting and disconnecting from the eye-tracker, starting and stopping the eye-tracking, and creating, saving, and loading the calibration. Additionally, two toggle buttons are used to enable or disable optional features. The *Show Pointer* button is to visualize the gaze by drawing a bubble on the screen that will be described in Section 3.4.3. The other button, *Click offset*, enables a gaze correction feature based on mouse clicks. This feature will be explained in Section 3.5.2.

3.3 Connecting to an Eye-Tracker

Many manufacturers are introducing eye-tracking devices on the market, each using their proprietary software. An adapter is needed to make sure that our plugin has the option to support the different eye-trackers and their software.

This adapter, *AEyeTracker*, describes the functionalities that are used within the plugin to communicate with an eye-tracker effectively. *AEyeTracker* sends the gaze information from the eye-tracker to the gaze messaging channel. Additionally, it also applies the filter described in Section 3.5.3 to the gaze information and sends it out.

A separate messaging channel is used to provide other functionalities, for example, to start the device. These messages are put into a separate channel to create a distinction between controlling the device and receiving gaze information. Gaze information is different

3. EYEIDEA

from control information because it is used by multiple layers while control information is only meant to be used by the *Control panel*.

An overview of this adapter is shown in Fig. 3.4.

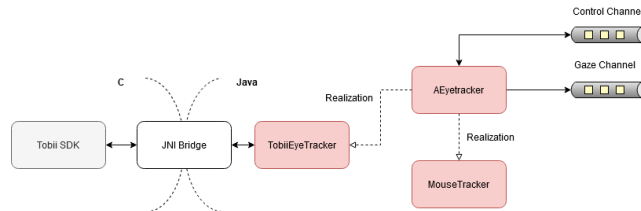


Figure 3.4: Instances of different eye-trackers.

For this thesis, we use an eye-tracker from Tobii [69] which comes with its own SDK's. However, none of these are written in Java, so a bridging component is developed with JNI¹ to have access to the required functionalities within Java. Additionally, another adapter is created for the mouse to simulate an eye-tracker for testing purposes.

3.4 Eye-tracker Setup

Before an eye-tracker can be used, it is recommended to calibrate the device. This calibration process is needed to create a 3d model of the users' eye since some parameters cannot be estimated without measurements [25]. Therefore, we have added a validation step to estimate the accuracy and precision of the eye-tracker to evaluate the current calibration.

All the measurements of both processes and the validation calculations are saved on disk. This also includes screenshots of both calibration and validation plots. An example plot is illustrated in Fig. 3.6.

Besides the validation process, we also added a gaze cursor option to visualize the gaze movements. This option can be used to let users test the current setup in the tool itself or use it as a pointer, similar to a mouse pointer.

3.4.1 Calibration

To calibrate an eye-tracker, the user starts by focusing on specific predefined locations on the screen. In turn, the eye-tracker uses these known locations together with additional information such as the eye-trackers distance from the screen and the screen dimensions to calibrate itself.

To provide this functionally, we developed an external Java application that provides a GUI that is used to perform the calibration. The GUI comes preconfigured with the typical three, five, or nine uniformly distributed calibration points on the screen. These points are

¹[https://openjdk.java.net/groups/hotspot/docs/RuntimeOverview.html#JavaNativeInterface\(JNI\)|outline](https://openjdk.java.net/groups/hotspot/docs/RuntimeOverview.html#JavaNativeInterface(JNI)|outline)

placed by using a relative coordinate system in order to support multiple screen resolutions. EyeIDEA itself uses nine calibration points since this configuration is typical for a fullscreen eye-tracking application. Additionally, this configuration balances between relatively good accuracy and precision and the duration of the calibration process.

During this calibration, the current calibration target shrinks to guide the users' eyes towards the center. In the next step, the eye-tracker collects the data it needs for the calibration. When it has collected enough data, the calibration target returns to its original size. If there are other destinations left, the target moves to that location. Otherwise, the eye-tracker computes the calibration, completing the process. This process is illustrated in Fig. 3.5.

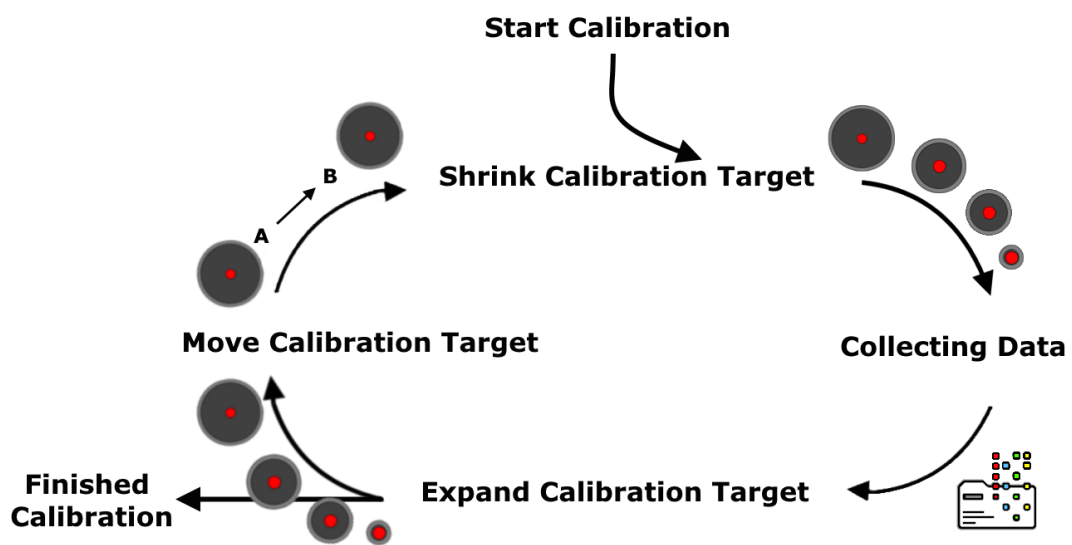


Figure 3.5: Life cycle of the calibration process.

After performing the initial calibration, the results are shown on the screen. These results show the measurements of each eye for each calibration point. Additionally, there are lines drawn from each calibration point to the measurements that belong to this point. A calibration plot is illustrated in Fig. 3.6.

Fig. 3.7a shows a zoomed-in illustration of a calibration point. The orange lines represent measurements taken from the left eye, and the green lines are measurements from the right eye.

The relationship between measurements and calibration points is also visualized by clicking on a point with the left mouse button. This action will show which measurements belong to the selected point. This is shown in Fig. 3.7b.

There also two filter buttons to only show the results of the left or right eye. These options can help to determine if the center correction (Section 3.5.1) needs adjusting for the current user. These buttons are shown in Fig. 3.7c.

If there is a calibration problem with one or more calibration points, it is possible to

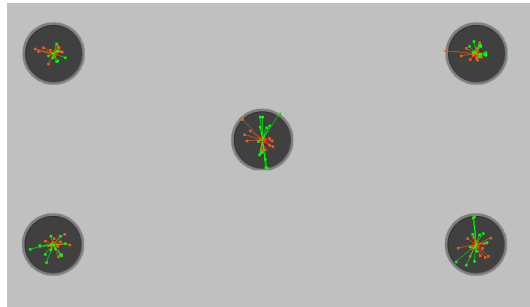


Figure 3.6: A calibration plot that illustrates the measurements taken.

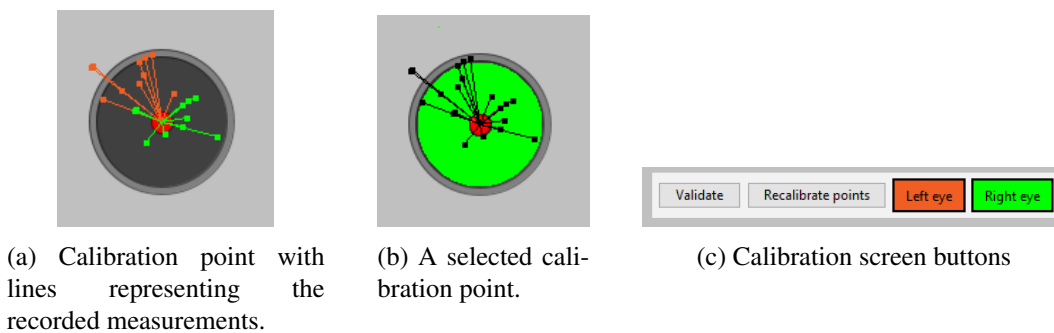


Figure 3.7: The different components of the calibration result screen.

select these points to rerun the calibration for only these. Otherwise, the calibration can be verified by running the validation. Each option is available by clicking on the buttons shown in Fig. 3.7c.

3.4.2 Validation

Understanding the performance of the eye-tracker is important to evaluate the quality of the eye-tracking data. For this study, we have chosen to evaluate the quality immediately after the calibration to ensure that the calibration process was successful. It is recommended by Holmqvist et al. [31] to validate the calibration to make sure that the eye-tracker performance is good enough for the current experiment. Accuracy and precision of the eye-tracker are indicators of the expected quality of the calibration. These will be explained later in this section. Both indicators provide information about the deviation between what the user is looking at and what the eye-tracker registers. These indicators are expressed in degrees of visual angle, which is the angle between two straight lines from two different points on the screen to the eye, as illustrated in Fig. 3.8. This type of measurement is often used in eye-tracking research since it takes the distance between the user and the screen into account instead of only using the distance between two points on the screen.

The following paragraphs explain the accuracy and precision indicators and the calculation of them.

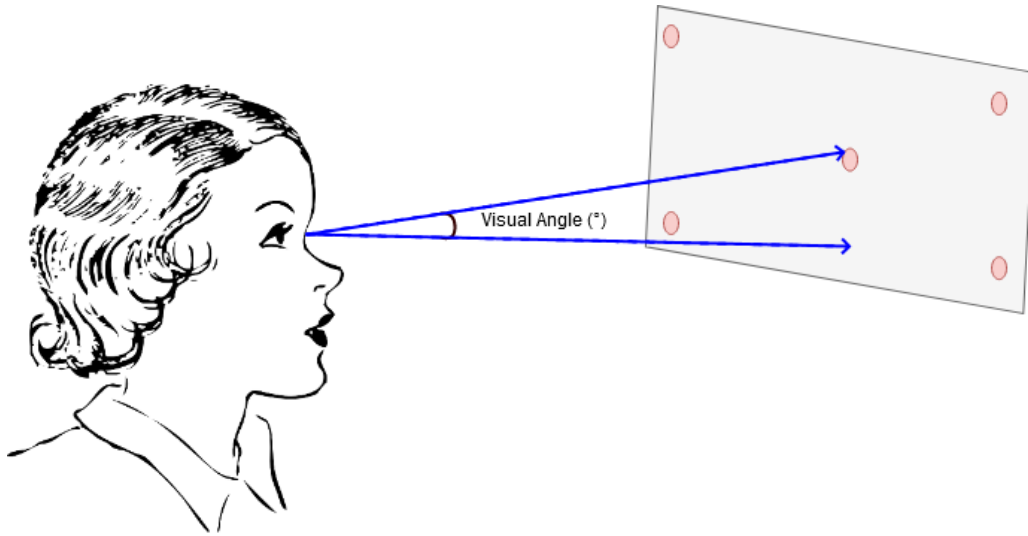


Figure 3.8: Illustration of degrees of visual angle.

Accuracy The average difference between the target calibration location and the measured gaze location [31]. To express this difference in degrees of visual angle, we start with averaging all measurements into a single point. Then, we calculate two 3D vectors, one for the averaged measurements and the other for the target location. The origin of both of these vectors is located at the users' eye as illustrated in Fig. 3.8. We use the mean of all the origin points for the two vectors since we are averaging all the measurements.

The construction of the measurement vector is straightforward. The used eye-tracker can report the measurements in the User Coordinate System ² which means that the measurements are already available as 3D points. The measurement vector is constructed according to Eq. (3.1).

$$\overrightarrow{OriginDataVector} = (Data.x - Origin.x, Data.y - Origin.y, Data.z - Origin.z) \quad (3.1)$$

To construct the target vector, we need to denormalize the 2D target point to the User Coordinate System. Each target point is normalized by using the top left corner of the screen as (0,0) and the bottom right as (1,1). We denormalize this point back by using the screen information taken from the eye-tracker and Eq. (3.2).

$$\begin{aligned} dx &= (ScreenTopRight - ScreenTopLeft) * Target.x \\ dy &= (ScreenBottomLeft - ScreenTopLeft) * Target.y \\ 3DTarget &= ScreenTopLeft + dx + dy \end{aligned} \quad (3.2)$$

Essentially, we calculate the width and height of the screen and multiply it with the x-coordinate or y-coordinate respectively of the target to obtain the coordinate in the User

²<http://developer.tobiipro.com/commonconcepts/coordinatesystems.html#UCS>

Coordinate System. Since the target does not move on the z-axis, we do not need to calculate the z-axis displacement separately.

With the constructed vectors, the visual angle is equal to the angle between these vectors. Since we are only interested in the direction of the vectors, they are normalized before the angle is calculated. To calculate this angle, we have used the Apache math3 API ³ that uses the cross product for vectors that are almost aligned and the dot product when they are not almost aligned.

Precision The variation between each successive measured gaze location, i.e., the ability of an eye-tracker to reproduce the gaze measurement [31]. It is common to calculate this value by using angular distances θ , which are defined as degrees of visual angle between successive gaze measurements. These distances are obtained by calculating the angle between two 3D Vectors as shown in Fig. 3.8. This process follows the same steps as described in the Accuracy paragraph. Then, the precision is calculated via the Root Mean Square(RMS) of these angular distances (see Eq. (3.3) where n is the number of measurements).

$$\theta_{RMS} = \sqrt{\frac{1}{n} \sum_{i=1}^n \theta_i^2} = \sqrt{\frac{\theta_1^2 + \theta_2^2 + \dots + \theta_n^2}{n}} \quad (3.3)$$

A second option to describe the precision is to compute the standard deviation of the measurements. In particular, the angular distance ϕ between each measurement and the mean of these measurements. This is equivalent to using the RMS normalized by the mean [68] and the equation is equivalent as that of Eq. (3.3). Both of the mentioned methods are illustrated in Fig. 3.9.

³[https://commons.apache.org/proper/commons-math/javadocs/api-3.6.1/org/apache/commons/math3/geometry/euclidean/threed/Vector3D.html#angle\(org.apache.commons.math3.geometry.euclidean.threed.Vector3D,%20org.apache.commons.math3.geometry.euclidean.threed.Vector3D\)](https://commons.apache.org/proper/commons-math/javadocs/api-3.6.1/org/apache/commons/math3/geometry/euclidean/threed/Vector3D.html#angle(org.apache.commons.math3.geometry.euclidean.threed.Vector3D,%20org.apache.commons.math3.geometry.euclidean.threed.Vector3D))

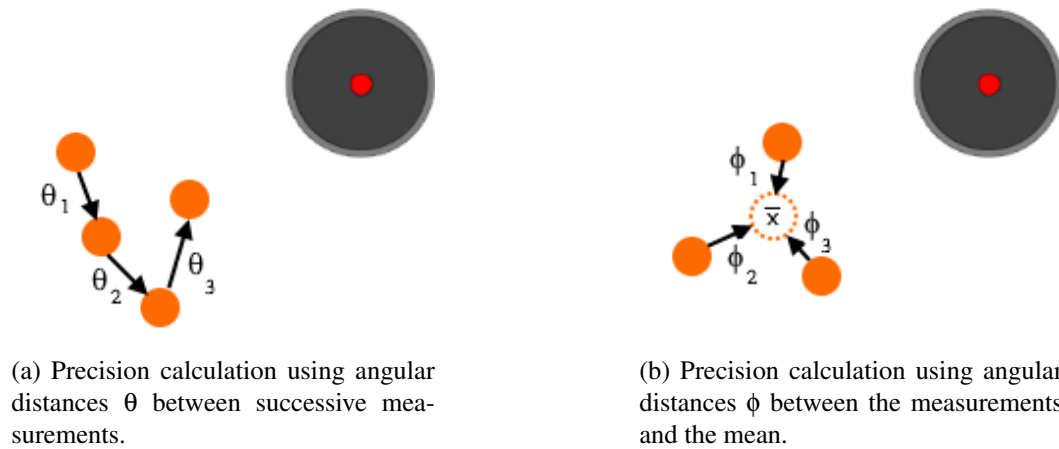
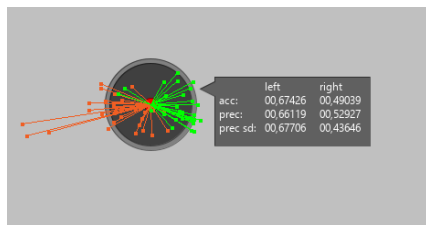


Figure 3.9: Two different methods to calculate the precision.

Both the accuracy and precision are calculated for each eye individually. To calculate these values, we use a new set of measurements that are taken after the calibration. The process of collecting these measurements is exactly the same as that of the calibration process, illustrated in Fig. 3.5. However, the difference is that the used Tobii eye-tracker operates in normal mode instead of calibration mode. The normal mode sends gaze information to the tool, whereas the calibration mode does not.

When all the measurements are complete, the results are shown on the screen. Initially, only the collected measurements are shown as illustrated in Fig. 3.7a. Hovering with the mouse over a calibration point reveals a popup with the accuracy and precision information as shown in Fig. 3.10.

The rest of the interface is similar to the calibration screen. It contains filter buttons that filter the left or right eye measurements. Additionally, the validation can be rerun by selecting one or multiple calibration points and then click on the appropriate button. The other two buttons either start the calibration again or closes the validation window. Fig. 3.10b shows the buttons to select these options.



(a) Calibration point with precision and accuracy statistics.



(b) Validation screen buttons.

Figure 3.10: The additional components of the validation result screen.

3.4.3 Gaze Cursor

The gaze cursor provides the user feedback based on where the eye-tracker thinks the users' current focus is. This cursor shows the differences between the users' focus and the eye-trackers understanding of the believed focus. Depending on this difference, it can be recommended to either instruct the user to re-adjust their head and posture, to adjust the gaze position using the methods described in Sections 3.5.1 and 3.5.2 or in the worst-case to re-calibrate the device again.

By default, a bubble represents the gaze on the screen. An algorithm draws a fictive circle around n successive measurements and merges these to form a bubble. This is illustrated in Fig. 3.11 for $n=2$.

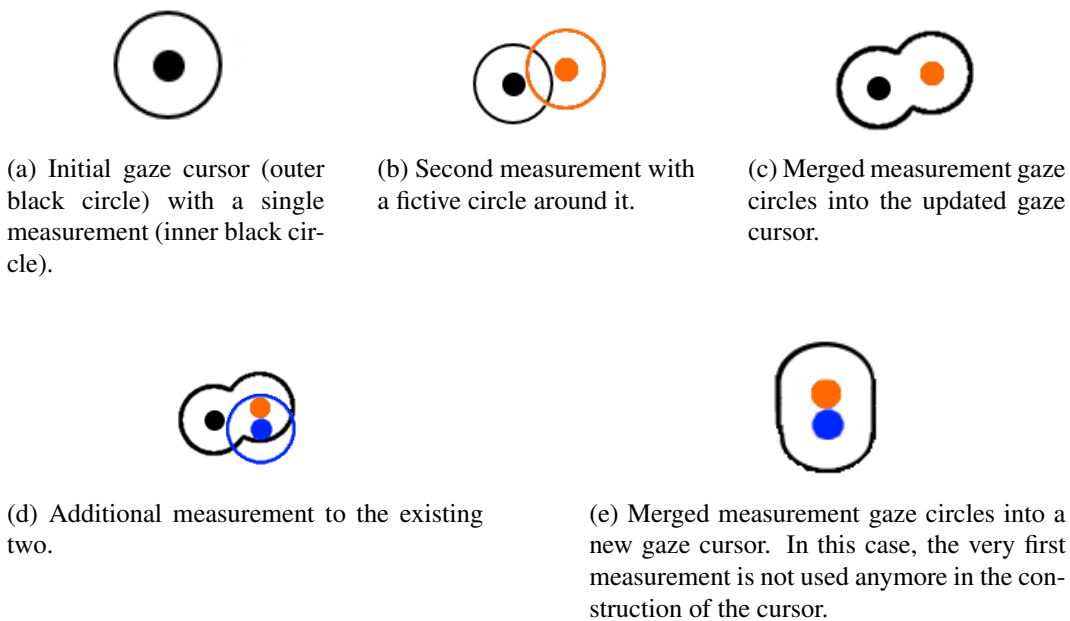


Figure 3.11: The construction of the gaze cursor.

Besides this implementation, the cursor overlay API also supports displaying a single dot for the current gaze measurement and multiple gaze cursors for different gaze data. However, EyeIDEA displays a single gaze cursor for the filtered gaze data since that is what we needed for our user study.

3.5 Preprocessing Gaze Data

Not all the raw data that is coming from the eye-tracker is useful. The data contains noise that is caused by involuntary saccade movements [14], head drift, and other measurement noise. Online correction and filter methods can correct these effects while eye-tracking is active. The following subsections explain the different options that are available within EyeIDEA.

3.5.1 Center Correction

This option can adjust how much weight should be given to each eye. Equations (3.4) and (3.5) show the center position is calculated.

$$x = x_{left} * (1 - weight) + x_{right} * weight \quad (3.4)$$

$$y = y_{left} * (1 - weight) + y_{right} * weight \quad (3.5)$$

The default configuration uses $weight = 0.5$. This configuration corresponds to an ideal situation in which both eyes behave identically. However, in some situations, one of the eyes corresponds more strongly with the users' vision than the other eye. In this case, changing the weight could compensate for this effect by relying more on the data for that particular eye. Setting the $weight < 0.5$ increases left eye usage, while setting the $weight > 0.5$, the right eye will be used more.

3.5.2 Click Drift Correction

There can be a displacement between the measured gaze position and what the user thinks that the center position should be. This displacement can occur because of head movements that the eye-tracker cannot adjust by itself or because the user sees the center differently than the eye-tracker expects. A simple algorithm can correct this displacement by adding it to the reported gaze data. The user can activate this by enabling the *Click drift* option. While this option is enabled, every left click with the mouse will adjust the gaze data by calculating the difference between the clicked location and the current gaze data location.

Using mouse clicks to adjust the gaze data is predominantly used when a webcam is used as an eye-tracker [50, 66].

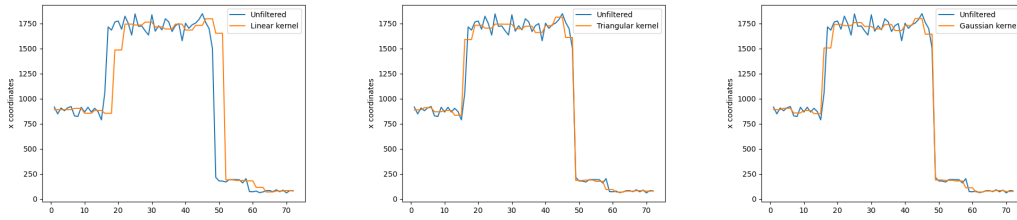
3.5.3 Filtering

Filters can improve the quality of the registered gazes and reduce the effort needed to analyze the recording. For interactive gaze support, all filtering needs to be done in real-time to keep the interaction responsive. This situation means that only online filters are possible. EyeIDEA incorporates a couple of different filter designs based on a Finite Input Response (FIR) design. This type of filter is recommended by Feit et al. [19]. An FIR filter creates a weighted average based on n gaze inputs.

$$X_t = \sum_{i=0}^n \frac{w_i}{\sum_j w_j} \cdot X_{t-i} \quad (3.6)$$

Equation (3.6) shows the general formula for the FIR filter where X_{t-i} stands for the gaze input at time $t - i$ and w is the calculated weight. Three different kernels are implemented to calculate these weights: linear, triangular, and Gaussian.

Linear The Linear kernel uses $w_i = 1$. This corresponds to averaging the last n gaze points.



(a) Filtering with a Linear kernel. (b) Filtering with a Triangular kernel. (c) Filtering with a Gaussian kernel.

Figure 3.12: Examples of filtering x-coordinates with different kernels.

Triangular The Triangular kernel uses Eq. (3.7) to calculate the weights. This formula assigns a higher weight to more recent points with increments of one. The result is that recent gazes are made more important than older ones.

$$w_i = n - i + 1. \quad (3.7)$$

Gaussian The Gaussian kernel uses Eq. (3.8) to calculate the weights. This formula assigns weights based on a Gaussian function. Recent gazes are assigned higher weights than older ones, similar to the **Triangular** kernel. However, this kernel uses non-linear smoothing instead of linear. This result changes the importance of older gazes as they decrease at a different rate than the **Triangular** kernel.

$$w_i = e^{-\frac{i^2}{2\sigma^2}} \quad (3.8)$$

3.6 IDE GUI Mapping

The eye-tracker provides gaze information that establishes a relation between the users' gaze and an (x,y) coordinate on the screen. However, this relationship is not sufficient to tell which visual object is relevant to the developer. A connection between the current gaze and the visual objects on the screen is needed to establish that. However, this is not an easy task since the IDE is developed with the Java Swing GUI toolkit that is designed to handle the keyboard and mouse input. The eye-tracker must either simulate the keyboard and mouse input or implement new methods to provide similar functionality.

After some trial and error, we concluded that it is easier to use a combination of these methods. For this alternative solution, all visual objects from the GUI need to be inspected to create a relation between the GUI and the current gaze location.

However, if a process needs to inspect the complete GUI every time a new gaze message comes in, it could become problematic to process everything before the next gaze is available. To mitigate this, the GUI is subdivided into several regions that we call *Components*. If a gaze falls into the visual area of the *Component*, the process only needs to inspect the visual objects within this area. Section 3.6.1 explains this further.

These visual objects also need their own representations in order to be identifiable. To that end, *Parts* are created. A *Part* is an abstraction of a visual object in the IDE, and it contains both visual and other information that relates to this object. *Parts* will be explained in Section 3.6.2.

The starting of the GUI mapping is handled by the *Mapper*. It is configured to take in the gaze information from the gaze channel and pass it into the available *Components*. Then, *Mapper* publishes the discovered *Parts* together with the corresponding *Component* on its own separate channel. Note that the *Mapper* uses the original gaze information and not the filtered gaze information. The *Mapper* uses this because it makes sure that it is possible to apply other filters and calculation methods on the original data after the experiments.

A complete overview of the mapping process is illustrated in Fig. 3.13.

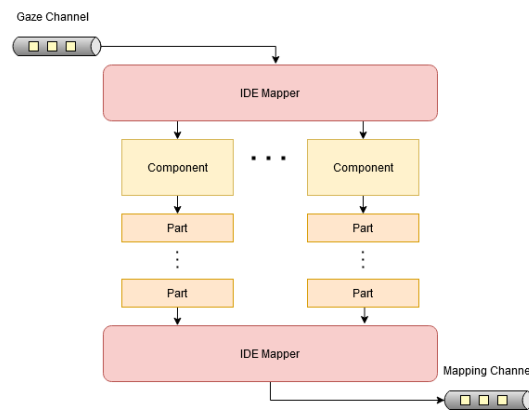


Figure 3.13: Overview of mapping architecture.

3.6.1 Components

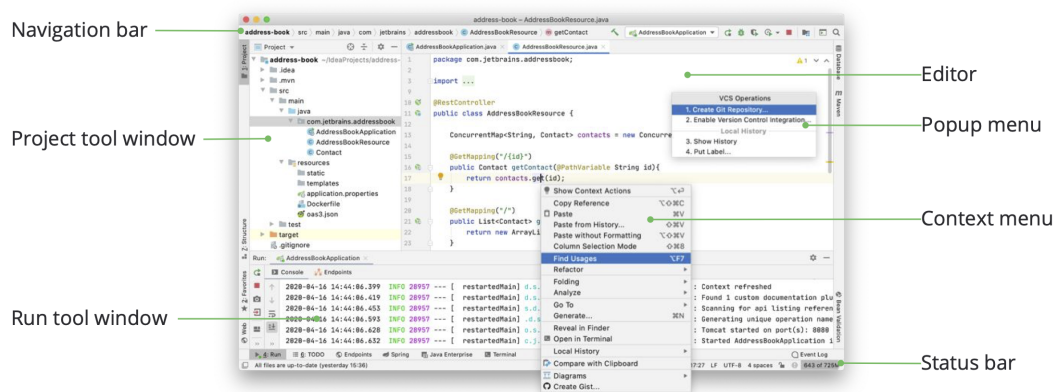


Figure 3.14: Overview of IntelliJ IDEA GUI, taken from their help guide⁴

3. EYEIDEA

A *Component* is a representation of a specific Window in the IDE. We classify these windows into a couple of main categories: the menubars, the editor, and the various tools. Some of these windows can be seen in Fig. 3.14.

Each window consists out of different visual objects. Therefore, a *Component* consists out of different *Parts*. A *Component* is thus responsible for selecting the *Parts* that corresponds to the gaze location. Unfortunately, due to some technical challenges regarding overlapping *Parts* from other *Components*, we needed to hard code these overlaps.

Another technical challenge we faced is in discovering which visual object is inside a Window. Luckily, IntelliJ provides a testing tool called UI Inspector ⁵ that tells us which classes belong to a visual object. However, this means that the construction of these *Components* and *Parts* relies on manually mapping the visual objects, which is very time-consuming. Therefore, not all visual objects are mapped to *Components*.

The following subsections will explain the different *Components* that EyeIDEA currently has implemented.

Tool window An IntelliJ Tool window provides various perspectives and access to regular development actions such as project navigation, running status, and debugging. The EyeIDEA plugin also adds several Tool windows to provide information and interactions with the eye-tracker.

Each Tool window consists out of two top-level *Parts*: the title bar and the content pane, as illustrated in Fig. 3.15.

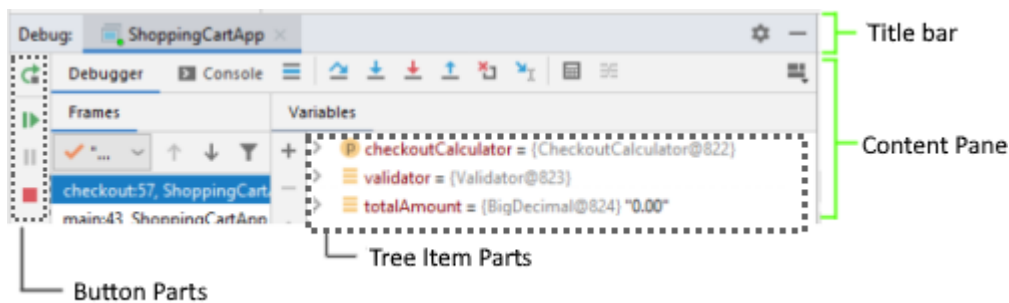


Figure 3.15: Overview of the Debugger Tool Window.

Additionally, the IntelliJ SDK provides which Tool windows are available. Altogether, this makes it possible to create a general *Tool Window Component* that can be used to map every Tool window. However, the content of the content panes differs between each Tool, which means that this *Component* only maps the title bar and Content Pane.

Besides the general *Component*, EyeIDEA contains a content pane mapping for the Debug tool since the debugger is one of the areas of interest for this research. Figure 3.15 illustrates two different kinds of *Parts* that are mapped. However, other visual objects such as the “Debugger” and “Console” tabs are also available as *Parts*.

⁴<https://www.jetbrains.com/help/idea/guided-tour-around-the-user-interface.html>

⁵https://jetbrains.org/intellij/sdk/docs/reference_guide/internal_actions/internal_ui_inspector.html

Editor In the Editor window, a scroll-able text box shows the source code with line markers beside it in a window called the *gutter*. Above the text box are tabs that are used to navigate between currently opened files. All of these items are shown in Fig. 3.16.

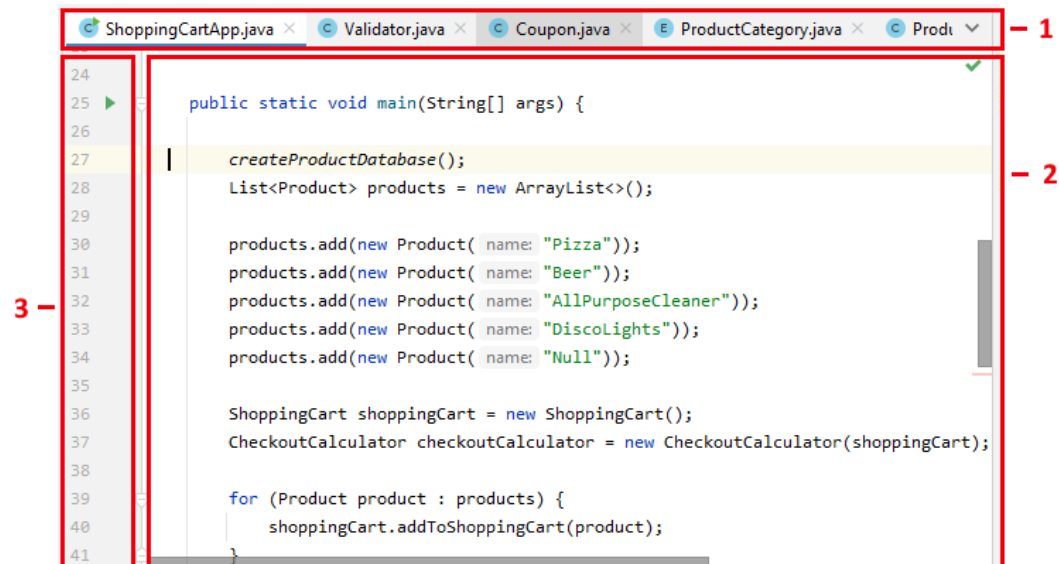


Figure 3.16: Overview of the Editor. 1) Opened files tabs 2) Code text box 3) Gutter

Each item in Fig. 3.16 is represented by a *Part*. Additionally, the system also maps the source code shown in the text box onto different *Parts*. However, unlike the other *Parts* that use an (x,y) coordinates system, these code entities use a different positioning system. This system uses the number of characters from the top left of the text box. This positioning is also called the logical position. Luckily, IntelliJ already provides methods that transform (x,y) coordinates to the logical position. From there, EyeIDEA maps the text to *Parts* by using two different mapping methods, the *Text Mapper* and *Java Mapper*.

Text Mapper The *text mapper* obtains the corresponding text by starting with scanning the characters left and right from the logical position. Then, it maps the gaze coordinates to the character, word, and line inside the text box. This is illustrated in Fig. 3.17.

Java Mapper The *Java mapper* maps the gaze coordinates to Java source code entities. This method makes use of IntelliJ's Program Structure Interface (PSI)⁶ which is responsible for parsing files and creates both semantic and syntactic code models. The *Java mapper* translates the logical position into a description of the corresponding Java source code entity by using the PSI model of the currently selected file. This description includes the text and an identification label. This label describes the type of entity, namely, a class, method, control statement, or variable. Alongside this description, the *Java mapper* also includes the entity position in the text box and its relative position to other identified source code

⁶https://jetbrains.org/intellij/sdk/docs/basics/architectural_overview/psi.html

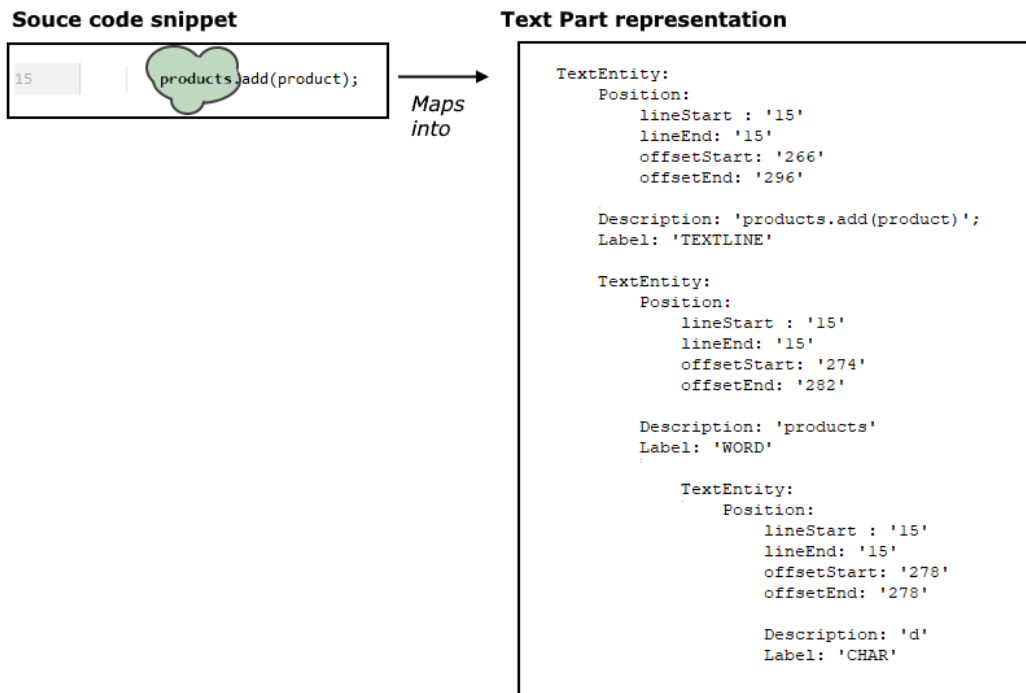


Figure 3.17: An example output of the text mapper. The green blob represents the current gaze.

entities. The last one is included to improve the mapping accuracy in case of code changes. An example of this process is illustrated in Fig. 3.18.

Popup A popup is a special type of GUI Window shown on top of other windows. This window automatically disappears when it is no longer active. Examples of these are shown in Fig. 3.14 as the *Popup Menu* and *Context Menu*. We have identified and implemented two different types of popups, namely popups with a list and with a table. Both of these consists out of separate items. For instance, Fig. 3.19 shows a popup that contains a list where each item represents the file location and code entity of the `SlowTrain` class.

3.6.2 Parts

A *Part* is a representation of a visual object on the screen. Most of these visual objects are implementations of IntelliJ IDEAs' version of Java Swing Components⁷. However, not all objects correspond to Swing Components. For instance, the GUI representation of source code elements is just text. However, the underlying PSI structure interprets it as Java source code objects. This is illustrated in Fig. 3.18.

⁷<https://docs.oracle.com/javase/tutorial/uiswing/components/index.html>

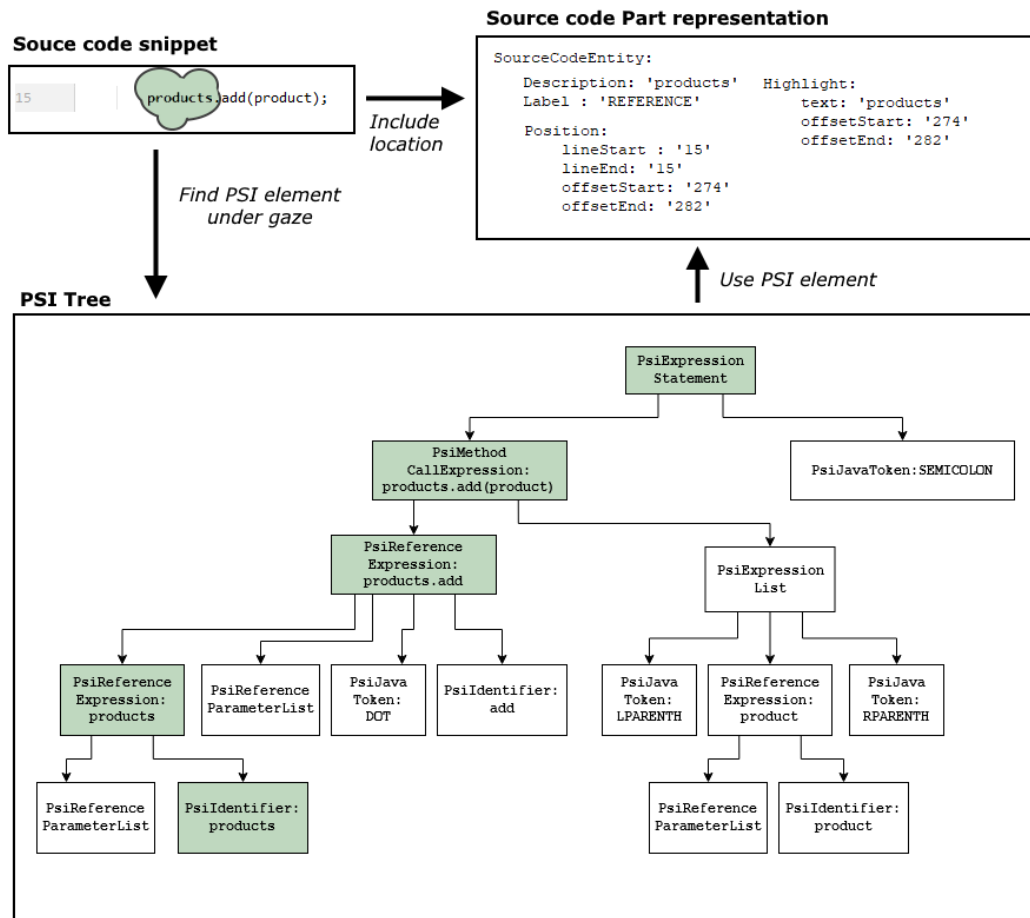


Figure 3.18: Example output of the Java mapper. The green blob represents the current gaze.

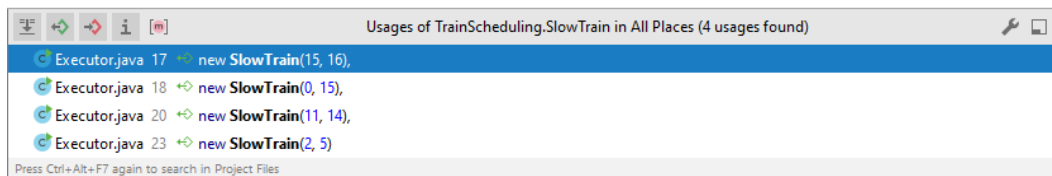


Figure 3.19: Example popup that uses a List to show items.

Besides representing source code, there is another reason why *Parts* represent a visual object, namely that some visual objects exist out of multiple objects. For instance, a list is a visual object on its own. However, the items inside the list are also separate visual objects. A *multi-Part* represents this relation by forming a linked list.

3. EYEIDEA

Therefore, a *Part* represents visual objects and provides access to the underlying visual objects whenever possible. With this information, the *Mapper* determines the object's location, which is needed to check if the current gaze coordinates fall onto it. For instance, Fig. 3.20 illustrates the process of representing a button as a *Part*.

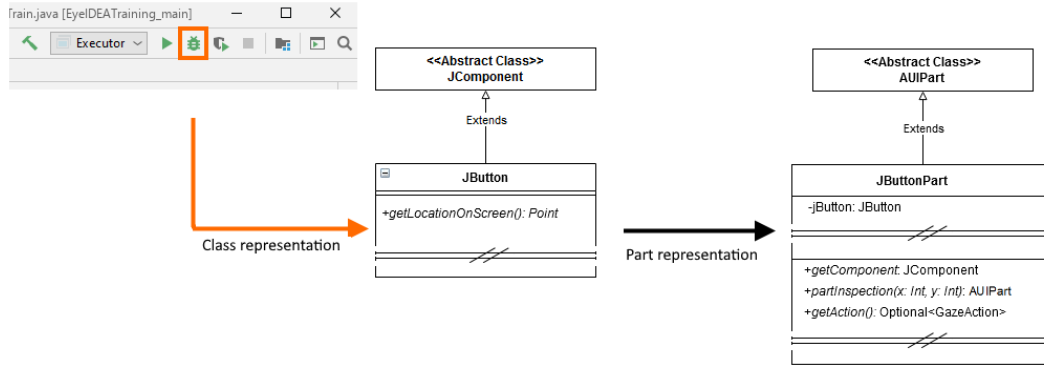


Figure 3.20: Example of a button(visual object) that is represented as a *Part*.

The result is a relationship between the gaze and the visual object. This relationship is not limited to knowing the visual object's type. It also enables interactions with this object if possible. These interactions will be explained in Section 3.8.

3.6.3 Mapper

The *Mapper* is responsible for detecting which *Component* and associated *Parts* can be mapped onto the current gaze during the discovery phase and making this information available to other processes.

To start the discover phase, the *Mapper* sends the gaze information to all the configured *Components*. These *Components* delegate the gaze to specific *Parts*. Eventually, the data has reached all *Parts* that are inside the gaze, and the discovery phase is completed.

After this discovery phase, the *Mapper* publishes the complete chain of the *Parts* together with the *Component* that corresponds to the given gaze.

With this process, it is possible to decide later on which visual objects are the most relevant to use. For instance, previous research into eye-tracking for software development used Area Of Interests that corresponds to an entire window [29] while others define more specific AOIs [36].

The described process is illustrated in Fig. 3.13.

3.7 Data Export

EyeIDEA generates a lot of data that could be useful for further analysis. This data can be grouped as gaze input (Section 3.3), IDE mappings (Section 3.6) and action usage (Section 3.8). The data is extracted by two different processes, the *Exporter* and the *Logger*.

These differ in their methods of extracting and exporting the data. The following subsections explain these processes.

3.7.1 Exporter

The *Exporter* shown in Fig. 3.1 exports the data from the eye-tracker and the *IDE Mapper* (Section 3.6) to a database. This database contains four different tables that are illustrated in Fig. 3.21.

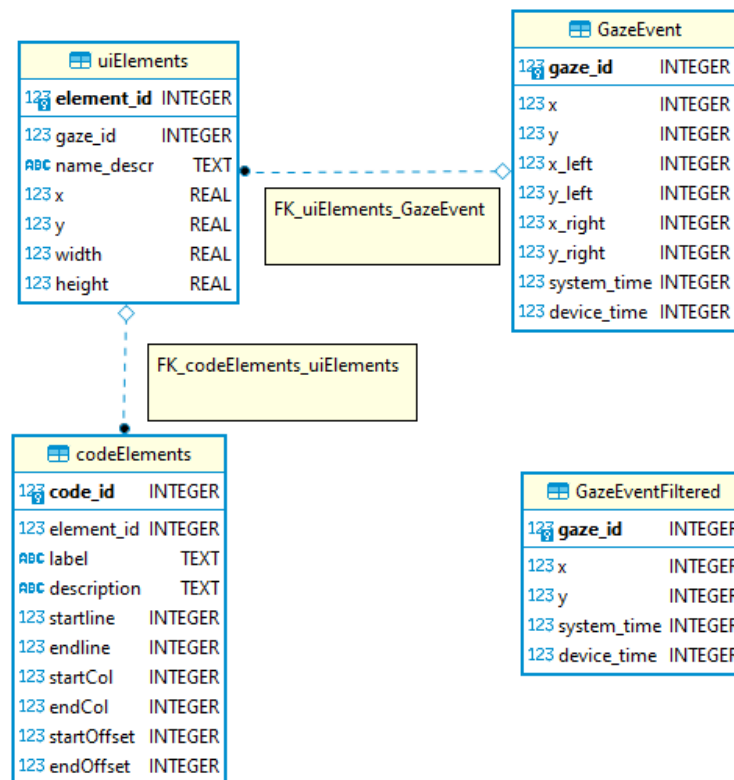


Figure 3.21: ER diagram of the database.

The first table, *GazeEvent*, stores information about the gaze itself. This table includes the individual (x,y) coordinates of both eyes and the combined coordinates using the method described in Section 3.5.1. Additionally, the time received from the eye-tracker and the current system time is included as well.

The second table, *uiElements*, stores the location, dimensions, and description of the mapped elements. These elements are either a *Component* (Section 3.6.1) or a *Part* (Section 3.6.2). Furthermore, the table uses a constraint on the gaze id from the corresponding gaze that is stored in the *GazeEvent* table.

The third table, *codeElements*, stores the code entity mappings. It is separated from the *uiElements* table because the location data is not an (x,y) coordinate but rather the position

3. EYEIDEA

inside the Editors text box, see Fig. 3.16. However, code entities are still *Parts*. Thus a constraint is placed between the corresponding element ids.

The fourth and final table, *GazeEventFiltered*, stores the results from the filtered gazes. This table has no relation to the other tables because the mapping process does not use the filtered gazes as mentioned in Section 3.6.

3.7.2 Logger

The purpose of the *Logger* is to record the performed gaze actions and which modes have been selected (Section 3.8). In order to record this, we use the Logback development tool⁸. This tool is one of the most well-known logging frameworks for Java, and it allows us to log the selected gaze actions and modes. A small snippet of the logged results is shown in Fig. 3.22.

```
2020-11-11T08:49:13,766Z INFO toolwindow.gazeAction.GazeButton - Informing listeners to do action, timestamp: 1605084553766
2020-11-11T08:49:13,766Z INFO actions.mode.ModeController - Changing Mode to actions.mode.ImplementationMode@1324b594
2020-11-11T08:49:17,791Z INFO toolwindow.gazeAction.GazeButton - Informing listeners to do action, timestamp: 1605084557791
2020-11-11T08:49:17,791Z INFO a.action.GoToImplementationAction - Do Action
2020-11-11T08:49:17,791Z INFO actions.mode.ModeController - Changing Mode to actions.mode.ButtonMode@660ae690
```

Figure 3.22: A snippet of the logging obtained with Logback.

3.8 Executing Gaze-based Actions

To execute an action, EyeIDEA needs to know what the intended action is of the developer. In order to figure this out, we created a distinction between selecting the action itself and selecting appropriate information if needed. For example, interacting with a regular button requires only clicking on that button. However, other actions such as navigating to a function require knowing that the developer wants to navigate and the function that the developer selected. Therefore, EyeIDEA uses *Modes* that control which *Action* can be executed and the different transitions between the Modes. Some *Modes* need additional input such as code selection. A straightforward method would be to let users dwell on additional input, such as a code entity, to select this as input. However, this solution suffers from the Midas Touch problem[33] which is the problem that it feels like every location where you look can trigger an action. Additionally, many additional inputs such as code entities are too small to be used as targets to use a dwelling method for [19]. Shakil et al. [60] approached this issue by highlighting the target and defer the actual activation of the *Action* to a separate target, a button, which is bigger than the actual target. All of the processes to execute a gaze-based action (except the highlighting that is described in Section 3.9) will be explained in the following subsections.

3.8.1 Modes

A *Mode* is the current interaction state of EyeIDEA and is responsible for executing the right *Action* with a certain input. A Mode uses a two-step approach similar to CodeGazer

⁸<http://logback.qos.ch/>

[60] to select the input. The first step is that every valid *Part* (Section 3.6.2) under the current gaze is highlighted with a unique color which corresponds to the eight Gazebuttons shown in panel 1 of Fig. 3.2. Then, the user can select this *Part* as input by dwelling on the gaze button with the same color as the highlight.

Table 3.1 lists all the available modes with a description and which the input is needed to execute the associated action.

Note that both Selector modes do not have a valid *Part*. The reason is that during this mode, a popup is shown on the screen to select other modes that do not require any additional input. These popups are shown in Fig. 3.23. Additionally, the *Call Evaluation Window* mode calls the Evaluation dialog directly and therefore does not need any other *Part*.

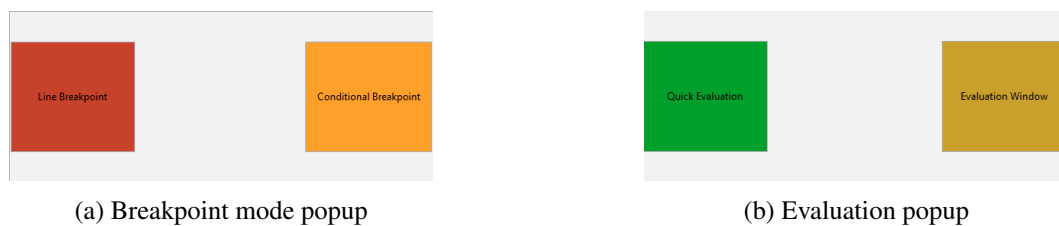


Figure 3.23: The two different popups

EyeIDEA makes use of gaze-selectable buttons so the user can select the listed modes with their eyes. These buttons are located in Panels 2 and 3 shown in Fig. 3.2. Panel 2 uses two gaze buttons to select the Implementation and Declaration modes directly. The other panel consists out of two tabs as explained in Section 3.2. The first tab, *Actions* has two buttons that trigger either the Breakpoint or Evaluation Selector mode. When either of these modes is selected, one of the popups from Fig. 3.23 will appear on the screen. Each of these popups has two buttons to select Modes 5, 6, 9, and 10, respectively. Fig. 3.24 shows how the relations between each mode. Additionally, Modes 7 and 11 can be selected by using the button in the *Code* tab shown in Fig. 3.3b.

However, as Fig. 3.24 illustrates, the Control Mode is not actively selected by the user but is selected by the system whenever the other modes are deselected. This mode is the default mode of EyeIDEA. Normally, a user selects a button with the mouse that only requires the user to click on that button. Therefore, if a user also had to select the Control Mode first, it will add another step in this process which adds more complexity. Another reason for this implementation is to test how much of a hindrance it is to select a mode from the user's perspective. During the user study, we will ask the participants about their experience with Mode selection. It will be helpful to the participants if they have a reference for the two different situations.

3.8.2 Actions

An *Action* executes one or more commands by optionally using the supplied input. This input can be specific visual objects, code entities, and character messages. The purpose of these commands is to interact with the IDE. For example, to start the debugger or to jump to

3. EYEIDEA

Mode	Description	Valid Parts
1. Control	Trigger a supported button or select an item from a supported list or table	Button, list item, table item
2. Implementation	Navigate directly to the implementation of a code entity	Code entity
3. Declaration	Navigate directly to the declaration or usage of a code entity	Code entity
4. Breakpoint Selector	Select between the Line Breakpoint and Conditional Breakpoint mode	-
5. Line Breakpoint	Set a Breakpoint at a specific line by using the position of a code entity	Position of a code entity
6. Conditional Breakpoint Input	Select and paste code entities that are used in the construction of a condition	Code entities
7. Set Conditional Breakpoint	Set a Breakpoint with the constructed condition at a specific line by using the position of a code entity	Position of a code entity
8. Evaluation Selector	Select between the Line Breakpoint and Conditional Breakpoint mode	-
9. Evaluation	Evaluate an existing code entity in the Editor	Code entity
10. Windowed Evaluation Input	Select and paste code entities that are used in the constructing of an arbitrary expression	Code entities, Line positions
11. Call Evaluation Window	Call the Evaluation dialog with the constructed expression	-

Table 3.1: Implemented modes in EyeIDEA. Only valid Parts can be selected when the mode is active.

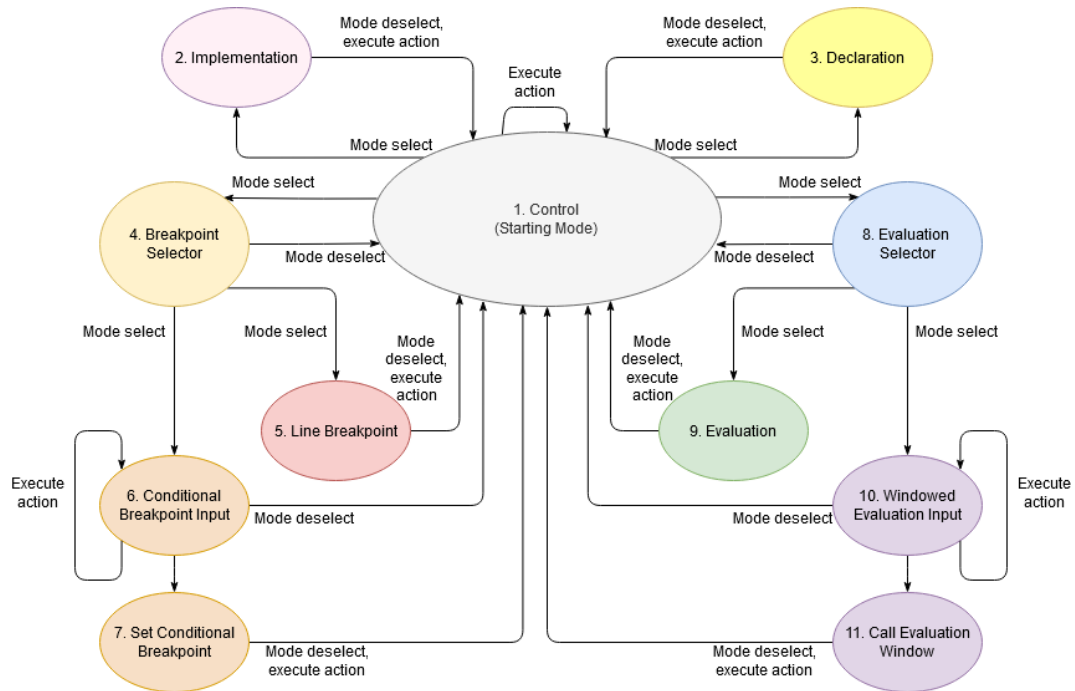


Figure 3.24: The relations between each mode and how they revert back to the control mode. Note, it is always possible to select Mode 2, 3, 4, and 8 from any other mode if this is not already the currently active mode.

the declaration of a code entity. Some of these commands correspond to commands already available in the IntelliJ SDK or commands specific to EyeIDEA.

However, not all commands are created this way. Instead, they are executed by emulating either a mouse click or a keyboard shortcut. For instance, to select an item from a list. The reason to use emulation in some situations is that they are not available through the SDK, and they are problematic to implement. IntelliJ expects a mouse or keyboard input in these situations and then interacts with many unknown parts to update the IDE.

Luckily, emulating mouse clicks does not interfere with using the mouse, but we could not locate the needed keyboard shortcuts in the system files of IntelliJ. Therefore, it is not possible to change the keyboard shortcuts because we manually added the key combinations to EyeIDEA.

3.8.3 Gaze Button

The Gaze Button allows a user to interact with EyeIDEA with their eyes. Each of these buttons is large enough to compensate for the inaccuracies of the eye-tracker. Feit et al. [19] recommends using a target that is at least 1.9x2.35cm in size to allow reliable interactions. The chosen target size is 2.7x2.7cm because this size was a better balance between reliable interaction and the overall screen estate to accommodate the buttons.

Each button has three different states, inactive, hover, and active. In the *inactive state*, the user has not selected the button. During *hover state*, the current gaze is on the button, and in the *active state*, the user has selected it.

Only a single mode is active, and thus only a single button from among the mode selection buttons can be activated. Therefore, the other buttons are automatically set to the inactive state if needed.

To activate the button, a user needs to dwell on the button long enough such that the button transitions from the hover state to the active state and vice versa to deselect the button. A button shows this state to the user by changing the color of the button to different shades of blue for the hover and activated state as shown in Fig. 3.25.

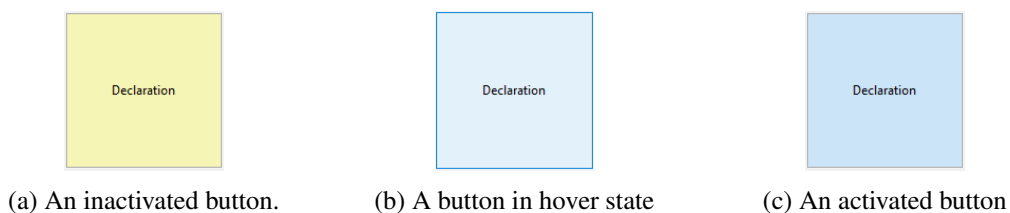


Figure 3.25: The three different Gaze Button states.

3.9 Creating highlights on the screen

EyeIDEA highlights items on the screen to indicate which items can be used as additional input to the currently selected *Mode* as explained in the previous section. There are two different technical implementations, one to highlight the code entities and the other to create highlights for all the other *visual objects*. The Editor of IntelliJ has a build-in highlighter which makes highlighting the code entities straightforward. However, no such highlighter exists natively for the other visual objects, so we implemented this.

A highlighting algorithm determines which item is highlighted and then uses one of these implementations. Depending on the currently active mode, only a specific subset of the *Parts* that the *Mapper* (Section 3.6.3) provides is used. Additionally, the algorithm can use a different set of colors in case of color blindness. These steps are illustrated in Fig. 3.26 and are explained in the upcoming subsections.

3.9.1 Highlighting Code Entities

As mentioned in the introduction, IntelliJ has a built-in highlighter to highlight text. By using the code mapping process explained in Section 3.6.1 Java Mapper, we can ask IntelliJ to highlight the text. The result of these highlights is shown in Fig. 3.27.

This system works with priorities to determine which color it should assign in case multiple processes want to highlight the same piece of code. Unfortunately, IntelliJ uses a priority system to assign the highlight, and it replaces our highlight with another one when another process uses the highest highlight priority.

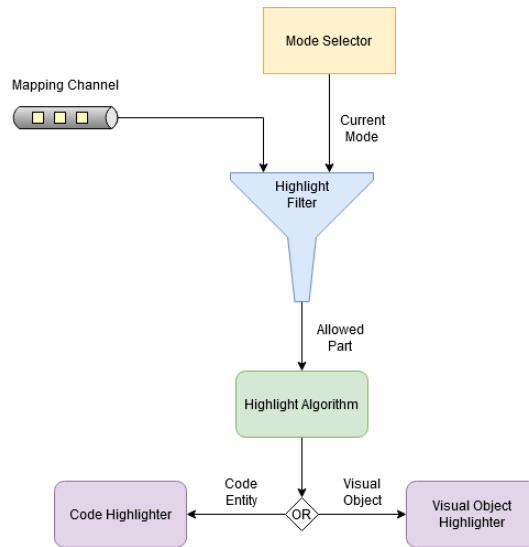


Figure 3.26: Overview of the process in creating highlights on the screen.

```

for(Train train = trainListIterator.next();
    if(!train.overlapWithTrain(scheduledTrain)
        scheduledTrain = train;
        scheduledTrains.add(scheduledTrain);
}
  
```

Figure 3.27: Example highlighting of the source code.

We know one instance where this happens, and that is when a whole line of the editor is highlighted when the debugger stops at a breakpoint.

Another issue that arises is that the used highlight colors are using the RGBA color model⁹. However, the highlighter of IntelliJ does not support transparent colors. This issue means that we need to blend the colors to obtain the right color. The result of Eq. (3.9) blends the given color with a background color. The chosen background, bg , is the same as that of the Editor of IntelliJ.

$$blend_{RGB} = src_{RGB} \cdot src_{alpha} + bg_{RGB} \cdot (1 - src_{alpha}) \quad (3.9)$$

3.9.2 Highlighting Visual Objects

The background color could be changed, or a highlight can be drawn on top of the object to highlight the visual objects. However, since the objects can only be accessed by discovery as explained in Section 3.6, any modifications done to the objects can result in unexpected behavior. An encountered example of this was that some objects kept reverting to the original background color. Therefore, we implemented the second option to draw the highlight

⁹https://en.wikipedia.org/wiki/RGBA_color_model

directly on top of the object. By using the width, height, and location of the object, a rectangle is drawn on the screen using a Glass Pane ¹⁰ which makes it possible to draw on top of IntelliJ. The used color for the highlight has a transparency value, so the underlying visual object is still visible. Two examples of these highlights are shown in Fig. 3.28.

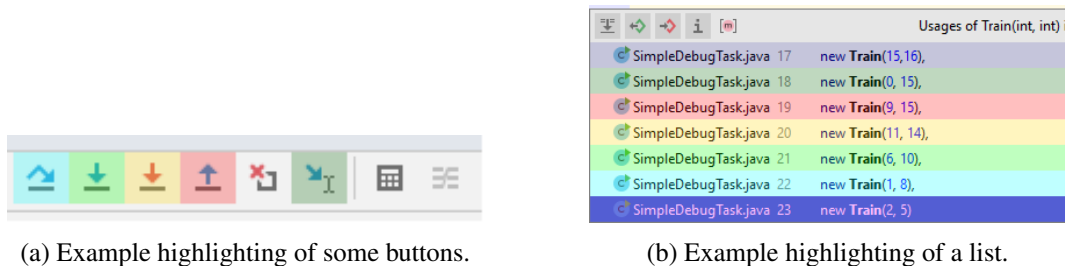


Figure 3.28: Two different examples of visual objects that are highlighted.

However, there are issues with this approach too. The first issue is if the underlying visual object changes color, it affects the used highlight because this approach blends the highlight color with the color underneath it. This can be seen by comparing Fig. 3.19 with Fig. 3.28b. The highlight on the last item on the list in Fig. 3.28b has a darker color because it blended with the highlight that can be seen in Fig. 3.19.

The second issue is related to the use of the GlassPane. The GlassPane catches any mouse clicks that are in the highlighted locations. Luckily, we found a solution when the IDE is running in *Microsoft Windows*. By changing the `WS_EX_TRANSPARENT`¹¹ Window Style constant of this pane, it instructs *Microsoft Windows* to make this specific window invisible to mouse clicks, effectively letting the mouse clicks pass through the GlassPane to the underlying visual object.

3.9.3 The Highlighting Algorithm

The highlights are used in combination with the action buttons described in Section 3.8, and there is a maximum of eight different colors available. It would not be possible to assign every element a unique color because there are far more selectable items on the screen than those eight, especially when the current mode requires a code entity as input. Additionally, the number of code elements can fluctuate, which means that two nearby code elements may wind up with the same color because these elements are put close together through a code change. This situation can cause flickering due to constant highlight removal to satisfy the one-to-one correspondence between a color and the selected element.

Therefore, an algorithm is needed that assigns these highlights. This algorithm needs to satisfy the following two requirements. Firstly, it should only create eight distinct highlights where the colors have a one-to-one correspondence to the action buttons. This way, there are no items that receive the same color. So, the algorithm assigns up to eight different

¹⁰<https://docs.oracle.com/javase/tutorial/uiswing/components/rootpane.html#glasspane>

¹¹https://docs.microsoft.com/en-us/windows/win32/winmsg/extended-window-styles#WS_EX_TRANSPARENT

colors that are evaluated at every incoming gaze. The algorithm then assigns a highlight on a valid item within this gaze, using the different highlighting options from the previous subsections. If all eight colors are used, the algorithm removes the highlight from the least recently looked item and uses this color for the next highlight.

Secondly, the assigned color should remain stable, i.e., it should not constantly change the color when the gaze is near the highlight and then on top of the highlight again. The algorithm remembers the color of the assignment to achieve this, and it is only changed when the algorithm removed the highlight to avoid assigning the same color twice.

3.9.4 Filtering Objects To Highlight

Depending on the mode, different *Parts* are eligible to be selected as input. The output from the *Mapper* must be limited to only the eligible parts to make sure that no other part can be selected. Therefore, a filter can be placed between the *Mapper* and the algorithm to allow certain parts. This filter is part of the currently active mode.

There are two different filter implementations used. The first only allows code entities to be highlighted, and the second only allows buttons that trigger specific IntelliJ actions. However, we designed the filter implementations such that they are easily adjusted to filter other parts if a new mode requires that.

3.9.5 Color blind mode

We chose the current colors for the highlighting using the website <https://mokole.com/palette.html>. These colors are tweaked somewhat because transparent colors have a lighter shade, so a small compensation was needed. However, for people that are color blind, the initially chosen colors can be very similar to each other. Therefore, we created three alternative color schemes that either avoid red, green, or blue tints. We tested these colors with the tool found on the website <https://www.color-blindness.com/cobli-s-color-blindness-simulator/>. This tool simulates a specific type of color blindness by reducing the affected tints in the uploaded image.

Chapter 4

Research Design

4.1 Research Questions

The main goal of this study is to get an initial impression of the developers' behaviors and perceptions when using an eye-tracking augmented IDE for debugging tasks. In particular, we choose to focus on debugging tasks as these kinds of tasks consist primarily out of navigation and understanding source code. As several studies show [47, 4, 45], developers spend most of their time on these activities when using an IDE.

To investigate this, we specifically focus on how developers interact with EyeIDEA, what the perceived advantages and disadvantages are, and prospects for using eye-tracking inside the IDE. Our goal is to identify relevant research areas for further development of an eye-tracking augmented IDE. Therefore, the main research question is:

MRQ: *Do developers perceive the eye-tracking augmented IDE as useful?*

Grudin [23] describes that the usefulness of a system can be determined by considering two quantities, usability and utility. *Utility* refers to if the developed system has a recognizable purpose for the intended users, whereas *usability* refers to how easy it is to learn and use the system. We consider three factors to determine the perceived usability and utility. These factors are the IDE usage, the difference between using eye-tracking and currently available inputs such as the mouse and keyboard, and the developers' perception. In the following subsections, we will explain and formulate the research questions based on these factors.

4.1.1 Research Question 1

IDE Usage An IDE provides multiple development tools to assist developers in the creation and understanding of source code. Developers interact with these tools through the GUI windows or by using hotkeys to trigger a specific action.

Consider, for example, a scenario in which a developer creates new functionality for existing code. This scenario involves reading and changing the source code displayed inside

4. RESEARCH DESIGN

the Editor, switching between different source files, and viewing the execution result in the *run tool window*. Developers would have to look at and interact with the GUI windows and use shortcuts in their day-to-day IDE activities.

A broader perspective can be obtained by analyzing both the usage data from the IDE. For example, by analyzing mouse clicks, keyboard usage, and gaze information captured by the eye-tracker. The purpose of the gaze information is to fill in the gaps left behind by the captured IDE usage data. These gaps occur because an IDE does not know that a developer is reading source code or is glancing over the debug window since there is no input activity from the developer.

Studying the actions and the coordination between GUI windows can reveal insights into how developers typically use the IDE [64]. This analysis can help in determining how the perceptions of the developers relate to their actual behavior.

RQ1: *How do developers interact with the eye-tracking augmented IDE compared to the non-augmented IDE?*

However, it is important to note that capturing and analyzing all the captured IDE data and gaze data would be impracticable, and not all of this data is relevant.

Therefore, this research focuses on how long a developer uses the different GUI Windows and the transitions between the GUI Windows. The specific GUI Windows that we consider are discussed in Section 3.6.1.

Additionally, the actions and related input techniques will also be analyzed, as stated in the upcoming paragraph.

RQ1.1: *How does the usage between the GUI windows differ?*

RQ1.2: *How frequently do developers transition between the GUI windows?*

Eye Tracking vs Keyboard and Mouse Another factor is to compare eye-tracking with the already existing input methods, keyboard and mouse, to determine the perceived usefulness of eye-tracking support.

The usage and preferences of keyboard and mouse differ between developers. Although there is no known study into these preferences, various online sources show that there is some debate about which input method is the best for developers [1]. There are even plugins to motivate to use keyboard shortcuts such as MouseFeed for Eclipse IDE [61] and Promotor X for IntelliJ IDEA [26].

Since this research is about the perceived usefulness of eye-tracking and not necessarily which different input technique has the best performance, both keyboard and mouse are combined to form a baseline of the developers' behavior.

We only test for the utility factor to investigate the usefulness of eye-tracking compared to mouse and keyboard. We do not test for usability here because it is difficult to compare the perceived usability of eye-tracking to keyboard and mouse. It is likely that the

target developers already have previous experience with using the target IDE, IntelliJ IDEA. This situation means that it is difficult to quantify their learning experience and compare it with the newly obtained information.

We analyze the utility of the different input techniques by investigating which IDE actions the developers use for every input technique. The reason is that if developers do not use an IDE action that is available with gaze-based input, then the perceived utility of that specific IDE action will probably be diminished.

RQ1.3: *What are the most used gaze input actions and how does this compare to mouse and keyboard?*

4.1.2 Research Question 2

Perception The perceptions of an eye-tracking-based IDE is the final factor that we use to determine the perceived usefulness. This factor is relevant to assess the thoughts of developers about using eye-tracking.

We test the perceived usability of our developed eye-tracking IDE to assess these thoughts. Nielsen [48] describes in his book five factors to consider in testing the usability of a system, namely learnability, efficiency, memorability, error tolerance, and subjective satisfaction. However, this research considers efficiency, memorability, and error tolerance, not within the scope of this study. The efficiency and error tolerance mainly test the performance of the system. However, this is not the main focus of this study. Additionally, memorability tests if the user can use the system successfully in the future, but this study focuses on the initial perceptions of the system. Therefore, we only consider the learnability and satisfaction factors to measure the developers' perceptions.

RQ2: *How do developers perceive the usability, learnability, and satisfaction of the developed eye-tracking IDE?*

Previous work by Shakil et al. [60] is an example of a study that also explored the usability of eye-tracking within an IDE. However, their approach differs from ours since they performed two studies: the first compares all the different inputs against each other, and the other tests the input preferences.

4.1.3 Research Question 3

Perception Besides measuring the perceived usability of EyeIDEA as stated above, we also investigate the perceived challenges for future prospects of using an eye-tracking IDE.

RQ3: *What are the perceived challenges with an eye-tracking IDE?*

4. RESEARCH DESIGN

More specifically, there are a couple of attributes that we want to investigate. The first attribute is to look into perceived limiting features of the developed eye-tracking IDE itself. The purpose is to get feedback on the implemented and missing features. This attribute will investigate the utility aspect of EyeIDEA.

RQ3.1: *What are the perceived limiting features of an eye-tracking IDE?*

The second attribute that we consider is the perceived limitations and opportunities of using eye-tracking as either an alternative or an addition to existing input methods. This attribute could shed light on possible routes to take to integrate eye-tracking in the development process.

RQ3.2: *What are the perceived challenges compared to the existing input methods?*

The final attribute that we consider is to investigate the general willingness of using an eye-tracking IDE and which perceived audience would benefit the most from such an IDE.

RQ3.3: *Is there an audience that would use an eye-tracking IDE?*

4.2 Methodology

The primary goal of this study is to examine the usability and perception of the developed eye-tracking IDE. In particular, to test the novel interaction method based on eye-tracking. This information could be used to figure out the knowledge gaps that need to be filled to understand an eye-tracking-based IDE.

To find out if it can compete with the existing interaction, we conduct a study to study how developers use the available input techniques, their preferences and why they prefer it.

The most logical data to collect would be to keep track of the interactions with the IDE, usage of the GUI windows by using eye-tracking and interviewing the participants about their experience and thoughts about the augmented IDE. Several other studies that tested the usefulness of using eye-tracking for interactions have used similar methods [60, 22].

To get a good idea about the perceptions of the augmented IDE, the participants should be able to voluntarily use the IDE such that it feels the most natural to them. This way, the study reflects a more realistic environment than measuring the different interactions individually. However, research into IDE user habits [47, 4, 45] show that while developers use the debugger in general, it does not mean that they will use it during the user study. There is a possibility that “whether to use the debugger or not is mostly a question of user habit.”[2]. So, there is a trade-off between giving the participants total freedom and be able to verify the usability and utility of the gaze interaction implementation of the debugging tools.

The following subsections will explain and discuss in-depth the procedure of the experiment for the participants, the design and trade-offs of this procedure, the collected data, and the analysis of this data.

4.2.1 Procedure Overview

The study consists out of four phases that are indicated with different colors in Fig. 4.1.

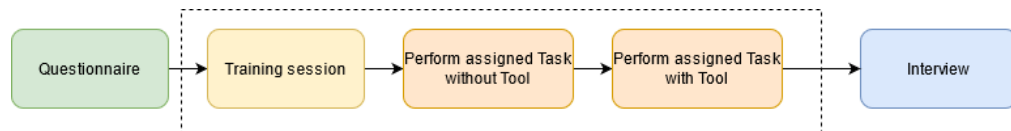


Figure 4.1: Overview of the study procedure.

Before the experiment, we ask the participants to fill out a questionnaire to determine the demographic information and which tasks they receive.

After the questionnaire, a training course introduces the participants to the tool and IntelliJ IDEA in case the participant is not familiar with it. The training starts with the calibration and validation of the eye tracker. During this training, the participants are asked to complete various objectives. We observe the time and effort that these objectives take and note these down. These serve as an indication of the learnability of the tool.

Right after the training session, the participants go through two tasks in which they have to discover and fix the bugs. These tasks use two different code snippets and are randomized between participants to take into account that some actions are more likely to be used during a specific task. Participants are also given a starting point to kick-start the debugging session. Before starting a task, we configure the IDE to enable the input technique under test and load in the associated code snippet.

The participants perform the first task without the tool to exercise their debugging skills first as they typically would. Additionally, this establishes a baseline to compare our tool against the regular IDE.

The second task is performed with the tool enabled. During this task, the participants are encouraged to use the tool as much as they see fit. This encouragement is to make sure that their actions are entirely voluntary.

After these tasks, we interview the participants about their initial opinion about the usefulness of an eye-tracking IDE. During this interview, the participants are asked to fill out the SUS questionnaire [9] which contains a couple of standardized questions to measure the subjective usability. Furthermore, observations made during the study are discussed and their overall perception of using an eye-tracker inside the IDE.

4.2.2 Programming environment

This study uses a Tobii Pro X2-30 eye-tracker [69]. The eye-tracker is connected to an HP Zbook 15 G5 Laptop(Intel i7-8750H 2.2Ghz with 16 GB RAM) which has a 1920x1080, 15.6" screen. The laptop runs Microsoft Windows 10 Home edition and IntelliJ IDEA

2019.2, in which two tools have been installed, EyeIDEA and Activity Tracker [17]. The latter is used to monitor the triggered actions as well as the keyboard and mouse activities inside IntelliJ IDEA. Additionally, a separate laptop is used with remote desktop capabilities to monitor the participants.

The programming language for the tasks is Java, which is a popular language and has great debug support in IntelliJ.

4.2.3 Pre-User Study Questionnaire

Before the user study starts, a participant fills in a questionnaire in order to establish demographic information. The specific information requested was age, current software development role or study program, previous experience with using an eye tracker, usage of glasses or contact lenses, existing eye deficiency such as color blindness, and previous experience with IntelliJ IDEA. We use this information to describe the audience that participated in this study. Furthermore, the mentioned items are further explained in Section 4.3.1 and the questionnaire can be found in Appendix B.1.

4.2.4 Training

Before the participants begin with the tasks, we ensure that they have an idea of how to operate EyeIDEA. During this training session, the participants receive information about the workings of EyeIDEA and will perform the actions until they can successfully perform the associated objective. This training session is based on recommendations given by Ko et al. [39].

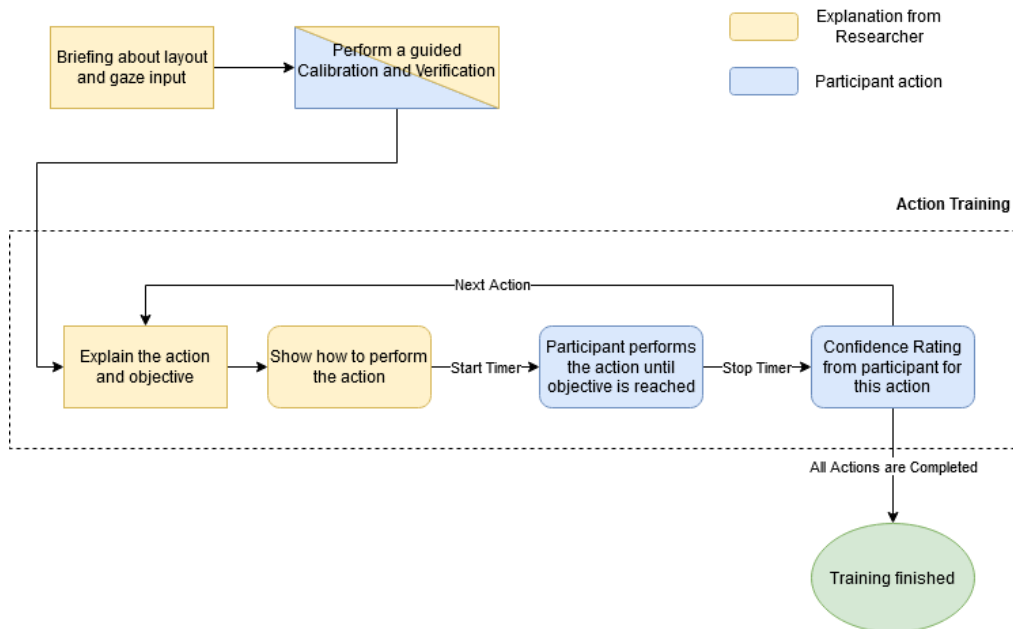


Figure 4.2: Overview of the comparative study procedure.

Action	Objective
Use Debug Buttons	Run the program and Start the Debugger
Go To Declaration	Starting at the SlowTrain file , find the usages of <i>SlowTrain</i> constructor and select the last usage of <i>new SlowTrain()</i> from the list and then find the declaration of <i>trains</i> variable
Go To Implementation	Starting at the Executor file , go to the implementation of <i>Scheduler</i> class and then to the implementation of <i>Train</i> class and select the <i>FastTrain</i> option from the list.
Set/Remove Line Breakpoint	Starting at the Scheduler file Set and remove a breakpoint at the only <i>if statement</i> in <i>Scheduler.java</i>
Set/Remove Conditional Breakpoint	Starting at the Scheduler file Set and remove a conditional breakpoint at the only <i>if statement</i> in <i>Scheduler.java</i> . The condition to be set is the same as the <i>if statement</i>
Quick Evaluation	Starting at the Scheduler file and while the debugger is active , evaluate the <i>sort</i> variable
Window Evaluation	Starting at the Scheduler file and while the debugger is active , select line 31 and activate the Window Evaluation and click evaluate

Table 4.1: List of actions and associated objectives.

Figure 4.2 shows an overview of the training. The first step is that the participants are briefed about the layout of EyeIDEA (Section 3.2), and how the gaze input action system works (Section 3.8). After we briefed the participants, we guide them through the calibration and validation process, which is explained in Section 3.4.

When participants successfully complete this, we move on to training every action that is available within EyeIDEA. Firstly, we explain the current action to the participant and how to complete the current training objective. These are listed in Table 4.1.

Then, the participant watches a small video on the screen that shows how to perform this action. After this video has ended, we ask the participant to enable the eye-tracker and complete the current objective. When the objective is met, the participant rates how confident they are to perform this action on a scale from one to five. All the links to the training videos can be found in Appendix B.4.

The training ends when the participant has completed all the objectives.

4.2.5 Task Design

During the user study, we use two different code snippets, one for each task. In this task, the participant finds and fixes as many bugs as possible within ten minutes. The primary goal of this study is to experience eye-tracking and not to test debugging experience of participants. Therefore, we chose the ten minutes limit. However, if a participant finds all bugs within the time limit, the task is marked as complete.

There are in total two different tasks, one for mouse plus keyboard input and the other adds gaze input to this list. Each of these tasks has an accompanied testing class to test if the code is working as expected. In the first task, the participants only use the mouse and keyboard. This task lets the participants exercise how they would typically debug. Additionally, it makes sure that everyone has a reference point to compare against the gaze input experience.

In the second task, the participants can use the gaze input, but they are not required to use it. We instruct them that it is up to them if they want to use specific actions to make sure that they voluntarily choose their actions.

In summary, the tasks' objectives are to let the participants experience the specific input techniques during debugging. For this reason, the code snippets are designed to be realistic and contain bugs that also could have occurred in actual production code.

4.2.6 Code Snippet Design

The main objective of this user study is to test how developers perceive the gaze input during debugging. Therefore, the code snippets are designed to let the participants exercise their debugging skills and experience the different input techniques.

Therefore, the design of the two different code snippets has to fulfill the following criteria:

1. The code snippets do not contain bugs or warnings that are spotted by the IDE.
2. The code snippets and bugs should not be too recognizable or trivial, or else a participant would not have a reason to use the debugging tools.
3. There are no unexpected bugs in the code snippets to have a controlled software environment.
4. The code snippets should be small and easy to understand to reduce the time it takes to figure out the structure of the code so that the participant has more time to use the debugging tools.
5. The code snippets should give the participant a reason to use the available debugging tools.
6. The bugs should not be too difficult, or else a participant would have no idea where to start.

Finding two different code snippets that can satisfy these criteria failed, and thus the decision was made to create our own code snippets. By designing these from scratch, a participant should not recognize the introduced bugs immediately. This decision will naturally fulfill a part of Item 2.

The code snippets are designed such that all the supplied tests will succeed if the participants fixed all bugs. This satisfies Item 3. Additionally, each of the code snippets uses topics that can be understood with only basic software knowledge to fulfill Item 4.

Only logical bugs are introduced such that the static analysis tools of IntelliJ do not flag the bugs already to satisfy Item 1. We carefully placed the bugs such that fixing a bug should lead to the discovery of another bug. This decision is to help the participant in finding a bug and fulfills Item 6.

Another factor in the design and placement of the bugs is that each code snippet elicits the use of breakpoints and variable evaluation. This design decision fulfills Item 5.

In order to add some realism in the form of non-trivial bugs, we based them on bug classifications found by earlier research [13, 10, 70, 51, 27, 41]

The exact selection of the bugs and an overview of the designed code snippets ¹ is found in the following subsections.

Shopping Cart Code Snippet

The first task is based on a Shopping cart module in which the total price is calculated for the products that are added to the shopping cart. In total, there are three bugs to be found.

The *first bug* is that the equals method of the product is missing that is needed to retrieve it from the database. This type of bug is classified in [70] as “not overriding equals and hashCode can cause both to be not defined”.

The *second bug* is that the tax calculation includes the product price instead of just the tax rate. This bug can be caused by copy and pasting code from an existing method from the same class without adjusting it to fit the control flow. Copy-and-paste type bugs are found to be a major source of bugs in operating systems [13].

The *third bug* introduces a non-existing product added to the cart that causes a negative price value for that product instead of 0. This type of bug is caused by missing a precondition check. This kind of check is found by Campos et al. [10] as a commonly used method to fix bugs.

Network Simulator Code Snippet

The second task is based on a simple Network simulator tool in which a simple Network can be build using switches, firewalls, and end devices. In total, there are three bugs to be found.

The *first bug* is that the connection links between a switch and a firewall can only flow in a single direction, which means that packages are lost. This type of bug is caused by missing an implementation that caused the error. Hamill et al. [27] classifies this as a coding fault and found that this type of error is a cause for failure.

¹code snippets can be found on <https://github.com/alangerak/EyeIdeaSampleProjects>

The *second bug* for this task is caused by the Switch that sends a package to itself if it does not know where to send it to, causing a stack overflow. This issue is caused by missing an edge case check to prevent this type of error. This type of bug can be seen as a Parallelism bug[51] because there could be an expectation that somehow, all switches can simultaneously receive and decide at the same time what to do with the package. If this were true, then the stack overflow would not happen since all switches would process the incoming packages at the same time and know that it should not send it again.

The *third bug* causes that newly added rules to the Tree-rule-based Firewall would override the source address if the destination address and port combination already exist for that source address. There is a missing condition that should check if there already exists a rule. The bug type is a missing corner case, but it could also be classified as a control flow bug since the instruction flow to add a new rule does not work as expected. This bug type is described and analyzed by Zhenmin et al. [41] for modern open-source projects. They show that these bugs are found a lot in these projects.

4.2.7 Post-Task Interview

After the participants have completed the second task, we conduct a post-task interview. The purpose of this interview is to obtain qualitative data about using an eye-tracking enhanced IDE. Before the interview starts, the participant is handed a SUS questionnaire that contains ten questions with five Likert items, from Strongly disagree to Strongly agree[9]. The result gives a score of the perceived usability of EyeIDEA.

At the beginning of the interview, we ask the participant to share their overall experience and thoughts about EyeIDEA. This opinion helps us to get a broad perspective of the perceptions. Additionally, it gives us a rough estimate of the participants' opinions about the specific components.

After these questions, we ask the participant about the learnability and perceptions about the gaze-based navigation and debugging actions. Depending on their overall experience, we ask participants to explain their experiences in the context of these questions.

The final subject that we ask about is a general opinion about using eye-tracking inside an IDE. This question is to evaluate whether the participant has any other expectations on using eye-tracking inside an IDE.

A complete overview of all the interview questions and the SUS form can be found in Appendices B.2 and B.3.

4.3 Data Collection and Analysis Procedure

This section covers the collected data and how it will be analyzed to answer the research questions. Each of the following subsections describes a specific set of data that is used to answer a question. These sections contain information about the data definitions, how we obtained them, and how they will be analyzed.

4.3.1 Demographic Information

Demographic information describes the background of the participants that show how they fit in the general population. This information is useful to show what audience participated in this study. All information is collected through a questionnaire (Section 4.2.3). We will analyze this data superficially as we will only show what kind of group participated in this study.

Age The age of the participant. The primary purpose is to provide information on the demographic distribution of the participants. However, due to the selection of participants, see Section 4.4, we expect that the age distribution will be small. However, it is common to ask the age when testing how useful a product is since age can influence how a particular input device is used due to factors such as motor control [63] and different movement strategies [71]. Age also can influence gaze patterns, as seen by Jennifer C et al. [8] when testing the differences in fixations during usability testing of various websites.

Background The current software role or study program of the participant. This information is only used to describe the current (non)involvement with software development.

Eye Deficiencies More than half of the Dutch population wears glasses or contact lenses [65] and this can potentially influence the accuracy and precision of an eye tracker [49]. The reduced accuracy and precision can greatly affect the usability of EyeIDEA because if the registered gaze does not match with the participants' expectation, it inhibits the ability to interact with the IDE.

Colorblindness could also impact the usability of EyeIDEA since it uses different colors to indicate interactions. Although EyeIDEA has different color schemes, colorblindness could still reduce the effectiveness of the implemented interaction system.

Eye Tracking Experience Whether or not the participant has used eye-tracking before. The purpose of this data is to check the widespread usage of eye-tracking technology among the participants.

IntelliJ IDEA Experience Indication if the participant has used IntelliJ IDEA before. If the participant has never used IntelliJ before, they will have a different learning experience than the others. This data shows how many participants needed to learn IntelliJ during this study.

4.3.2 Research Question 1

Data about the IDE interactions is needed to answer RQ1. This data can be obtained from the input devices (keyboard, mouse, and eye-tracker) as well as the actions that the IDE performs. The following paragraphs explain how we turn this into data points and how we will analyze this data.

GUI Window usage The usage of a GUI Window is measured by capturing the gazes that fall within the GUI Window. EyeIDEA already registers these gazes, and EyeIDEA puts these into a database.

This data is transformed into numerical data by summing up the *time spend* in each GUI Window.

To account for different completion times and time spend inside GUI Windows between participants, we normalized the total time for each Window by dividing it by the completion time of that task. This way, we can compare the relative usages of the Windows.

We analyze this data by comparing the data gathered from both debug sessions against each other to see if there are notable differences.

GUI Window transitions A transition takes place when the fixation changes from a GUI Window to a different one. Similar to the GUI Window usage, we consult the database of EyeIDEA to get the needed data.

In order to transform this into numerical data, two approaches are taken: The *first approach* is to count the number of transitions between GUI Windows. This data is normalized by dividing it by the total number of transitions because the number of transitions varies between participants. This approach follows a similar approach taken by Hansen et al. [28].

This data is analyzed in the same way as the GUI Window usage data. However, this approach does not take temporal effects into account that participants change their behavior during their experiments [7]. To account for these effects, we use a *second approach*. This angle follows the approach that Hejmady et al. [29] took to study visual patterns during debugging.

For this approach, we transformed the time spent into a binary variable, *timeSpan*, to indicate if the time spent before switching to another GUI Window is greater or smaller than 500 ms. Additionally, this variable is used to encode the usage of a GUI window in a character. For example, the character 'A' represents a short time spend in the Editor, while the character 'B' represents a long time spent in the Editor. This results in an encoding of a participant's session that takes time into account.

The Sequential Pattern Mining (SPAM) algorithm [5] is used to analyze these sequences. This algorithm uncovers frequently occurring switching patterns for each participant. The result is a collection of switching patterns for both the mouse with keyboard input and gaze input. Then, the top three patterns are extracted and plotted together into graphs to explore the temporal effects.

Used actions An action occurs when a developer uses either a specific key combination, mouse click, or a gaze button. This research excludes typing-related actions such as writing code and code completion as they are outside of the study's scope. These actions are collected by parsing the output file obtained by Activity Tracker [17] to obtain a collection of actions.

To differentiate between gaze input triggered action and keyboard/mouse actions, we use the action system's log file of EyeIDEA. This file keeps track of when a gaze-related action is used. We can retrieve the input type for every action, by manually comparing the timestamps of both files.

With this annotated list, we obtain for both gaze input and keyboard/mouse the total amount per action-type. However, this list could still contain typing-related actions that we remove manually. In the end, we will have two of these lists, one for the usage of the augmented IDE and another for the non-augmented IDE. Then, we calculate the relative frequencies for each action based on the total number of actions in that list. With these frequencies, we will order the actions in each list and compare them to each other. Additionally, we will also use results from the conducted interview to check their answers against the obtained results. We are particularly interested in missing eye-tracking actions and whether the implemented gaze actions are comparable to the most popular mouse and keyboard actions.

Furthermore, we also investigate the situations in which the participants use the mouse and keyboard instead of eye-tracking when it is available to them. These situations can tell us when the participants value the mouse and keyboard above using the eye-tracker. This value difference is an indicator of the effort it takes to use the eye-tracking input method.

4.3.3 Research Question 2

To answer RQ2, we need data about the participants' experience with EyeIDEA. This data is obtained by monitoring the participants during the user study and conducting an interview which is described in Section 4.2.7.

Satisfaction Satisfaction refers to how pleasant it is to use the augmented IDE for that particular input technique and has a sizeable impact on the acceptance of using that input technique. We measure the perceived satisfaction by asking the participants questions during the interview about how they rate their experience with mouse and keyboard but also their satisfaction with eye-tracking.

To analyze the satisfaction, we compare the answers given by the participants based on the questions that relate to satisfaction.

Usability Usability is a measure of how well the user can use the system to satisfy their needs and requirements to complete a specific goal. A tool to measure subjective usability is the SUS questionnaire [9]. This questionnaire contains ten questions with a one-to-five scoring scale. These questions alternate between positive and negative items in order to avoid response biases. The result of the questionnaire indicates the usability of this system.

The usability scores are calculated according to the following method:

- For each of the odd-numbered questions, subtract 1 from its value.
- For each of the even-numbered questions, subtract its value from 5.
- Take these new values and add up the total score. Then multiply this by 2.5.

Brooke [9] used the following rationale for this calculation. The maximum score that should be achieved is 100, and how higher the score, the more usable the system is perceived. In order to get that score, each question has to have a maximum score of 10. Since

all odd-numbered questions are asked in a positive tone, a “agree completely” should have a score of 10 while a “strongly disagree” has a score of 0. When 1 is subtracted from the lowest value, the score will be 0. In addition, when you multiply the highest value with 2.5 after the subtraction, the score is $(5 - 1) \cdot 2.5 = 10$.

Similarly, the even-numbered questions are asked in a negative tone and should have a reverse scoring effect on the usability score. This means that a “agree completely” should receive a score of 0 and “strongly disagree” a score of 10.

To interpret these results, we calculate the mean and standard deviation over all the scores. We also add an adjective rating based on these statistics by using the scoring system created by Bangor et al. [6].

Besides this questionnaire, we also look into the answers given during the interview. In particular, we use the answers to the questions that relate to the systems’ usability, e.g., the evaluation of the interaction system and individual actions.

Learnability Learnability is defined as the amount of time that a novice needs to successfully perform a specific goal. During the training, each participant is observed to see how long it takes until they complete a specific goal. At the end of the training, we ask the participants how confident they feel on a score between one and five in performing the actions available to EyeIDEA. This score is used as a perceived proficiency level for the actions.

We will analyze this data by plotting these results and then look for downward and upwards trends by treating each objective as its own category. Additionally, we look into if participants perceive specific modes as more difficult to learn than other modes. For this approach, we will use the detailed answers that we collect during the interview.

Another option we will explore is to investigate the actions a participant took during an objective. We compare these against the ideal action path to see which kind of errors they made.

4.3.4 Research Question 3

For answering RQ3, we only need the opinions and experience with EyeIDEA. These opinions and experiences are obtained directly from the interview Section 4.2.7.

Perception The perception is an overall subjective measure of the perceived usefulness of the eye-tracking IDE. In particular, the design of EyeIDEA is judged and whether eye-tracking is suited for IDE-related tasks. We address the perception by asking participants to express their opinions about EyeIDEA and their overall experience of using an eye tracker. These questions are asked during the interview, see Section 4.2.7.

All the responses are manually analyzed and grouped to check for overlapping and unique answers to analyze the participants’ answers. We will then use these results to answer the three research questions that we listed in Section 4.1.3 Perception.

4.4 Participants

We wanted to get a diverse group of participants with different amounts of programming experience for this study. This way, we obtain a broad range of responses from participants, ranging from experienced programmers that know how to use the IDE effectively to participants that are still learning how to use the IDE and develop software.

Additionally, we wanted that a part of the group wore glasses or contact lenses for two reasons. The first reason is to test the influence of the glasses on eye-tracking. Secondly, to test if the possibly altered performance impacts the perceived usefulness of an augmented IDE.

To achieve this, we invited students from the TU Delft since they are readily available to us. We sent the invitations using direct chat messages to fellow students, master student group chats, Teaching Assistant (TA) group chats, and distributed posters on the campus to invite students. This poster is included in Appendix D. These invitations led to an on-line Survey form that contained more information about the user study and that they could choose to receive a € 10,- gift card. The survey is included in Appendix E.

4.5 Pilot study

A pilot study has been performed to verify the training procedure (Section 4.2.4), the code snippet (Section 4.2.6), and the questions for the interview (Section 4.2.7). We conducted this pilot with an experienced software lead developer, aged 41, and had no prior experience with using an eye-tracker. The participant was asked to go through the whole session first. We discussed the participants' experience directly after the session to provide us with feedback.

For the training procedure, we received feedback on the training videos on how to make them shorter, so there is less time between the video and performing the objective. This solution should make it easier to remember how to perform the shown action. We shortened the videos by not repeating how to select an item in each video. This procedure is always the same, and the participant deemed it unnecessary to repeat this. Additionally, we originally had plans to use an overview card to guide the participants, but the training videos proved to be sufficient.

The feedback for the code snippets was that the code reflects the difficulty of production code. This meant that the participant used the augmented IDE just like in a regular working day, so we did not change the code snippets.

For the interview, we noticed that there were a couple of questions that received the same answer. To keep the flow of the interview, the participant recommended which questions could be removed and which ones could be rewritten to encourage extensive answers. Apart from that, the questions themselves were clear, and there were no obvious missing questions to evaluate the experience with using the augmented IDE.

Chapter 5

Results

The previous chapter listed the research questions, what kind of data is needed to answer them, the design and analysis of the user study, and which participants were invited to the user study. In this chapter, we report the results of this study and analyze them accordingly to answer the stated research questions. This chapter starts with going over which participants took part in this study. We use descriptive demographics statistics to describe the participants. Next up, we address the collected IDE Usage data. After that, we analyze the training results and the overall usability of the developed augmented IDE, for which we use the results obtained from the interview. Lastly, we will focus on the perceptions of an eye-tracking IDE and what challenges and solutions they have proposed.

5.1 Participants statistics

A total of eight students signed up, but seven of them participated in this study, of which 28% were female, and the others were male. The participants' age ranged between 19 and 29 years (mean = 22.7; standard deviation = 3.0; median = 22). Furthermore, 71% of the students wore glasses or contact lenses when entering the study, but this dropped to 43%, because of issues with tracking the participants' eyes. Some had to take their glasses off to fix this issue.

None of the participants were color-blind, so we have not used the alternative color schemes described in Section 3.9. Only a single participant had no prior experience using IntelliJ IDEA but did inform us to have experience with other IDEs such as Eclipse and Visual Studio. Additionally, 43% of the participants have used an eye-tracker, either for another research project or for gaming.

Lastly, two of the participants were in their first and second year of their Bachelor in Computer Science, four were Master students for Computer Science, and a single Complex Systems Engineering and Management Master student.

5. RESULTS

Factor	Percentage	Factor	Percentage
Gender		IntelliJ Experience	
Male	72%	Yes	86%
Female	28%	No	14%
Age		Eye-tracker Experience	
19-22	57%	Yes	43%
23-29	43%	No	57%
Glasses / Contact Lenses		Study	
Yes	71%*	BSc Computer Science	29%
No	29%	MSc Computer Science	57%
		MSc C.S.E.M.	14%

Table 5.1: Demographics of the seven participants, *Some participants had issues with their glasses and took them off, resulting in 43% that used them.

5.2 RQ1: IDE Usage

The IDE usage data provides a perspective about the differences between using the eye-tracking augmented IDE and not using it. This information is captured by the eye-tracking data, collected actions, and answers provided during the interview. The upcoming subsections will show the results and analyze the differences between using the eye-tracking capabilities and not using it as described in Section 4.3.

5.2.1 GUI Window Usage

During both debugging sessions, we turned eye-tracking on to analyze the participants' attention regarding GUI windows. Then, we summed up all the participants' gaze duration for each GUI window. However, we only included the windows that at least three participants looked at to remove significant outliers. This data is shown in Table 5.2.

From this data, we observe that the participants spent the most time on the Editor. Moreover, we also observe some differences in the usage patterns between the participants. Firstly, participant P1 is the only participant that never looked at the DebugTool during both debugging sessions.

Secondly, participant P3 never looked at the DebugPanel and Breakpoint Chooser, which means that this participant never placed breakpoints using eye-tracking. Additionally, P3 spent a negligible amount of time on the other EyeIDEA windows. This result shows that P3 barely used eye-tracking. A reason could be that this participant indicated that the eye-tracking felt too sensitive and that EyeIDEA responded too quickly to the gaze input.

Thirdly, the participants spent an insignificant amount of time in the Usage Popup. This window only shows up when a participant used the *Go To Declaration* navigation action, and there exist multiple navigation locations. This observation shows that either this

GUI Window	No Eye-tracking Available							Eye-tracking Available						
	P1	P2	P3	P4	P5	P6	P7	P1	P2	P3	P4	P5	P6	P7
Common:														
Editor	8m47s	7m14s	8m40s	10m07s	7m27s	7m51s	8m39s	7m17s	7m13s	9m41s	8m43s	7m34s	5m25s	5m49s
DebugTool	0m0s	0m32s	0m39s	0m16s	1m07s	0m50s	0m46s	0m0s	0m22s	0m0s	0m21s	0m44s	1m23s	0m47s
RunTool	0m16s	0m10s	0m16s	0m01s	0m17s	0m48s	0m01s	0m25s	0m01s	0m18s	0m03s	0m05s	0m03s	0m01s
Usage Popup	0m01s	≈0m0s	0m05s	0m0s	0m0s	≈0m0s	0m12s	0m17s	0m03s	0m0s	0m0s	0m0s	0m0s	0m13s
NavigationBar	0m04s	0m05s	0m0s	0m01s	0m04s	0m01s	0m0s	≈0m0s	0m03s	0m0s	≈0m0s	0m06s	0m01s	≈0m0s
EyeIDEA:														
ButtonPanel	-	-	-	-	-	-	-	0m33s	0m40s	0m07s	0m53s	0m39s	0m48s	0m55s
NavigationPanel	-	-	-	-	-	-	-	0m25s	0m25s	0m04s	0m46s	0m06s	0m29s	0m32s
Breakpoint Chooser	-	-	-	-	-	-	-	0m12s	0m02s	0m0s	0m04s	0m08s	0m22s	0m05s
DebugPanel	-	-	-	-	-	-	-	0m16s	0m04s	0m0s	0m03s	0m08s	0m07s	0m03s

Table 5.2: The total time that each participant spent in each window. All windows below the EyeIDEA label are only available with eye-tracking enabled.

situation does not often happen or that the participants did not use this navigation action. We will revisit this in Section 5.2.3.

Finally, we also observe that the NavigationBar received hardly any gaze time. This window contains the run program, start debugging buttons, among other buttons, and is located in the upper right corner just above Panel 1 in Fig. 3.2. The participants had to spend little time in this window since clicking on these buttons takes up almost no time.

To further investigate these numbers, we calculated the percentage of time spent inside a window by dividing this time by the total time spent in all windows, which we repeated for each participant individually. This way, we compensated that not every participant spent exactly ten minutes looking at the screen during both debugging sessions.

In Table 5.3, we show the obtained data when eye-tracking interactions were unavailable. We observe that the Editor window is the number one most looked at window with a significant usage gap between it and the second most used window, DebugTool.

Additionally, we observe that the difference between the minimum and maximum usage of the Editor is 15%. This difference is the largest one among these windows, which also means that there is quite a difference in how important the Editor is between participants. However, it does remain the window on which the participants spend the most time on.

We also observe that there are two additional significant gaps. The first gap is between the DebugTool and the RunTool. The second gap is between the RunTool and the other windows below that. These gaps mean that both of these windows take second and third place, respectively. The other windows do not receive as much time compared to the other windows. A reason could be that participants do not use them often or quickly find what they need in these windows.

Moreover, we also observe a big difference between the quartiles Q1, Q3, and the maximum time for the Usage Popup window. This difference indicates that, in this case,

5. RESULTS

GUI Window	No Eye-tracking Available							Eye-tracking Available						
	Mean	STD	Min	Q1	Q2	Q3	Max	Mean	STD	Min	Q1	Q2	Q3	Max
Common:														
Editor	89.0%	5.9	81.4%	85.1%	89.0%	92.7%	97.0%	77.2%	10.2	62.2%	72.7%	78.7%	79.7%	94.9%
DebugTool	6.44%	4.11	0%	4.56%	6.75%	8.33%	12.5%	5.79%	5.74	0%	1.62%	4.18%	8.53%	16.1%
RunTool	2.85%	2.70	0.30%	1.16%	2.76%	3.11%	8.32%	1.47%	1.67	0.26%	0.41%	0.71%	1.96%	4.56%
Usage Popup	0.53%	0.82	0%	0.06%	0.14%	0.63%	2.22%	0.90%	1.35	0%	0%	0%	1.66%	3.03%
NavigationBar	0.47%	0.44	0%	0.11%	0.29%	0.85%	1.06%	0.34%	0.41	0%	0.01%	0.13%	0.48%	1.13%
EyeIDEA:														
ButtonPanel	-	-	-	-	-	-	-	7.10%	3.05	1.28%	6.37%	7.44%	8.65%	11.0%
NavigationPanel	-	-	-	-	-	-	-	4.33%	2.47	0.80%	2.77%	4.77%	6.05%	7.09%
Breakpoint Chooser	-	-	-	-	-	-	-	1.46%	1.43	0%	0.62%	1.05%	1.83%	4.29%
DebugPanel	-	-	-	-	-	-	-	1.13%	0.96	0%	0.58%	0.89%	1.45%	2.97%

Table 5.3: Percentage of the total time spent in each window based on the data obtained from the seven participants. All windows below the EyeIDEA label are only available with eye-tracking enabled.

the distribution is skewed as only P7 spent some time inside this window as seen in Table 5.2. Additionally, we observe that the participant did not often look at the NavigationBar. The distribution is also right skewed as the percentage significantly increases from Q2 to Q3. This result means that only a few participants have looked at the buttons in this window because they either used the buttons more or the other participants clicked on these buttons without looking.

On the right side in Table 5.3, we show the obtained data when eye-tracking was available. This side of the table includes additional EyeIDEA windows that were not available the first time. From this, we can immediately observe that the time is spread out between the different windows. However, just as in the previous table, we observe that the Editor window remains the most examined window. Additionally, the difference between the minimum and maximum for the Editor increased to 32%. There is an even greater difference in using the Editor compared to the left side of Table 5.3. We assume that this is caused by the addition of the EyeIDEA windows. These windows require that participants have to look at them to execute an action. However, some participants did not use them very often or found the correct gaze button very quickly. This behavior could explain the increased differences in the time spent inside the Editor window.

Additionally, we observe that there are three additional significant gaps. The first gap is between the EyeIDEA ButtonPanel (Panel 1 in Fig. 3.2) and the DebugTool as the gap is bigger than the difference of mean time spent between the (Panel 2 in Fig. 3.2) and the DebugTool. Secondly, we observe a gap between the Gaze Navigation Panel and Breakpoint Chooser (Fig. 3.23a). The final gap is between the Usage Popup and the NavigationBar. These gaps show that we can distinguish between five different groups based on these gaps. If we look at the average times, we observe that the most time is spent in the first three

groups and drops significantly for the other windows. As before, this could mean that participants quickly find what they need in the other windows or that they use these windows less often. Moreover, we also observe a skew in the RunTool usage as the mean and median differ by 70%. P1 and P3 caused this issue as they spent significantly more time in the RunTool, which we observed in Table 5.2.

We also observe something about the time spent on the introduced EyeIDEA windows. Participants spent on average the most time on the ButtonPanel followed by the NavigationPanel. Since the ButtonPanel plays a significant role in the interaction system, it is no surprise that participants spend the most time in this window. This situation happens primarily when most participants need to locate the right Gaze Button before activating a function which adds additional time that they need to spend. However, there is a big difference between the minimum and maximum time. This difference indicates that some participants have more issues with locating the right button than the others.

Another surprising observation we found is that participants spend a significant portion of their time on EyeIDEAs' NavigationPanel. This panel only consists out of two gaze-based buttons that activate the navigation modes. Perhaps, this indicates that the participants perform numerous navigation actions. Moreover, based on these observations alone, we see that participants did not use the debug modes offered by EyeIDEA. If they were, we would expect to observe something similar for the Gaze Debug panels. We will look further into this in Section 5.2.3 where we make observations about which actions the participants used.

Finally, if we ignore the introduced EyeIDEA panels and then compare both sides of Table 5.3, we observe that the order did not change. This observation shows that the participants did not significantly change their behaviors. However, we do notice a significant drop in RunTool times and a rise in the Usage Popup times. Maybe, more navigation actions took place, and participants relied less on running the program and looking at the output. Another reason is that in order to use the Usage Popup a participant has to remember the highlight color in addition to reading the list as seen in Fig. 3.28b. This activity increases the time spent in this window.

Key Points

- The participants spend by far the most time on the Editor, regardless of the presence of eye-tracking. Additionally, if we exclude the additional new windows introduced by EyeIDEA, then the order of the windows based on the time spent did not change.
- The introduction of the tool caused the participants to spend on average less time on the Editor and spend this time on the new windows introduced by EyeIDEA since the used buttons are activated based on the gaze time.
- Participant spent more time on the Usage Popup (i.e., the popup that displays selection list of source code location which can be seen in Fig. 3.28b) when eye-tracking was active, probably because a participant has to remember the highlight color in addition to reading the source code list in this popup.

5.2.2 GUI Window transitions

The way that participants transitioned between windows tells us how much they have changed their attention between them. To capture these transitions, we created an encoding as described in Section 4.3.2 and used the open-source data mining program SPMF [20] for running the SPAM algorithm to uncover patterns in these sequence of transitions. The algorithm was configured such that it only considered either one or two consecutive transitions to prevent double transition counting. We have extracted the top ten transitions that we will discuss below.

	Label	Transition		Label	Transition
1	RN	LONG DebugTool - LONG Editor	1	VN	LONG Gaze Buttons - LONG Editor
2	NR	LONG Editor - LONG DebugTool	2	AI	SHORT Editor - SHORT Gaze Buttons
3	NB	LONG Editor - SHORT NavigationBar	3	IA	SHORT Gaze Buttons - SHORT Editor
4	BN	SHORT NavigationBar - LONG Editor	4	NV	LONG Editor - LONG Gaze Buttons
5	NRN	LONG Editor - LONG DebugTool - LONG Editor	5	NI	LONG Editor - SHORT Gaze Buttons
6	PN	LONG RunTool - LONG Editor	6	IN	SHORT Gaze Buttons - LONG Editor
7	NP	LONG Editor - LONG RunTool	7	QN	LONG Navigation Panel - LONG Editor
8	NBN	LONG Editor - SHORT NavigationBar - LONG Editor	8	NVN	LONG Editor - LONG Gaze Buttons - LONG Editor
9	NE	LONG Editor - SHORT DebugTool	9	NQ	LONG Editor - LONG Navigation Panel
10	RNR	LONG DebugTool - LONG Editor - LONG DebugTool	10	ND	LONG Editor - SHORT Navigation Panel

(a) without having eye-tracking available.

(b) with eye-tracking available.

Table 5.4: The top ten GUI Window transitions. The annotation (LONG) and (SHORT) indicate if the time spent inside a Window is ≥ 500 ms or shorter, respectively.

We can immediately observe from Table 5.4 that the top ten differs greatly between the two situations. For instance, none of the labels in Table 5.4a are found in Table 5.4b and vice versa. This result tells us that the participants changed their gaze patterns significantly. However, this is to be expected since numerous interactions take place in the newly

introduced EyeIDEA windows. These interactions are caused by interacting with the gaze-based buttons. Additionally, all the transitions in Table 5.4b include at least one window introduced by EyeIDEA that further strengthen the idea that participants transitioned more between the EyeIDEA windows, rather than the previously existing ones.

Moreover, we also observe that the participants transitioned a lot between the Gaze button (panel 1 in Fig. 3.2) and the Editor when eye-tracking was available because position one to six in Table 5.4b includes the Gaze button Panel. The most probable reason is that participants needed to return to this panel frequently to select and execute a gaze-based action. Additionally, we observe that the time spent in the *Gaze Button window* before transitioning to the Editor varied a lot. A reason for this behavior is that participants searched for the Gaze button with the right color. This variation is indicated in Table 5.4b with the SHORT and LONG transitions annotations, and every possible permutation of this transition can be seen in the top six.

During the interview, participants P1, P2, P4, and P7 indicated that they encountered situations in which they had to spend more time to find the right button because the highlight colors changed too much. This behavior means that there were some difficulties in remembering the right locations.

A second observation we make is that the additional transitions reduced the transition amount to the Editor and DebugTool compared to Table 5.4a. We can observe from Table 5.3 that on average, the participants spent significantly less time in the Editor. However, there is only a small difference between the DebugTool times. This observation means that the participants spent more time before transitioning between them. Additionally, we do observe that the participant also spend at least 500ms before transitioning when eye-tracking was not available as indicated by the top two transitions in Table 5.4a. In contrast to the earlier situation, these transitions happened quite often as positions five and ten in Table 5.4a shows that these followed each other often.

To better understand how the transitions changed during the experiment, we spliced the transitions into 15-second intervals. Then, we calculated the transition frequency per participant and averaged these values.

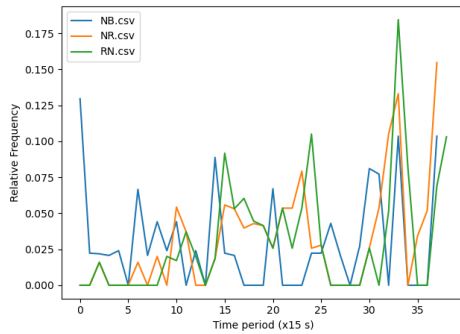
We first looked at the transition differences in the top three in both situations. This is plotted in Fig. 5.1. In the left figure, we observe that the most activity during in the first 200 seconds is that participants looked for more than 500 ms at the Editor before transitioning to the NavigationBar in which they spend little time. This behavior indicates that participants started with running the code or starting the debugger since the buttons that activate this are inside the NavigationBar. After this period, the participants transitioned between the Editor and the DebugTool, in which they spend at least 500 ms. This behavior indicates that they were in the process of debugging the code snippets.

In the right figure, we observe that for the first 200 seconds, the participants transitioned very quickly between the Editor and the Gaze Buttons as they spend very little in either of these windows. The participants knew exactly which Gaze button they had to look at to perform an action. This situation could occur when there are only a couple of highlighted items since it is easier to remember which color to find. This situation mostly happens when a participant tries to select a regular GUI button as there are not many different GUI buttons.

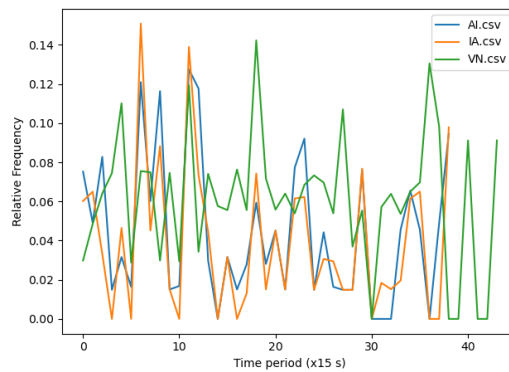
5. RESULTS

Furthermore, this also coincides with the same behavior described above.

After this period, the participants take more time looking at the Gaze Buttons since they probably use them to select source code which introduces a lot more changing highlight colors.



(a) Top three transitions over time without having eye-tracking available.



(b) Top three transitions over time with eye-tracking enabled.

Figure 5.1: Individual plots of the top three transitions found in Table 5.4.

If we compare the figures in Fig. 5.1, an interesting pattern emerges. The transitions varied are a lot more when eye-tracking was available. This situation makes sense since a user needs to look at the Gaze Buttons for the gaze interactions and back to the original window. Additionally, it shows that there are a lot more transitions when eye-tracking was available. Participants P5 and P6 indicated during the interview that they found the amount of transitions between the Editor and Gaze Buttons had a negative impact on the usability which was phrased as:

“[...] I did not like this when I wanted to do multiple actions in quick succession [...]”
- P5

Another interesting situation arose when we took the top two transitions of Table 5.4a and compared it with the eye-tracking situation as illustrated in Fig. 5.2.

In Fig. 5.2a which shows when eye-tracking was not available, the participants had two periods in which they transitioned slowly between the DebugTool and the Editor. These periods show when they selectively used the debugging information. However, when the eye-tracker was available as shown in Fig. 5.2b, the participants were slow in using the debugger, but this increased at the same time as the first period, which is around 200 seconds as in Fig. 5.2a.

An explanation is that the participants were navigating during this time when eye-tracking was available instead of debugging.

When we plot the transitions that relate to the Gaze Navigation Panel as shown in Fig. 5.3, we observe that this seems to be the case. The amount of transitions between the

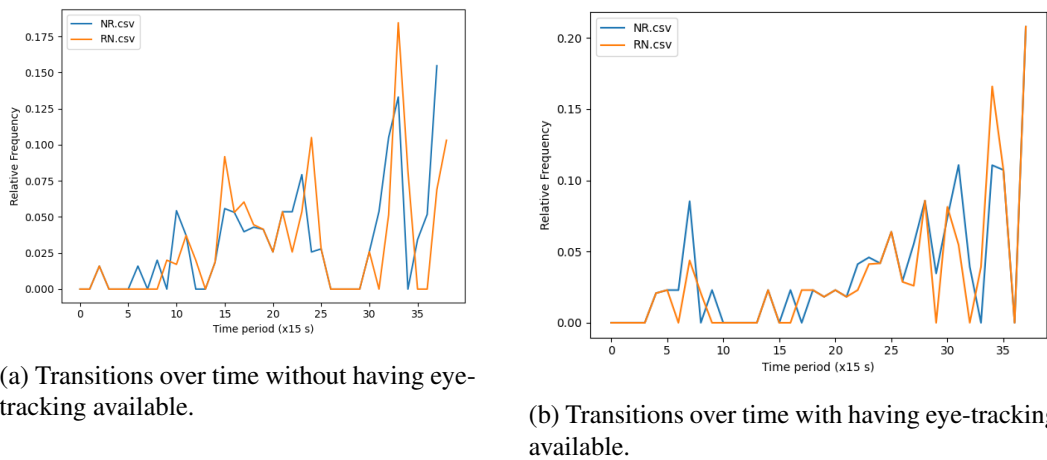


Figure 5.2: Individual plots of transitions between the DebugTool and Editor where the participants spend at least 500ms on the respectable window before transitioning.

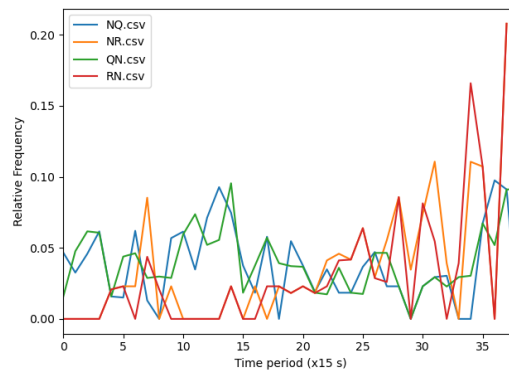
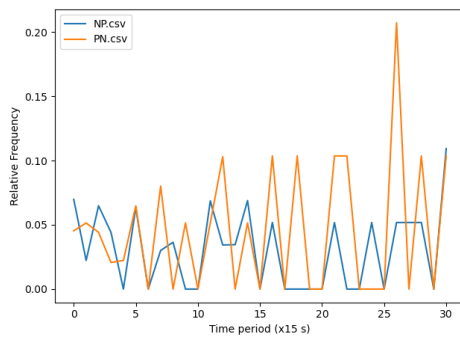


Figure 5.3: Navigation and DebugTool transitions to the Editor when eye-tracking is available.

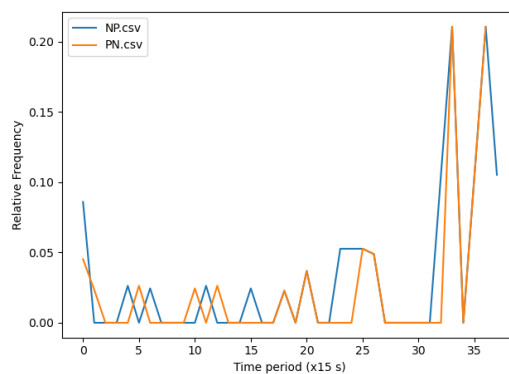
Editor and the Gaze Navigation Panel decreases significantly after 200 seconds.

Another explanation for this phenomenon was that participants P2, P3, and P6 said that they found it more difficult to use eye-tracking interactions when they needed to think a lot. This situation always occurred when they were debugging, which means that they found it difficult to combine debugging and eye-tracking. Instead, they used the gaze-based navigations instead of debugging.

5. RESULTS



(a) Transitions over time without having eye-tracking available.



(b) Transitions over time with having eye-tracking available.

Figure 5.4: Individual plots of transitions between the RunTool and Editor where the participants spend at least 500ms on the respectable window before transitioning.

The final situation that we looked at is the difference in transitions between the RunTool and the Editor since the RunTool can play a major role in debugging the source code. These results are plotted in Fig. 5.4. We observe that the transition frequency is consistent when the eye-tracking was not available, as shown in Fig. 5.4a. However, in Fig. 5.4b, we observe that it was only visited frequently at the end when eye-tracking was available. It seems that the participants prioritized other windows. We can also observe that from Table 5.3 since the average percentage of the time spent in the RunTool is half of the time spent when the eye-tracking was not available.

Key Points

- There are more transitions observed between windows when eye-tracking is available, which participants perceived as a negative contribution to the usability.
- The DebugTool is used more selectively when the eye-tracking is not available since there is a period in which a lot more transitions are observed between the Editor and the DebugTool.
- When eye-tracking is not available, the RunTool (i.e. window that shows the program output) is consulted faster and more often.

5.2.3 Used Actions

We captured the used actions that the participants used during both debugging sessions and created a top five, which is shown in Table 5.5. Since they had only the eye-tracking available in the second debugging session, all actions in Table 5.5a were activated with a

mouse or keyboard.

In the top five, we observe that the declaration navigation action is the most popular, followed closely by the StepOver debug action, which steps over a code line when the debugger is active. Interestingly, action three and five are also related to debugging. Only action four is related to navigation as this action jumps back to the previously visited code location.

In Table 5.5b, we listed all the actions when eye-tracking was available. We observe again a navigation action is the most popular action. However, this time there is a bigger usage gap between the first and second action. This gap means that participants likely relied on this navigation a lot more.

Action	Usage	Action	Input	Usage
1 GoTo Declaration	26.9%	1 GoTo Implementation	Eye-Tracking	22.6%
2 StepOver	21.0%	2 Set LineBreakpoint	Eye-Tracking	10.6%
3 Resume	8.89%	3 Undo	Keyboard/Mouse	9.36%
4 Back	7.06%	4 GoTo Declaration	Eye-Tracking	8.10%
5 Set LineBreakpoint	5.24%	5 GoTo Declaration	Keyboard/Mouse	4.26%

(a) without having eye-tracking available.

(b) with eye-tracking available.

Table 5.5: The top five IDE actions:

Something that we also observed in Table 5.5b was that the participants used both eye-tracking and keyboard/mouse to navigate to a declaration. The majority of the participants said during the interview, that they tended to navigate with the keyboard or mouse whenever they had to think a lot but fell back to eye-tracking right after. One of the reasons that was given:

“[...] I needed to think about how to use it because it takes more effort compared to muscle memory that it already there.” - P7

Other participants also indicated that eye-tracking felt nicer whenever they thought that the keyboard was not needed, i.e., writing code or doing an action that eye-tracking did not support.

The second thing was that navigation was the most popular action for both, as mentioned before. However, during the first debugging session, the participants preferred the declaration action, while in the second, it was the implementation action. During the training, P1, P2, P5, P6, and P7 said that they did not know the difference between these two actions, only that they knew that a ctrl+click always navigated them to the expected location. That shortcut is to activate the *Go To Declaration* action. So, a reason for the top one difference could be that some participants did not know that the other navigation action existed and that this worked better. This explanation also showed up in the total amount of

5. RESULTS

used actions since no one used the *Go To Implementation* action with keyboard or mouse input in both debugging sessions.

Another thing we noticed was that the navigation actions were more popular when eye-tracking was available since 35% of the actions were navigation actions compared to 27%. On the other hand, the participants used more actions to actively use the debugger with actions such as *Step Over* and *Resume*; and were setting fewer breakpoints compared to the other actions. More interestingly was that during the interview, almost every participant said that they did not notice any difference in their action usage between the different debug sessions using statements such as “*Felt no difference*”, “*did the same things*” and “*coding style did not change*”. Participant P6 said the following about this:

“*It is pretty hard to change debugging habits and the tool in itself did not influence me[...]*” - P6

An explanation for this usage difference is that several participants found that the navigation actions felt better to use because the eye-tracking was more reliable with code element selection than button selection. So, they were inclined to navigate a lot more.

We also investigated the number of actions performed. When eye-tracking was not available, the participants performed a total of 439 actions from 30 unique actions. However, when it was available, a total of 235 actions were performed from 26 unique actions. So the participants executed almost half the amount of actions when the eye-tracking was available while performing only four less unique action types. P3 and P4 did indicate during the interview that they had to be careful when looking around. They felt that the system reacted too quickly to their gaze and performed fewer actions. Therefore, they only chose actions that felt more reliable such as code selections as mentioned before.

We extracted the top five actions when eye-tracking was available based on the input types to investigate if the participants used the debugging actions often. This result will show more details than Table 5.5b since we now look at both input types.

Action	Usage	Action	Usage
1 Undo	9.36%	1 GoTo Implementation	22.6%
2 GoTo Declaration	4.26%	2 Set LineBreakpoint	10.6%
3 Resume	3.40%	3 GoTo Declaration	8.10%
4 StepOver	2.98%	4 StepOver	3.83%
5 SaveAll	2.56%	5 Run	2.98%

(a) keyboard or mouse input.

(b) eye-tracking input.

Table 5.6: The top five IDE actions when eye-tracking available, split by input type.

If we exclude the actions that use code selection, which in this case the *Set LineBreakpoint* action from Table 5.6b, we observe that the *StepOver* debugging action is still used more often than the keyboard or mouse counterpart. However, the participants did not like to use the *Resume* debugging action with eye-tracking as it is not in the top five found in Ta-

ble 5.6b. Based on this information, we observe that the participants slightly favored using debugging actions with the keyboard and mouse rather than using eye-tracking.

Moreover, we observe that participants liked to activate the *Run* action with eye-tracking input. We think that the participants did not activate it with the keyboard or mouse because it is unlikely that a participant repeatedly performs this action which means that it is less of an issue that the button selection is less reliable than code selection.

Key Points

- The choice of when to use eye-tracking actions depends on the current programming activity. If it requires lots of thinking, the tendency is to use the keyboard and mouse over eye-tracking.
- The navigation actions are used more often when eye-tracking is available while debugging actions are more popular when eye-tracking is not available. A reason is that the single code selection felt more stable to use than the other eye-tracking interactions.

5.3 RQ2: EyeIDEA Perceptions

One of the other factors to determine the usefulness of an eye-tracking IDE, is to assess the perceptions of the developed eye-tracking IDE, EyeIDEA. There are three criteria that we used to assess these perceptions namely, satisfaction (Section 4.3.3 Satisfaction), usability ((Section 4.3.3 Usability) and learnability (Section 4.3.3 Learnability). In the following subsections, we present and summarize the results of each criterion.

5.3.1 Satisfaction

When describing the satisfaction of using EyeIDEA, participants P1, P2, P3, P6, and P7 found that the system was “Pleasant to use” and that they enjoyed the experience. Additionally, P1 and P7 stated that they expected a less enjoyable and difficult system and were positively surprised about it.

A feature that contributed to the overall satisfaction was that the system felt very similar to the original IDE, which meant that it was easy to learn.

Another feature that all participants enjoyed was that the action usage was fast and responsive. However, P3 and P4 said that the system sometimes responded too quickly to their gaze with P3 saying:

“I felt that I had to be careful where I was looking at which made it feel restrictive to use during certain moments.” - P3

A common dissatisfaction among the participants was that the gaze tracking was not always accurate enough to pinpoint the gaze to the intended target, which contributed to some annoyance. Additionally, all participants felt that the additional gaze buttons took up too much screen space, which made it more difficult to use the Editor.

Key Points

- The overall impression is that eye-tracking is “Pleasant to use”. However, the inaccuracies of the eye-tracking contribute to some annoyance, and in some cases, the system responds too quickly.

5.3.2 Usability

As mentioned in Section 4.3.3 Learnability, we evaluate the usability based on the score of the System Usability Score (SUS) questionnaire and the answers given by the participants during the interview.

System Usability Score After the experiment, the participants filled out the SUS questionnaire, which indicates the systems’ usability. The resulting score (Mean = 70.31, Standard Deviation = 7.22) can be translated to an adjective scale as “Good” [6]. Moreover, the acceptability of EyeIDEA can be seen as “Passable,” which is reserved for systems with a score above 70. This score means that the developed IDE seems promising, but it does not guarantee high acceptability when it is used outside a lab setting [6].

Overall System Usability More than half of the participants found that the system is simple to use and feels responsive. Out of the seven participants, four of them described the systems’ usability with words as “Easy to Use” (P7), “Simple” (P2, P5), and “Not Difficult” (P1). For the responsiveness, five participants described it either as “Feels faster than the mouse” (P2, P4), “Reacts Quickly” (P7), or “Responsive” (P1, P5).

Something that all the participants liked was that the implemented actions worked the same way that they are used to. Particularly, P6 described it as:

“To me, this felt very similar as if I would use the mouse.” - P6

Unfortunately, there were some inconsistencies felt by three participants regarding the eye-tracking performance. This was described by P4 as:

“[...] the gaze did not always reach far enough.” - P4

The participants worked around this issue by readjusting themselves or by quickly looking back and forth at another point on the screen.

Despite this issue, nobody felt fatigued while using the system. However, some were a bit tired because they had to remove their glasses to get through the calibration procedure successfully.

Another topic that arose when asking about the overall usability was that of the color selection system. Most of the participants said that they liked it. This was described by P3 as:

“It works quite good to select the right code because it gives you a conformation, so it also works without the gaze cursor.” - P3

An issue that was raised by many participants though was that the color assignment did not feel consistent and that you had to spend some effort remembering the correct color. For instance, P1 described it as:

“[...] when I first looked at the run button it was first green but then another time it was red” - P1

We also received feedback about some difficulties with selecting modes as described in Section 3.8.1. Participants P3 and P4 found that the buttons activated too quickly. They described it as if the system could activate an action without their input. This issue means that the current system did not prevent the Midas Touch[33] problem for everyone.

Usability of Controlling Buttons Two out of the seven participants (P4 and P7) chose the IDE button interaction as one of their favorite modes. They mentioned that it was the most logical to use, although we did observe that most of the participants primarily used this mode to activate the “run code” and “start debugger” buttons. Participant P5 mentioned that the distance from the buttons of the debugger window (located at the same position as panel 4 in Fig. 3.2) to the Gaze selection buttons (panel 1 in Fig. 3.2) is too big and that it felt cumbersome to use. Additionally, P4 and P7 had some issues with selecting the buttons because of their small sizes. This problem made it more difficult to see the highlight color and get the registered gaze at the correct position due to inaccuracies.

The participants were divided about the feature that the Button Control mode did not require an activation as shown in Fig. 3.24. P2, P5, and P7 liked the idea that it required fewer steps to activate a button, but the other half found it somewhat inconsistent compared to the other modes, and P3 even forgot this feature because it did not require a mode selection.

Navigation Actions Usability Most participants were positive about the usability of the navigation actions and described it as “Natural”. Six out of the seven participants (P1, P2, P4, P5, P6, P7) described that it felt similar to using the mouse for navigation. P2, P4, and P6 even mentioned that it felt faster than using the mouse. Additionally, they liked the responsiveness of these actions.

Debugging Actions Usability All the participants were positive about the usability of setting a line breakpoint with P1 and P3, liking that it had a direct relation with the code rather than having to locate the correct line first. We do have to note that this is not a unique feature with our tool as there is a shortcut in IntelliJ to place a breakpoint at the current cursor location in the Editor, but none of the participants knew that.

However, while the participants used line breakpoints, the conditional breakpoint was barely used. Most participants noted that they do not use this feature regularly and, as such, did not use them. Additionally, they noted that it felt complex to use since it required keyboard input to turn the code selection into a valid conditional expression.

A similar problem was also found for both of the evaluation modes. None of the participants are used to using the quick evaluation option during debugger and instead used the debug window as shown in Fig. 3.15 to view the variables. When asked about this, P2 said:

5. RESULTS

“[...] IntelliJ already shows a short summary of the value of the variable in the editor anyway.”. - P2

However, P1, P3, and P4 noted that they found the action itself was quick and easy to use. However, they needed more time during the debugging session and rather preferred to try to understand the code by reading it before using debugging tools.

Key Points

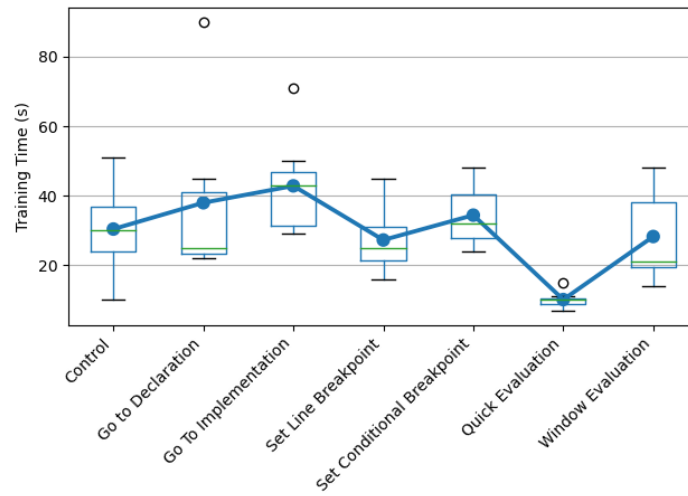
- The participants find EyeIDEA easy to use, responsive and the implemented actions feel similar as they are used to despite some inconsistencies with the eye-tracking.
- The perceived usability of the color highlights depends largely on the used colors as well as how consistent the assignment of a color to the same selectable element.
- Most participants find that the navigation actions are the most usable because the code selection feels the most natural and response.
- The participants perceived the debugging actions that do not require additional code input as more usable than the actions that do require it.

5.3.3 Learnability

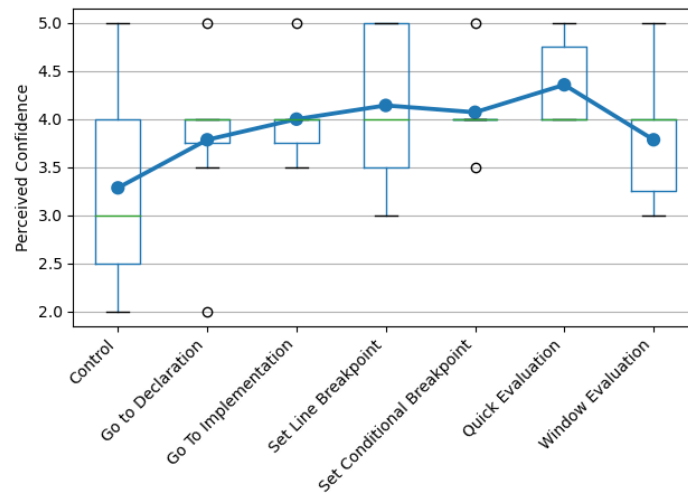
During the training, we timed the participants how long they took to complete the objectives listed in Table 4.1. Additionally, we asked the participants to rate their confidence in remembering how to use these actions.

Based on the results of the training times, which are shown in Fig. 5.5a, we observe that initially, the average training time goes up but seems to go down after the navigation modes. The interesting thing is that the *Set Conditional Breakpoint* and *Window Evaluation* takes more steps to complete, which should result in a higher training time, but this is not the case. We do observe this trend after the navigation modes with *Set Line Breakpoint* and *Quick Evaluation* taking considerably less training time. However, we also observe that the participants required more training time in the initial stages as they are getting accustomed to the interaction system. Additionally, we see that the spread of the training time decreases as participants are learning each mode, except for the very last mode in which the spread increases again. This indicates that the *Window Evaluation* mode could be more difficult to learn than the other modes.

Moreover, based on the confidence results shown in Fig. 5.5b, we observe that the participant had a wide varied opinion about the confidence after the very first mode. However, we do observe that this spread decreases significantly for the navigation modes and the *Set Conditional Breakpoint* which shows that the participants were probably unsure about using the *Control* mode as this was the very first time that they used EyeIDEA. There are some outliers in both the low and high-end spectrum, which could indicate that there are



(a) Individual results of the completion times for the different modes. The blue line represents the average training times for each mode.



(b) Individual results of the confidence for the different modes. The blue line represents the average confidence rating for each mode.

Figure 5.5: Individual plots of the training time results.

fast and slow learners present in the participant group. For the remaining modes, the spread is larger, which indicates that these modes could be seen as more difficult to learn for some participants.

Additionally, we also observe that on average, the participants gained more confidence between each mode during the training session except of the conditional breakpoint and window evaluation mode. Both of these modes were perceived by five participants (P1, P2,

5. RESULTS

P4, P6, P7) as the most difficult mode when asked *Which mode was the hardest to learn?*. The two main issues mentioned were from P4 and P7:

“I had to write stuff and not only use the eye-tracker.” - P4

“[...] it was not very clear to me how I can select the variables and then create a condition with it” - P7

So the main issue with this mode was the unclear selection of code. This relative difficulty can also be seen in Fig. 5.5a as it required more time to learn the conditional breakpoint than the similar Line Breakpoint mode.

A similar note can also be made about the Window Evaluation since it is very similar to the quick evaluation mode, which took less time to learn. Additionally, the confidence rating for the window evaluation was lower than the quick evaluation. This difference also paints the picture that participants perceived this mode as more difficult than the other ones.

However, both the *Set Line Breakpoint* and *Quick Evaluation* mode required fewer steps, and thus less time was to be expected. On the other hand, the navigation modes required around the same amount of steps but took more time to train. This increase was probably due to the ordering; P3 mentioned this when asked about which action was the easiest to learn. One thing that we conclude from Fig. 5.5a was that the participants were getting more proficient in using the interaction system, which drove the completion time down.

Furthermore, when asked about the easiest mode to learn, five out of the seven participants (P2, P3, P4, P5, P6) said they found the navigation actions the easiest to learn:

“Navigation requires fewer things to remember [...]” - P3

“I liked the navigation [...] felt very similar as if I would use the mouse.” - P6

This is somewhat reflected by the rising average confidence levels from the very first mode to both of the navigation modes. However, we did not observe this trend in the training times which is probably caused by the ordering of the tested modes. Besides the navigation modes, P1, P3, P6, and P7 mentioned that they also found that the line breakpoint mode was one of the easier modes:

“[...] the line breakpoints only involved selecting and clicking just like a mouse.” - P6

An interesting remark by participant P5 which was also shared by P3 was about the difficulty of the modes:

“[...] It just requires more steps that you have to do but most of them overlaps with the other modes, so you don’t have to learn anything different.” - P5

In addition to the completion times and perceived confidence, we also investigated the total amount of mistakes made during the training. These mistakes should also give some information about learnability. Particularly, namely which type of mistakes the participants made and how this changes during the training. We identified four different types of errors:

Error Type I - Selecting the wrong mode. This means that the participant switched to the wrong intended mode and had to switch back.

Error Type II - Forgetting to activate a mode before using the Gaze Buttons. The participant tried to use a Gaze Button but forgot to activate the mode, resulting in that the Gaze Buttons did nothing.

Error Type III - Looking at the wrong Gaze Button. Since each Gaze Button is color-coded, it can happen that a participant forgot the correct color or had trouble distinguishing between the colors, which resulted in selecting the wrong Gaze Button.

Error Type IV - Selected the wrong action or obtained the wrong result based on the current objective. Each of the training exercises had a certain objective that the participant had to follow to succeed. If the participant obtained a different result or executed the wrong action, the participant had to do it again.

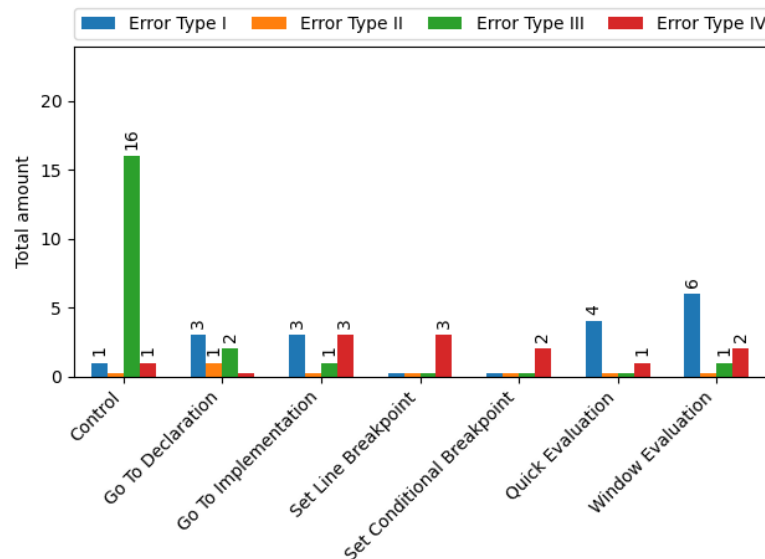


Figure 5.6: The total amount of faults made during the training session.

Based on the total amount of errors made during the complete training, we can observe in Fig. 5.6 that the different kinds of modes were easy to learn, and the objectives were easy to follow since the participants made few Type IV errors.

Additionally, we observe that there is a high amount of type III errors during the first objective. However, these type III errors shrank very fast for the subsequent objectives. In the beginning, the participants had to learn to differentiate between the different colors, but they picked it up very easily, and few mistakes were made in the remaining objectives.

Moreover, we observe that some participants selected the wrong modes during the navigation and evaluation modes as seen by the existence of Type I errors. P2 and P5 selected

the breakpoint selector on accident during the evaluation mode training because they forgot that the evaluator selector was a different button. Additionally, P2 and P6 mixed up the selection of the navigation modes. However, they quickly corrected themselves as they noticed that they navigated to the wrong source code location. Perhaps this confusion was that they normally use the Go To Declaration action as we described in Section 5.2.3 and forgot the other option, despite the training video.

Finally, we also observed only a single type II error right during the very first mode, *Go To Declaration*, that requires an activation before a user can use it. This indicates that it is very easy to adapt to select a mode before performing the associated action.

Key Points

- On average, the interaction system is easy to learn except for modes that require additional input from the keyboard. Additionally, it is easy to learn to localize the correct Gaze button based on the highlight color as the number of mistakes decreases rapidly.
- The participants learn the actions that feel the same as using the mouse faster than the other actions.

5.4 RQ3: Eye-tracking IDE Prospects

The final aspect that we considered was to investigate the perceived limitations and opportunities in using an eye-tracking IDE. During the interview, we made it clear to the participants that some of these questions were not about EyeIDEA itself but rather their experience with using an eye-tracker within an IDE.

5.4.1 Missing features and limitations

We started by asking about missing EyeIDEA features during the debugging session. An overwhelming response of the participants was that they missed the ability to switch between the editor tabs. We observed this as one of the primary methods that the participants used to navigate within the code. Additionally, four participants missed the “go back to the previous location” action. With this action, a participant can quickly jump between two files using eye-tracking. They especially missed it since the mouse in the user study did not have an additional side button which they would use to perform this action.

Another nice to have feature mentioned by P3, P4, and P6 was to include a scrolling option. However, their sentiment was that it would be difficult to create a good scrolling feature based on the current perceived limitations of the eye-tracker:

“[...] I honestly would have no idea how reliable it would be [...] eye-tracking is probably too sensitive for scrolling compared to a mouse scroll wheel or using a touchscreen.”
- P4

Besides the mentioned limitations during the regular questions, we also gave the participant an opportunity to talk about other limitations of EyeIDEA. An often heard complaint was that the current layout takes up too much screen space:

“[...] it feels like the editor is boxed in [...].” - P3

Currently, the size of the gaze buttons mostly determines the layout size. However, the participants found that the gaze buttons had a comfortable size since they had no issues selecting a button if their gaze could reach it. However, this means that there is a limit on the number of gaze buttons that can be used based on the screen size to prevent the “boxed in feeling of an editor”.

Another often heard issue was about the usage of the highlight colors. In particular, some colors were less distinguishable from others or felt too vibrant. Most notably, P1, P2, P4, and P7 found that the two different shades of green were too similar, especially when they highlighted anything other than source code. P1 also included that:

“[...] there are always differences between how good everyone can see the difference between colors [...].” - P1

This means that user preferences play a major role in the color palette, limiting the number of colors that we can use. In return, this also puts a limit on the maximum amount of gaze buttons. In addition, four participants (P1, P2, P4, P7) said that the current assignment of highlight colors changed too often:

“[...] when I first looked at the run button, it was first green but then moments after it was red so I had to always spend some time to remember the current color.” - P1

This issue limited the participants’ willingness to use the system whenever they were under a heavy mental load. However, it is not possible to find a unique color assignment for every element as mentioned in Section 3.9.3.

There was another limitation raised that also increased the mental load:

“Looking does not equal reading. [...] I sometimes just wanted to read some code before an action but I had to consciously switch to the action activation button and back to the code which broke my reading thoughts as it were. That is why I chose to not use the eye-tracking when I needed to think very deeply [...].” - P6

Other participants also mentioned something very similar where they used the mouse when they needed to think very deeply.

Key Points

- Common missing features are navigating between editor tabs and go back to a previous location.
- A limitation is that the Gaze buttons take up too much space on the used 15.6" laptop screen. The number of Gaze Buttons is dependent on the size of the screen.
- The participants are only comfortable with a limited number of different colors that are assigned to a selectable element.
- There are difficulties when switching between reading code and looking at the interface to select the right Gaze Button, especially under heavy mental load.

5.4.2 Challenges Compared to Keyboard and Mouse Usage

During our observations, we noticed that most participants (P1, P2, P3, P6, P7) lifted their hand from the mouse whenever they were using the eye-tracker and put it back when they needed the mouse again. According to those participants, this felt the most natural to them.

This behavior introduces a challenge to incorporate the eye-tracker into the current workflow. The participants said it felt difficult or unintuitive to switch between the eye-tracker or mouse to execute an action.

All the participants said they preferred not having to switch between different input methods during the execution of an action and therefore ranked those actions higher. For P3 and P5, it also felt that the current system required more steps to accomplish something. One of the reasons for this feeling was that these participants always used a combination of using a keyboard and mouse to perform something, i.e., selecting an action with the keyboard while simultaneously clicking on a code fragment. However, we mentioned above that it was difficult to use eye-tracking in combination with the mouse or keyboard.

A second challenge is that the eye-tracker had lower accuracy and precision compared to the mouse. While the implemented Gaze buttons and selection system took this into account, there were still some issues with using it. Primarily, the posture of the participants had a big influence on eye-tracking performance. Sometimes, the changing posture caused an issue that the eye-tracking did not reach far enough. Additionally, it also means that the performed calibration only works when the complete working setup is constant, i.e., same desk, same chair height, which makes it cumbersome to use if there is no fixed workplace, for instance, when traveling.

The final challenge is to avoid premature activation of modes. Some participants let their eyes drift more easily than others which activated a mode in some cases. Especially for P3:

“Sometimes, it felt like an action was activated without me looking at the button, or at least, it felt that way.” - P3

Key Points

- A part of the test group finds it difficult to switch between mouse usage and eye-tracking.
- A small change in posture has a big influence on the current eye-tracking performance, which forces users to keep their body more still compared to mouse or keyboard.
- It is more difficult to avoid a premature mode activation with the implemented eye-tracking interaction system.

5.4.3 Perceived Audience

Who would benefit from an eye-tracking IDE? According to all the participants, programmers are not able to use the mouse. However, the reason and the eventual audience differed. For instance, participants P1, P4, P5, and P7 stated that programmers with medical issues that relate to their hands, either permanently or temporarily like an injury such as RSI, could benefit from an eye-tracking IDE in order to replace the mouse.

The second reason for not being able to use the mouse that was stated by P3, P4 and P6, was “*programming on the go*” or “*programming on a couch*”. In these situations, a programmer would probably not have access to a physical mouse and would most likely replace it with a track-pad or touchscreen. According to P6:

“A track-pad feels tedious to use and it always feels as if I am a lot slower when working with it. I think that an eye-tracker would be faster and more responsive than a track-pad [...]”. - P6

However, an issue that was raised by the same participant was that it would require a fast calibration process otherwise too much time would be spent on setting up the system. Additional, P6 mentioned:

“[...] the eye-tracker should be reliable too since you shift a lot with your position, something that this system unfortunately has some trouble with.” - P6

A second perceived audience can be grouped as developers that seeks additional tools to help with their focus. For instance, as a guidance during tutorials for beginning developers to indicate which part of the source code on the screen is interesting based on what they are looking at:

“[...] something like a more user focused tutorial.” - P1

Another example from P3 was to use eye-tracking as an aid during collaboration sessions because:

5. RESULTS

“[...] you don’t have to or can’t point to a shared screen to indicate code fragments [...]” - P3

This idea could help to get the other attendants’ attention such that they look at the same code fragment as the active developer, resulting in fewer misunderstandings among the attendants. D’Angelo et al. [15] explored this idea already for pair programming and found it developers found it helpful.

The third identified audience are developers that seek additional productivity by improving their navigation or debugging actions. The stated requirements were to offer “quick and easily accessible” actions that to prevent situations as:

*“[...] having to reach for the mouse while I prefer to keep my hands on the keyboard.”
- P7*

This sentiment was also described earlier in Section 5.3.3. In that section, we mentioned that the participants preferred the simpler actions such as setting line breakpoints over the complex conditional breakpoint action that required multiple selections or keyboard input.

But what about the participants themselves? From the seven participants, P1 and P2 would definitely use the implemented eye-tracking IDE, while P4, P5, and P7 needed more time in order to have an opinion about it. The remaining participants liked the idea, but they found the accuracy and precision of the eye-tracking left something to be desired, which made it feel unwieldy to use at some moments.

Key Points

- Programmers that cannot use the mouse because of health issues or do not have access to a mouse are the most popular perceived audience.
- Less important audiences are programmers who search for tools to help with their focus or productivity.

5.4.4 Proposed Ideas for an Eye-Tracking IDE

Instead of only criticizing EyeIDEA and eye-tracking for IDEs, we also gave the participants an opportunity to suggest new ideas for an eye-tracking IDE based on their current impressions. We made a selection of these ideas and grouped them based on popularity and their relation to the current challenges mentioned in Sections 5.4.1 and 5.4.2.

The most popular ideas we found were about changing the current layout such that the gaze controls take up less space, reduce accidental mode selections, and make the Part selection less cumbersome. These ideas revolved around the following ideas:

Changes to the Gaze action button panel Transform the Gaze action button panel (marked as 1 in Fig. 3.2) into a floating popup, such that its position is closer to the highlighted elements. This change makes it less cumbersome to select a Part. The number of Gaze buttons should be reduced to make sure that this change does not take up too much space. This change also reduces the complexity of picking enough distinguishable colors since there are fewer Gaze buttons. Additionally, the Gaze buttons should include the corresponding code element to reduce the issue of switching between looking and reading. However, the participant found it challenging to come up with an idea to trigger the placement of this popup. There were suggestions to either use a keyboard shortcut, another Gaze button, or to implement a Gaze gesture to activate it.

Grouping modes together A similar idea was to group the different modes into a single popup to eliminate the left mode panel to make room for the Project tool window shown in Fig. 3.14. The remaining panel (marked as 3 in Fig. 3.2) could be used for the Gaze buttons to call the Part and Mode selection popups. By removing these Gaze buttons, we could reduce the accidental mode selections. The reason is that there is only a single central place to activate the mode selection instead of all around the Editor.

Simplify Control Panel The last idea was to simplify the Control panel (marked as 4 in Fig. 3.2) by moving it into a menu to free up room for the Run tool window as shown in Fig. 3.14.

We illustrate all of these ideas in Fig. 5.7 which includes the new floating gaze selection popup, the removal of the left mode panel, and the new menu for the gaze controls.

The second most popular ideas were to add customization options such as custom color selections, changing which mode can be quickly accessed, and adjustable sensitivity of the Gaze buttons. By adding these customizations, a developer could get a better experience at the cost of some setup time. However, a suggestion was to add different default profiles to prevent this. These profiles are based on activities and skill levels, such as a travel profile where emulated mouse control is more important or a novice profile where the buttons' sensitivity is lowered.

Besides these ideas, we also received feedback that the eye-tracking felt at its best when it gave an immediate reaction like it is currently with navigation or selecting a button. This feedback means that new modes and resulting action should be implemented with this in mind. For instance, P5 and P7 suggested that the current conditional breakpoint mode could be an addition to the line breakpoint mode, i.e., the user places the breakpoint first and then to the code selection for the condition. This way, the user does not have to construct the complete condition first but always gets an immediate response.

5. RESULTS

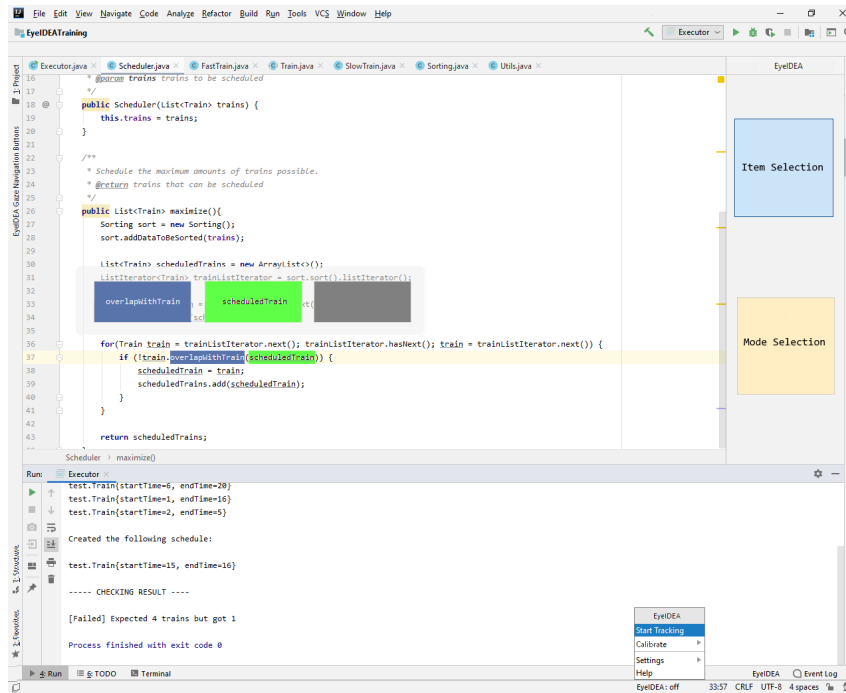


Figure 5.7: A mock-up of the ideas to change the current layout instead of the current fixed window at the top of the screen.

Key Points

- The most popular proposed idea we received is to change the current layout such that it takes up less space, less accidental mode selection, and closer to the selection to reduce the switching between reading and looking.
- The eye-tracking IDE feels at its best when an action does something immediately, rather than having to perform multiple (selection) steps.

Chapter 6

Discussion

In this chapter, we start with formulating answers to the research questions. We will use the results of our user study and compare this against relevant literature whenever possible. Then, we present the implications of our findings for an augmented eye-tracking IDE. Lastly, we address the risks that can affect the validity of our study.

6.1 Research Questions

In this section, we revisit the research questions and use the user study results to answer them. The findings of our research will also be compared against related work whenever possible.

RQ1: *How do developers interact with the eye-tracking augmented IDE compared to the non-augmented IDE?*

Based on our user study, the developers used the eye-tracking augmented IDE differently than the non-augmented IDE. With EyeIDEA, the developers showed a lot more volatility in switching between different windows. This behavior is to be expected since the implemented interactions rely on activating dwell-based buttons that are placed around the Editor. However, the amount of effort that this takes negatively impacts the usability of EyeIDEA. A probable cause is that these transitions result in numerous context switches, i.e., developers have to switch between thinking about the code and locating the right gaze button based on the highlight color. Meyer et al. indicate that these context switches take a developer “out of the flow” and reduces the perceived productivity [43].

Another difference is that developers rely on navigation a lot more with the eye-tracking IDE as they are looking more selectively to the DebugTool and RunTool, and use more navigation actions. Developers found the navigation actions more usable because they were simple, and single code selections were more stable and thus used more. Additionally, it could be that the screen estate taken up by the tool influenced the usability of the DebugTool and the readability of the Editor as multiple students reported that they liked to have a lot of screen space to display source code.

RQ2: *How do developers perceive the usability, learnability and satisfaction of the developed eye-tracking IDE?*

Overall, the developers were positive about using the eye-tracking system. However, the eye-tracking lacked some accuracy and precision at times, which introduced usability issues such as that a specific location could not be reached on the screen or that a developer activated a dwell-based button on accident. These issues were less apparent when selecting code because it is displayed in the middle of the screen, which is an area that generally has better eye-tracking quality [19]. This observation means that to mitigate the remaining “Midas touch problems”, the interaction elements should be moved away from the edges of the screen.

We also found that developers perceive the actions which require only a single selection as more usable than actions that require multiple selections. This could be because multiple selections take too much time, which reduces the ability to think about the code to form a mental model. By keeping it simple and fast, this interruption can be kept at a minimum.

The interaction system was perceived as easy to learn, and the training results showed that the number of errors went down rapidly and confidence went up during the training. This result means that this type of interaction is not confusing to developers, which is vital as developers have trouble with learning how to use an IDE efficiently [37].

RQ3: *What are the perceived challenges with an eye-tracking IDE?*

We found three main challenges with an eye-tracking IDE. The first challenge is that large dwell-based buttons are needed to compensate for the eye-tracking performance, which reduces the screen space for the Editor. The developers found that this made the Editor feel “boxed in”. This challenge means that the screen size dictates how many buttons can be placed beside the Editor. CodeGazer used the same type of interaction method [60] but did not report about this issue, possibly because they used a much larger screen in their study.

The second challenge is that developers perceived a big difference between reading code and looking at code. This difference caused issues when using eye-tracking under heavy mental load as they are forming a model of the code inside their head. This issue is also noted in the CodeGazer tool as developers preferred the tool for “light code exploration work”.

The final challenge is that there are difficulties in using eye-tracking alongside the keyboard and mouse. In particular, our results show that the students found it difficult to keep their hands on the mouse as they are using the eye-tracker. A reason could be that since both mouse and eye-tracking use an (x,y) coordinate system, that the developers avoid using the mouse because it is confusing to look at the screen without moving the mouse cursor. Instead, they let go of the mouse to treat eye-tracking as a mouse replacement. Also, since developers adjust their posture unconsciously when using the keyboard and mouse, the eye-tracking performance degrades, which forces the developers to keep their heads unnatural still. However, it is meaningful to note that as eye-trackers are continuously improved, we expect that it becomes less of an issue.

MRQ: *Do developers perceive the eye tracking augmented IDE as useful?*

Based on the formulated answers to the previous research questions, developers perceive the eye-tracking-augmented IDE as useful under certain conditions. Firstly, the quality of eye-tracking has a significant impact on the usability of the IDE, which limits the design of the interaction systems. Secondly, simple actions that only require a single selection are perceived as useful, while complex actions are not seen as useful. Finally, the audience and environment play a role when developers perceive eye-tracking as useful. Most notably is the situation in which a developer either cannot use the mouse because of health-related issues, or there is no mouse available.

6.2 Implications

We observed several implications based upon the results and answers described in the previous section.

- The availability of eye-tracking-based actions can influence the behavior of developers. They relied more on navigation actions and less on debugging. The implication here is that developers perceived the navigation actions as more useful than controlling the DebugTool buttons. Our goal was to establish areas that need further attention, and this seems a good candidate for further investigation.
- Eye-tracking interactions are perceived as less usable when under heavy mental load. This mental load could range from navigating and using debugger buttons in quick succession to create a mental picture of a code snippet while staring at the screen. This issue indicates another area that can benefit from additional research, especially since the developers seemed to have trouble switching between using the mouse and eye-tracking.
- Depending on the eye-tracking technology, developers have to keep their heads in a similar position as during the calibration. This proved to be somewhat difficult for the developers. Therefore, the augmented IDE should include tools to help the developers to adjust their posture.
- The design of the tool should keep the layout size in mind to make sure that the Editor is large enough to avoid the “boxed in” feeling. Therefore, it is advisable to consider in other designs that additional interaction elements should be kept hidden and only made visible when the developer wants to interact.

6.3 Threats to Validity

An eye-tracking augmented IDE opens up new possibilities to investigate how developers are using an IDE. Together with interviews and questionnaires, it can provide insights into the usefulness of said augmented IDE. However, these kinds of user studies have their own issues that can challenge the validity of the results and findings.

6.3.1 Internal Validity

For our study, the participants always conducted the first debugging session without eye-tracking capabilities before the second session that did include it. Therefore, it could be that the order of these sessions influenced the usability of eye-tracking since they first experience the IDE as they normally would. This experience gives them certain expectations on what eye-tracking interactions should feel. However, the participants did receive a training session beforehand to familiarize themselves with the eye-tracking tool to make sure that they already have some experience, although this is not fully comparable with their experience with mouse and keyboard. Furthermore, half of the participants used a different order of two different code snippets to make sure there was no learning effect with using the IDE twice with the same code. The random ordering of the code snippets was also to make sure that the differences between the code snippets themselves did not influence the results.

Some questions during the interview left some room for interpretation as some participants provided more information to a question than necessary. Some participants even gave a partial answer to another question, which could have influenced their answer to that question. However, we did make sure to avoid putting pressure on the participants to provide us with more information by making sure that they understood that this study was about their perceptions and that we are interested in understanding the usefulness of an eye-tracking IDE. Particularly, that we investigate both the advantages and disadvantages to figure out limiting and positive aspects of an eye-tracking IDE such that we can determine if further research is beneficial.

Similarly, the participants might have responded more positively to not disappoint us. However, we believe that this did not happen for multiple reasons. Firstly, the participants were free to use the eye-tracking under any circumstance that suited their needs. We can confirm that the participants chose their actions voluntary as they switched between eye-tracking, keyboard and mouse input under specific situations. Additionally, we observed significant variances in eye-tracking usage between participants which shows that they used it only when they wanted to use it. Secondly, we observed that the participants' answers coincided with the eye-tracking data that we collected. This observation means that the participants truthfully answered the questions. Finally, we received negative feedback about the eye-tracking IDE and multiple participants expressed the same negative feedback. This feedback means that the participants felt comfortable to express their negative opinions.

6.3.2 External Validity

For our study, we only invited students, which diminishes the generalizability across all types of developers. While the students had different education levels and real-world programming experience, they do not have the experience level of a senior developer. This lack of experience level is especially true for working with unfamiliar code since the invited students indicated that they mostly write their own code and do not often work with unfamiliar code. While it is possible to give a separate group more time to familiarize themselves with the code, it would have increased the study duration, making it more difficult to recruit enough students.

Furthermore, the used code bases varied somewhat in size, bug types, and their perceived complexity. While they were created to simulate real-world code, there is still a difference with real-world code. Most notably, the size is generally much larger, which requires more effort to understand the code before developers can find bugs. Additionally, we only used Java as the programming language and IntelliJ IDEA as the IDE. Since there are numerous differences between programming languages and IDEs, our findings are not generalizable across all types of languages and IDEs.

Additionally, we only used a specific eye-tracker, which has certain performance properties that relate to the accuracy, precision, and how much a developer could move and adjust their posture. Therefore, different results could be obtained with eye-trackers that exhibit different performance properties.

Another point is that the participants had only a limited amount of time to evaluate the developed tool. However, this is not an issue since the goal was to investigate the initial perceptions about an eye-tracking IDE, which we used to gather information about areas that need more investigation.

We also cannot exclude the novelty effect, especially since most developers have never used eye-tracking IDE before. However, we obtained similar positive responses as CodeGazer [60] which we have based our interaction system on. This result suggests that the responses to the interaction system were genuine.

Chapter 7

Conclusions and Future Work

In this paper, we introduced an augmented IDE that uses eye-tracking to provide alternative interaction methods and fine-grained information about the usage of this IDE. We conducted a user study to research the initially perceived usefulness of this augmented IDE with a particular focus on *(i)* IDE usage between non-augmented and augmented IDE *(ii)* perceptions of the developed tool regarding usability, learnability, and satisfaction *(iii)* future prospects of using eye-tracking in general for software development tasks inside an IDE.

The main findings of our user study are that *(i)* navigation and setting breakpoints are perceived as the most useful eye-tracking interactions, *(ii)* actions that do not require multiple selections or are not selected in quick succession are perceived as the most suited for eye-tracking interactions, *(iii)* switching between eye-tracking input and mouse/keyboard input is perceived as difficult, *(iv)* eye-tracking feels natural, but accuracy and precision problems reduces the overall usefulness, *(v)* gaze-based actions are easy to learn, and developers adapt quickly to it and *(vi)* an eye-tracking IDE is perceived as suitable for light programming work and could work as a substitute for a mouse in case of health issues or traveling.

Based on these findings, we speculate that an augmented eye-tracking IDE is perceived as useful under certain conditions. Additionally, there are some implications in designing such an IDE to improve their perceptions. It seems that eye-tracking interactions are perceived as less useful when under heavy mental load, indicating that special care should be taken in the design of interactions to no disturb the developer when under heavy mental load. This result could be achieved by making sure that there are no unwanted visual elements on the screen because of the interaction system. Additionally, it seems that the interaction between mouse, keyboard, and eye-tracking poses some challenges, and further research is needed into these issues. Finally, the design should keep the layout size into account to keep the Editor as large as possible and not surround it with large GUI components.

Our methodology and design of the augmented IDE have some limitations which need to be addressed. Firstly, the user study is only conducted in a lab setting with students. In order to generalize this, an industry study is needed to test how our findings stand with professional developers in a different environment. Secondly, our study only used a specific IDE and programming language, which decreases the generality of all possibly augmented eye-tracking IDEs.

7. CONCLUSIONS AND FUTURE WORK

Our long-term vision is that new tools and interaction methods that use eye-tracking to support developers can be beneficial. As future work, we call upon other researchers to improve and expand our work by integrating eye-tracking within other existing IDEs, developing different interaction methods, and using different investigation methods into developers' behaviors to understand developers' needs from an IDE.

Bibliography

- [1] Why do computer programmers dislike using the mouse? - Quora, 2017. URL <https://www.quora.com/Why-do-computer-programmers-dislike-using-the-mouse>. [Online; accessed 15-October-2020].
- [2] Afsoon Afzal and Claire Le Goues. A Study on the Use of IDE Features for Debugging. In *Proceedings of the 15th International Conference on Mining Software Repositories, MSR '18*, pages 114–117, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450357166. doi: 10.1145/3196398.3196468. URL <https://doi.org/10.1145/3196398.3196468>.
- [3] Maike Ahrens and Kurt Schneider. Using Eye Tracking Data to Improve Requirements Specification Use. In Nazim Madhavji, Liliana Pasquale, Alessio Ferrari, and Stefania Gnesi, editors, *Requirements Engineering: Foundation for Software Quality*, pages 36–51, Cham, 2020. Springer International Publishing. ISBN 978-3-030-44429-7.
- [4] S. Amann, S. Proksch, S. Nadi, and M. Mezini. A Study of Visual Studio Usage in Practice. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 124–134, 2016. doi: 10.1109/SANER.2016.39.
- [5] Jay Ayres, Jason Flannick, Johannes Gehrke, and Tomi Yiu. Sequential pattern mining using a bitmap representation. In *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '02*, pages 429–435, New York, NY, USA, 2002. Association for Computing Machinery. ISBN 158113567X. doi: 10.1145/775047.775109. URL <https://doi.org/10.1145/775047.775109>.
- [6] Aaron Bangor, Philip T. Kortum, and James T. Miller. An Empirical Evaluation of the System Usability Scale. *International Journal of Human–Computer Interaction*, 24(6):591–593, 2008. doi: 10.1080/10447310802205776. URL <https://doi.org/10.1080/10447310802205776>.

- [7] Roman Bednarik and Markku Tukiainen. Temporal Eye-Tracking Data: Evolution of Debugging Strategies with Multiple Representations. pages 100–102, January 2008. doi: 10.1145/1344471.1344497.
- [8] Jennifer C. Romano Bergstrom, Erica L. Olmsted-Hawala, and Matt E. Jans. Age-Related Differences in Eye Tracking and Usability Performance: Website Usability for Older Adults. *International Journal of Human–Computer Interaction*, 29(8):545–546, 2013. doi: 10.1080/10447318.2012.728493. URL <https://doi.org/10.1080/10447318.2012.728493>.
- [9] John Brooke. SUS: A quick and dirty usability scale, 1996. URL <https://hell.maiert.org/core/pdf/sus.pdf>. [Fetched from site; accessed 15-October-2020].
- [10] E. C. Campos and M. d. A. Maia. Common Bug-Fix Patterns: A Large-Scale Observational Study. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, page 407, 2017. doi: 10.1109/ESEM.2017.55.
- [11] K. R. Chandrika and J. Amudha. A fuzzy inference system to recommend skills for source code review using eye movement data. *Journal of Intelligent & Fuzzy Systems*, 34:1743–1754, 2018. ISSN 1875-8967. doi: 10.3233/JIFS-169467. URL <https://doi.org/10.3233/JIFS-169467>. 3.
- [12] K. R. Chandrika, J. Amudha, and Sithu D. Sudarsan. Identification and Classification of Expertise Using Eye Gaze—Industrial Use Case Study with Software Engineers. In Jagdish Chand Bansal, Mukesh Kumar Gupta, Harish Sharma, and Basant Agarwal, editors, *Communication and Intelligent Systems*, pages 391–405, Singapore, 2020. Springer Singapore. ISBN 978-981-15-3325-9.
- [13] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An Empirical Study of Operating Systems Errors. *Operating Systems Review (ACM)*, 35: 83–86, September 2001. doi: 10.1145/502059.502042.
- [14] Tom N. Cornsweet. Determination of the Stimuli for Involuntary Drifts and Saccadic Eye Movements*. *J. Opt. Soc. Am.*, 46(11):987–993, November 1956. doi: 10.1364/JOSA.46.000987. URL <http://www.osapublishing.org/abstract.cfm?URI=josa-46-11-987>.
- [15] Sarah D’Angelo and Andrew Begel. Improving Communication Between Pair Programmers Using Shared Gaze Awareness. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, CHI ’17, pages 6245–6255, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450346559. doi: 10.1145/3025453.3025573. URL <https://doi.org/10.1145/3025453.3025573>.
- [16] Robert DeLine, Amir Khella, Mary Czerwinski, and George Robertson. Towards Understanding Programs through Wear-Based Filtering. In *Proceedings of the 2005 ACM Symposium on Software Visualization*, SoftVis ’05, page 186, New York, NY, USA, 2005. Association for Computing Machinery. ISBN 1595930736. doi: 10.1145/1056018.1056044. URL <https://doi.org/10.1145/1056018.1056044>.

-
- [17] Dmitry Kandalov. Activity Tracker - plugin for IntelliJs IDE — JetBrains, 2020. URL <https://plugins.jetbrains.com/plugin/8126-activity-tracker>. [Online; accessed 15-October-2019].
- [18] Sarah Fakhoury, Yuzhan Ma, Venera Arnaoudova, and Olusola Adesope. The Effect of Poor Source Code Lexicon and Readability on Developers' Cognitive Load. In *Proceedings of the 26th Conference on Program Comprehension, ICPC '18*, pages 286–296, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5714-2. doi: 10.1145/3196321.3196347. URL <http://doi.acm.org/10.1145/3196321.3196347>.
- [19] Anna Maria Feit, Shane Williams, Arturo Toledo, Ann Paradiso, Harish Kulkarni, Shaun Kane, and Meredith Ringel Morris. Toward Everyday Gaze Input: Accuracy and Precision of Eye Tracking and Implications for Design. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems, CHI '17*, pages 1125–1126, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450346559. doi: 10.1145/3025453.3025599. URL <https://doi.org/10.1145/3025453.3025599>.
- [20] Philippe Fournier Viger, Chun-Wei Lin, Antonio Gomariz, Ted Gueniche, Azadeh Soltani, Zhi-Hong Deng, and Hoang Lam. The SPMF Open-Source Data Mining Library Version 2. volume 9853, pages 36–40, September 2016. ISBN 978-3-319-46130-4. doi: 10.1007/978-3-319-46131-1_8.
- [21] T. Fritz and S. C. Müller. Leveraging Biometric Data to Boost Software Developer Productivity. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 5, pages 66–77, 2016. doi: 10.1109/SANER.2016.107.
- [22] Hartmut Glücker, Felix Raab, Florian Echtler, and Christian Wolff. EyeDE: Gaze-enhanced Software Development Environments. In *Proceedings of the Extended Abstracts of the 32Nd Annual ACM Conference on Human Factors in Computing Systems, CHI EA '14*, pages 1555–1560, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2474-8. doi: 10.1145/2559206.2581217. URL <http://doi.acm.org/10.1145/2559206.2581217>.
- [23] J. Grudin. Utility and Usability: Research Issues and Development Contexts. *Interact. Comput.*, 4:209–217, 1992. doi: 10.1016/0953-5438(92)90005-Z.
- [24] Drew Guarnera, Corey Bryant, Ashwin Mishra, Jonathan Maletic, and Bonita Sharif. iTrace: Eye Tracking Infrastructure for Development Environments. pages 1–3, June 2018. doi: 10.1145/3204493.3208343.
- [25] E. D. Guestrin and M. Eizenman. General theory of remote gaze estimation using the pupil center and corneal reflections. *IEEE Transactions on Biomedical Engineering*, 53(6):1125–1126, June 2006. ISSN 1558-2531. doi: 10.1109/TBME.2005.863952.

BIBLIOGRAPHY

- [26] Hal's Corner. Key Promotor X - plugin for IntelliJ's IDE — JetBrains, 2020. URL <https://plugins.jetbrains.com/plugin/9792-key-promoter-x>. [Online; accessed 15-October-2020].
- [27] M. Hamill and K. Goseva-Popstojanova. Common Trends in Software Fault and Failure Data. *IEEE Transactions on Software Engineering*, 35(4):492–493, 2009. doi: 10.1109/TSE.2009.3.
- [28] Michael E. Hansen, Robert L. Goldstone, and Andrew Lumsdaine. What Makes Code Hard to Understand? *CoRR*, abs/1304.5257, 2013. URL <http://arxiv.org/abs/1304.5257>.
- [29] 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*, ETRA '12, pages 197–200, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450312219. doi: 10.1145/2168556.2168592. URL <https://doi-org.tudelft.idm.oclc.org/10.1145/2168556.2168592>.
- [30] Haytham Hijazi, Ricardo Couceiro, João Castelhana, Paulo Carvalho, Miguel Castelo-Branco, and Henrique Madeira. Intelligent Biofeedback Augmented Content Comprehension (TellBack). *IEEE Access*, PP:1–14, February 2021. doi: 10.1109/ACCESS.2021.3058664.
- [31] Kenneth Holmqvist and Richard Andersson. *Eye-tracking: A comprehensive guide to methods, paradigms and measures*, pages 33–35,132. November 2017. ISBN 978-1979484893.
- [32] Constantina Ioannou, Indira Nurdiani, Andrea Burattin, and Barbara Weber. Mining reading patterns from eye-tracking data: method and demonstration. *Software and Systems Modeling*, 19(2):345–369, March 2020. ISSN 1619-1374. doi: 10.1007/s10270-019-00759-4. URL <https://doi.org/10.1007/s10270-019-00759-4>.
- [33] Robert J. K. Jacob. The Use of Eye Movements in Human-Computer Interaction Techniques: What You Look at is What You Get. *ACM Trans. Inf. Syst.*, 9(2):156, April 1991. ISSN 1046-8188. doi: 10.1145/123078.128728. URL <https://doi.org/10.1145/123078.128728>.
- [34] Ahmad Jbara and Dror G. Feitelson. How programmers read regular code: a controlled experiment using eye tracking. *Empirical Software Engineering*, 22(3):1440–1477, June 2017. ISSN 1573-7616. doi: 10.1007/s10664-016-9477-x. URL <https://doi.org/10.1007/s10664-016-9477-x>.
- [35] Silge Julia, Taylor Anita, and Devine Beth. Developer Survey Results 2019, 2020. URL <https://insights.stackoverflow.com/survey/2019#development-environments-and-tools>. [Online; accessed 16-February-2021].

- [36] 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, ESEC/FSE 2015*, pages 202–213, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3675-8. doi: 10.1145/2786805.2786864. URL <http://doi.acm.org/10.1145/2786805.2786864>.
- [37] Rex Bryan Kline and Ahmed Seffah. Evaluation of integrated software development environments: Challenges and results from three empirical studies. *International Journal of Human-Computer Studies*, 63(6):607–627, 2005. ISSN 1071-5819. doi: <https://doi.org/10.1016/j.ijhcs.2005.05.002>. URL <https://www.sciencedirect.com/science/article/pii/S1071581905001102>.
- [38] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung. An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks. *IEEE Transactions on Software Engineering*, 32(12):971–987, 2006. doi: 10.1109/TSE.2006.116.
- [39] Andrew J. Ko, Thomas D. Latoza, and Margaret M. Burnett. A Practical Guide to Controlled Experiments of Software Engineering Tools with Human Participants. *Empirical Softw. Engg.*, 20(1):123–124, February 2015. ISSN 1382-3256. doi: 10.1007/s10664-013-9279-3. URL <https://doi.org/10.1007/s10664-013-9279-3>.
- [40] Seolhwa Lee, Danial Hooshyar, Hyesung Ji, Kichun Nam, and Heuseok Lim. Mining biometric data to predict programmer expertise and task difficulty. *Cluster Computing*, 21(1):1097–1107, March 2018. ISSN 1573-7543. doi: 10.1007/s10586-017-0746-2. URL <https://doi.org/10.1007/s10586-017-0746-2>.
- [41] Zhenmin Li, Lin Tan, Xuanhui Wang, Shan Lu, Yuanyuan Zhou, and Chengxiang Zhai. Have things changed now?: An empirical study of bug characteristics in modern open source software. pages 28–29, January 2006. doi: 10.1145/1181309.1181314.
- [42] Christof Lutteroth, Moiz Penkar, and Gerald Weber. Gaze vs. Mouse: A Fast and Accurate Gaze-Only Click Alternative. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology, UIST '15*, pages 385–394, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450337793. doi: 10.1145/2807442.2807461. URL <https://doi.org/10.1145/2807442.2807461>.
- [43] André N. Meyer, Thomas Fritz, Gail C. Murphy, and Thomas Zimmermann. Software Developers' Perceptions of Productivity. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2014*, pages 25–26, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450330565. doi: 10.1145/2635868.2635892. URL <https://doi.org/10.1145/2635868.2635892>.

- [44] R. Minelli, A. Mocci, M. Lanza, and L. Baracchi. Visualizing Developer Interactions. In *2014 Second IEEE Working Conference on Software Visualization*, pages 147–156, 2014. doi: 10.1109/VISSOFT.2014.31.
- [45] R. Minelli, A. Mocci, and M. Lanza. I Know What You Did Last Summer - An Investigation of How Developers Spend Their Time. In *2015 IEEE 23rd International Conference on Program Comprehension*, pages 25–35, 2015.
- [46] R. Minelli, A. Mocci, and M. Lanza. Measuring Navigation Efficiency in the IDE. In *2016 7th International Workshop on Empirical Software Engineering in Practice (IWESEP)*, pages 1–6, 2016. doi: 10.1109/IWESEP.2016.11.
- [47] G. C. Murphy, M. Kersten, and L. Findlater. How are Java software developers using the Eclipse IDE? *IEEE Software*, 23(4):76–83, 2006.
- [48] Jakob Nielsen. *Usability Engineering*, pages 23–37. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993. ISBN 0125184050. URL <https://dl.acm.org/doi/10.5555/2821575>.
- [49] M. Nyström, Richard Andersson, K. Holmqvist, and Joost Weijer. The influence of calibration method and eye physiology on eyetracking data quality. *Behavior Research Methods*, 45:281, 2013. doi: 10.3758/s13428-012-0247-4.
- [50] Alexandra Papoutsaki, Patsorn Sangkloy, James Laskey, Nediya Daskalova, Jeff Huang, and James Hays. WebGazer: Scalable Webcam Eye Tracking Using User Interactions. In *Proceedings of the 25th International Joint Conference on Artificial Intelligence (IJCAI)*, page 3841. AAAI, 2016.
- [51] Roy D. Pea. Language-Independent Conceptual “Bugs” in Novice Programming. *Journal of Educational Computing Research*, 2(1):27–29, 1986. doi: 10.2190/689T-1R2A-X4W4-29J2.
- [52] F. Petrillo, Z. Soh, F. Khomh, M. Pimenta, C. Freitas, and Y. Guéhéneuc. Towards Understanding Interactive Debugging. In *2016 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 152–163, 2016. doi: 10.1109/QRS.2016.27.
- [53] Jan Pilzer, Raphael Rosenast, André N. Meyer, Elaine M. Huang, and Thomas Fritz. Supporting Software Developers’ Focused Work on Window-Based Desktops. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, CHI ’20, pages 1–13, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450367080. doi: 10.1145/3313831.3376285. URL <https://doi.org/10.1145/3313831.3376285>.
- [54] David J. Piorkowski, Scott D. Fleming, Irwin Kwan, Margaret M. Burnett, Christopher Scaffidi, Rachel K. E. Bellamy, and Joshua Jordahl. The Whats and Hows of Programmers’ Foraging Diets. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI ’13, pages 3063–3072, New York, NY,

- USA, 2013. Association for Computing Machinery. ISBN 9781450318990. doi: 10.1145/2470654.2466418. URL <https://doi-org.tudelft.idm.oclc.org/10.1145/2470654.2466418>.
- [55] Sebastian Proksch, Sven Amann, and Sarah Nadi. Enriched Event Streams: A General Dataset for Empirical Studies on in-IDE Activities of Software Developers. In *Proceedings of the 15th International Conference on Mining Software Repositories*, MSR '18, pages 62–65, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450357166. doi: 10.1145/3196398.3196400. URL <https://doi.org/10.1145/3196398.3196400>.
- [56] Stevche Radevski, Hideaki Hata, and Kenichi Matsumoto. EyeNav: Gaze-Based Code Navigation. In *Proceedings of the 9th Nordic Conference on Human-Computer Interaction*, NordiCHI '16, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450347631. doi: 10.1145/2971485.2996724. URL <https://doi.org/10.1145/2971485.2996724>.
- [57] 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*, ICSE 2014, pages 390–401, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2756-5. doi: 10.1145/2568225.2568247. URL <http://doi.acm.org/10.1145/2568225.2568247>.
- [58] Jonathan Saddler. *Understanding Eye Gaze Patterns in Code Comprehension*. Dissertation, University of Nebraska - Lincoln, May 2020.
- [59] 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*, ESEC/FSE 2015, pages 954–957, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3675-8. doi: 10.1145/2786805.2803188.
- [60] Asma Shakil, Christof Lutteroth, and Gerald Weber. CodeGazer: Making Code Navigation Easy and Natural with Gaze Input. In *CHI 2019 - Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, CHI : Conference on Human Factors and Computing Systems, pages 1–12, USA United States, May 2019. Association for Computing Machinery. doi: 10.1145/3290605.3300306.
- [61] Sigasi. MouseFeed — Eclipse Plugins, Bundles and Projects - Eclipse Marketplace, 2020. URL <https://marketplace.eclipse.org/content/mousefeedr>. [Online; accessed 15-October-2020].
- [62] A. Singh, A. Z. Henley, S. D. Fleming, and M. V. Luong. An Empirical Evaluation of Models of Programmer Navigation. In *2016 IEEE International Conference on*

- Software Maintenance and Evolution (ICSME)*, pages 9–19, 2016. doi: 10.1109/ICSM.2016.84.
- [63] Michael W. Smith, Joseph Sharit, and Sara J. Czaja. Aging, Motor Control, and the Performance of Computer Mouse Tasks. *Human Factors*, 41(3):395–396, 1999. doi: 10.1518/001872099779611102. URL <https://doi.org/10.1518/001872099779611102>. PMID: 10665207.
- [64] Will Snipes, Emerson Murphy-Hill, Thomas Fritz, Mohsen Vakilian, Kostadin Damevski, Anil R. Nair, and David Shepherd. Chapter 5 - A Practical Guide to Analyzing IDE Usage Data. In Christian Bird, Tim Menzies, and Thomas Zimmermann, editors, *The Art and Science of Analyzing Software Data*, pages 85–138. Morgan Kaufmann, Boston, 2015. ISBN 978-0-12-411519-4. doi: <https://doi.org/10.1016/B978-0-12-411519-4.00005-7>. URL <http://www.sciencedirect.com/science/article/pii/B9780124115194000057>.
- [65] Centraal Bureau Statistiek. Ruim 6 op de 10 mensen dragen een bril of contactlenzen, 2013. URL <https://www.cbs.nl/nl-nl/nieuws/2013/38/ruim-6-op-de-10-mensen-dragen-een-bril-of-contactlenzen>. [Online; accessed 16-October-2020].
- [66] Yusuke Sugano, Yasuyuki Matsushita, Yoichi Sato, and Hideki Koike. An Incremental Learning Method for Unconstrained Gaze Estimation. volume 5304, pages 656–667, October 2008. doi: 10.1007/978-3-540-88690-7_49.
- [67] Jerry Chih-Yuan Sun and Kelly Yi-Chuan Hsu. A smart eye-tracking feedback scaffolding approach to improving students’ learning self-efficacy and performance in a C programming course. *Computers in Human Behavior*, 95:66–72, 2019. ISSN 0747-5632. doi: <https://doi.org/10.1016/j.chb.2019.01.036>. URL <http://www.sciencedirect.com/science/article/pii/S0747563219300457>.
- [68] Tobii. Tobii Accuracy and Precision Test Method for Remote Eye Trackers. pages 9–10, 2012. URL <https://www.tobiipro.com/siteassets/tobii-pro/learn-and-support/use/what-affects-the-performance-of-an-eye-tracker/tobii-test-specifications-accuracy-and-precision-test-method.pdf/>. [Online; accessed 14-October-2020].
- [69] Tobii. Tobii Pro X2 eye tracker, 2020. URL <https://www.tobiipro.com/product-listing/tobii-pro-x2-30/>. [Online; accessed 25-Februari-2020].
- [70] V. Vipindeep and P. Jalote. List of Common Bugs and Programming Practices to avoid them. page 14, April 2005. URL <https://www.cse.iitk.ac.in/users/jalote/papers/CommonBugs.pdf>. [Online;].
- [71] Neff Walker, David Philbin, and Arthur Fisk. Age-Related Differences in Movement Control: Adjusting Submovement Structure To Optimize Performance. *The journals of gerontology. Series B, Psychological sciences and social sciences*, 52:47, February 1997. doi: 10.1093/geronb/52B.1.P40.

- [72] 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, ICPC 2014*, pages 201–204, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450328791. doi: 10.1145/2597008.2597795. URL <https://doi.org/10.1145/2597008.2597795>.

Appendix A

Glossary

In this appendix, we give an overview of frequently used terms and abbreviations.

AOI: Area of Interest is a designated area that is used to link eye-movements to a particular part of an object.

Debugger: A specialized software development tool that aids a software developer in debugging a software application.

Debugging: The process of finding and resolving issues within software applications.

Dwell-base: An interaction mechanism that uses a specific fixation time to activate an action.

Editor: A software application that is used to write and maintain source code, data, or text.

EyeIDEA Action: The execution of one or more commands by using optional input.

EyeIDEA Component: A representation of a specific Window in the IDE.

EyeIDEA Mode: The current interaction state of EyeIDEA.

EyeIDEA Part: A representation of a visual object on the screen.

Eye-tracker: A device that measures eye movements.

Eye-tracking: The process of measuring eye movements to determine where a person is looking at.

Fixation: A location in which the eyes stopped moving.

Gaze: The location where someone is looking at.

GUI: Graphical User Interface that displays content on the screen and allows interacting with the software application.

A. GLOSSARY

IDE: Integrated Development Environment is an application that provides a comprehensive set of software development tools that are integrated into the program.

Saccade: A small and rapid eye movement to shift attention between different points.

Tab: A Graphical User Interface component that allows multiple documents to be embedded in a single window and allows a user to switch between documents quickly.

Toolbar: A Graphical User Interface component which can contain input and output components, icons, menus, and text.

Window: A Graphical User Interface component that provides a container to present the main content.

Appendix B

Interview Questionnaires

In this appendix, we provide the material that was used during the interview.

B.1 Demographic Questionnaire



EyeIDEA Questionnaire
User nr:

Please fill in the following information

How old are you? _____

Gender? _____

What is your profession or what do you study? _____

Do you wear glasses or contact lenses? Yes No

Are you color blind? Yes No

► Which blindness? _____

Have you used an eye tracker before prior to this study? Yes No

Have you used IntelliJ IDEA before prior to this study? Yes No

B.2 SUS Questionnaire



EyeIDEA Interview Questions
User nr:

System Usability Scale

agree completely →
← strongly disagree

1. I think that I would like to use EyeIDEA frequently	<input type="checkbox"/> 1	<input type="checkbox"/> 2	<input type="checkbox"/> 3	<input type="checkbox"/> 4	<input type="checkbox"/> 5
2. I found EyeIDEA unnecessarily complex	<input type="checkbox"/> 1	<input type="checkbox"/> 2	<input type="checkbox"/> 3	<input type="checkbox"/> 4	<input type="checkbox"/> 5
3. I thought EyeIDEA was easy to use	<input type="checkbox"/> 1	<input type="checkbox"/> 2	<input type="checkbox"/> 3	<input type="checkbox"/> 4	<input type="checkbox"/> 5
4. I think that I would need the support of a technical person to be able to use EyeIDEA	<input type="checkbox"/> 1	<input type="checkbox"/> 2	<input type="checkbox"/> 3	<input type="checkbox"/> 4	<input type="checkbox"/> 5
5. I found the various functions in EyeIDEA were well integrated..	<input type="checkbox"/> 1	<input type="checkbox"/> 2	<input type="checkbox"/> 3	<input type="checkbox"/> 4	<input type="checkbox"/> 5
6. I thought there was too much inconsistency in EyeIDEA	<input type="checkbox"/> 1	<input type="checkbox"/> 2	<input type="checkbox"/> 3	<input type="checkbox"/> 4	<input type="checkbox"/> 5
7. I would imagine that most people would learn to use EyeIDEA very quickly	<input type="checkbox"/> 1	<input type="checkbox"/> 2	<input type="checkbox"/> 3	<input type="checkbox"/> 4	<input type="checkbox"/> 5
8. I found EyeIDEA very cumbersome to use	<input type="checkbox"/> 1	<input type="checkbox"/> 2	<input type="checkbox"/> 3	<input type="checkbox"/> 4	<input type="checkbox"/> 5
9. I felt very confident using EyeIDEA	<input type="checkbox"/> 1	<input type="checkbox"/> 2	<input type="checkbox"/> 3	<input type="checkbox"/> 4	<input type="checkbox"/> 5
10. I needed to learn a lot of things before I could get going with EyeIDEA	<input type="checkbox"/> 1	<input type="checkbox"/> 2	<input type="checkbox"/> 3	<input type="checkbox"/> 4	<input type="checkbox"/> 5

B.3 Interview Questionnaire

1 EyeIDEA Overall experience

The following questions are to evaluate your experience of using EyeIDEA.

1. In a couple of words, how would you sum up your experience with EyeIDEA? (RQ2)
2. What did you like in general of an IDE that uses eye-tracking? *Please describe it concretely by using examples during the task. For example, "It helped me to check the value of variable X. So, I did Y and Z and then I saw the value"* (RQ2)
3. What did you not like in general of an IDE that uses eye-tracking? (RQ2)
4. Would you use EyeIDEA for professional use? (RQ3.3) Yes No Don't know

► Why?

2 EyeIDEA Interaction System

The following questions are to evaluate the interaction system of EyeIDEA.

5. Which action did you found the easiest to learn how to execute it? (RQ2)

► Why?

6. Which action did you found the most difficult to learn how to execute it? (RQ2)

► Why?

7. Would you change something about the interaction system? (RQ3.1)

3 EyeIDEA navigating effectiveness

The following questions are to evaluate the effectiveness of navigating with EyeIDEA.

8. What do you think about the navigation features? *Please describe it concretely by using an example.* (RQ2)

9. Are there other navigation features that you missed? (RQ3.1)

4 EyeIDEA debugging effectiveness

The following questions are to evaluate the effectiveness of debugging with EyeIDEA.

10. What was your favorite and least favorite debugging action? You can choose from: button interaction, placing breakpoints, placing conditional breakpoints, using quick evaluator, using window evaluator) (RQ2)

Most:

Least:

► Why? *Please describe it concretely by using an example.*

11. Did you notice any differences in your usage of the debugging features when you could use your eyes instead of the mouse and keyboard? (RQ3.2)

5 General IDE Eye tracking evaluation

The following questions are to evaluate what kind of impact the plugin had on using eye tracking in general for an IDE.

12. In your opinion, which audience would benefit the most with an eye tracking augmented IDE? (RQ3.3)

13. Do you think it is better to use an eye-tracker as standalone just like the gazebutton or is it better to mix it with mouse and keyboard just like you saw with the evaluator? (RQ3.2)

14. Could eye tracking replace existing IDE actions that requires a mouse or keyboard at the moment? How do you envision this? (RQ3.2)

6 Eye Tracking experience

The following question is to evaluate the eye tracking itself.

15. Did you feel getting fatigued using the eye-tracker? (RQ3.1)

7 Tasks Evaluation

The following questions are to evaluate the quality of the tasks. (They dont have any connection with the research questions.)

How would you rate the difficulty of debugging your first task? Trivial Easy Neutral Difficult Imposs.

How would you rate the difficulty of debugging your second task? . Trivial Easy Neutral Difficult Imposs.

16. Did you have any issues with understanding the code from your first task?

17. Did you have any issues with understanding the code from your second task?

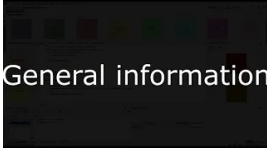

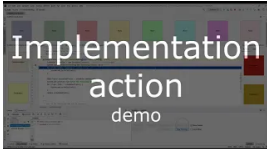
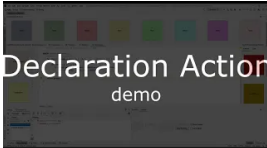
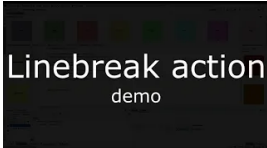
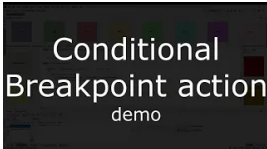
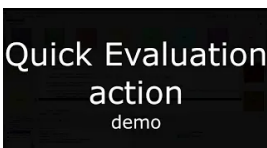
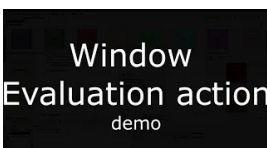
8 Closing questions

The participant gets some room here to comment on other aspects there were not asked.

18. Do you have any other remarks about EyeIDEA that you would like to share?

19. Do you have any other remarks about this research that you would like to share?

B.4 Training Videos

Thumbnail	Description	Link
	This training video shows the concepts behind using EyeIDEA.	https://www.youtube.com/watch?v=VB2W4-3f8Jo
	This training video shows how to select and activate buttons.	https://www.youtube.com/watch?v=C46rQdwGy9E
	This training video shows how to use the “Go To Implementation” navigation action.	https://www.youtube.com/watch?v=0PL05VECOYo
	This training video shows how to use the “Go To Declaration” navigation action.	https://www.youtube.com/watch?v=xvNHNyT5FcA
	This training video show how to use the line breakpoint action to set a breakpoint.	https://www.youtube.com/watch?v=7vV8xTQzADk
	This training video shows how to use the conditional breakpoint mode to set breakpoints that only activate under a certain condition.	https://www.youtube.com/watch?v=eGkMyYC_kYA
	This training video shows how to activate and use the quick evaluation action..	https://www.youtube.com/watch?v=X6hixqdiQQg
	This training video shows how to use the window evaluation action which activates the Evaluator Tool.	https://www.youtube.com/watch?v=p16saDxt6Cg

Appendix C

Overview Card EyeIDEA

C. OVERVIEW CARD EYEIDEA



1 UI Overview



Panel 1 is **Gaze action button panel**, it is composed out of **action buttons**.
 Panel 2 is **Navigation Modes panel**, it is composed out of **navigation mode buttons**.
 Panel 3 is **Debugging Modes button panel**, it is composed out of **debug mode buttons**.
 Panel 4 is **Control panel**, it controls the eye tracker.

2 Gaze Button



To **activate a button**, look at an inactivate button. This will change into the hover state. If the gaze is over the button for long enough, the button will be activated. **The deactivation of a button** follows the same steps.

3 Actions

3.1 Highlights



Highlights are used to show which items are currently linked to the gaze action buttons. Only a maximum of 8 items can be linked at any time. If there are more, the oldest highlighted item will be replaced.

3.2 Executing an action

To be able to perform an action, the following steps need to be performed:

- Step one**, select the desired mode.
- Step two**, look at an item to highlight it.
- Step three**, activate the action button that has the same color as of the highlighted item.

4 Modes

A Mode is responsible for linking the appropriate item on the screen with a specific action. These actions are set to the action buttons. This link between the item and the button is made clear by using highlights.

4.1 Selecting a mode

To switch between the different modes, gaze buttons are used. Each group is located on its own panel.

The button mode does not have a dedicated button, so it is not activated by looking at a specific button. Instead, looking at the current activated gaze mode button will deactivate that mode and the plugin will return to the button mode.

If the gaze mode button was just activated, the mode will not be deactivated. Instead, look at the editor for just a moment and then look at the button to deactivate it.

4.2 Different modes

IDE control mode. Links regular IDE buttons and lists to the gaze action buttons. These ui elements are normally activated by using the mouse or keyboard. The mode is active when no other modes are active.

4.2.1 Navigation modes

Navigation modes links the source code elements with a specific navigation action. To trigger these modes, look at the corresponding gaze button found in navigation mode panel.

Implementation mode. Links a source code element to the implementation action. Code elements are highlighted and the implementation action can be triggered with the action buttons.

Declaration mode. Links a source code element to the declaration action. Code elements are highlighted and the declaration action can be triggered with the action buttons.

4.2.2 Debug modes

Debug modes links the source code elements with a specific debug action. Some of the modes uses the location of the code elements to perform the action.

Selecting the specific mode requires 2 steps.
 1. activate the gaze mode button that corresponds with the group. A popup will show up in the middle of the screen. This popup has multiple gaze mode buttons.
 2. Select the specific submode by activating its gaze button.



Breakpoint modes

Line breakpoint mode. Links the location of a source code element to set a new breakpoint or remove an existing one. To set or remove an existing breakpoint, use the steps from 3.2

Evaluation modes

Quick Evaluation mode. Links a source code element to be evaluated and shows the result in the editor.



Code tab

Conditional breakpoint mode. Uses one or multiple source code elements to create a condition for a breakpoint. To select the elements, use 3.2. These are added to the text area in the code tab.
 Create a valid condition of the added elements and then activate the gaze button below the text area. To set the location of the breakpoint, it uses the same as of **Line breakpoint mode**.

Evaluation Window mode. Uses one or multiple source code elements to be evaluated and shows this in a separate evaluate window. To select the elements, use 3.2. These are added to the text area in the code tab.
 To activate the evaluation popup, activate the button right below the text area. This will create the Evaluation Window and the code from the text area will be put into this popup.

5 Additional Features

5.1 Show Pointer

To show the gaze on the screen, simply select the SShow Pointer"box. This will display a bubble on the screen that expands and retracts depending on the gaze.

5.2 Click offset

Recenter the gaze based on the clicked position with the left mouse button. When the option is turned of, the correction will be removed and the original gaze position from the eye tracker will be used again.

Appendix D

Invitation Poster for the User study

This appendix includes the poster that we distributed to invite students to join the user study.



The poster features a blue background with a white and cyan diagonal design. At the top, there are two large, stylized eyes. The main title is 'Volunteers needed to test an eye-tracking IDE.' Below this, it says 'Participate in the evaluation of an interactive eye-tracking IDE for debugging'. The details listed are: 'Who: TU Delft students', 'When: December 10-18 (weekdays)', 'Where: Mathematics & Computer Science (TU Delft Building 28)', and 'Compensation: €10,- giftcard'. A QR code is shown with a black arrow pointing to it and the text 'Sign up' written above the arrow. Below the QR code is a cyan button with the URL 'HTTPS://FORMS.GLE/7SS6EDJVXZ0TRD4X6'. At the bottom, a dark blue banner contains the text 'All Covid-19 guidelines will be respected'.

Volunteers needed to test an eye-tracking IDE.

Participate in the evaluation of an interactive eye-tracking IDE for debugging

Who: TU Delft students
When: December 10-18 (weekdays)
Where: Mathematics & Computer Science (TU Delft Building 28)
Compensation: €10,- giftcard

[Sign up](https://forms.gle/7SS6EDJVXZ0TRD4X6)

[HTTPS://FORMS.GLE/7SS6EDJVXZ0TRD4X6](https://forms.gle/7SS6EDJVXZ0TRD4X6)

All Covid-19 guidelines will be respected

Appendix E

Survey invitation form

Registration for participating in User study about an interactive eyetracking IDE

Hi, My name is Arjan and I developed a plugin to interact with IntelliJ IDEA using an eyetracker as part of my Master Thesis and it needs to be evaluated. During this study, you will debug some Java code with and without using the eyetracker. However, dont worry about your debugging skills, the goal is to test the eyetracking interactions and not your debugging performance.

The study will consist out of a training, interacting with the IDE and an interview afterwards so you can give me feedback. It will take around 45-60 minutes.

For your efforts, I will give you a €10,- giftcard of your choice if you participate.

A timeslot based on your availability will be emailed to you along with some details about the procedure.

All Covid-19 measures currently in place will be respected and your data will be collected anonymously.

Note, only TU Delft students can participate due to Covid-19 measures.

Dates: 7 December- 18 December
Location: Mathematics & Computer Science (building 28)
Van Mourik Broekmanweg 6
2628 XE Delft
Contact: a.c.langerak@student.tudelft.nl

*Required

Name *

Your answer _____

Email (will only be used to send you a timeslot and more information) *

Your answer _____

Giftcard (select an option if you want the giftcard)

Choose ▾

Please indicate your preferred availability (You can select multiple options)

9.30 - 10.30 11.00 - 12.00 13.30- 14.30 15.00 - 16.00 16.30 - 17.30