Master's Thesis

AN EYE-TRACKING STUDY ON THE EFFECTS OF TYPE ANNOTATIONS AND IDENTIFIER NAMING ON PROGRAM COMPREHENSION

NILS ALZNAUER

20.05.2024

Advisors:

Dr. Norman Peitek	Chair of Software Engineering
Dr. Marvin Wyrich	Chair of Software Engineering
Annabelle Bergum	Chair of Software Engineering

Examiners:

Prof. Dr. Sven Apel	Chair of Software Engineering
Prof. Dr. Antonio Krüger	Professor for Computer Science CEO & Scientific Director DFKI

Chair of Software Engineering
Saarland Informatics Campus
Saarland University

UNIVERSITÄT DES SAARLANDES

Nils Alznauer: *An Eye-Tracking Study on the Effects of Type Annotations and Identifier Naming on Program Comprehension,* © May 2024

Erklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken,_____ _____

(Datum/Date) (Unterschrift/Signature)

ABSTRACT

Software engineering is an essential discipline in today's world. At the core of applied software engineering is the developer's ability to read, understand, and modify the code. These tasks are collectively called program comprehension and take up a significant portion of a developer's time.

Program comprehension is very complex and involves a variety of cognitive processes. It highly depends on the programming language and how the code is written. One aspect of program comprehension investigated in research is the impact of type annotations on program comprehension. While some effects of type annotations on program comprehension have been identified, the results are still inconclusive.

We aim to investigate the relationship between type annotations and measures of program comprehension. To assist with this investigation, we consider the impact of the combination of type annotations and identifier names on program comprehension.

To achieve this, we conducted a controlled experiment with 31 participants using eye tracking. Participants were asked to comprehend 20 Python source code snippets, and their performance was evaluated using commonly employed eye-tracking and behavioral measures.

We found that the combination of type annotations and identifier names has a significant impact on some line-based visual attention measures but not on behavioral measures or the reported subjective difficulty. Type annotations on their own do not carry a significant impact on any of the measures, but nearly all participants report that type annotations helped with program comprehension.

The identified effects of type annotations and the combination of type annotations and identifier names on visual attention measures can be used to improve the design of programming languages and the usage of type-based tools to support developers in their daily work.

ACKNOWLEDGMENTS

First and foremost, I extend my gratitude to all the people who enriched my life at the university and accompanied me for the last years on this journey and the successful completion of this thesis.

I want to express my most profound appreciation to Dr. Norman Peitek for his guidance, support, and encouragement throughout the whole journey of this thesis. His expertise, patience, and motivation have been invaluable to me and greatly enhanced the quality of this work. Furthermore, I appreciate his willingness to provide constructive feedback and help when needed.

Furthermore, I want to thank Dr. Marvin Wyrich for his support and guidance during the study. His insights and critical eye for statistics helped me improve the study's quality and results.

My gratitude extends to Annabelle Bergum, with whom I started this thesis, who helped me get started with the study and the setup.

I am immensely grateful to Prof. Dr. Sven Apel for his confidence in my potential and for giving me the opportunity to work on this thesis.

I want to extend my sincere thanks to Prof. Dr. Antonio Krüger for reviewing this thesis. I am honored and truly appreciate the time and effort he dedicates to this task.

I am also thankful to everyone at the Software Engineering chair for always lending a helping hand and an open ear when needed. Their assistance and support have been invaluable to this study.

Special thanks to Friederike Repplinger for her help with administrative hurdles and regulations.

I also appreciate everyone who participated in my study and made this research possible.

A special thanks goes to my two proofreaders for giving constructive feedback and improving the quality of this thesis.

Thank you to my parents, who put up with me for the last weeks before the deadline and always supported my decisions.

I want to give a special thanks to my partner, who supported me throughout the whole journey and always believed in me.

Last, I thank my family and friends for their unwavering support, encouragement, and love throughout this journey. Their belief in me and encouragement have been a constant source of strength and motivation.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

OSS Open-Source Software

ERB Ethical Review Board

FMRI Functional Magnetic Resonance Imaging

EEG Electroencephalography

IDE Integrated Development Environment

1 INTRODUCTION

Software is one of the cornerstones of the world. Nearly everyone relies on software, and in many cases, lives or huge investments depend on its flawless running. While it enables better connectivity and global progress, it can lead to monetary losses and even loss of human life. One example is a software error that exposed patients to a deadly amount of radiation, killing five people [18]. Another software error led to the crashes of two Boeing 737 MAX airplanes, resulting in the loss of 346 lives [17]. Because these incidents are not singular but only some examples of long-standing software errors with huge impacts, it is imperative to have reliable, correct, and secure software.

Assuring correctness, reliability, and security gets increasingly difficult as the development continues. This primarily occurs in big codebases that include more than 100 million lines of code[1]. Thus, developers must not only be able to write code and add it to a codebase, but they also need to be able to *comprehend* what they are doing, which takes up about 58% of the time during development [37]. Despite program comprehension and the underlying cognitive processes being a big part of development, they need to be better understood [32].

Even though research into program comprehension has existed for multiple decades, it still needs to be clearly understood which cognitive processes are involved in the comprehension part of development. This lack of understanding is especially prevalent in which code parts significantly influence program comprehension. One such concept that can influence program comprehension is type annotations. Conventional measures, like response time and response correctness, can sometimes identify the influence type annotations have on program comprehension. However, they can not explain why differences in these measures can be observed. Another concept that can influence program comprehension is meaningful identifier names. For this concept, research has already suggested that meaningful identifier names positively influence program comprehension. What remains unknown is that meaningful identifier names and obfuscated identifier names influence type annotations. To explain the effect of both concepts, measures for visual attention and reading patterns can be used in addition to conventional measures.

The measure of choice for visual attention and reading patterns is an eye tracker because it is an inexpensive and fast solution to measure how developers approach the actual comprehension of a given task. Thus, we can analyze the order in which developers looked at the code and what viewing strategies they used.

In this thesis, we propose to explore how the existence or absence of type annotations influences developers' comprehension of the program. To that end, we developed and ran a

1 `https://assets.kpmg.com/content/dam/kpmg/jm/pdf/protecting-the-fleet-webfile.pdf`, visited 06 June 2023

study that uses eye tracking to measure the cognitive load of participants and their strategies to comprehend the given tasks.

Thus, we investigate the following research questions:

1. How do type annotations influence program comprehension?
2. How does the combination of type annotations and identifier names influence program comprehension?

OUTLINE

Chapter 2, *Background*, explains the background information for the program comprehension, type annotations, and eye tracking. After that, in Chapter 3, *Related Work*, we provide an overview of the current research into program comprehension and the research into type annotations and identifier names. Then, we present the study itself in Chapter 4, *Study Design*. In Chapter 5, *Results*, we evaluate the study and present all results. After that, we discuss the results and threats to its validity in Chapter 6 *Discussion*. Finally, we conclude the thesis in Chapter 7 with a conclusion and an outlook for future work.

2 BACKGROUND

This chapter explains the most essential concepts and terms used in this thesis. We start by explaining the concept of program comprehension and the strategies used by developers to comprehend code. Then, we discuss the concept of type annotations and how they are used explicitly in Python. Finally, we explain the concept of eye tracking in program comprehension research and the different measures that can be derived from it.

2.1 PROGRAM COMPREHENSION

Program comprehension is an area in software engineering, especially in software maintenance and development, which is concerned with the cognitive process of how developers understand (comprehend) source code. It is a vital part of any software development process because this program comprehension is needed to continue working on the source code, and it takes most of the time a developer spends on code [37]. This work on the source code can be anything from e. g., adding new functions or features to testing and fixing bugs. Since this has been well-known for years, there has been some research into this topic.

However, even though there has been much progress, the core problem is still that it is possible to measure if the developer is comprehending the source code objectively. However, finding out *why* the source code is comprehended in a certain way is challenging. This is due to both the complexity of the cognitive processes and the fact that it is challenging and expensive to measure them. To better understand *why* developers comprehend source code in a certain way, we need to use qualitative measures.

One of the methods is the use of Functional Magnetic Resonance Imaging of Electroencephalography to measure the cognitive load. However, both these methods are either very expensive or not practical in everyday use. The setup and the time needed to prepare and execute the study are especially high and locally bound.

One way to circumvent or at least weaken these problems is by using eye tracking. Andaloussi et al. [1] have found that eye tracking can be used even at a fine-grained level or measurement to understand the cognitive processes of developers better.

2.1.1 *Program Comprehension Strategies*

According to Peitek [20] based on the work of Exton [14], "[program comprehension] describes the transition in which a programmer understands an existing implementation formed in source code and constructs an analogous mental model". This means that program comprehension is the process used by developers to understand the source code and retain it in their memory. There are different strategies for developers to comprehend the source code. The two main strategies are *bottom-up comprehension* and *top-down comprehension*.

BOTTOM-UP COMPREHENSION Bottom-up comprehension is the most fundamental program comprehension strategy [25]. In this strategy, developers read through the code line by line, extracting information for each statement and grouping it into a larger model. This process needs to be repeated until every line is comprehended and the whole mental model is available. Because this strategy requires every line to be read and understood, it is taxing on the developer, takes much time, and is tedious work. Combining this information shows that this strategy is not very efficient and requires much cognitive load for the developers [33].

TOP-DOWN COMPREHENSION Top-down comprehension is in comparison to bottom-up comprehension the faster and more efficient process [8]. In this strategy, developers can use their domain knowledge as well as their previous contextual knowledge to comprehend the content faster and guess the source code snippet intent [33]. Because developers can try to guess the intent of the source code, they can skip parts of the code that are irrelevant and look for parts that would either validate or invalidate their hypothesis of what the code does. This strategy is thus more efficient and requires less cognitive load than bottom-up comprehension.

In this thesis, we use both strategies to analyze the source code snippets and the type annotations in the source code snippets.

2.2 TYPE ANNOTATIONS

The type of a variable is a crucial part of any programming language [7, 15]. It provides the developer and the compiler with a way to understand what kind of object they are dealing with. This also puts constraints on the variable and what functions can be used by it or on it. This provides developers with the ability to more easily understand code and make less mistakes when writing code [15]. In general, two kinds of type systems are differentiated: static and dynamic types.

STATIC TYPING Static typing is a type system that checks the type of a variable at compile-time. This means that the variables' type is known before the program is run, and thus, the compiler can check for any discrepancies within the type logic. This is very helpful to not only find logical errors early but also to prevent any type mismatches that could lead to runtime errors. Most programming languages that need to be compiled use static typing, e. g., Java and C++.

DYNAMIC TYPING Dynamic typing is a type system that checks the type of a variable at runtime. This means that the variables' type is only known when the program is run. This can lead to type mismatches and runtime errors that are hard to find and debug. However, dynamic typing is more flexible, easier to write, and faster to write code. Many scripting languages use dynamic typing, e. g., Python and JavaScript.

Both of these type systems have their advantages and disadvantages. That is why Python introduced a way to add type annotations to the code.

TYPE ANNOTATIONS IN PYTHON Type annotations are used to add type information to a variable. They were introduced to Python in version 3.5[1] and can be used to add information while at the same time not changing the advantages of dynamic typing. This means that the type annotations are only used by the developer and not by the compiler. Additionally, developers have written plugins or extensions for the current Integrated Development Environments (IDEs) to quickly check for type mismatches and other type-related errors. The type annotations can not only be used for variables but also for functions and classes.

To use type annotations in Python, the `typing` module needs to be imported. This module provides developers with the ability to the standard types like `int`, `str`, `float`, etc., as well as more complex types like `List`, `Dict`, `Tuple`, `Union`, and others. It also gives a developer the ability to add a `return` type and `input parameter` types to the function definition.

2.3 EYE TRACKING IN PROGRAM COMPREHENSION

Eye tracking is a method to measure a persons visual attention through the observation of their eye movements. It is used in many different research fields and various disciplines like psychology, medicine and health care, neuroscience, mathematics and computer sciene, etc. [12]. Because the visual attention triggers cognitive processes required for comprehension, it is a very useful tool in program comprehension research [1, 29].

A possible visual stimulus are source code snippets or rather any object that is necessary to perform a task and whose perception stimulates a cognitive process. In the case of source code snippets, the source code snippets is the biggest entity to perceive, while variables, type annotations, keywords etc. are some of the smallest.

Because eye tracking is an inexpensive technique for measuring cognitive processes, it is a very useful tool for program comprehension research. It can be used to measure the cognitive load, the cognitive processes, and the strategies used by developers to comprehend the source code [1, 3, 10, 11, 24].

2.3.1 *First Order Data*

First Order Data is data that is collected immediately from the eye tracker itself. In most cases this includes the X and Y coordinate of where the person is looking at, the time the person is looking at a certain point, and the pupil size. This data can be extended by stronger eye trackers that also measure the distance between the eye and the screen, the angle of the eye, and the distance between the two eyes.

According to Beatty [2] and Poole and Ball [26], both pupil size and the number of blinks can be correlated to a higher cognitive load.

2.3.2 *Second Order Data*

Second Order Data is data that can be directly inferred from the *First Order Data*. It consists of both *fixations* and *saccades* which are derived by using certain thresholds.

1 `https://docs.python.org/3/library/typing.html`

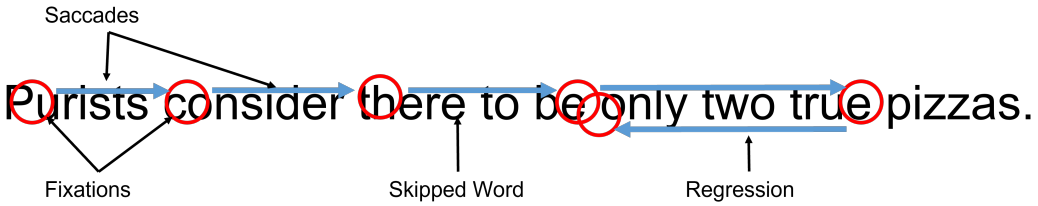


Figure 2.1: Example of saccades and fixations modeled after Carter and Luke [12]

2.3.2.1 *Fixation*

One of the basic building stones is fixation. As any eye moves across a surface it needs to stop the movement on the surface to gain visual information. According to Carter and Luke [12] these fixations are short because the eye can only collect little visual information upon focussing. This leads to frequent small movements and thus the general time of a fixation, even though highly dependent on many factors, is between 180 and 330 ms.

In Figure 2.1, the fixations are marked with red rings and show where exactly the user stopped their eye movements. We can see that there are gaps between the circles meaning that the eye does not linger on every single bit of information but also sees in the peripheral.

2.3.2.2 *Saccades*

Saccades, on the other hand, are the quick movements of the eye between the different fixations. The visual input during this movement is suppressed [12] such that humans are effectively blind. As saccades vary in size and time and we use them to read text, they will last about 30 ms. If a saccade is flowing "backward" within the text, it is called a regression.

In Figure 2.1, we can see the saccades between the single fixations as well as the direction they are traveling. This can be easily inferred from the information about when the different fixations took place.

2.3.3 *Third Order Data*

This data is derived from the *Second Order Data* and consists of counts and durations. This includes the number of fixations and saccades, as well as the fixation duration and the saccade length. In most of these cases, a higher count of fixations and saccades correlates with a higher cognitive load [29]. Additionally, one of the most crucial analyses are done through the use of AOI.

2.3.3.1 *Areas of Interest (AOI)*

AOI is a crucial concept in eye-tracking research. They provide the examiner with the ability to mark specific, but broad enough areas [20], as interesting such that one may analyze whether a participant stopped their eye movement in an AOI. In Figure 2.2, we can see how AOI can be applied. The three boxes of different sizes show us when `arrayAverage([2,4,1,9])` is being looked at.

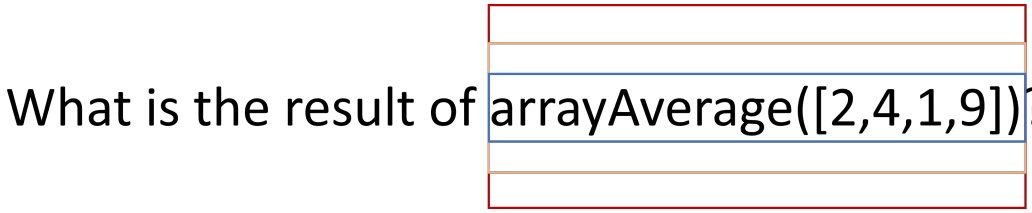


Figure 2.2: Example of areas of interest modeled after Peitek et al. [24]

These AOI can then be used for simple measures like the time to the first fixation within a certain AOI, but also the percentage of fixations within an AOI or the total time spent within an AOI. This can be extended to not only include fixations but saccades as well.

2.3.4 *Fourth Order Data*

This data builds on top of *Third Order Data* and especially uses the measures gained through AOI. As an example, we describe the two measures *Scan Path* and *Transition Matrix*. There is however no limit to the ingenuity of the measures that can be derived from the AOI.

2.3.4.1 *Scan Path*

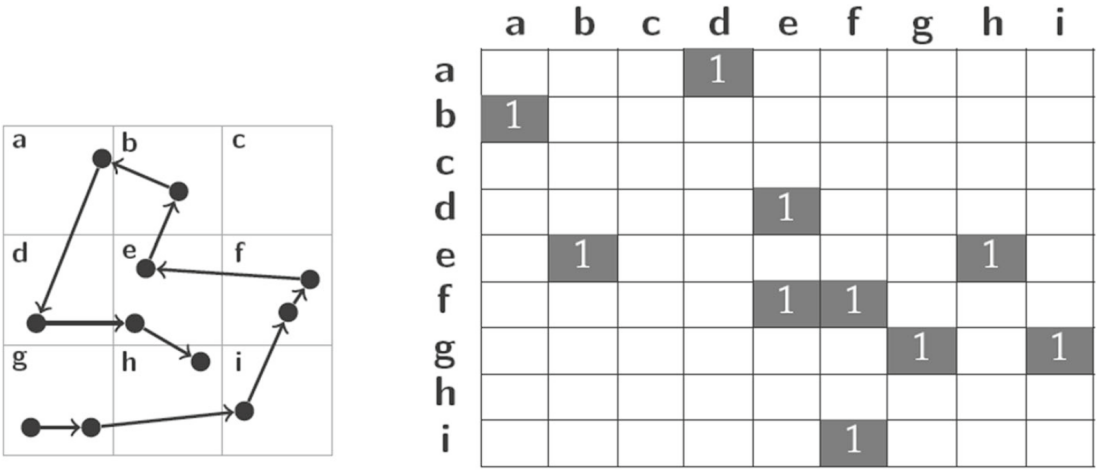


Figure 2.3: Example of scan path and corresponding transition matrix from Smet et al. [35]

A *scan path* is a sequence of fixations or AOIs. We can chain them together and therefore describe the duration that an eye gazed at a certain space on the screen. According to Sharafi et al. [29], they are also an indicator of search efficiency since overall longer-lasting *scan paths* indicate that the user had to take more time comprehending similarly-sized programs. Thus they are an indicator of inefficient searching or scanning of the source code. In Figure 2.1 we can easily see such a *scan path* if we connect all the fixations through the saccades.

2.3.4.2 *Transition Matrix*

If the *scan path* is set up and found, we can also easily compute a *transition matrix* that consists of a two-dimensional matrix that is empty except for the case that an edge, in our case saccades, between the different fixations or AOIs, exists. The *transition matrix*, therefore, shows us all the directed edges with start and end points (Figure 2.3). Through this matrix, we can easily see if there are certain AOIs that have many connections. This would indicate that these AOI are in correlation with each other.

3

RELATED WORK

After the background, we take a closer look at the research in this area. First, we review research on program comprehension and then how the different independent variables have been analyzed in the past.

3.1 PROGRAM COMPREHENSION

Program comprehension is an area in software engineering in which research has increased in the last few years as the methods for measuring the cognitive load have improved or become cheaper.

One of the methods is the use of Functional Magnetic Resonance Imaging (fMRI) or Electroencephalography (EEG) to measure cognitive load. With this method much information about cognitive processes is gained and the brain of the participant is studied on a fine-grained level.
Peitek et al. [24] first studied the question of whether it is feasible for researchers to use a combination of fMRI and eye tracking. They find that eye tracking is a possible way to match different brain activation sequences with certain areas of code even though it has drawbacks due to the usage of an fMRI machine. This shows how eye tracking can be used for program comprehension research.

In their paper on the reading order of programmers Peitek et al. [23] find that there is a significant effect between the linearity of source code and how programmers read a program. If the source code is less linear they find that there are large unnecessary vertical eye movements that slow programmers down.

In another paper, Peitek et al. [22], look at the connection between efficacy and programming experience with the help of an eye tracker and EEG. Here they use the eye-tracking data because it was shown before that the reading behavior of the groups with differing programming experiences differ.

These studies can be very exact when looking for the correct brain area and also when which brain area is activated, however, they are not always feasible nor easy to run. Additionally, these methods are not always the best, since only so much information can be collected when strong magnets are in a room and no magnetic metal can be within the room without getting destroyed. Furthermore, we can not get a definitive answer to many questions without looking at them from multiple angles. One other angle from which to approach is eye tracking. It is a very cheap method to measure cognitive load in a participant and is quickly set up. It also creates new data points and gives an overview of how programmers read through a program and which parts of the program are actually relevant for the actual comprehension.

Eye tracking has over the years become a reliable, fast, and cheap version for researchers to study the reaction of users to different source codes. One of the first in the field were Bednarik and Tukiainen [3] that found that much information can be gained from the usage

of eye-tracking software when looking at the behavior of programmers even though the information might otherwise be hard to access.

Andaloussi et al. [1] use eye tracking to improve the estimation of a developer's cognitive load through machine learning. They use different eye-tracking measures to train machine learning models which should then better estimate which parts of the code lead to a higher cognitive load. The authors find that their pupil-based and saccade-based measures show the highest importance scores in the ML models.

As we are using eye tracking and other measures to find data, we need to take a look at the research already done in the scope of the research questions.

3.2 TYPE ANNOTATIONS

In the scope of type annotations, Hanenberg et al. [15] have checked whether the use of a static-type system is helping programmers to comprehend the program better. They did this by checking for bug fixes. A rather obvious improvement is that it is easier for programmers to fix type errors. However, they could not find any statistically significant evidence that semantic errors were easier to fix when static-type systems were available.

Additionally, Bogner and Merkel [7] found in their comparison of JavaScript and TypeScript on Open-Source Software (OSS) systems that while measures show there is better code quality to be found in TypeScript, there is no clear improvement on the accounts of bug proneness and bug resolution. This strongly suggests that it is not yet clear whether type annotations help with program comprehension.

To better understand whether experimenters and their participants are biased towards the usage of static-type annotations in their experiment, Okon and Hanenberg [19] created tasks that were specifically designed to show that dynamic-type annotations are beneficial. In their experiment, they found that even though these tasks were specifically created to introduce bias, half of the tasks showed that static-type annotations were still beneficial. This clearly shows that the impact of type annotations should be visible even to the eye tracker.

For better understanding in combination with identifier names, Spiza and Hanenberg [36] checked whether type names already improve the usability of APIs even if no static-type checking is applied. This was overall the case, however, contrary to their expectation, they also found that there was a statistically significant negative impact when the type names were incorrect. This suggests that, on the one hand, type annotations seem to play a role in understanding and additionally that false contextual information can lead developers astray.

3.3 IDENTIFIER NAMES

On the question of the different identifiers, Casalnuovo et al. [13] find evidence that programmers prefer it when the code includes more predictable and thus less surprising variants of identifier names. This indicates that not only do programmers find sensible and predictable identifier names better than others but it hints that programmers at least in their mind have a better program comprehension when sensible identifier names are found.

Furthermore, Hofmeister et al. [16] find that code is comprehended 19% faster when whole words are used as identifier names rather than letters or abbreviations. This was found in an effort to find whether the length or the semantics of an identifier name were more relevant to program comprehension. This suggests that in order to create good and meaningful identifier names, we need to use whole words. It also suggests that there is better program comprehension when these identifier names are meaningful and thus they reduce the cognitive load on the participant. In their study, they used code snippets that included static types. We will thus further check if this effect can still be seen when the type annotations are missing.

On the other hand, Scanniello et al. [27] find that in the realm of bug fixing this effect of increased program comprehension does not seem to be available. This can on the one hand be interpreted as contrary to [16], however, it can also mean that the kind of task provided to the participants is very important to the measurement of program comprehension.

Additionally, Schankin et al. [28] did a study on the question of whether longer but more descriptive identifier names helped with program comprehension. They found that semantic defects were found about 14% faster with descriptive names. These effects however disappeared when the participants looked for syntax errors suggesting that this was especially so when no program comprehension was required. Schankin et al. also find that the effect in favor of longer and more descriptive identifier names is more pronounced with participants that have more programming experience, which was measured through the use of a subjective question.

Overall, the research on identifier names is not yet conclusive. Some studies report on the positive effects of naming schemes for identifier names and how predictable and meaningful names can help with program comprehension. However, other studies, especially those that look at bug fixing, do not find a clear effect. This suggests that the effect of identifier names might be dependent on the task at hand and the experience of the participants.

4 STUDY DESIGN

This chapter describes the study design of the controlled experiment conducted to investigate the effect of type annotations and the combination of type annotations and identifier names on program comprehension. To get precise and objective measurements a two-by-two factorial design was chosen to compare all groups. First, we explain the research questions and how they are operationalized. Then, we describe the independent and dependent variables, the experimental design, the participants, the materials, and the procedure. Finally, we discuss the deviations that occured during the study.

4.1 RESEARCH QUESTIONS

In this study, we investigate the influence of type annotations on program comprehension. We are interested in the behavior, the linearity of reading order, and the subjective code difficulty of participants when they take in source code snippets with and without type annotations. We want to find out if we can find clear differences between the two cases and then if these differences are significant. Thus, we pose the first research question:

> RQ 1: How do type annotations influence program comprehension?

Since type annotations may not be as relevant to program comprehension in a well-documented and well-structured codebase, we simulate a codebase that forces the participant to use bottom-up comprehension instead of using their context knowledge. This is done by changing the type annotations and also changing the identifier names to be either meaningful or obfuscated. This gives the second research question:

> RQ 2: How does the combination of type annotations and identifier names influence program comprehension?

Since program comprehension is a very broad term, the research questions need to be properly operationalized. To do this, we add multiple sub-questions to each research question. Since both research questions are similar and only differ in the independent variable or the combination thereof, we use the same sub-questions for both research questions.

4.1.1 *Operationalization of Research Questions*

The following sub-questions are used to operationalize the research questions. They always start with the same phrase and then continue with the specific question replacing the program comprehension part in the research question. The sub-questions are then used to create hypotheses that can be tested in the study.

Table 4.1: Null hypotheses with corresponding p-values for behavioral measures

	Hypothesis	p-value
H_1	The response time is not influenced by the presence of type annotations.	≤ 0.05
H_2	The correctness is not influenced by the presence of type annotations.	≤ 0.05
H_3	The response time is not influenced by the presence of type annotations and identifier names.	≤ 0.05
H_4	The correctness is not influenced by the presence of type annotations and identifier names.	≤ 0.05

... INFLUENCE BEHAVIORAL MEASURES? This research question aims to find out how the response time and the correctness are influenced. The response time is a good start in measuring, how difficult the source code snippets are and how much time it takes the participants to understand the source code snippets. Additionally, the correctness is very important for the behavioral measures, since it shows if the participant understood the source code snippets and was able to answer a question about it. These measures are operationalized by having participants look at the source code snippets and then answer a question afterwards, while at the same time measuring response time and correctness.

... INFLUENCE THE LINEARITY OF READING ORDER? This reasearch question aims to investigate, if the participants read the source code snippets linearly in execution order or if their gaze is jumping around within the source code snippets. A more linear reading order suggests that participants were able to better comprehend the source code snippets, because they did not have to backtrack as much as others. If they then also answered the question correctly, it would suggest that this participant had a better comprehension of the source code snippets. This can especially be influenced by adding type annotations to the source code snippets, since this creates more information in each line which might lead to less backtracking. This measure is operationalized by measuring the scan path of the participants as well as some measures of backtracking.

... INFLUENCE THE SUBJECTIVE CODE DIFFICULTY? This research question aims to find out how participants subjectively rate the source code snippets. This provides us with a good measure of how difficult the source code snippets were individually and for the participants themselves. It also provides us with a comparison to other participants and how they were rating the source code snippets. The subjective difficulty is measured by asking the participants at the end of the study to rate each source code snippet that they have seen and done on a scale from 1 (very easy) to 5 (very hard). The participants were also asked to write down any additional comment they had to the source code snippets within the source code snippets itself.

Table 4.2: Null hypotheses with corresponding p-values for linearity measures

	Hypothesis	p-value
H_5	VerticalNext is not influenced by the presence of type annotations.	≤ 0.05
H_6	VerticalLater is not influenced by the presence of type annotations.	≤ 0.05
H_7	HorizontalLater is not influenced by the presence of type annotations.	≤ 0.05
H_8	RegressionRate is not influenced by the presence of type annotations.	≤ 0.05
H_9	LineRegressionRate is not influenced by the presence of type annotations.	≤ 0.05
H_{10}	StoryOrder (Naive) is not influenced by the presence of type annotations.	≤ 0.05
H_{11}	StoryOrder (Dynamic) is not influenced by the presence of type annotations.	≤ 0.05
H_{12}	VerticalNext is not influenced by the presence of type annotations and identifier names.	≤ 0.05
H_{13}	VerticalLater is not influenced by the presence of type annotations and identifier names.	≤ 0.05
H_{14}	HorizontalLater is not influenced by the presence of type annotations and identifier names.	≤ 0.05
H_{15}	RegressionRate is not influenced by the presence of type annotations and identifier names.	≤ 0.05
H_{16}	LineRegressionRate is not influenced by the presence of type annotations and identifier names.	≤ 0.05
H_{17}	StoryOrder (Naive) is not influenced by the presence of type annotations and identifier names.	≤ 0.05
H_{18}	StoryOrder (Dynamic) is not influenced by the presence of type annotations and identifier names.	≤ 0.05

Table 4.3: Hypotheses with corresponding p-values for the subjective code difficulty.

	Hypothesis	p-value
H_{19}	The subjective code difficulty is not influenced by the presence of type annotations.	≤ 0.05
H_{20}	The subjective code difficulty is not influenced by the presence of type annotations and identifier names.	≤ 0.05

4.2 STUDY PLAN

The study needs to match the research questions and their operationalization. For this, we opt for a mixed-subject design by using both a within-subject and between-subject approach, as well as a qualitative and quantitative part.

The within-subject design is chosen for the independent variable type annotation. This is done to create a better understanding of how type annotation influence each participants program comprehension. A participant that might have been slower before, might become faster with type annotations and vice versa. This also means that we change as few variables as possible when analyzing type annotations themselves. Furthermore, the change between with and without type annotations might not be as prevalent and can thus easily accommodated without the participants catching on to the meaning of the study immediately.

The between-subject design is chosen for the independent variable identifier names. Through this, we force part of the participants to use bottom-up comprehension instead of using their context knowledge [31].

This gives us the possibility to investigate how the participants' program comprehension is influenced by the type annotations when they are forced to use bottom-up comprehension vs possible context knowledge. Through this, we can get a better image of the influence of type annotations.

The final overview of this decision and the interplay is described in Table 4.4.

4.2.1 *Independent Variables*

The independent variables are whether or not type annotations are available and if the identifier names are obfuscated or meaningful.

4.2.1.1 *Type Annotations*

The first and most important independent variable is type annotations. Type annotations are used in many programming languages to provide the developer with the ability to correctly use the functions and variables of a program. In many programming languages, type annotations are mandatory to run a static analysis on the program when it is compiled. In Python however, type annotations are optional and were only introduced in PEP 484[1].

In this thesis, the type annotations used are type annotations that are usually classified as simple tpyes. These simple types consist of `int`, `float`, `str`, `bool`, and `list`. For `list`, all possible combinations of the other simple types are also considered. Furthermore, the `return type` as well as the `input types` of the function are also included.

In this thesis, we decided against taking complex types and composite types into account (i. e., `dict`, `set`, etc.). This is due to the fact that the source code snippets need to be comparatively small to fit on one screen and be so big that there can be a clear distinction between the lines when using the eye tracker. Additionally, custom types are also not used due to the previous reasons but also because they would have introduced even more

1 https://peps.python.org/pep-0484/

Table 4.4: The independent variables in a 2x2 factorial design mixed design. Identifier names are used as the between-subject grouding factor and Type annotations are used as the within-subject factor.

	Type Annotations		
Identifier Names	Non-Annotated	Annotated	
Obfuscated	Obfuscated | Non-Annotated	Obfuscated | Annotated	Between-Subject
Meaningful	Meaningful | Non-Annotated	Meaningful | Annotated	
	Within-Subject		

overhead, would not have been used in literature before, and created new confounding factors.

Through these criteria, we get a good overview, of how type annotations can influence the comprehension of a program. In the case that the type annotations are added to the source code snippets, all possible type annotations are added to the source code snippets.

4.2.1.2 *Identifier Names*

The second independent variable are the identifier names. In our case, they can either be *meaningful* or *obfuscated*. This forces the participant to use bottom-up comprehension instead of using their context knowledge [31]. Through this, we can get a better image of the influence of type annotations.

In this thesis, a meaningful identifier name is defined as an identifier name that consists of words found in a dictionary and explains the usage of the identifier while at the same time not giving away the whole function's purpose (e. g., `array_to_sort`).

Through this, the developer is provided with context about the identifier's intended usage and functionality. Should an identifier name not provide enough information then it is extended to add more context by using underscores as suggested by Binkley et al. [6].

An obfuscated identifier name is defined as an identifier name that consists only of letters in their order as found in the alphabet. These obfuscated identifiers do not include any underscores. To make the source code snippets comparable using eye-tracking measures between obfuscated and meaningful identifier names, all obfuscated identifier names are the same length as the meaningful ones. We can thus overlay the source code snippets and compare the two groups. Furthermore, we want that all identifiers within a source code snippets are easily distinguishable and not surprising to the participant [13, 16]. For this reason, we created the obfuscated identifier names by starting with the first identifier name, taking its length x in its meaningful form and using the first x letters of the alphabet to create the obfuscated identifier name. In this case the first variable with a lenght of 6 charactes would be written in this way `abcdef`. The next identifier name is then created by taking the next variable and taking the next number of letters from the alphabet as provided by the meaningful identifiers name. Should we at any point reach the end of the alphabet,

we start again at the beginning. Through this, we ensure that no two identifier names are the same or look similar.

4.2.2 *Dependent Variables*

The dependent variables are the measures used to evaluate the influence of both the independent variables. They can be categorized into behavioral, visual attention, and subjective measures.

4.2.2.1 *Behavioral Measures*

Behavioral measures can be objectively and quantifiably observed and thus easily compared and evaluated. We use them to get a good first understanding of measureable program comprehension. Even though these measures are high-level, we can already get a first understanding of the individual difficulty of the source code snippets and the overall general understanding. All measures, however, can not say *why* any participant made mistakes in any given task.

CORRECTNESS Correct source code snippets measures the amount of correctly solved source code snippets for every participant. This gives us an insight into the understanding of certain source code snippets and whether their difficulty was inappropriate. Additionally, we can measure if some participants incorrectly answered many source code snippets suggesting that they either did not understand the task or that their thinking was flawed.

RESPONSE TIME Response time is the time that a participant takes to solve a source code snippet. Through this measure, we find how demanding the tasks were for the participants and possibly their individual complexity. We can also compare if there were significant differences between participants based on the total response time. The exact measure will be started from the moment the source code snippet appeared on the screen until the time that the participant read and answered the question.

4.2.2.2 *Linearity of Reading Order*

To measure the different *linearity of reading order* that programmers employ and find how they use linearity to comprehend a program Peitek et al. [22] and Busjahn et al. [9] created a set of eye-tracking measures. The measures are described in detail in Table 4.5. Through these measures it is possible to measure most of the reading behavior of the participants and find out how they read the source code snippets.

In order to compute the story order, we compute the Needleman-Wunsch alignment score of the fixation order with the linear text reading order. The Needleman-Wunsch alignment score is a measure of the similarity between two sequences and was used by Busjahn et al. [9] among others.

Table 4.5: Gaze-based measures as proposed by Peitek et al. [22] based on the work of Busjahn et al. [9]

	Measure	Definition
Local	*Vertical Next*	% of forward saccades that either stay on the same line or move one line down
	Vertical Later	% of forward saccades that either stay on the same line or move down any number of lines
	Horizontal Later	% of forward saccades within a line
	Regression Rate	% of backward saccades of any length
	Line Regression Rate	% of backward saccades within a line
	Saccade Length	Average Euclidean distance between every successive pair of fixations
Global	*Story Order (Naive)*	Needleman-Wunsch alignment score of fixation order with linear text reading order
	Story Order (Dynamic)	Needleman-Wunsch alignment score of fixation order that tolerates multiple reads

4.2.2.3 *Subjective Feeling*

SUBJECTIVE MEASURES Subjective measures provide the participants with a possibility to let the examiner know how the participants themselves perceived their own program comprehension. These subjective measures target, especially, the overall difficulty, the subjective difficulty for each group, and a free text answer concerning the enjoyment. For this, the participant is provided with all tasks they did.Additionally, we ask about if the type annotations did help with program comprehension in this study and if they help in general. This will give us a clue if the subjective and the objective measures fit well together. All questions except the free-text ones are given as a Likert scale answer.

4.3 SOURCE CODE SNIPPETS

To create a sizable set of source code snippets such that the participants would take a reasonable amount of time to comprehend and solve them, while at the same time not overwhelming them, we collected source code snippets from eye-tracking related literature [5, 21, 34] and created some additional ones. This pool consisted of 44 source code snippets.

The decision process on which source code snippets to use involved an evaluation of the snippets based on the time needed, the subjective complexity, as well as the possibility to add type annotations. The source code snippets should encompass both simple and more complex algorithms so as to more effectively test the influence of the independent

Table 4.6: All main study source code snippets including number of lines of code and number of type annotations added

Source Code Snippet	Lines of Code	Number of Type Annotations
arrayAverage	8	5
binarySearch	12	6
binaryToDecimal	10	2
bubbleSort	8	4
commonChars	10	6
containsSubstring	11	3
countIntegerInterval	9	8
countLetters	7	4
crossSum	4	2
factorial	4	2
forwardBackward	7	4
leastCommonMultiple	7	4
linearSearch	5	3
palindrome	8	4
power	6	3
prime	6	3
squareRoot	11	3
unrolledSort	12	5
validParentheses	12	2

variables. This inlcudes both shorter and longer source code snippets as well as the concept of recursion.

This led to a set of 27 source code snippets which were evaluated in the pilot study. To remove as many confounding factors from the code as possible, all comments were removed, the language was changed from Java to Python, the code was indented correctly, and the function names were changed from their descriptive names to `compute`, so as to obscure the function itself. Furthermore, all variable names were changed to fit Python's naming convention[2] and Sharif and Maletic [30] recommendation to use underscores to separate words in identifier names. After this, the source code snippets were modified by obfuscating identifier names and using type annotations. All of these source code snippets were tested in the pilot study and the results concerning time and difficulty were used to remove some of the source code snippets from the main studies source code snippet list. The final list of 20 source code snippets can be found in Table 4.6.

2 PEP 8: `https://peps.python.org/pep-0008/`

Listing 4.1: Binary search source code snippet computing the index of a key in an array with meaningful identifier names and included type annotations.

```
def compute(array: list[int], key: int) -> int:
    index1: int = 0
    index2: int = len(array) - 1
    while index1 <= index2:
        m: int = (index1 + index2) // 2
        if key < array[m]:
            index2 = m - 1
        elif key > array[m]:
            index1 = m + 1
        else:
            return m
    return -1
```

SOURCE CODE SNIPPET In order to remove further confounding factors, the source code snippets were created as pictures in the monospace font JETBRAINSMONONL-REGULAR[3] without any ligatures. Furthermore, the whole code was written in black on a white background, i. e., without any syntax highlighting so as to not draw the gaze of the participant and thus influence the eye-tracking measures.

For each source code snippets, we created four versions, one for each field in the 2x2 factorial design. This means that each source code snippets was created with and without type annotations and with meaningful and obfuscated identifier names. This gives us a total of 80 different source code snippets. The difference between the source code snippets is visible in Listing 4.1 and Listing 4.2. The first listing displays a source code snippets that includes both type annotations as well as meaningful identifier names. This was the default version of the source code snippets from which the others were created. The second listing shows the same source code snippets but with obfuscated identifier names and no type annotations. For reading purposes in this study, we used some code highlighting, however, the code highlighting is removed for the participants. An important note is that the identifier names in both the meaningful and obfuscated versions are the same length and are easily distinguishable from each other.

4.3.1 *Programming Language: Python*

Python was chosen as the programming language of choice since it is a simple language that is close to Pseudo Code and most developers, even if they do not have any previous knowledge in Python, can comprehend the code to a certain extent. Furthermore, Python is a language which can be written with and without type annotations as is necessary for the independent variables. Also, it is easy to follow the official style guide for Python which makes it easy to have a consistent style throughout all source code snippets.

3 https://www.jetbrains.com/lp/mono/

Listing 4.2: Binary search source code snippet computing the index of a key in an array with obfuscated identifier names and without type annotations.

```
def compute(abcde, fgh):
    ijklmn = 0
    opqrst = len(abcde) - 1
    while ijklmn <= opqrst:
        u = (ijklmn + opqrst) // 2
        if fgh < abcde[u]:
            opqrst = u - 1
        elif fgh > abcde[u]:
            ijklmn = u + 1
        else:
            return u
    return -1
```

4.4 PRE-QUESTIONNAIRE

The pre-questionnaire is a questionnaire for every participant that consist of demographic questions, questions about the general experience with programming languages and Python, as well as a self-assessment concerning their experience in comparison to fellow students. The pre-questionnaire is implemented within the study software and can thus be answered by all participants themselves. A special regard was given to the question whether the participants already took part in either the advanced lecture *Competitive Programming* or in the core lecture *Algorithms and Data Structures*, since both of these lectures contain a lot of information about algorithms, how to recognize them and how to implement them. This is relevant because the participants' ability to comprehend the source code snippets might be influenced. For most answers, the participants were only able to choose one of the given answers, for the self-assessment and the programming languages they could fill in open fields.

4.5 POST-QUESTIONNAIRE

The post-questionnaire is done in an interview style shortly after the main study and was read aloud by the study conductor. The post-questionnaire consists of general questions regarding the subjective difficulty of the source code snippets, how the participants feel, what their self-assessment is, and how they think about type annotations during the study and in general. The full post-questionnaire can be found in Section A.2. For each question with an open field the study conductor wrote down the answer of the participant in summarized and abbreviated form.

4.6 PARTICIPANTS

4.6.1 *Number of Participants*

The study should be conducucted with a sizable number of participants. Because there are no effect sizes reported for type annotations, we would need to approximate the effect size. This would lead to uncertainty in the results, which is why, we use the number of other eye-tracking studies in computer science as a reference. Some of the studies used as a reference are [9, 21, 22, 29, 31]. In these papers, the number of participants ranges from 14 participants to 37. To be on the safe side, we chose to have at least 30 participants.

4.6.2 *Prerequisite/Criteria of Participants*

For this study, the participants had to fit certain criteria. Any participant has to be at least 18 years of age, study in a computer-science related course of study, be an undergraduate student (Bachelor), and be able to program in Python.

AGE All participants had to be at least 18 years old to give their informed consent. This is in accord with the regulations of the Ethical Review Board (ERB)[4] and the Universität des Saarlandes.

COURSE OF STUDY All participants needed to be in a computer-science related course of study. As a point of reference, all courses of study that are part of the Faculty of Mathematics and Computer Science are considered computer-science related[5]. This criteria was chosen to ensure that the participants have a basic level of comprehension of programming, programming languages, and the concepts of computer science. This might also be true for other courses of study, however, we used this to filter beforehand.

OCCUPATION All participants needed to be undergraduate students. This was chosen to ensure that all participants have a similar level of experience in both programming and computer science. Through this, we create a more homogeneous group of participants. According to Siegmund and Schumann [34], this is a good criterium to get a more homogeneous group and even out the randomization. This especially excludes professional programmers who are likely very different in overall experience.

EXPERIENCE WITH PYTHON Each participant needed to be able to write and comprehend small programs in Python. Since there is no simple and fast way of checking for this, the participants had to self-assess their experience. When applying for the study, the participants had to answer the question on whether they had any experience with Python. Only if they did, were they invited to come in and do the study. A soft requirement was that the participants should have taken the *Programming 2 Pre-Course* which is taught in Python. If participants did not take part in the pre-course, but their self-assessment was that they were able to program in Python, they were also allowed to participate.

4 https://erb.cs.uni-saarland.de/

5 https://saarland-informatics-campus.de/en/studium-studies/

4.6.3 *Recruitment*

All 31 participants were recruited at Universität des Saarlandes by handing out flyers with exact requirements after exams, convincing students through personal means, hanging up posters all over campus, and advertising within lectures. As incentives, participants received a small number of sweets, a softdrink, and the chance to win one of two Amazon vouchers worth 25 € each.

4.7 ETHICAL CONSIDERATIONS

Before running the study itself, the study plan (Section 4.2), all questionnaires (Section 4.4, Section 4.5), and the declaration of consent (Figure A.4) were handed in to the Ethical Review Board of the Faculty of Mathematics and Computer Science. The ERB is responsible for checking the ethical consideration of any study at the Faculty of Mathematics and Computer Science. The study was approved on 2023-12-04 with approval number 23-11-07. All documents concerning this application can be found in the appendix (Section A.1).

MAIN ETHICAL CONSIDERATIONS Some of the most important ethical considerations and privacy considerations were whether the participants could be fully informed about the study, would be put under stress or misled about the study's purpose. Overall this study did not take any special measures within the study and the participants could be informed about everything. The privacy considerations were taken into account and any data related to the participants was anonymized and stored only on the study computer, a USB-drive, as well as the university GitLab[6].

4.8 EYE-TRACKING SOFTWARE AND HARDWARE

The eye tracking software used is the Tobii EyeX Core. The eye tracker is a Tobii X60[7] with a sampling rate of 70 Hz that uses backlight assisted near infrared illumination. The eye tracker is mounted at the bottom of a monitor and is used to track the participants' gaze. The display has a resolution of 2560 by 1440 pixels.

The Tobii X60 is able to capture the gaze of participants in a 40 x 30 cm area at a distance of 75 cm. To enable a good measurement, the participants were seated in a chair in front of the monitor and were asked to keep their head still and in the same position throughout the study. The participants were also asked to keep their distance from the monitor and to not move their head too much.

The source code snippets are displayed using an adapted version of Peitek et al. [23]'s eye-tracking software. This software is able to display the source code snippets, collect answers to the pre-questionnaire, display questions and possible answers to the source code snippets, calibrate the eye tracker with the participant's help.

6 https://gitlab.cs.uni-saarland.de/

7 https://help.tobii.com/hc/en-us/articles/212818309-Specifications-for-EyeX

4.9 PILOT STUDY

The pilot study was organized a few weeks before the main study should start. All 4 participants were recruited out of the field of computer science and were not part of the main study. These participants were shown all previously selected source code snippets. They ran through both the pre-questionnaire and post-questionnaire and each of the participants were shown 27 source code snippets with their gaze being measured. This was done to receive feedback on the procedure, as well as the subjective difficulty, and time taken for each of the source code snippets. From the feedback we removed those source code snippets that our pilot study participants did not solve within the given time limit of 5 minutes or solved too quickly (i. e., under 20 seconds). Furthermore, we chose the number of source code snippets to be 20 such that the main study would take around 45 minutes to complete. It was also used to check for any obvious mistakes in the study implementation.

4.10 DEVIATIONS

During the main study, some deviations from the study plan occured. Due to a problem with the timer, the first participant took a whole hour to finish all snippets. To mitigate this, we removed the snippets done after 45 minutes from the data analysis. The 27th participant found a small bug in the source code snippet `capitalizeFirstLetter` when type annotations are provided. The type annotation in line 2 needed to be changed from `list[int]` to `list[str]`. This should not have a huge impact on the results, especially, since no other participant detected it.

5 RESULTS

In this chapter, we present the results of the study. We start with the demographic data of the participants and then move on to how the data analysis was done. We then present the results for each research question and its sub-questions. We start with the behavioral measures, then move on to the linearity measures, and finish with the subjective difficulty of the participants. In the end, we also present general findings that can not be put into one of the research questions.

5.1 PARTICIPANTS

Overall, 31 participants were recruited for the study and considered according to the exclusion criteria in Table 5.1. Out of these intial 31 participants, 28 fit the criteria for the study and were included in the analysis. Three of the four participants that were excluded did not study a computer-science related course of study, while one participant did not have enough eye-tracking data points remaining after the data cleaning process. Considering the remaining participants, 30% studied *Computer Science* and 30% studied *Cybersecurity*, while the rest was distributed over other computer-science related courses of study. Out of the 27 participants, 3 reported their gender as *Female*, 23 as *Male* and 1 person *Prefer not to say*. The average age of the participants was 22.68 (SD = 4.02) years. Additionally, the participants were asked whether they had any impaired eye-sight which was the case for 6 participants but all reported that they had corrective glasses and with these had 100% eyesight. Furthermore, to establish whether the participant had already seen most algorithms in a algorithms lecture at the university, they were asked whether they had taken either the core lecture *Algorithms and Data Structures* or *Competitive Programming*. This was the case for 2 participants.

GROUPING Since there were two groups and the distribution into the groups was completely random, 9 participants were assigned to the *Obfuscated Group* and 18 to the *Meaningful Group*. The exact values are reported in Table 5.2.

Table 5.1: Exclusion criteria for participants

	Criterion		n
Age	< 18	No	0
Higher Education	Bachelor?	No	9
Course of Study	Computer-science related	No	3
Python Experience	Average or Prog2 Prep Course?	No	0

Table 5.2: Demographic data of our participants

	Meaningful (n=18)	Obfuscated (n=9)
Male	14 (82%)	9 (90%)
Female	2 (12%)	1 (10%)
Prefer not to say	1 (6%)	-
Age (in Years)	22.89 ± 4.5	22.30 ± 3.16
Semester	5.78 ± 2.39	5.10 ± 3.14
Years of Programming	4.83 ± 2.48	5.60 ± 3.84
Have done a study before	7	3
Impaired Eyesight (but still 100%)	2	6

5.2 DATA ANALYSIS

The data analysis was done in three steps. First the data was cleaned, then it was processed, and finally the analysis was done.

5.2.1 *Data Cleaning*

When analyzing the data, we found that some participants were not able to finish the study in the alloted time or took very long with some source code snippets. Similar to Peitek et al. [22], we removed the top 5% (Response Time > 180 seconds) of completion times for all snippets, so that there is less influence from outliers. This led to one participant being removed completely from the study, because 75% of their data was removed. Additionally, we checked whether to remove the bottom 5% as well, but since the cutoff point would have

been 20 seconds, we kept them in, as it is possible to comprehend the source code snippets in that time.

Furthermore, through the use of our own eye-tracking software, the pictures were slightly scaled and there was a continue button in the bottom right corner to go to the next snippet. Any data points that are found within the 140 pixels at the bottom are not relevant because they are not on the source code snippets. We thus removed all data points that were within 140 pixels of the bottom of the screen and then scaled the eye-tracking data back to the original size.

MISSING DATA Out of the 27 participants, 3 did not have any eye tracking data. This was due to an incorrect setup for the eye tracking device which was not immediately found and due to too many movements of some participants. For one participant there is no data concerning the subjective difficulty of the source code snippets. Even though the study was setup such that participants would take 45 minutes to complete the study, most participants finished all 20 within the alloted time. Six participants did not finish the full 20, four of them missed less than 4 while one missed 8 source code snippets.

5.2.2 *Data Processing*

For the data processing, we used a modified version of the analysis tool that Peitek et al. [23] created and published with their replication package. This tool was used to preprocess the eye-tracking data and then create the linearity measures. For the creation of the AOI, another tool from the chair was used that creates AOI on a source code snippets with different colors according to the underlying text.

5.2.2.1 *Preprocessing Data*

After the data was cleaned and scaled, it was smoothed using a Savitzky-Golay filter with a window size of 5 and a polynomial order of 3. Then a velocity-based algorithm was run to distinguish between fixations and saccades. For this a velocity of 150 pixel in 100 milliseconds was used as a threshold. If the velocity was below this threshold, the point was considered a fixation, otherwise it was considered a saccade [22].

5.2.2.2 *Areas of Interest*

The AOI were created using a small program used previously in other eye-tracking related studies. This included defining the AOI for each source code snippets. We created them for each line, for the whole source code snippets, and for the type annotations if they were present. Because there might be some inaccuracies in the eye-tracking data, we followed the examples of Peitek et al. [24] by adding 5 pixels to all sides of each AOI. Furthermore, following Busjahn et al. [9], all fixations that are within 100 pixels horizontally of an AOI are considered to be part of that AOI otherwise smaller AOI could be easily missed.

A typical source code snippets with type annotations and the added AOI can be seen in Figure 5.1. The AOI are colored in different colors and the type annotations are marked with a red border, while the return type is turquoise.

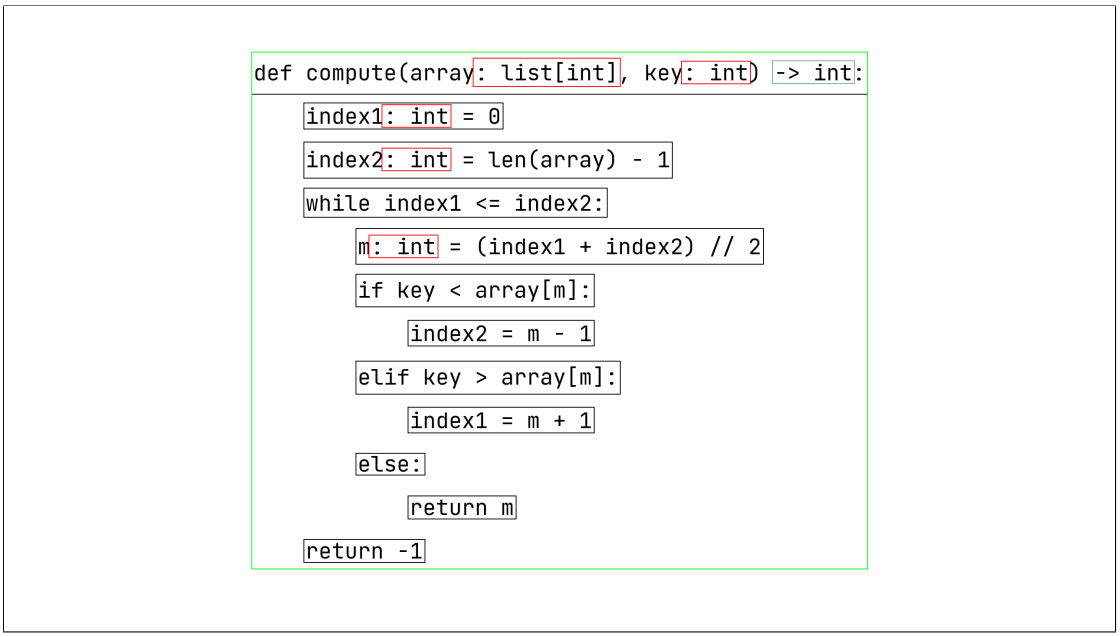


Figure 5.1: An example of a source code snippets with the type annotations and the AOI marked. The type annotations are marked with a red border, while the AOI are colored in different colors.

5.2.3 *Analysis Procedure*

After preprocessing, we used Peitek et al. [23]'s tool to create the linearity measures for each participant and each source code snippet.

Since we have a different amount of independent variables for the research questions, we use statistical measures that take only one variable into account to measures statistical significance. This includes the *Mann-Whitney U* test for the behavioral measures and the linearity measures. For the subjective difficulty, we use the *Chi-Square Test of Independence*. Both of these are provided by thy Python library *statsmodels*. For the second research question, we use a mixed linear regression model to account for the different independent variables and then use the *Wald Chi-Square* test to find the p-values. This is also provided by the *statsmodels* library.

5.3 RESEARCH QUESTION 1

In this section, we report all the results for the first research question and all its sub-questions. We start with the behavioral measures, then move on to the linearity measures, and finish with the subjective difficulty of the participants.

5.3.1 *Behavioral Measures*

Under behavioral measures, we count the number of correct and incorrect answers and the time it took to answer the source code snippets. In this research question, we consider the hypotheses H_1 and H_2 from Table 4.1 with their respective threshold of ≤ 0.05.

Table 5.3: The 2x2 matrix for the mean and standard deviation of response time for each group created by the independent variables

	Type Annotation		
Identifier Names	Annotated	Not Annotated	Total by Identifier Names
Meaningful	74.75 ± 50.38	64.27 ± 38.07	69.75 ± 45.17
Obfuscated	89.98 ± 55.92	88.85 ± 67.44	89.52 ± 60.74
Total by Type Annotation	79.46 ± 30.33	72.36 ± 28.35	

RESPONSE TIME The overall average time spent for annotated snippets was 79.46 (SD = 30.33) seconds. For non-annotated snippets, the average time was 72.36 (SD = 28.35) seconds. The mean time for the annotated source code snippets was thus longer, while both the standard deviations were roughly the same.

To check if the response time's increase when type annotations are added is statistically significant, we first check for both normality with *Shapiro-Wilks Test* and for homogeneity in the variance with *Levene's Test*. We can reject the hypothesis for normality for both with p-values of < 0.001 and < 0.001 respectively. The variance however is equal with a p-value of 0.13.

Because our data is not normally distributed and also not of the same length, we choose a length robust test, namely the *Mann-Whitney U*, which yields a p-value of 0.07 (U = 28663) (Table 5.5). This means that we can not reject the null hypothesis of H_1 that there is no difference between the annotated and non-annotated source code snippets.

CORRECTNESS The overall correctness of the study was 87% out of 504 possible correct answers. For the two groups, we have 85% for the annotated source code snippets and 88% for the non-annotated source code snippets. The respective table can be found in Table 5.6.

To check for the null hypothesis (H_2), that the two groups are independent of each other, we use a *Chi-Square Test of Independence* on the contingency table. The p-value for this test is 0.39 (Table 5.5) which is above the threshold and thus we can not reject the null hypothesis of H_2 that the groups are independent of each other. This means that there is no statistically significant difference between the correctness of the source code snippets with and without type annotations.

SUMMARY We find that there is no statistically significant difference between the medians of the times for the source code snippets with and without type annotations. We also find no statistically significant difference between the correctness of the source code snippets with and without type annotations. We can thus conclude that the presence of type annotations does not influence the behavior of the participants in this study.

Table 5.4: Code snippets with the respective correct answers and the time it took to answer them

Source Code Snippet	Correct Answers	Time (in seconds)
arrayAverage	25/25 (100%)	56.36 ± 24.06
binarySearch	19/22 (86%)	93.0 ± 40.12
binaryToDecimal	21/25 (84%)	79.4 ± 28.18
bubbleSort	22/26 (85%)	92.04 ± 45.24
capitalizeFirstLetter	26/26 (100%)	97.77 ± 38.35
commonChars	18/27 (67%)	57.67 ± 30.91
containsSubstring	17/21 (81%)	91.9 ± 36.11
countIntegerInterval	18/25 (72%)	78.48 ± 22.81
countLetters	22/26 (85%)	60.54 ± 39.03
crossSum	18/25 (72%)	36.2 ± 15.11
factorial	25/26 (96%)	23.88 ± 15.66
forwardBackward	25/25 (100%)	77.56 ± 22.33
leastCommonMultiple	24/26 (92%)	53.96 ± 22.76
linearSearch	24/27 (89%)	31.59 ± 13.08
palindrome	25/25 (100%)	70.72 ± 37.59
power	24/27 (89%)	48.78 ± 26.08
prime	22/26 (85%)	73.73 ± 40.18
squareRoot	21/22 (95%)	72.59 ± 25.45
unrolledSort	19/26 (73%)	71.19 ± 35.42
validParentheses	22/26 (85%)	111.35 ± 36.58
Overall	437/504 (87%)	68.39 ± 37.96

5.3.2 *Linearity Measures*

For this research question, we consider the null hypotheses H_5 to H_{11} from Table 4.2 with their respective threshold of ≤ 0.05. Each of these hypotheses is created for one particular linearity measure and thus we will report the results for each of them separately.

To get a good overview of eye-tracking measures and how the participants look onto the screen we chose 8 different linearity and eye-tracking measures. These measures are *VerticalNext*, *VerticalLater*, *Regression*, *HorizontalLater*, *LineRegression*, *StoryOrder (Naive)*, *StoryOrder (Dynamic)*, and *SaccadeLength*. For each measure the *Mann-Whitney U* test was used because the analyzed data was not normally distributed (Table 5.7). Following this, none of the tests was statistically significant and thus we can not reject the hypotheses (H_5 to H_{11}) that there is no difference between annotated and non-annotated source code snippets. The false-discovery-rate correction [4] was not necessary since all p-values were above the threshold.

Table 5.5: Display the time and correctness measures with the respective tests and p-values. We mention for each group if they are normally distributed and if the variance is equal.

Dependent Variable	Response Time		Correctness	
Type Annotation	Yes	No	Yes	No
Mean ± SD	79.46 ± 30.33	72.36 ± 28.35	85% ± 13%	87% ± 13%
Normality (p-value)	Yes (< 0.001)	Yes (< 0.001)	Yes (< 0.001)	Yes (< 0.001)
Variance (p-value)	Yes (0.13)		Yes (0.32)	
Test	*Mann-Whitney U*		*Chi-Square Test of Independence*	
Stat (p-value)	U = 28663 (0.07)		χ^2 = 0.71 (0.39)	

Table 5.6: The contingency table for the correctness grouped by both independent variables type annotations and identifier names

	Type Annotation		
Identifier Names	Annotated	Not Annotated	Total by Identifier Names
Meaningful	156/182 (86%)	147/166 (89%)	303/348 (87%)
Obfuscated	95/109 (87%)	65/76 (86%)	160/185 (86%)
Total by Type Annotation	251/291 (88%)	212/242 (88%)	

5.3.3 *Subjective Difficulty*

In this research question, we consider the null hypothesis H_{19} from Table 4.3 with the threshold of ≤ 0.05.

The individiual difficulty of each source code snippets was rated by the participants on a likert scale from 1 (Very Easy) to 5 (Very Difficult). We find that more than 70% of the source code snippets were rated as *Very Easy* or *Easy* by the participants independent of the type annotations.

The results of this scale can be found in Figure 5.2 in which all values are displayed proportional to their respective groups size. We see a slight trend that on average the source code snippets with type annotations were rated as *Very Easy* more often than the non-annotated source code snippets. However, they were also more often rated as *Difficult*.

To find if there is a statistically significant difference between annotated and non-annotated source code snippets we use a *Chi-Square Test of Independence* test on the respective contingency table. This test is used because the input and output variable are each categorical. The *Chi-Square Test of Independence* yields a p-value of 0.72 which is above the threshold and thus we can not reject the null hypothesis (H_{19}) that there is no difference in difficulty between the groups.

Table 5.7: All linearity measures with the test used, the p-value and the test value. If any test result is statistically significant they are marked.

Linearity Measure	Test	p-value	Test Value
Vertical Next	*Mann-Whitney U*	0.76	U = 22061.0
Vertical Later	*Mann-Whitney U*	0.57	U = 22369.5
Horizontal Later	*Mann-Whitney U*	0.85	U = 21447.0
Regression Rate	*Mann-Whitney U*	0.54	U = 21534.5
Line Regression Rate	*Mann-Whitney U*	0.85	U = 20584.0
Saccade Length	*Mann-Whitney U*	0.14	U = 19678.0
Story Order Naive	*Mann-Whitney U*	0.07	U = 23934.5
Story Order Dynamic	*Mann-Whitney U*	0.28	U = 23004.5

SUMMARY This study finds that there is no statistically significant evidence that the subjective difficulty is influenced by the presence of type annotations.

5.3.4 *Summary of RQ 1*

This study has not found any statistically significant evidence that the presence of type annotations influences the behavior of the participants. It has also found that it does not meaningfully influence eye-tracking measures or the assigned difficulties of the source code snippets. Subjectively, the participants found the type annotations helpful both in this study and in general. They also found that the source code snippets were not too difficult and that the type annotations were helpful in understanding the source code snippets. Therefore this research question finds, that the presence of type annotations does not significantly influence program comprehension.

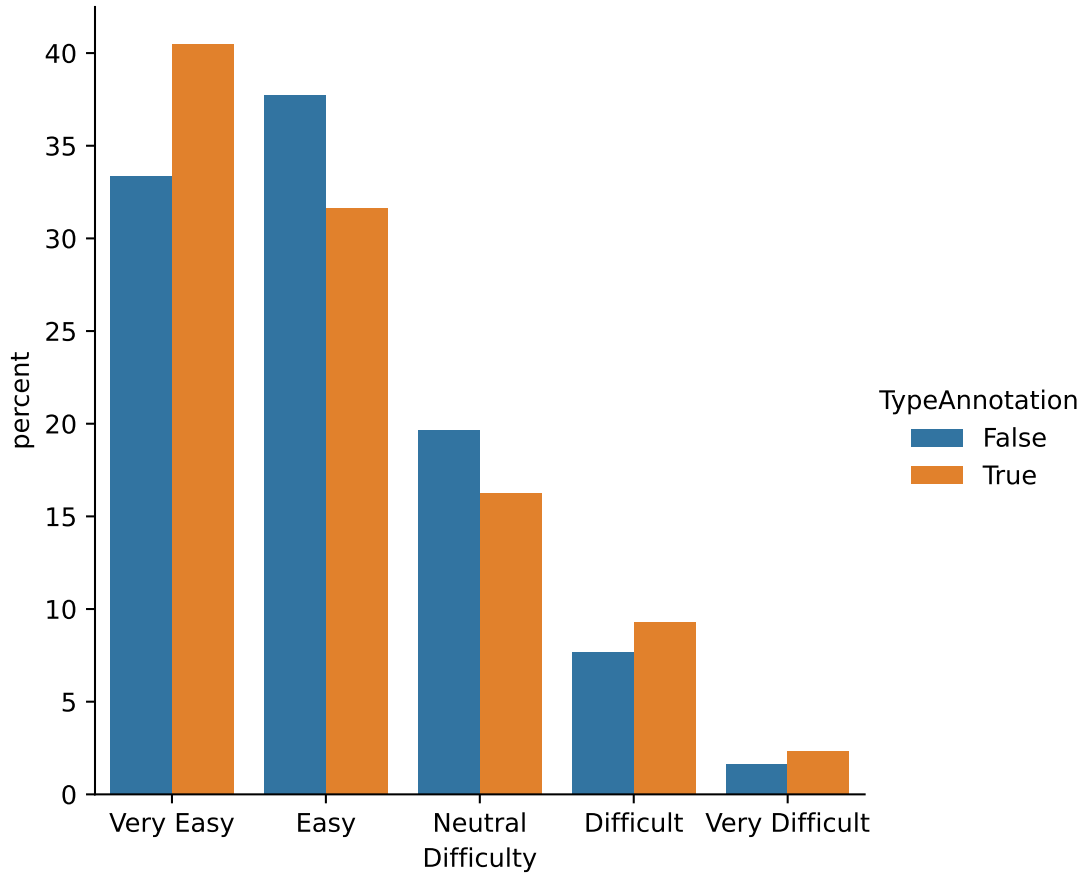


Figure 5.2: The difficulty answers divided into two groups of annotated and non-annotated source code snippets with the respective proportion of their group.

5.4 RESEARCH QUESTION 2

In this section, we report all the results for the second research question and all its sub-questions. We start with the behavioral measures, then move on to the linearity measures, and finish with the subjective difficulty of the participants.

5.4.1 *Behavioral Measures*

For this research question, we consider the null hypotheses H_3 and H_4 from Table 4.1 with their respective threshold of ≤ 0.05.

RESPONSE TIME We find that the time taken to solve the source code snippets is on average longer for the obfuscated group than for the meaningful group. The average time for the meaningful group was 74.75 (SD = 50.38) seconds. For the obfuscated group, the average time was 89.98 (SD = 55.92) seconds (Table 5.3). The mean time for the obfuscated group was thus longer, while the standard deviation was also higher.

To check if these differences are statistically significant, we use a mixed linear regression model and use the source code snippets as our group. Based on this, we then compute the *Wald Chi-Square* test which gives us the p-values of 0.007 for the meaningful independent variable, 0.43 for the type annotations independent variable, and 0.67 for the combination of both. Thus we conclude that we can not reject H_3 that the time taken to solve the source code snippets is not influenced by the independent variables.

CORRECTNESS We find that there is no real difference in the correctness between the different groups and whether a source code snippets was annotated or not. Even though there is a strong contrast in the numbers of correct answers between the groups, the percentage of correct answers is nearly the same for all of them, namely roughly 86%. The exact values for each group are reported in Table 5.6.

In order to find out if the correctness is influenced by both of the independent variables, we create a logistic regression model and use the *Wald Chi-Square* test to find the respective p-values. This provides us with a p-value of 0.63 for the combination of both independent variables. Since this p-value is above the threshold, we can not reject the H_4. This means that the correctness of the source code snippets is not influenced by the type annotations or the identifier names.

SUMMARY This study shows that there is a statistically significant difference in the time taken to solve the source code snippets between the meaningful and obfuscated groups. However, there is no statistically significant difference in the correctness of the source code snippets. This means that source code snippets with meaningful identifier names are faster to solve but not more correct.

5.4.2 *Linearity Measures*

For this research question, we consider the null hypotheses H_{12} to H_{18} from Table 4.2 with their respective threshold of ≤ 0.05. Each of these hypotheses is created for one particular linearity measure and thus we will report the results for each of them separately.

For the eye-tracking measures, we use a mixed linear regression model and then run the *Wald Chi-Square* test to find the p-values. Because we use multiple measures on the same data and to account for multiple testing, we use the false-discovery-rate correction [4]. The results for the combination of both independent variables can be found in Table 5.8.

We find that for the combination of both independent variables type annotations and identifier names, no measure is statistically significant. Thus, we can not reject any of the hypotheses H_{12} and H_{18}.

5.4.3 *Subjective Diffulty*

For this research question, we consider the null hypothesis H_{20} from Table 4.3 with the threshold of ≤ 0.05.

To assess the reported difficulty, the partcipants are grouped by the identifier names and displayed proportional to their appearance within the group. For the meaningful values, we observe that more than 70% of all source code snippets were rated as *Very Easy* or *Easy* by

Table 5.8: Results of mixed linear regression models on the linearity measures for the combination of both independent variables, before and after false-discovery-rate correction

Linearity Measures	Combination before FDR correction	Combination after FDR correction
Vertical Next	0.43	0.57
Vertical Later	0.033	0.13
Horizontal Later	0.056	0.15
Regression Rate	0.031	0.13
Line Regression Rate	0.33	0.53
Saccade Length	0.10	0.2
Story Order Naive	0.55	0.63
Story Order Dynamic	0.97	0.97

TM = Annotated and Meaningful, NO = Not Annotated and Obfuscated, T = Annotated, N = Not Annotated

the participants (Figure 5.3). It is especially important to see, that there is a much higher number for *Very Easy* when type annotations are available and only one participant gave any source code snippet, *Very Difficult* as a rating. For the obfuscated group, we have a different picture. The *Very Difficult* ratings are much higher and the meaningful group has a higher number of *Difficult* and *Very Easy* ratings. The *Very Easy* and *Easy* ratings, however, are higher for the obfuscated group, especially when the source code snippets are annotated.

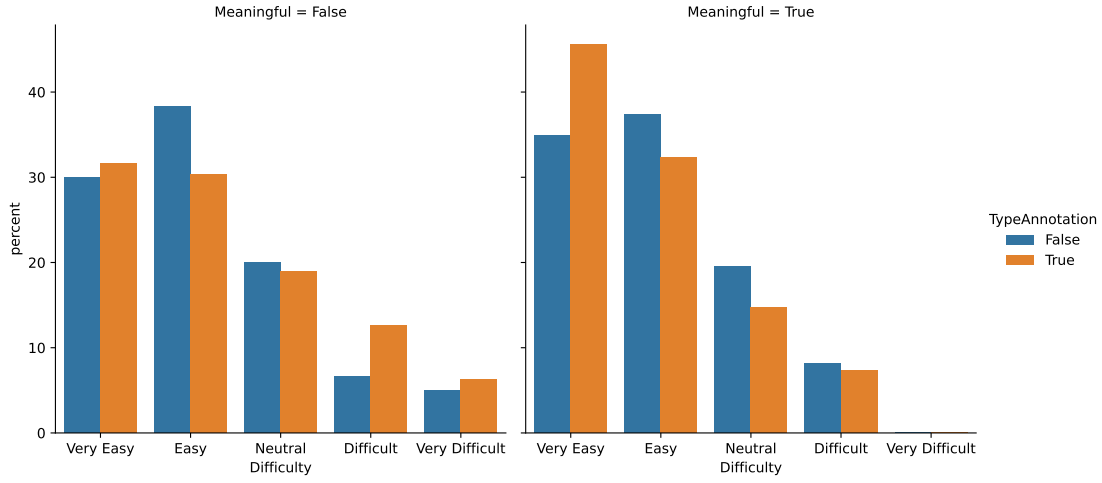


Figure 5.3: The difficulty grouped by identifier names and type annotations with their proportional appearance in the groups.

When analyzing the difficulty for significance, we use a *Wald Chi-Square* test on a logistic regression model because all variables are categorical. This yields a p-value of 0.55 for identifier names, a p-value of 0.78 for the type annotations, and a p-value of 0.65 for the combination of both. Since all of these p-values are above the threshold, we can not reject the hypothesis that there is no difference in difficulty between the groups.

SUMMARY We observe that subjective difficultyis not significantly influenced to show difference in the groups created by the independent variables.

5.4.4 *Summary of RQ 2*

This study has not found any statistically significant influence that the combination of type annotations and identifier names influences the behavior of the participants. For the eye-tracking measures, we find that the combination of both independent variables has a statistically significant influence on *Regression Rate* and *Vertical Later*. None of the other combinations yield a statistically significant result. The subjective can also not reject its hypothesis. Thus, this research question finds that the combination of type annotations and identifier names influences program comprehension, but only in some special cases.

5.5 GENERAL FINDINGS

In this section, we report additional data that can not be directly included in one of the research questions but is relevant for the study and the discussion thereof. This includes the subjective helpfulness of type annotations on the source code snippets and in general.

Table 5.9: Contingency table of reported frequency for the subjective helpfulness of type annotations within this study and in general grouped by identifier names

Group	Within Study		Outside Study	
	Helpful	Not Helpful	Helpful	Not Helpful
Meaningful	17	1	14	4
Obfuscated	8	1	9	-
Total Helpful	25	2	23	4

TYPE ANNOTATIONS WITHIN THE STUDY All participants were asked if the type annotations were helpful during the study. We observe that 89% of all participants found the type annotations helpful. This is further broken down into the identifier names meaningful and obfuscated where exactly one participant of each group did not find the type annotations helpful. We check if this is statistically significant for the meaningful identifier names. For this, we use a contingency table (Table 5.9) and because two values are below 5, the *Fisher's Exact* test is appropriate. This yields a p-value of 0.26 which is above the threshold of 0.05 and thus we can not reject the hypothesis that there is no difference between the groups.

TYPE ANNOTATIONS IN GENERAL The participants were also asked if they found type annotations helpful in general. We observe that 93% of all participants found the type annotations helpful. We break this down further into the identifier names meaningful and obfuscated where overall 4 participants did not find the type annotations helpful. These 4 participants were all in the meaningful group. We check if this is statistically significant for identifier names with a contingency table (Table 5.9) and because two values are below 5, the *Fisher's Exact* test is appropriate. This yields a p-value of 1.0 which is above the threshold of 0.05 and thus we can not reject the hypothesis that there is no difference between the groups.

SUMMARY GENERAL FINDINGS This study finds that the subjective helpfulness of type annotations is very high both within the study and in general.

6 DISCUSSION

This study aimed to explore the influence of type annotations and identifier names on program comprehension, focusing on how they affect the behavioral measures, the linearity of reading order and participants' perception.

Relevant to our investigation was not only the collection of quantitative data but also the collection of qualitative feedback from the participants. This feedback was collected through the post-questionnaire asking the participants about their experience with the study and their general opinion on type annotations in the scope of this study and in general.

In the following discussion, we discuss the sub-research questions individually and then summarize the results for the main research questions. We then discuss the results' implications and the study's limitations.

6.1 RESEARCH QUESTION 1

RQ 1.1: How do type annotations influence behavioral measures?

We expect that the participants will generally be faster and more correct in solving the source code snippets if type annotations are present, irrespective of whether the identifier names were obfuscated or meaningful. The study found that, on average, the annotated source code snippets took longer to solve than the non-annotated ones. However, this result is not statistically significant. Similarly, the average correctness does not change significantly for the presence of type annotations. Peitek et al. [22] also reported that for a closely related study, the correctness is not significantly different when changing the difficulty of the source code snippets.

Since both of these studies have similar findings, this might suggest that the participants only answered the source code snippets' question when they were ready and understood the code independently of what the underlying code looked like or whether it was harder or easier to comprehend. This should result in a higher average time taken for the source code snippets, which we can see in the data. However, the statistical test could not reject the hypothesis that the groups had no difference. This might be because the source code snippets or the questions were too easy, and the answers were too obvious. On the other hand, Peitek et al. [23] asked their participants to answer each question in a free-text field. The answers were then manually checked for correctness, which is arguably harder than answering a multiple-choice question. They reported very similar correctness values to this study and found no significant results.

RQ 1.1: Overall, we find that the behavioral measures are not significantly influenced by type annotations.

RQ 1.2: How do type annotations influence the linearity of reading order?

We expect that the presence of type annotations results in a more linear reading order because the type annotations provide additional information, which makes it easier to understand the source code snippets. Additionally, we expect that there will be lower regression scores both within a line and between the lines of code. The study found that the presence of type annotations did not result in a significantly more linear reading order or fewer regressions. Possible reasons are the complexity of the source code snippets and the fact that the information added by simple types is not as much as expected. The simple types and the complexity of the source code snippets were a design choice such that the source code snippets could easily fit on the screen, and eye tracking could be used without scrolling.

RQ 1.2: Overall, we find that the eye-tracking measures are not significantly influenced by type annotations.

RQ 1.3: How do type annotations influence the subjective code difficulty?

We expect that participants would report a greater difficulty if the source code snippets were not annotated versus when they were annotated. This was not the case; the statistical test could not reject the hypothesis that there was no difference between the groups. Even though the participants did not report a higher difficulty, the reported difficulty shows that the source code snippets were overwhelmingly (> 70%) considered to be *Easy* or *Very Easy*. This might be the reason why the participants did not report a higher difficulty when the source code snippets were not annotated, which suggests that the source code snippets were too easy to show a real difference.

However, nearly all (89%) participants reported that the type annotations were helpful in comprehending the source code snippets. This is a very high percentage and shows how well-liked the type annotations are. Two participants declined that the type annotations were helpful. The declining participants' reason for this was that the type annotations did not provide any accurate new information and that they were more of a distraction than a help. The participants who found the type annotations helpful said that they were a good reminder of what the code was supposed to do and that they got additional information, especially when they were not able to immediately infer the types from the statements. They also mentioned that the type annotations were helpful when the code was new to them.

This is corroborated by the fact that 93% of the participants found that type annotations are generally helpful in understanding code. One of the two participants who declined mentioned that they did so because they see type annotations as only necessary when they are getting into a new language and are not yet sure of what the code is doing. This participant also declined that type annotations were helpful with the program comprehension source code snippets for a similar reason. The other participant said that they think that type annotations are not helpful unless accompanied by a comment, which is especially the case when the programming language is Python. However, they mentioned before that they did like the type annotations in the case of this study, even though there were no comments on any of the source code snippets. The participants who found type annotations helpful in

general said that they are very helpful when the code is not their own or when there are more complicated types. They can not necessarily infer what an object can do. They also mentioned that the type annotations are helpful when the code is poorly documented or part of a bigger codebase.

RQ 1.3: Overall, we find that the subjective measures are not significantly different when grouped by type annotations.

Result RQ 1

None of the behavioral, eye-tracking, or subjective measures could measure clear and statistically significant differences for the independent variable type annotations. However, we find that even though the type annotations did not significantly influence the measures, the participants overwhelmingly found the type annotations helpful and sometimes instrumental in comprehending the source code snippets. They also reported that type annotations are generally helpful. This suggests that type annotations are a good tool to help developers be more confident in their code comprehension. Many of the participants mentioned that they liked the type annotations because they were a good reminder of what the code was supposed to do and that they could more easily figure out certain parts of the code.

RQ 1: Overall, we can conclude that type annotations do not significantly influence program comprehension.

6.2 RESEARCH QUESTION 2

R 2.1: How does the combination of type annotations and identifier names influence behavioral measures?

As in RQ_1, we expect that participants would be faster and more correct when switching from obfuscated to meaningful identifier names and from non-annotated source code snippets to annotated source code snippets. Because the participant gained the most information, we expect that the combination of both independent variables would have the biggest influence. This can not be supported by the data collected. In the case of correctness, we observe that all values are similar to each other independent of the independent variables. The statistical test reported the same, and there is no statistically significant evidence that the combination of the two independent variables influences the correctness. The source code snippets' response time shows that switching from obfuscated to meaningful identifier names resulted in a significant difference. However, neither type annotations nor the combination of both independent variables resulted in a significant difference in the time needed. This suggests that even if the participants are forced to use bottom-up comprehension instead of top-down comprehension and are suspected to receive the most information through the type annotations, the completion time is not significantly influenced.

RQ 2.1: Overall, we find that the correctness is not significantly influenced by type annotations or identifier names while the time is significantly influenced by switching from bottom-up comprehension to top-down comprehension.

RQ 2.2: How does the combination of type annotations and identifier names influence the linearity of reading order?

Similar to the first research question, we expect that there is a difference between the different groups created by the independent variables. Similar to the first research question, we observe that no significant differences appear for any of the models. This is due to false-discovery rate correction, which removed two previously significant results. A possible reason for this is that the participants were not challenged enough by the source code snippets and that the type annotations were not as helpful as they could have been, which was partially by design. The source code snippets were chosen to be simple and easy to understand so as to not overwhelm the participants and to make sure that eye tracking could be used without scrolling.

RQ 2.2: Overall, we find that none of the eye-tracking measures are significantly influenced by the independent variables.

RQ 2.3: How does the combination of type annotations and identifier names influence the subjective code difficulty?

We expected that the participants would report a higher difficulty when switching from meaningful to obfuscated identifier names and from non-annotated to annotated source code snippets. These expectations could not be satisfied, and the null hypothesis, which was that there was no difference, could not be rejected. However, we saw that the reported difficulty (Figure 5.3) was overwhelmingly (> 70%) considered to be *Very Easy* or *Easy* for the meaningful identifier names. At the same time, for the obfuscated identifier names, we saw slightly more than 60% of the difficulty being *Very Easy* or *Easy*, while the categories *Difficult* and *Very Difficult* made up 10% of the reported difficulties for the not-annotated source code snippets, while they made up 20% for the annotated source code snippets. This shows that there is some indication that the source code snippets' subjective code difficulty is influenced by the type annotations and that the type annotations might make the source code snippets more difficult to comprehend if the identifier names are obfuscated, but this is not statistically significant. A reason for this could be that, especially with the longer snippets, the type annotations were distracting and made the source code snippets harder to comprehend.

Most of the participants reported that the type annotations were helpful in comprehending the source code snippets of this study and in general, even if we take into account the grouping for different identifier names.

RQ 2.3: The subjective measures are not significantly influenced by the combination of the independent variables identifier names and type annotations.

Result RQ 2

Both the behavioral measures and the subjective measures are not significantly influenced by the combination of the independent variables. Furthermore, the eye-tracking measures are not significantly influenced by the independent variables as well. As with RQ_1, the participants' opinion (subjective) is that type annotations are very helpful for program comprehension, while the objective measures do not show a significant difference. This points to a sense of security that is gained by the participants when they have the type annotations and meaningful identifier names available to them. This sense is only reflected in some measures and not in others, which might be because the source code snippets were too easy for the respective participants.

RQ 2: We can conclude that the combination of type annotations and identifier names does not influences program comprehension.

6.3 GENERAL DISCUSSION

From the results for both research questions and program comprehension, we conclude that type annotations are not as relevant for the developers in the way that they were presented in this study. For this study and the possibility of successfully using eye tracking, the type annotations had to be relatively simple and short to fit on one screen. This might have led to the type annotations not being as helpful as they could have been. This is especially suggested by the answers to our subjective questions. The overwhelming number of participants like type annotations and find them to make code easier, better to comprehend, and better to keep in mind what specific parts of the code do.

To avoid the problem of the source code snippets being too easy, we checked them with a pilot study and all participants within the pilot study showed that the source code snippets, even though they were solved at different speeds, they were still approachable for a wide range of participants. However, our main study participants showed a mean of 66.53 seconds and a standard deviation of 37.49 seconds, which is an indicator for a diverse group of source code snippets that were challenging to some while not as challenging to others.

Still, the participants claimed that out of all source code snippets, 70% or more were *Very Easy* or *Easy*. Thus, we can conclude that the participants were not challenged enough by the source code snippets, and we need to use more challenging source code snippets in the future. Considering that the participants are all in their bachelor's, the claim that all source code snippets were that easy is surprising. By revisiting the participant's demographics, we find a possible answer. On average, the participants were in their 5.54 (SD = 4.02) semester and had been programming for 5.11 (SD = 2.99) years. 29% of participants considered themselves to be more experienced than their classmates, while 82% reported themselves as students who are at least average in their experience or more experienced than their classmates. This suggests that, overall, the participants were more experienced than the

average student and that most of them were already at the end of their degree. This might be a reason why the source code snippets were not as challenging as they could have been.

SUMMARY All of this suggests that type annotations are intuitively important for program comprehension, while we were not able to objectively quantify this. We suspect that with more challenging source code snippets that use complex and composite types, the type annotations would be more helpful.

6.4 THREATS TO VALIDITY

In any empirical study, there is a need to recognize possible threats to the validity of the results. That is why we will discuss the possible threats to validity and explain how we designed the study to minimize them.

6.4.1 *Construct Validity*

Construct validity relates to how accurately the study measures what it is supposed to. In this study, the main constructural validity threats are matching fixations to the measures and whether the tasks' descriptions are clear and unambiguous.

MATCHING FIXATIONS The eye-tracking measures rely on a correct matching of a fixation to a line of code. This might be problematic if participants use peripheral vision and thus not focus on the line of code. Similar to Peitek et al. [23], we interpret a fixation to be on the same line of code if the fixation is less than 100 pixels away horizontally.

PARTCIPANTS AND QUESTIONS There is always room to interpret a posed question differently than intended. To avoid this threat, the questions were always written in the same way and formulated such that they could be answered by checking a multiple-choice box. To further avoid misunderstandings, the participants were given a warmup source code snippets with a question that was similar to the questions in the main study. Furthermore, one examiner was always present to answer any questions posed as long as they would not interfere with the data collection or the execution of the study.

6.4.2 *Internal Validity*

The internal validity concerns the extent to which the observed effect can be attributed to the independent variables rather than other factors. Here, the main threats are possible order and learning effects, setting up the environment, and selecting participants.

ORDER AND LEARNING EFFECTS To ensure that the participants' measurements had cause-and-effect relationships, it is crucial that there is no bias introduced by the selection of the source code snippets as well as the order in which they are presented. The source code snippets were selected from a large pool of program comprehension tasks and checked for feasibility through a pilot study. To reduce any learning and ordering effects for either

independent variable, each time a participant would arrive, they were randomly assigned to one group of identifier names and then provided with all 20 source code snippets in a completely randomized order, with each source code snippet being present exactly once. All source code snippets were then also randomly assigned whether to display the type annotations or not. The question was the same for each mutated source code snippet. This further reduces potential bias.

CONTROLLED ENVIRONMENT We further created a controlled environment at the chair by using a quiet room not used for anything else to not be disturbed. For all participants, the blinds were closed, and the light was turned on to ensure that the eye tracker could work correctly and that the participants would not let their eyes wander. We also made sure that the fatigue would not get too high by only running the study for 45 minutes. Furthermore, because eye tracking is a sensitive method, all participants were instructed beforehand how to sit and how to look at the screen. For each participant, the eye tracker was newly calibrated according to the instruction manual.

PARTICIPANTS To make sure that participants were not confused about how the study worked, they were read instructions beforehand, got additional information on the screen, and needed to complete a warmup source code snippet before the actual study started. Additionally, the participants might have a considerable difference in their programming knowledge. To reduce this threat, we only allowed bachelor students to participate in the study so as to keep the level similar. Through this, we can assume that all participants have a certain base level of programming knowledge while not yet being at the level of most professional developers.

6.4.3 *External Validity*

External validity is concerned with the generalizability of the results. In this study, the main threats are the generalizability of the participants and the source code snippets.

GENERALIZABILITY This study exhibits the same threats to external validity as all other studies that use students as participants and small source code snippets to test program comprehension. This includes the fact that the source code snippets can not necessarily be generalized to real-world projects and their size, as well as that the source code snippets are generally algorithmic and mono-method and thus not representative of all programming tasks. It also includes that the source code snippets were all written in Python and can thus not be generalized to other programming languages.

PARTICIPANTS The participants were all bachelor's students in computer science related course of study from Saarland University and can thus not be generalized to other universities, professional developers, or non computer scientists. Nonetheless, the results can carefully be generalized to other contextual factors, since they also targeted a relevant population.

7 CONCLUDING REMARKS

7.1 CONCLUSION

Inspired by the work of Peitek et al. [22], Busjahn et al. [9], and Siegmund et al. [32], we investigated the effect of type annotations and identifier names on Program Comprehension. We conducted an eye-tracking study with 27 participants who were asked to solve program comprehension tasks. Our results find that type annotations do not significantly affect the behavioral measures, eye-tracking measures, and reported difficulty. We observe the same for the combination of both type annotations and identifier names.

However, 89% of all participants reported that the type annotations helped with program comprehension within this study and in general. This indicates that the type annotations might positively affect the program comprehension but that the effect was not necessarily measurable with the given study setup.

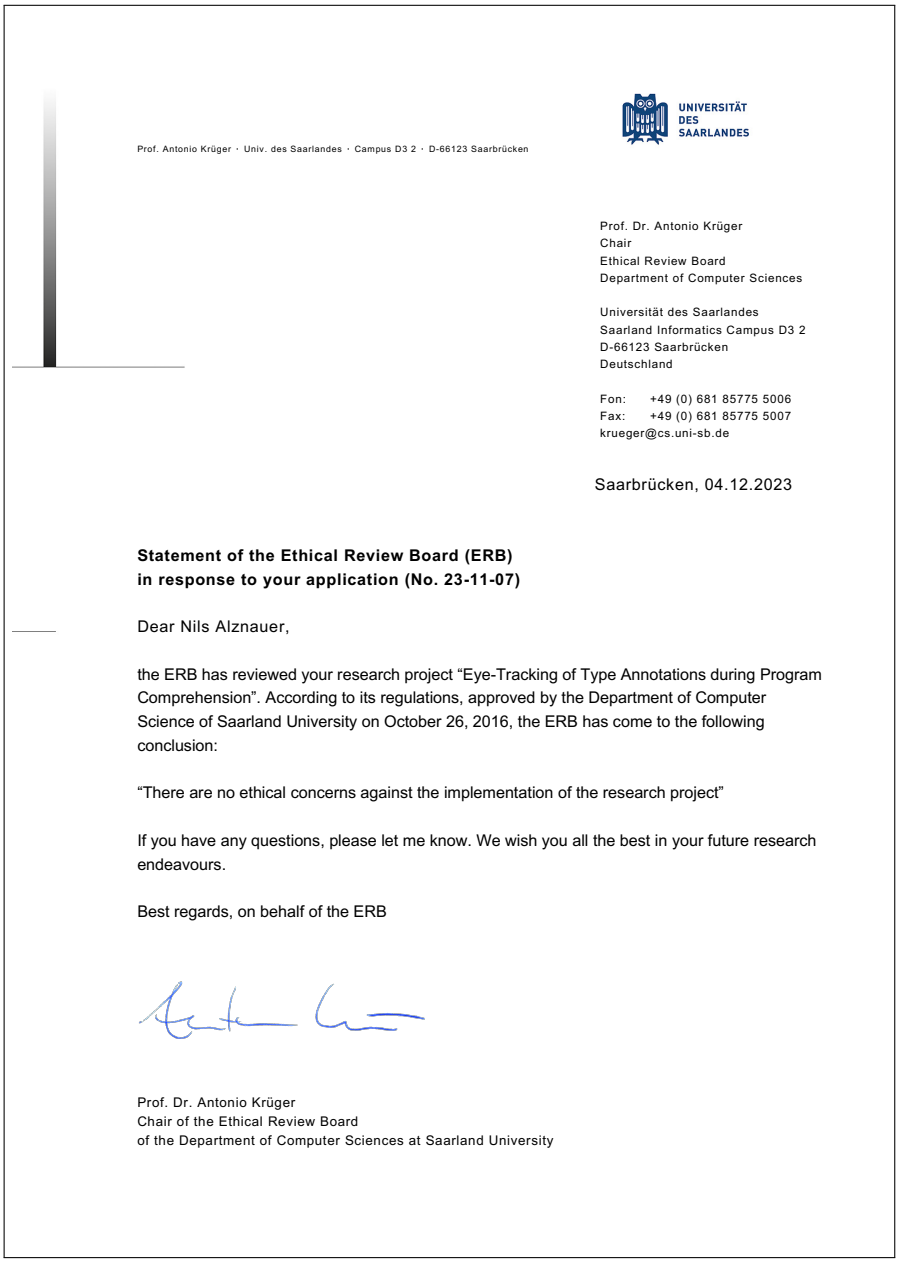
7.2 FUTURE WORK

In future studies, we aim to investigate the effect of type annotations on program comprehension in more detail. This includes opening up the study to a broader population to more easily generalize the study's findings. Furthermore, future studies could investigate source code snippets closer to real-world projects with bigger source code snippets that can include complex types and composite types to use the full range of type annotations. This could also entail that users create simple source code snippets based on some underlying codebase with and without type annotations. This would allow us to investigate the effect of type annotations on the creation of source code snippets and the comprehension of the underlying codebase.

Ultimately, this thesis lays the groundwork for a better understanding of the comprehension and the strategic use of type annotations in program comprehension and the software development process.

APPENDIX

A.1 ETHICAL-REVIEW BOARD



Prof. Antonio Krüger · Univ. des Saarlandes · Campus D3 2 · D-66123 Saarbrücken

UNIVERSITÄT DES SAARLANDES

Prof. Dr. Antonio Krüger
Chair
Ethical Review Board
Department of Computer Sciences

Universität des Saarlandes
Saarland Informatics Campus D3 2
D-66123 Saarbrücken
Deutschland

Fon: +49 (0) 681 85775 5006
Fax: +49 (0) 681 85775 5007
krueger@cs.uni-sb.de

Saarbrücken, 04.12.2023

Statement of the Ethical Review Board (ERB)
in response to your application (No. 23-11-07)

Dear Nils Alznauer,

the ERB has reviewed your research project “Eye-Tracking of Type Annotations during Program Comprehension”. According to its regulations, approved by the Department of Computer Science of Saarland University on October 26, 2016, the ERB has come to the following conclusion:

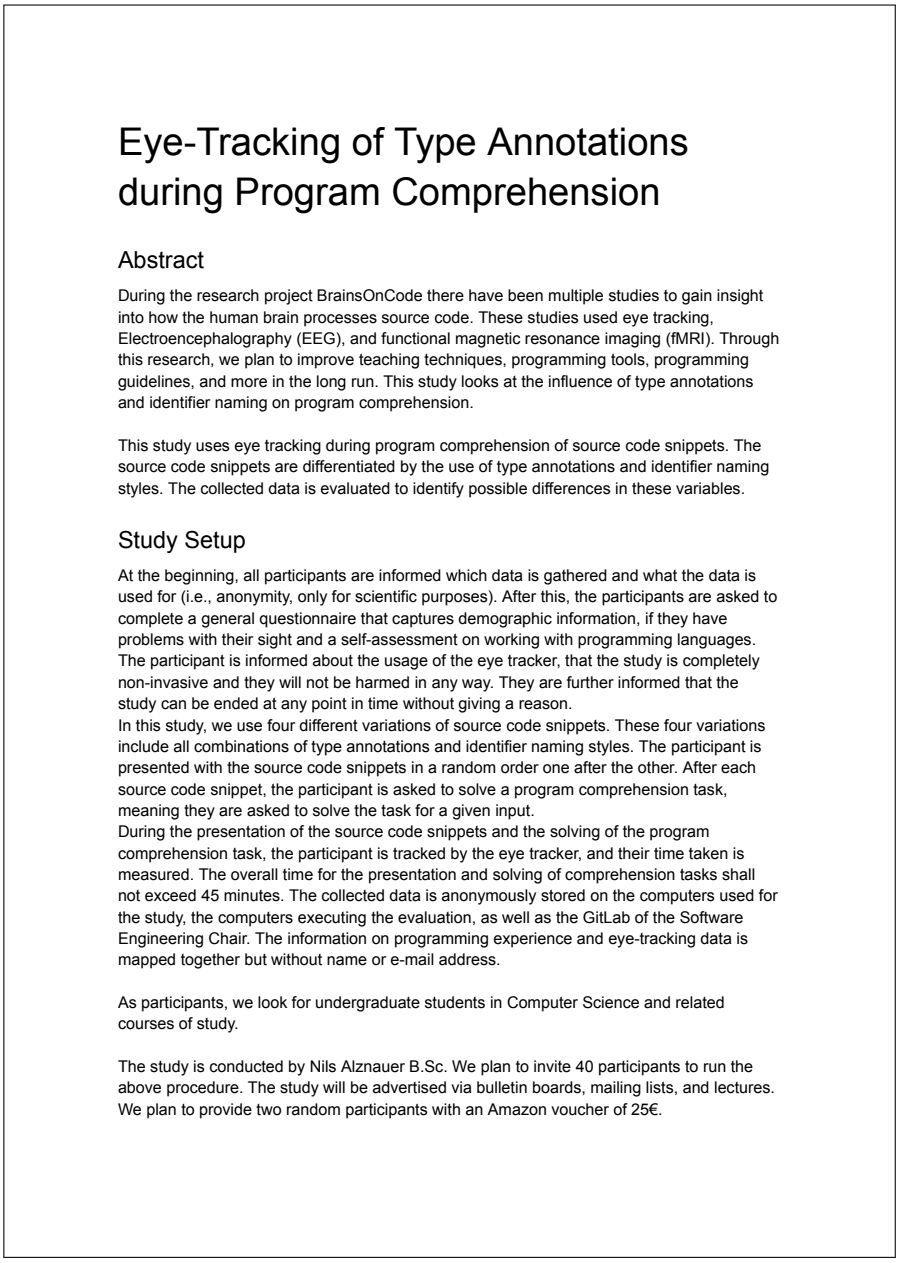
“There are no ethical concerns against the implementation of the research project”

If you have any questions, please let me know. We wish you all the best in your future research endeavours.

Best regards, on behalf of the ERB

Prof. Dr. Antonio Krüger
Chair of the Ethical Review Board
of the Department of Computer Sciences at Saarland University

Figure A.1: Confirmation from the Ethical Review Board on the implementation of the study design



Eye-Tracking of Type Annotations during Program Comprehension

Abstract

During the research project BrainsOnCode there have been multiple studies to gain insight into how the human brain processes source code. These studies used eye tracking, Electroencephalography (EEG), and functional magnetic resonance imaging (fMRI). Through this research, we plan to improve teaching techniques, programming tools, programming guidelines, and more in the long run. This study looks at the influence of type annotations and identifier naming on program comprehension.

This study uses eye tracking during program comprehension of source code snippets. The source code snippets are differentiated by the use of type annotations and identifier naming styles. The collected data is evaluated to identify possible differences in these variables.

Study Setup

At the beginning, all participants are informed which data is gathered and what the data is used for (i.e., anonymity, only for scientific purposes). After this, the participants are asked to complete a general questionnaire that captures demographic information, if they have problems with their sight and a self-assessment on working with programming languages. The participant is informed about the usage of the eye tracker, that the study is completely non-invasive and they will not be harmed in any way. They are further informed that the study can be ended at any point in time without giving a reason.
In this study, we use four different variations of source code snippets. These four variations include all combinations of type annotations and identifier naming styles. The participant is presented with the source code snippets in a random order one after the other. After each source code snippet, the participant is asked to solve a program comprehension task, meaning they are asked to solve the task for a given input.
During the presentation of the source code snippets and the solving of the program comprehension task, the participant is tracked by the eye tracker, and their time taken is measured. The overall time for the presentation and solving of comprehension tasks shall not exceed 45 minutes. The collected data is anonymously stored on the computers used for the study, the computers executing the evaluation, as well as the GitLab of the Software Engineering Chair. The information on programming experience and eye-tracking data is mapped together but without name or e-mail address.

As participants, we look for undergraduate students in Computer Science and related courses of study.

The study is conducted by Nils Alznauer B.Sc. We plan to invite 40 participants to run the above procedure. The study will be advertised via bulletin boards, mailing lists, and lectures. We plan to provide two random participants with an Amazon voucher of 25€.

Figure A.2: Summary of the study design handed in to the Ethical Review Board

SAARLAND UNIVERSITY
COMPUTER SCIENCE

Faculty of
Mathematics and Computer Science
Saarland University

Evaluation of research projects by the ethical review board of the Faculty of Mathematics and Computer Science, Saarland University

Basic Questionnaire

For each relevant study every executive researcher has to complete and sign this basic questionnaire, if an evaluation by the ethics commission is required. If the researcher is a student, the basic questionnaire must be additionally signed by the responsible supervisor.

Abbreviated designation of the study:

Eye-Tracking of Type Annotations during Program Comprehension

1. General data

This is a (series of) study(ies), which is (are) to be performed in the following context (please check):

- ☐ Study, e.g. internship or project
- ☐ Bachelor thesis
- ☒ Master thesis
- ☐ Dissertation
- ☐ Habilitation
- ☐ Research project (e.g. funded project)
- ☐ Other (please specify)

Executive researcher:

Name, given name(s): Nils Alznauer
Research group: Chair of Software Engineering
Email address: s9nialzn@stud.uni-saarland.de

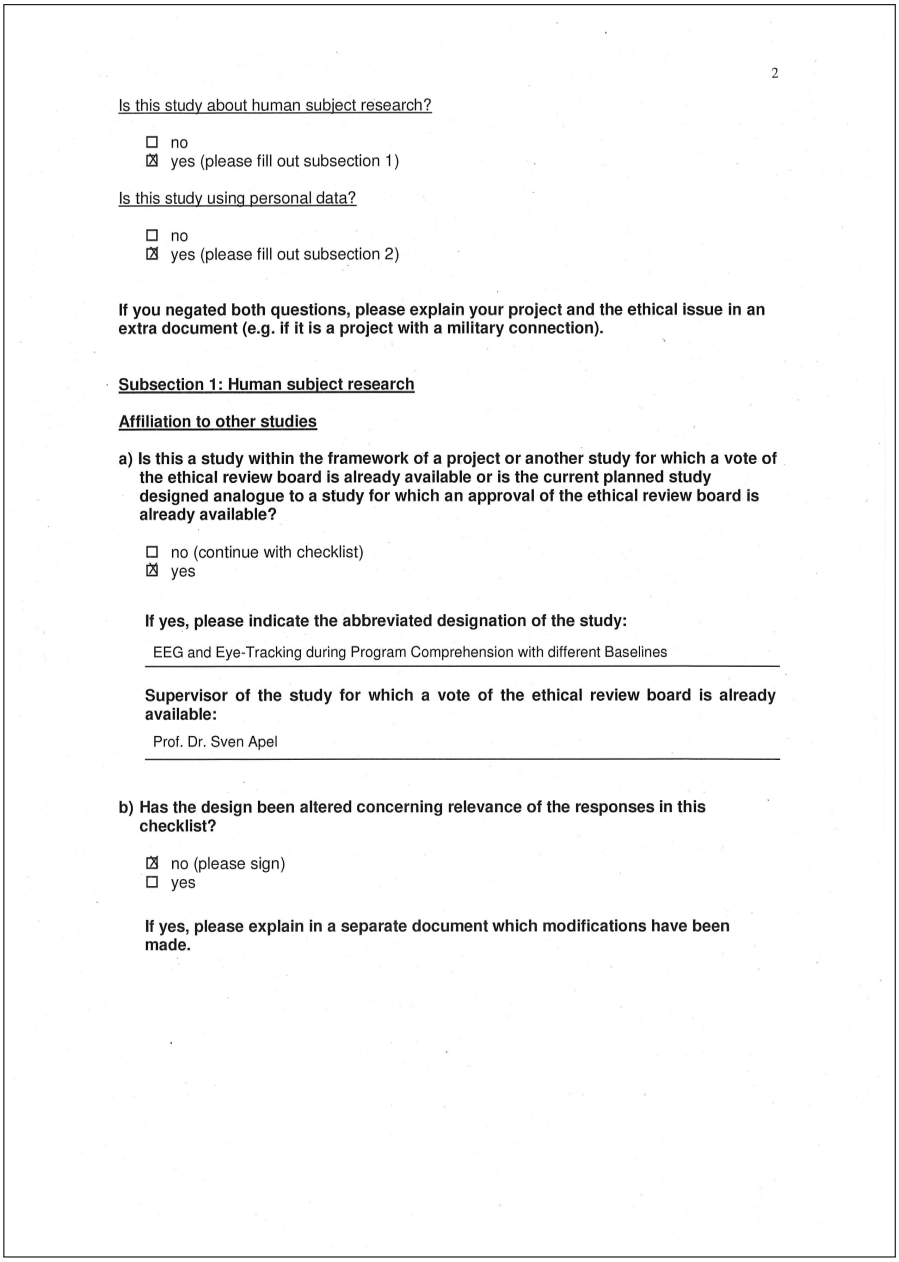
Status (please check):

- ☐ Student bachelor's degree
- ☒ Student master's degree
- ☐ Research assistant
- ☐ Other (please specify):

Responsible supervisor, if applicable:

Name, given name(s): Prof. Dr. Sven Apel
Research group: Chair of Software Engineering
Email address: apel@cs.uni-saarland.de

Figure A.3: Ethical Review Board Questionnaire page 1



2

Is this study about human subject research?

☐ no
☒ yes (please fill out subsection 1)

Is this study using personal data?

☐ no
☒ yes (please fill out subsection 2)

If you negated both questions, please explain your project and the ethical issue in an extra document (e.g. if it is a project with a military connection).

Subsection 1: Human subject research

Affiliation to other studies

a) Is this a study within the framework of a project or another study for which a vote of the ethical review board is already available or is the current planned study designed analogue to a study for which an approval of the ethical review board is already available?

☐ no (continue with checklist)
☒ yes

If yes, please indicate the abbreviated designation of the study:

EEG and Eye-Tracking during Program Comprehension with different Baselines

Supervisor of the study for which a vote of the ethical review board is already available:

Prof. Dr. Sven Apel

b) Has the design been altered concerning relevance of the responses in this checklist?

☒ no (please sign)
☐ yes

If yes, please explain in a separate document which modifications have been made.

Ethical Review Board Questionnaire page 2

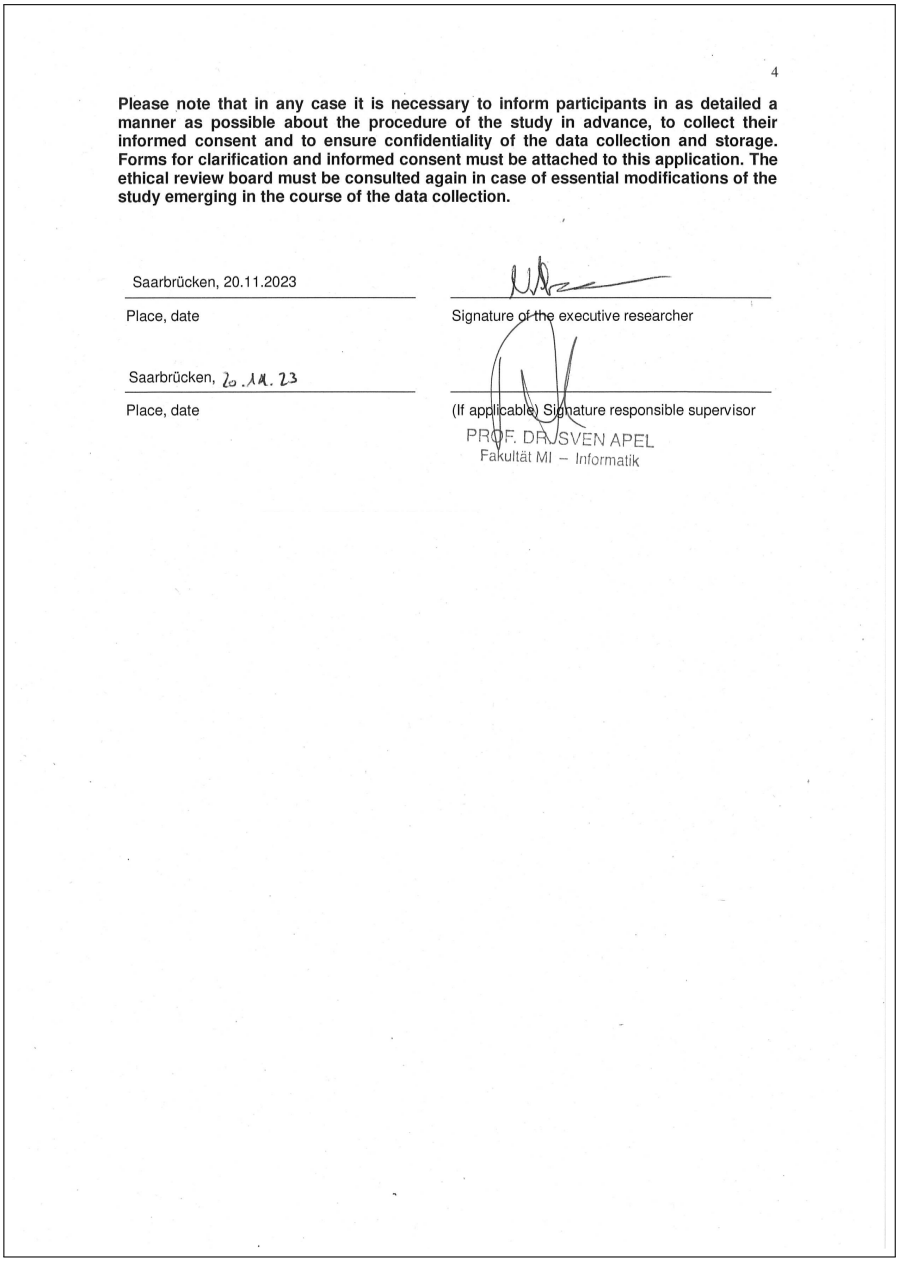
Checklist 1

Comments:
Detailed information on individual topics can be seen on the following website of the ethics guidelines of the German Psychological Society (DGPs):
http://www.dgps.de/index.php?id=185
If one or more of questions 1–9 on the checklist have been answered with yes, please describe the study scheme in a separate document and particularly go into detail about the necessity of the point(s) answered with yes. Please go also into detail about how you will ensure compliance with the ethics guidelines regarding these point(s).

	yes	no
1. Does the study involve participants who are unable to give informed consent (e.g. people under the age of 18 or people unable to consent in a legal sense)?		X
2. Does the study involve participants who belong to a particularly vulnerable group (e.g. participants of clinical samples, people with learning disabilities, residents of a hospital or nursing home or people serving a sentence)?		X
3. Is it necessary that people participate without being informed about their participation or without having given informed consent (e.g. covert observation) at this point?		X
4. Is it necessary that the participants are not entirely informed about purpose and content of the study? (Remark: the entire information does not imply the disclosure of the hypothesis but refers to the purpose and procedure of the study. For example, an incomplete or false information is on hand if a cover story is necessary to be able to address the questioning).		X
5. Is it necessary actively to mislead people concerning the purpose of the study?		X
6. Is it necessary to ask questions on subjects of an intimate nature for the respondents or the answer to which could be conceived as stigmatizing (e.g. relating to illegal or deviant behavior)?		X
7. Is it expected that participants are going to suffer from physiological stress, anxiety, exhaustion, physical pain or other negative effects beyond the anticipated everyday life dimension?		X
8. Does the study involve the administration of medicine, placebo or any other substances?		X
9. Will the participants be subject to any invasive or potentially harmful procedures?		X
10. Will personal data be collected which cannot be processed anonymously (e.g. video or audio recordings of the participants, collection of body substances such as saliva samples)? If yes, please specify what kind of data: _____ Will the participants be informed about this? ☐ yes ☐ no May the participants demand the deletion/destruction of this data at any time and will they be informed about this? ☐ yes ☐ no Will this data only be stored internally? ☐ yes ☐ no		X

If question 10 has been answered with yes, please answer directly the corresponding additional questions. If in question 10, one or both additional question(s) have been answered with no, please describe in a separate document why this is necessary and how you will ensure compliance with the ethics guidelines regarding these point(s).

Ethical Review Board Questionnaire page 3



4

Please note that in any case it is necessary to inform participants in as detailed a manner as possible about the procedure of the study in advance, to collect their informed consent and to ensure confidentiality of the data collection and storage. Forms for clarification and informed consent must be attached to this application. The ethical review board must be consulted again in case of essential modifications of the study emerging in the course of the data collection.

Saarbrücken, 20.11.2023 — Signature of the executive researcher

Place, date

Saarbrücken, 20.11.23 — (If applicable) Signature responsible supervisor

Place, date

PROF. DR. SVEN APEL
Fakultät MI – Informatik

Ethical Review Board Questionnaire page 4

5

If this study is using personal data:

Subsection 2

	yes	no
1. Can you ensure that all personal data is processed and stored anonymously?	X	
2. Did you check if the quality and extent of used data is appropriate according to the study (economical use)?	X	
3. Did you check if the data will be used legally?	X	
4. Will personal data be collected which can't be processed anonymously (e.g. video / audio recordings of participants, removal of body substances like salvia sample)? If yes, which data: _____ Will the participants be informed about this? ☐ yes ☐ no Can the participants demand a disposal of this data and will they be informed about this as well? ☐ yes ☐ no Will this data only be stored internally? ☐ yes ☐ no		X
5. Does the research contain the processing of already collected personal data?		X

I certify that all information in this application is accurate to the best of my knowledge.

If you negated one or more of the questions 1-3 of this subsection, please describe in a separate document the issue which prevents an answer with yes. Please consider for this purpose the data protection law of Saarland:
https://recht.saarland.de/bssl/document/jlr-DSGSL2018rahmen

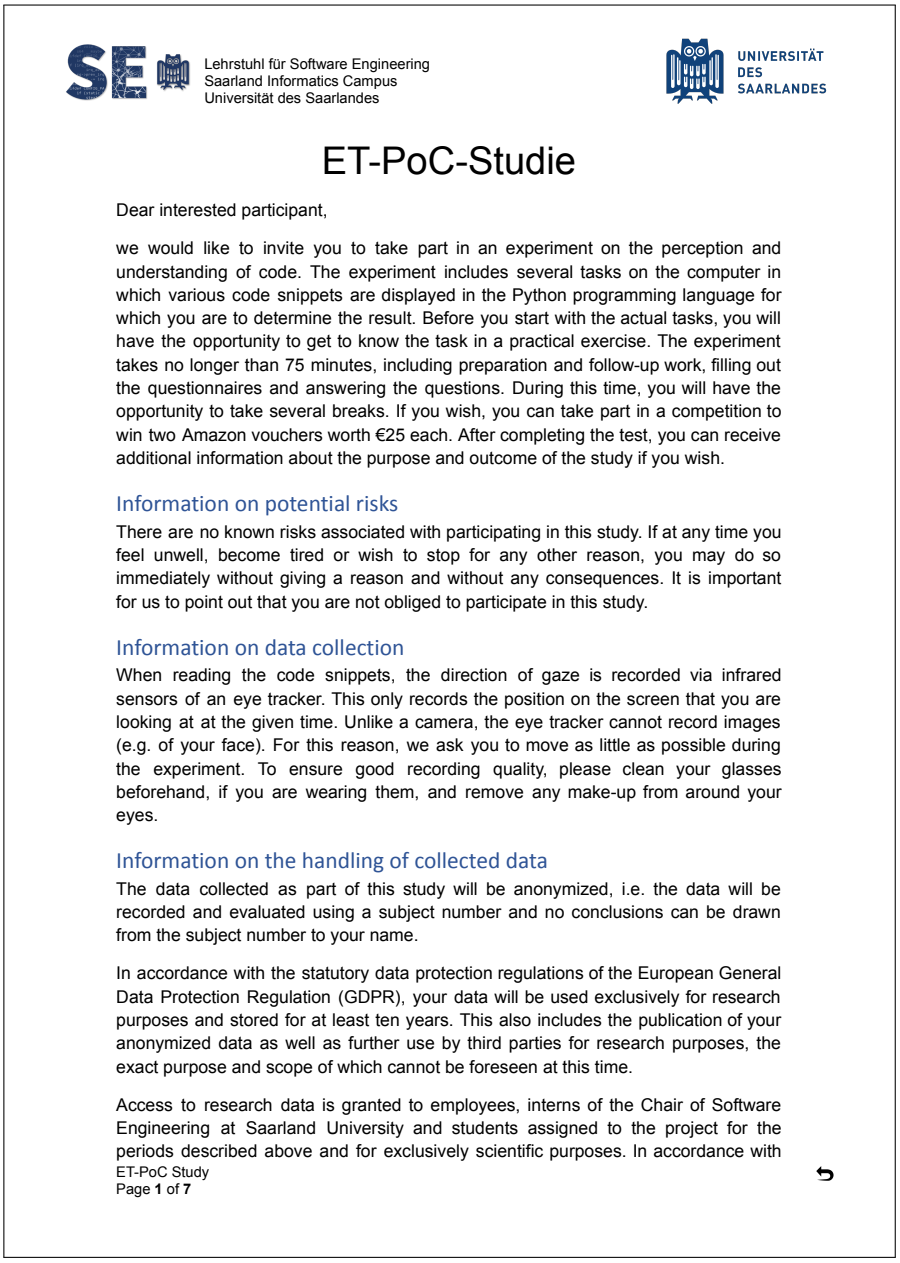
Saarbrücken, 20.11.2023
Place, date

Signature of the executive researcher

Saarbrücken, 20.11.2023
Place, date

If applicable) Signature responsible supervisor
PROF. DR. SVEN APEL
Fakultät MI – Informatik

Ethical Review Board Questionnaire page 5



Lehrstuhl für Software Engineering
Saarland Informatics Campus
Universität des Saarlandes

UNIVERSITÄT DES SAARLANDES

ET-PoC-Studie

Dear interested participant,

we would like to invite you to take part in an experiment on the perception and understanding of code. The experiment includes several tasks on the computer in which various code snippets are displayed in the Python programming language for which you are to determine the result. Before you start with the actual tasks, you will have the opportunity to get to know the task in a practical exercise. The experiment takes no longer than 75 minutes, including preparation and follow-up work, filling out the questionnaires and answering the questions. During this time, you will have the opportunity to take several breaks. If you wish, you can take part in a competition to win two Amazon vouchers worth €25 each. After completing the test, you can receive additional information about the purpose and outcome of the study if you wish.

Information on potential risks

There are no known risks associated with participating in this study. If at any time you feel unwell, become tired or wish to stop for any other reason, you may do so immediately without giving a reason and without any consequences. It is important for us to point out that you are not obliged to participate in this study.

Information on data collection

When reading the code snippets, the direction of gaze is recorded via infrared sensors of an eye tracker. This only records the position on the screen that you are looking at at the given time. Unlike a camera, the eye tracker cannot record images (e.g. of your face). For this reason, we ask you to move as little as possible during the experiment. To ensure good recording quality, please clean your glasses beforehand, if you are wearing them, and remove any make-up from around your eyes.

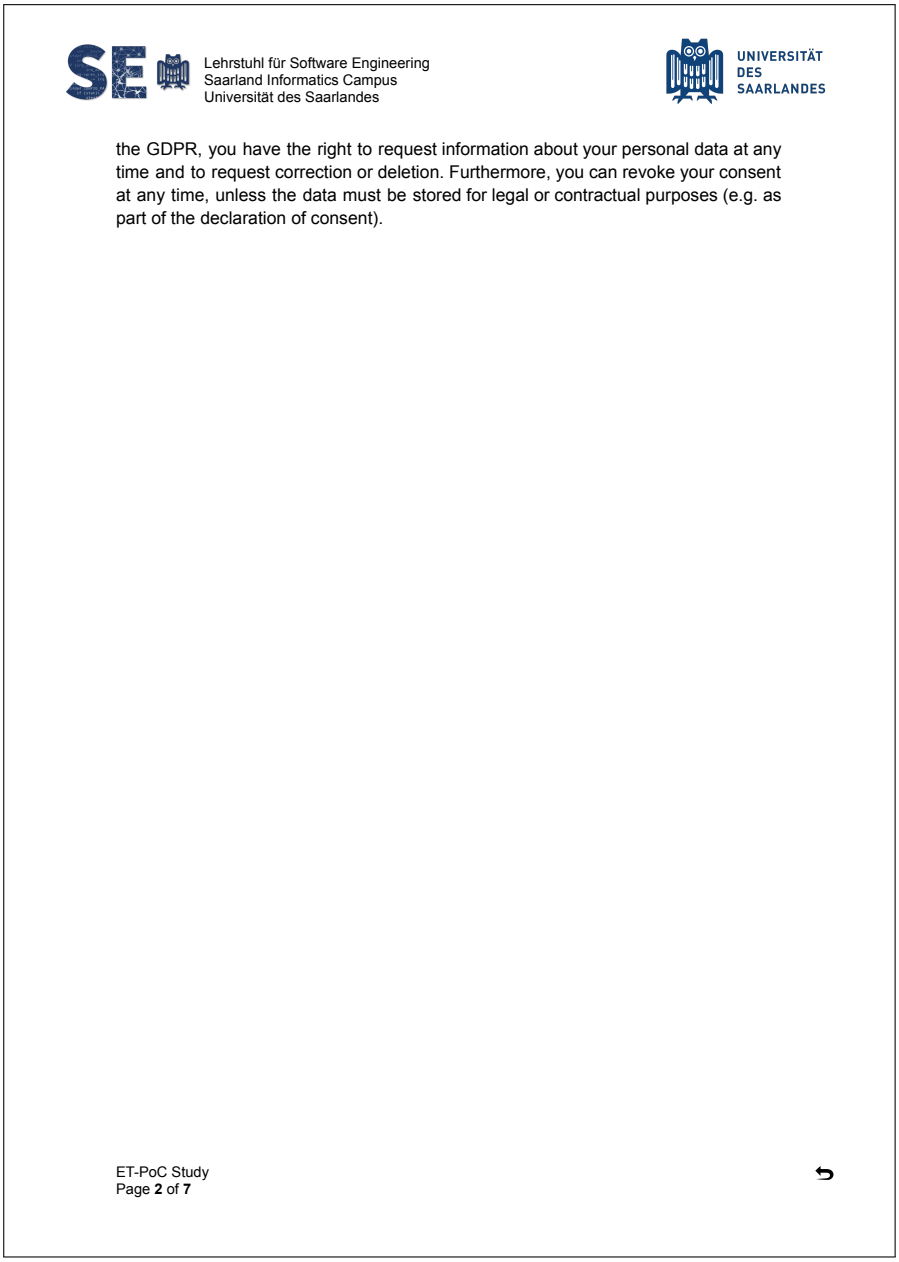
Information on the handling of collected data

The data collected as part of this study will be anonymized, i.e. the data will be recorded and evaluated using a subject number and no conclusions can be drawn from the subject number to your name.

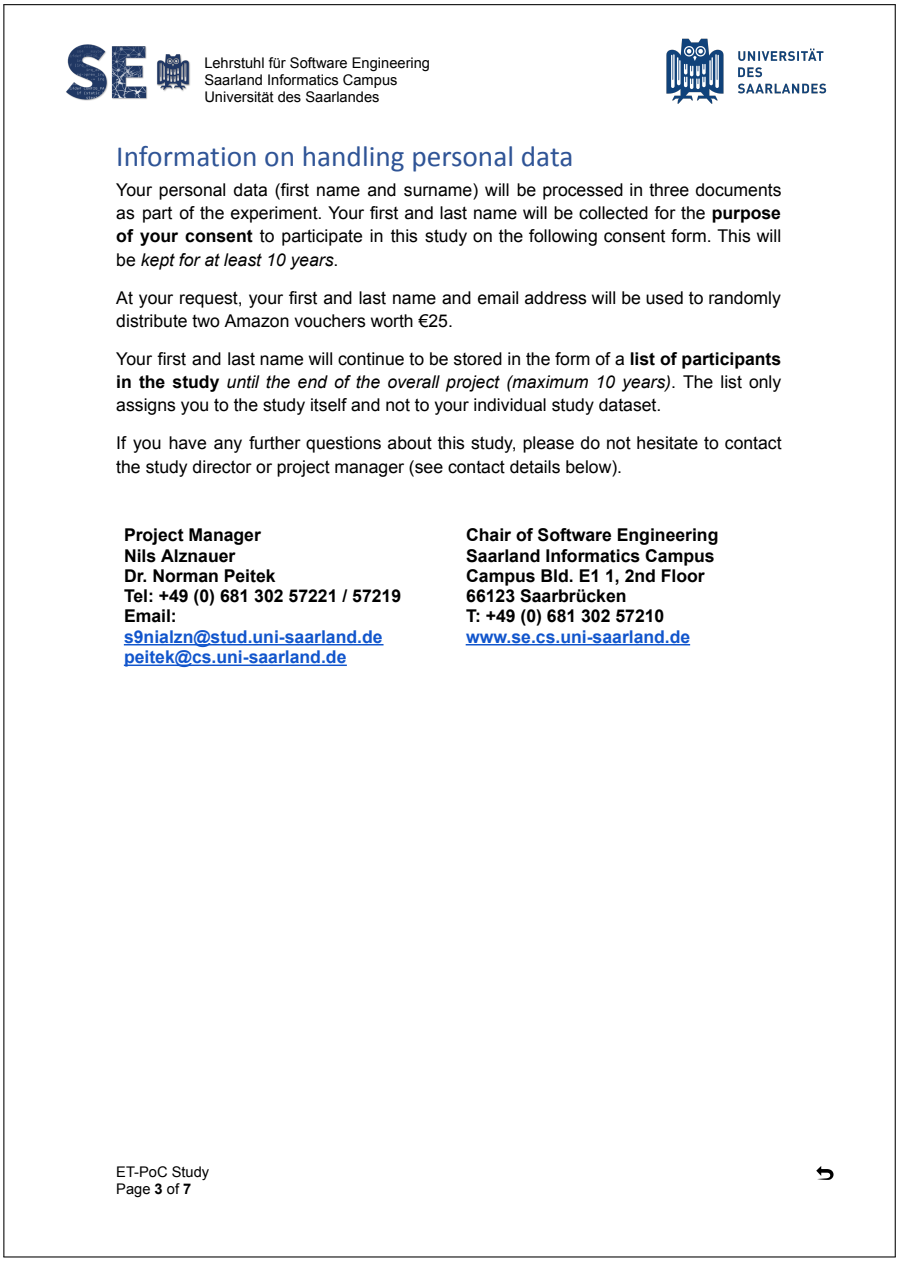
In accordance with the statutory data protection regulations of the European General Data Protection Regulation (GDPR), your data will be used exclusively for research purposes and stored for at least ten years. This also includes the publication of your anonymized data as well as further use by third parties for research purposes, the exact purpose and scope of which cannot be foreseen at this time.

Access to research data is granted to employees, interns of the Chair of Software Engineering at Saarland University and students assigned to the project for the periods described above and for exclusively scientific purposes. In accordance with

Figure A.4: Consent form for the study page 1



Consent form for the study page 2



Lehrstuhl für Software Engineering
Saarland Informatics Campus
Universität des Saarlandes

UNIVERSITÄT DES SAARLANDES

Information on handling personal data

Your personal data (first name and surname) will be processed in three documents as part of the experiment. Your first and last name will be collected for the **purpose of your consent** to participate in this study on the following consent form. This will be *kept for at least 10 years*.

At your request, your first and last name and email address will be used to randomly distribute two Amazon vouchers worth €25.

Your first and last name will continue to be stored in the form of a **list of participants in the study** *until the end of the overall project (maximum 10 years)*. The list only assigns you to the study itself and not to your individual study dataset.

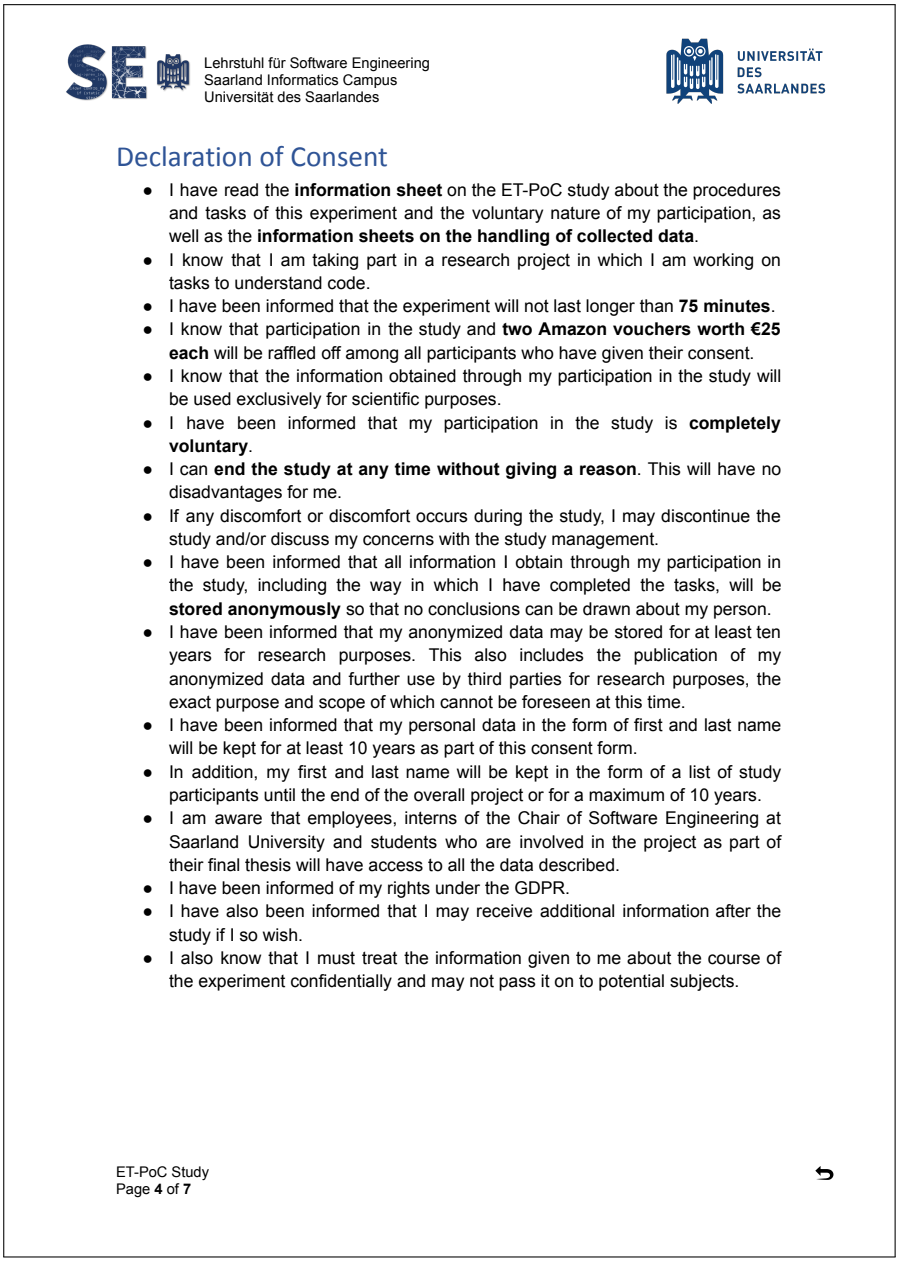
If you have any further questions about this study, please do not hesitate to contact the study director or project manager (see contact details below).

Project Manager
Nils Alznauer
Dr. Norman Peitek
Tel: +49 (0) 681 302 57221 / 57219
Email:
s9nialzn@stud.uni-saarland.de
peitek@cs.uni-saarland.de

Chair of Software Engineering
Saarland Informatics Campus
Campus Bld. E1 1, 2nd Floor
66123 Saarbrücken
T: +49 (0) 681 302 57210
www.se.cs.uni-saarland.de

ET-PoC Study
Page **3** of **7**

Consent form for the study page 3



Lehrstuhl für Software Engineering
Saarland Informatics Campus
Universität des Saarlandes

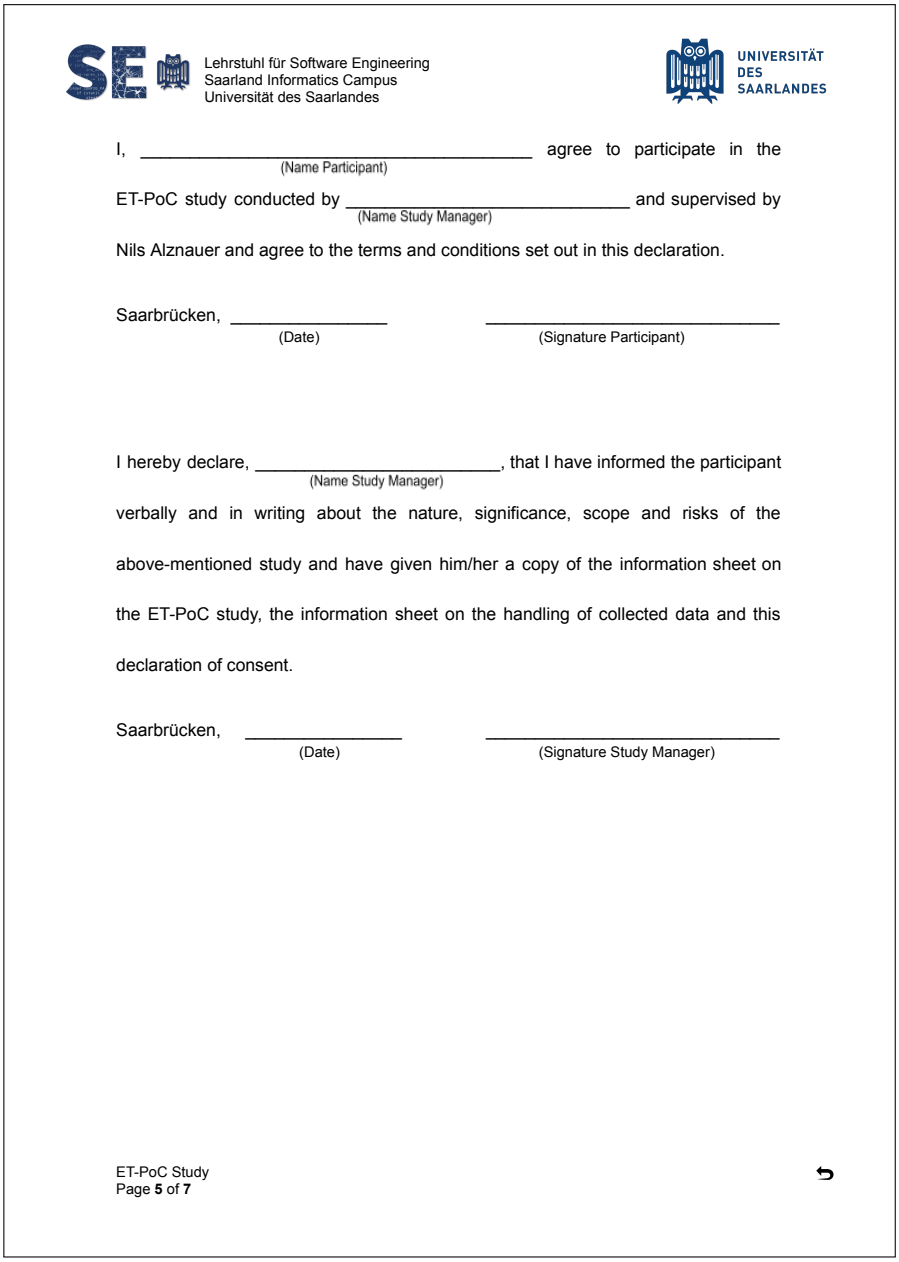
UNIVERSITÄT DES SAARLANDES

Declaration of Consent

- I have read the **information sheet** on the ET-PoC study about the procedures and tasks of this experiment and the voluntary nature of my participation, as well as the **information sheets on the handling of collected data**.
- I know that I am taking part in a research project in which I am working on tasks to understand code.
- I have been informed that the experiment will not last longer than **75 minutes**.
- I know that participation in the study and **two Amazon vouchers worth €25 each** will be raffled off among all participants who have given their consent.
- I know that the information obtained through my participation in the study will be used exclusively for scientific purposes.
- I have been informed that my participation in the study is **completely voluntary**.
- I can **end the study at any time without giving a reason**. This will have no disadvantages for me.
- If any discomfort or discomfort occurs during the study, I may discontinue the study and/or discuss my concerns with the study management.
- I have been informed that all information I obtain through my participation in the study, including the way in which I have completed the tasks, will be **stored anonymously** so that no conclusions can be drawn about my person.
- I have been informed that my anonymized data may be stored for at least ten years for research purposes. This also includes the publication of my anonymized data and further use by third parties for research purposes, the exact purpose and scope of which cannot be foreseen at this time.
- I have been informed that my personal data in the form of first and last name will be kept for at least 10 years as part of this consent form.
- In addition, my first and last name will be kept in the form of a list of study participants until the end of the overall project or for a maximum of 10 years.
- I am aware that employees, interns of the Chair of Software Engineering at Saarland University and students who are involved in the project as part of their final thesis will have access to all the data described.
- I have been informed of my rights under the GDPR.
- I have also been informed that I may receive additional information after the study if I so wish.
- I also know that I must treat the information given to me about the course of the experiment confidentially and may not pass it on to potential subjects.

ET-PoC Study
Page **4** of **7**

Consent form for the study page 4



Lehrstuhl für Software Engineering
Saarland Informatics Campus
Universität des Saarlandes

UNIVERSITÄT DES SAARLANDES

I, _____ agree to participate in the
(Name Participant)

ET-PoC study conducted by _____ and supervised by
(Name Study Manager)

Nils Alznauer and agree to the terms and conditions set out in this declaration.

Saarbrücken, _____ _____
(Date) (Signature Participant)

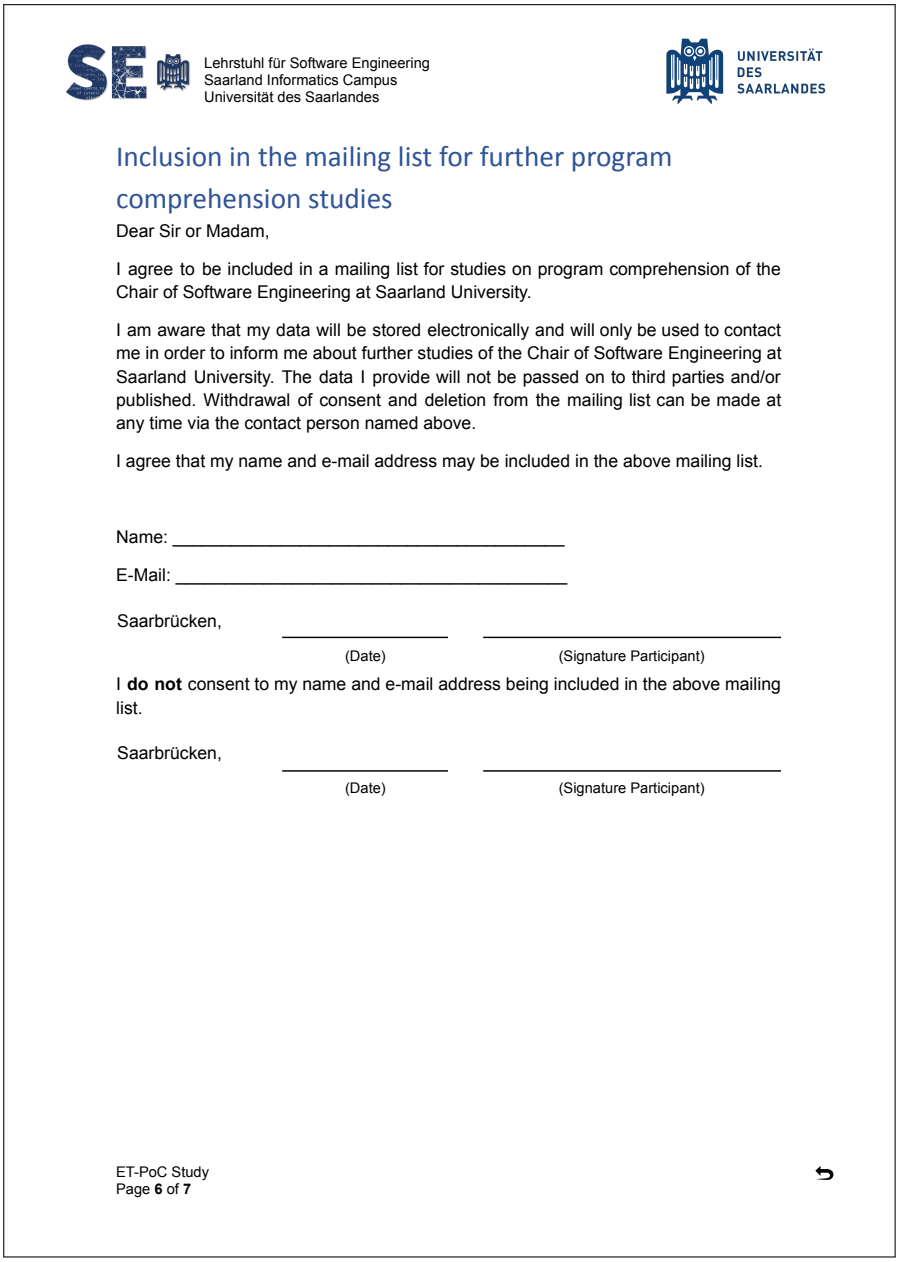
I hereby declare, _____, that I have informed the participant
(Name Study Manager)

verbally and in writing about the nature, significance, scope and risks of the above-mentioned study and have given him/her a copy of the information sheet on the ET-PoC study, the information sheet on the handling of collected data and this declaration of consent.

Saarbrücken, _____ _____
(Date) (Signature Study Manager)

ET-PoC Study
Page **5** of **7**

Consent form for the study page 5



Lehrstuhl für Software Engineering
Saarland Informatics Campus
Universität des Saarlandes

UNIVERSITÄT DES SAARLANDES

Inclusion in the mailing list for further program comprehension studies

Dear Sir or Madam,

I agree to be included in a mailing list for studies on program comprehension of the Chair of Software Engineering at Saarland University.

I am aware that my data will be stored electronically and will only be used to contact me in order to inform me about further studies of the Chair of Software Engineering at Saarland University. The data I provide will not be passed on to third parties and/or published. Withdrawal of consent and deletion from the mailing list can be made at any time via the contact person named above.

I agree that my name and e-mail address may be included in the above mailing list.

Name: _____

E-Mail: _____

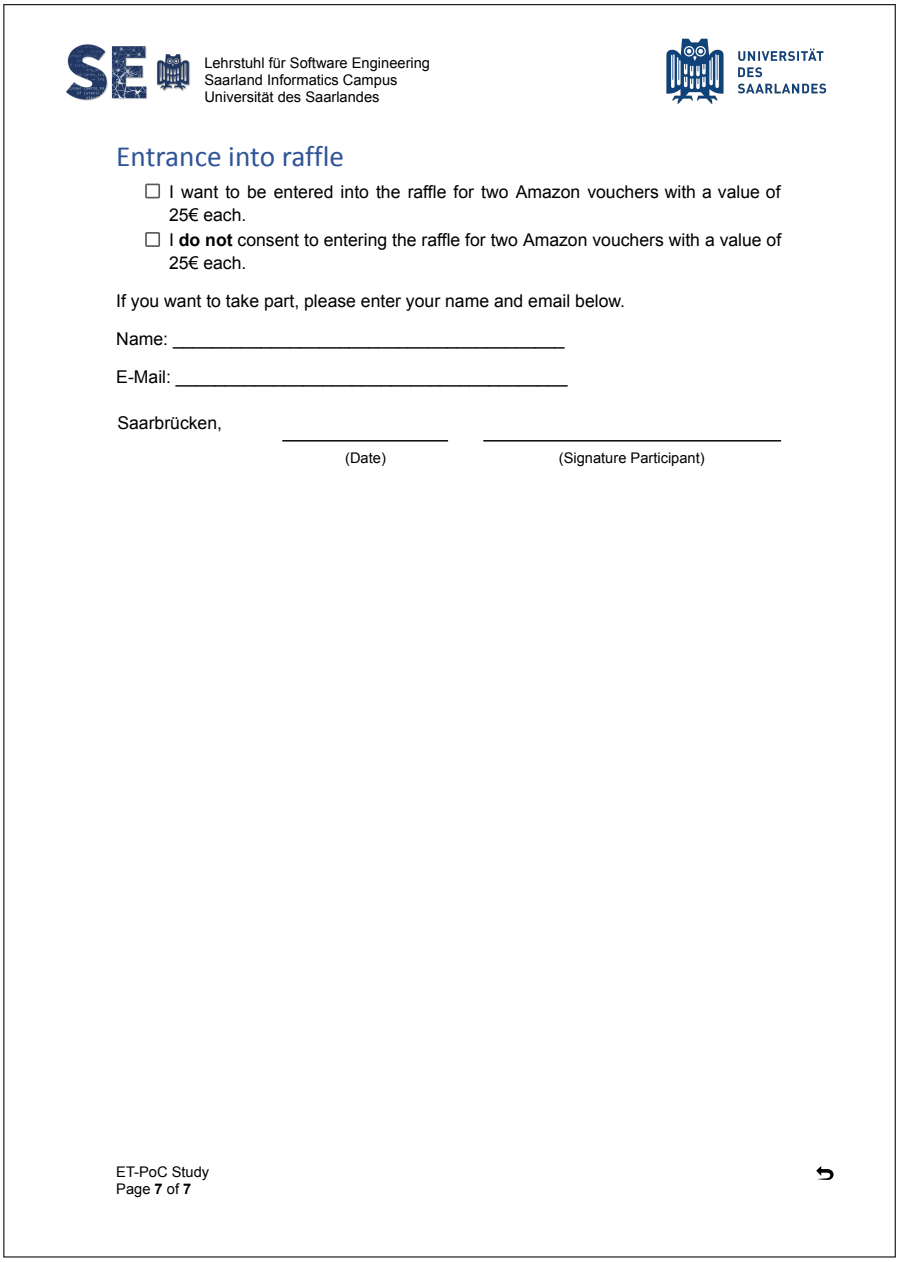
Saarbrücken, _____ _____
(Date) (Signature Participant)

I **do not** consent to my name and e-mail address being included in the above mailing list.

Saarbrücken, _____ _____
(Date) (Signature Participant)

ET-PoC Study
Page **6** of **7**

Consent form for the study page 6



Lehrstuhl für Software Engineering
Saarland Informatics Campus
Universität des Saarlandes

UNIVERSITÄT DES SAARLANDES

Entrance into raffle

☐ I want to be entered into the raffle for two Amazon vouchers with a value of 25€ each.

☐ I **do not** consent to entering the raffle for two Amazon vouchers with a value of 25€ each.

If you want to take part, please enter your name and email below.

Name: _____

E-Mail: _____

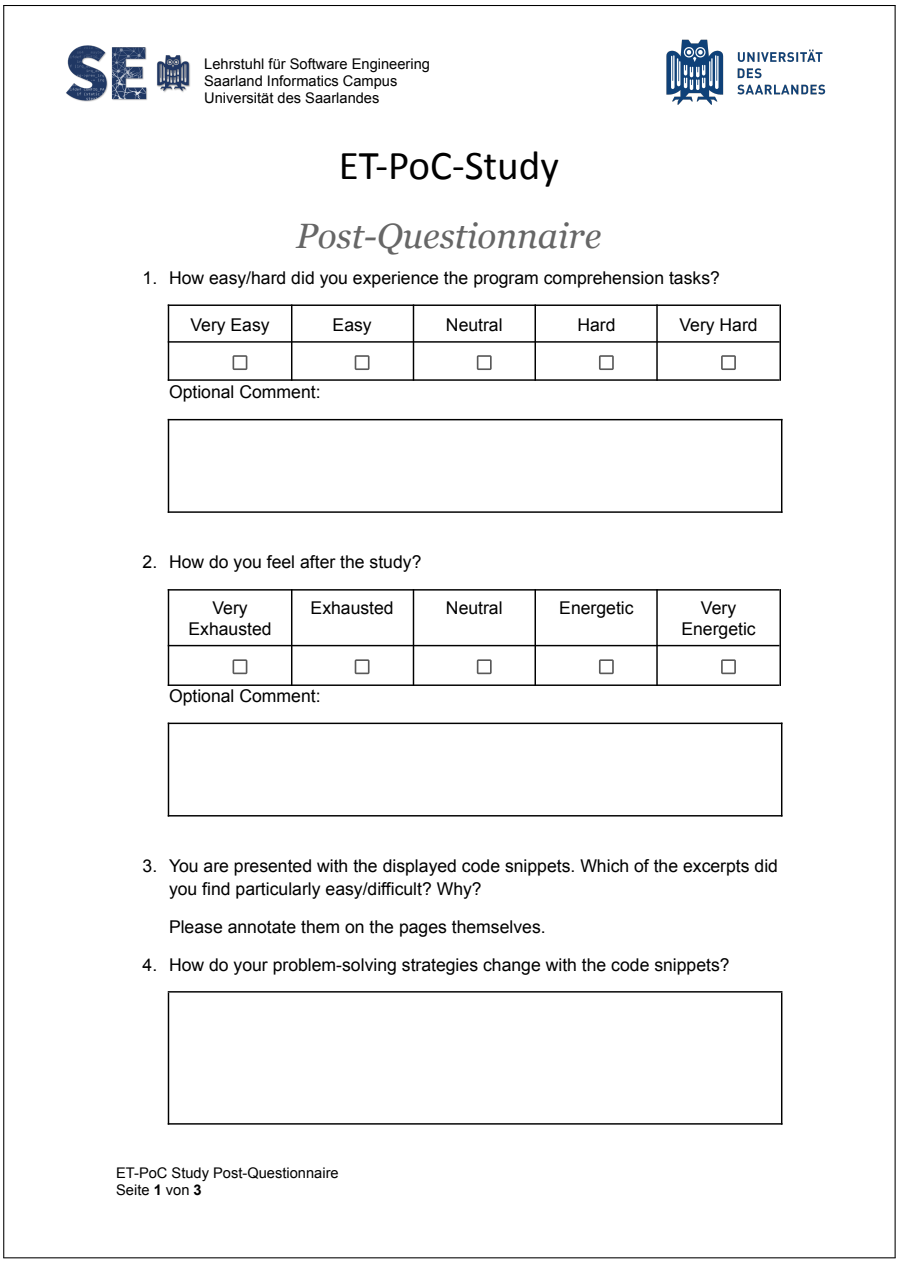
Saarbrücken, _____ _____

(Date) (Signature Participant)

ET-PoC Study
Page **7** of **7**

Consent form for the study page 7

A.2 POSTQUESTIONNAIRE



Lehrstuhl für Software Engineering
Saarland Informatics Campus
Universität des Saarlandes

UNIVERSITÄT DES SAARLANDES

ET-PoC-Study

Post-Questionnaire

1. How easy/hard did you experience the program comprehension tasks?

Very Easy	Easy	Neutral	Hard	Very Hard
☐	☐	☐	☐	☐

Optional Comment:

2. How do you feel after the study?

Very Exhausted	Exhausted	Neutral	Energetic	Very Energetic
☐	☐	☐	☐	☐

Optional Comment:

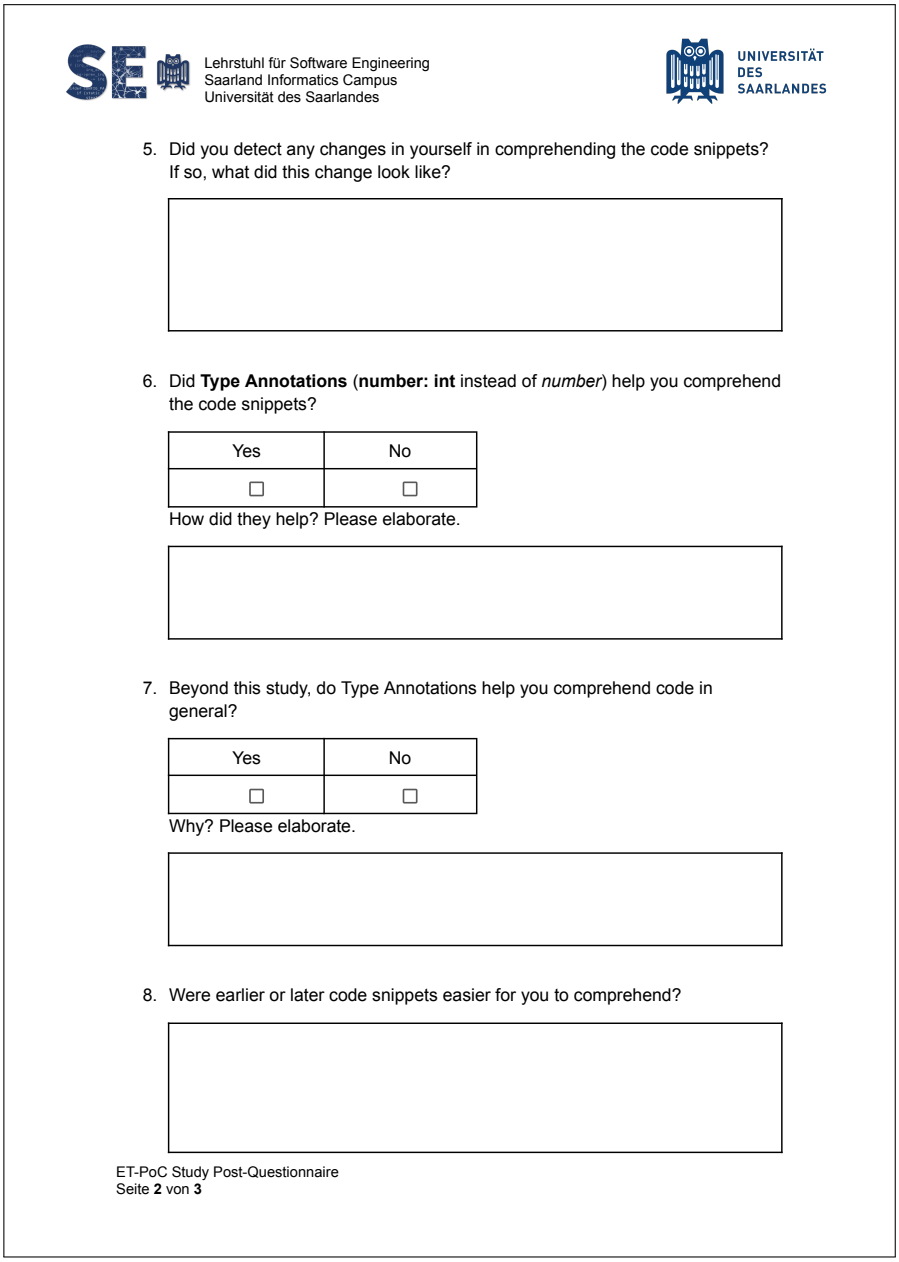
3. You are presented with the displayed code snippets. Which of the excerpts did you find particularly easy/difficult? Why?

   Please annotate them on the pages themselves.

4. How do your problem-solving strategies change with the code snippets?

ET-PoC Study Post-Questionnaire
Seite **1** von **3**

Figure A.5: Post-Questionnaire for the study page 1



Lehrstuhl für Software Engineering
Saarland Informatics Campus
Universität des Saarlandes

UNIVERSITÄT DES SAARLANDES

5. Did you detect any changes in yourself in comprehending the code snippets? If so, what did this change look like?

6. Did **Type Annotations** (**number: int** instead of *number*) help you comprehend the code snippets?

Yes	No
☐	☐

How did they help? Please elaborate.

7. Beyond this study, do Type Annotations help you comprehend code in general?

Yes	No
☐	☐

Why? Please elaborate.

8. Were earlier or later code snippets easier for you to comprehend?

ET-PoC Study Post-Questionnaire
Seite **2** von **3**

Post-Questionnaire for the study page 2

A.3 SOURCE CODE SNIPPETS

All non-annotated with obfuscated identifier names and annotated with meaningful identifier names source code snippets. The remaining groups can be inferred from these source code snippets by either removing the annotations or the obfuscation.

Listing A.1: Source Code Snippets arrayAverage non-annotated and obfuscated identifier names

```
def compute(abcdefg):
    hijklmn = 0
    opq = 0
    while hijklmn < len(abcdefg):
        opq = opq + abcdefg[hijklmn]
        hijklmn = hijklmn + 1
    rstuvw = opq / hijklmn
    return rstuvw
```

Listing A.2: Source Code Snippets arrayAverage annotated and meaningful identifier names

```
def compute(numbers: list[int]) -> float:
    counter: int = 0
    sum: int = 0
    while counter < len(numbers):
        sum = sum + numbers[counter]
        counter = counter + 1
    result: float = sum / counter
    return result
```

Listing A.3: Source Code Snippets binarySearch non-annotated and obfuscated identifier names

```
def compute(abcde, fgh):
    ijklmn = 0
    opqrst = len(abcde) - 1
    while ijklmn <= opqrst:
        u = (ijklmn + opqrst) // 2
        if fgh < abcde[u]:
            opqrst = u - 1
        elif fgh > abcde[u]:
            ijklmn = u + 1
        else:
            return u
    return -1
```

Listing A.4: Source Code Snippets binarySearch annotated and meaningful identifier names

```
def compute(array: list[int], key: int) -> int:
    index1: int = 0
    index2: int = len(array) - 1
    while index1 <= index2:
        m: int = (index1 + index2) // 2
        if key < array[m]:
            index2 = m - 1
        elif key > array[m]:
            index1 = m + 1
        else:
            return m
    return -1
```

Listing A.5: Source Code Snippets binaryToDecimal non-annotated and obfuscated identifier names

```
def compute(abcdef):
    if abcdef == "0":
        return 0
    if abcdef == "1":
        return 1
    if abcdef[-1] == "0":
        return 2 * compute(abcdef[:-1])
    if abcdef[-1] == "1":
        return 1 + 2 * compute(abcdef[:-1])
    return -1
```

Listing A.6: Source Code Snippets binaryToDecimal annotated and meaningful identifier names

```
def compute(number: str) -> int:
    if number == "0":
        return 0
    if number == "1":
        return 1
    if number[-1] == "0":
        return 2 * compute(number[:-1])
    if number[-1] == "1":
        return 1 + 2 * compute(number[:-1])
    return -1
```

Listing A.7: Source Code Snippets bubbleSort non-annotated and obfuscated identifier names

```
def compute(abcde):
    for f in range(len(abcde)):
        for g in range(f, 0, -1):
            if abcde[g-1] > abcde[g]:
                abcde[g-1], abcde[g] = abcde[g], abcde[g-1]
    return abcde
```

Listing A.8: Source Code Snippets bubbleSort annotated and meaningful identifier names

```
def compute(array: list[int]) -> list[int]:
    for i in range(len(array)):
        for j in range(i, 0, -1):
            if array[j-1] > array[j]:
                array[j-1], array[j] = array[j], array[j-1]
    return array
```

Listing A.9: Source Code Snippets capitalizeFirstLetter non-annotated and obfuscated identifier names

```
def compute(abcdefgh):
    ijklmn = []
    opqrs = abcdefgh.split()
    for t in range(len(opqrs)):
        if t > 0:
            ijklmn.append(" ")
        ijklmn.append(opqrs[t][0].upper() + opqrs[t][1:])
    return "".join(ijklmn)
```

Listing A.10: Source Code Snippets capitalizeFirstLetter annotated and meaningful identifier names

```
def compute(sentence: str) -> str:
    result: list[int] = []
    words: list[str] = sentence.split()
    for i in range(len(words)):
        if i > 0:
            result.append(" ")
        result.append(words[i][0].upper() + words[i][1:])
    return "".join(result)
```

Listing A.11: Source Code Snippets commonChars non-annotated and obfuscated identifier names

```
def compute(abcdefghijk, lmonpqrstuvx):
    yzabcde = 0
    if len(abcdefghijk) < len(lmonpqrstuvx):
        fghijklmnopqrstuvwxyz = len(abcdefghijk)
    else:
        fghijklmnopqrstuvwxyz = len(lmonpqrstuvx)
    for y in range(fghijklmnopqrstuvwxyz):
        if abcdefghijk[y] == lmonpqrstuvx[y]:
            yzabcde += 1
    return yzabcde
```

Listing A.12: Source Code Snippets commonChars annotated and meaningful identifier names

```
def compute(first_input: str, second_input: str) -> int:
    counter: int = 0
    if len(first_input) < len(second_input):
        length_shortest_input: int = len(first_input)
    else:
        length_shortest_input: int = len(second_input)
    for i in range(length_shortest_input):
        if first_input[i] == second_input[i]:
            counter += 1
    return counter
```

Listing A.13: Source Code Snippets containsSubstring non-annotated and obfuscated identifier names

```
def compute(abcd, efghijklm):
    for n in range(len(abcd)):
        for o in range(len(efghijklm)):
            if n + o >= len(abcd):
                break
            if abcd[n + o] != efghijklm[o]:
                break
            else:
                if o == len(efghijklm) - 1:
                    return True
    return False
```

Listing A.14: Source Code Snippets containsSubstring annotated and meaningful identifier names

```
def compute(word: str, substring: str) -> bool:
    for i in range(len(word)):
        for j in range(len(substring)):
            if i + j >= len(word):
                break
            if word[i + j] != substring[j]:
                break
            else:
                if j == len(substring) - 1:
                    return True
    return False
```

Listing A.15: Source Code Snippets countIntegerInterval non-annotated and obfuscated identifier names

```
def compute(abcdef, ghijk, lmn):
    opqrst = 0
    uvwx = abcdef.split(" ")
    for y in range(len(uvwx)):
        zab = int(uvwx[y])
        cdefg = (zab >= ghijk and zab <= lmn)
        if cdefg:
            opqrst += 1
    return opqrst
```

Listing A.16: Source Code Snippets countIntegerInterval annotated and meaningful identifier names

```
def compute(string: str, start: int, end: int) -> int:
    result: int = 0
    keys: list[str] = string.split(" ")
    for i in range(len(keys)):
        key: int = int(keys[i])
        check: bool = (key >= start and key <= end)
        if check:
            result += 1
    return result
```

Listing A.17: Source Code Snippets countLetters non-annotated and obfuscated identifier names

```
def compute(abcd, efghijk):
    lmnopqrstuv = 0
    for w in range(len(abcd)):
        for x in range(len(efghijk)):
            if abcd[w] == efghijk[x]:
                lmnopqrstuv += 1
    return lmnopqrstuv
```

Listing A.18: Source Code Snippets countLetters annotated and meaningful identifier names

```
def compute(word: str, letters: list[str]) -> int:
    letterCount: int = 0
    for i in range(len(word)):
        for j in range(len(letters)):
            if word[i] == letters[j]:
                letterCount += 1
    return letterCount
```

Listing A.19: Source Code Snippets crossSum non-annotated and obfuscated identifier names

```
def compute(abcdef):
    if abcdef == 0:
        return 0
    return (abcdef % 10) + compute(abcdef // 10)
```

Listing A.20: Source Code Snippets crossSum annotated and meaningful identifier names

```
def compute(number: int) -> int:
    if number == 0:
        return 0
    return (number % 10) + compute(number // 10)
```

Listing A.21: Source Code Snippets factorial non-annotated and obfuscated identifier names

```
def compute(abcde):
    if abcde == 1:
        return 1
    return compute(abcde - 1) * abcde
```

Listing A.22: Source Code Snippets factorial annotated and meaningful identifier names

```
def compute(value: int) -> int:
    if value == 1:
        return 1
    return compute(value - 1) * value
```

Listing A.23: Source Code Snippets forwardBackward non-annotated and obfuscated identifier names

```
def compute(abcde):
    f = ""
    g = ""
    for i in range(len(abcde) - 1, -1, -1):
        f = abcde[i] + f
        g = g + abcde[i]
    return f + g
```

Listing A.24: Source Code Snippets forwardBackward annotated and meaningful identifier names

```
def compute(input: str) -> str:
    a: str = ""
    b: str = ""
    for i in range(len(input) - 1, -1, -1):
        a = input[i] + a
        b = b + input[i]
    return a + b
```

Listing A.25: Source Code Snippets leastCommonMultiple non-annotated and obfuscated identifier names

```
def compute(abcdefg, hijklmn):
    opqrst = abcdefg * hijklmn
    for u in range(1, abcdefg * hijklmn):
        if u % abcdefg == 0 and u % hijklmn == 0:
            opqrst = u
            break
    return opqrst
```

Listing A.26: Source Code Snippets leastCommonMultiple annotated and meaningful identifier names

```
def compute(number1: int, number2: int) -> int:
    result: int = number1 * number2
    for i in range(1, number1 * number2):
        if i % number1 == 0 and i % number2 == 0:
            result = i
            break
    return result
```

Listing A.27: Source Code Snippets linearSearch non-annotated and obfuscated identifier names

```
def compute(abcde, f):
    for g in range(len(abcde)):
        if abcde[g] == f:
            return g
    return -1
```

Listing A.28: Source Code Snippets linearSearch annotated and meaningful identifier names

```
def compute(array: list[int], x: int) -> int:
    for i in range(len(array)):
        if array[i] == x:
            return i
    return -1
```

Listing A.29: Source Code Snippets palindrome non-annotated and obfuscated identifier names

```
def compute(abcd):
    efghij = True
    for k in range(0, len(abcd) // 2):
        l = len(abcd) - 1 - k
        if abcd[k] != abcd[l]:
            efghij = False
            break
    return efghij
```

Listing A.30: Source Code Snippets palindrome annotated and meaningful identifier names

```
def compute(word: str) -> bool:
    result: bool = True
    for i in range(0, len(word) // 2):
        j: int = len(word) - 1 - i
        if word[i] != word[j]:
            result = False
            break
    return result
```

Listing A.31: Source Code Snippets power non-annotated and obfuscated identifier names

```
def compute(a, b):
    if b == 0:
        return 1
    if b == 1:
        return a
    return a * compute(a, b - 1)
```

Listing A.32: Source Code Snippets power annotated and meaningful identifier names

```
def compute(a: int, b: int) -> int:
    if b == 0:
        return 1
    if b == 1:
        return a
    return a * compute(a, b - 1)
```

Listing A.33: Source Code Snippets prime non-annotated and obfuscated identifier names

```
def compute(abcde):
    fghi = True
    for j in range(2, (abcde // 2) + 1):
        if abcde % j == 0:
            fghi = False
    return fghi
```

Listing A.34: Source Code Snippets prime annotated and meaningful identifier names

```
def compute(input: int) -> bool:
    flag: bool = True
    for i in range(2, (input // 2) + 1):
        if input % i == 0:
            flag = False
    return flag
```

Listing A.35: Source Code Snippets squareRoot non-annotated and obfuscated identifier names

```
def compute(abcdefg):
    hijklm = [0.0] * len(abcdefg)
    for m in range(len(abcdefg)):
        if abcdefg[m] == 0:
            hijklm[m] = 0.0
            continue
        if abcdefg[m] < 0:
            hijklm[m] = math.sqrt(-1 * abcdefg[m])
        else:
            hijklm[m] = math.sqrt(abcdefg[m])
    return str(hijklm)
```

Listing A.36: Source Code Snippets squareRoot annotated and meaningful identifier names

```
def compute(numbers: list[int]) -> str:
    result: list[float] = [0.0] * len(numbers)
    for i in range(len(numbers)):
        if numbers[i] == 0:
            result[i] = 0.0
            continue
        if numbers[i] < 0:
            result[i] = math.sqrt(-1 * numbers[i])
        else:
            result[i] = math.sqrt(numbers[i])
    return str(result)
```

Listing A.37: Source Code Snippets unrolledSort non-annotated and obfuscated identifier names

```
def compute(a, b, c, d):
    if a > b:
        b, a = a, b
    if c > d:
        d, c = c, d
    if a > c:
        c, a = a, c
    if b > d:
        d, b = b, d
    if b > c:
        c, b = b, c
    return [a, b, c, d]
```

Listing A.38: Source Code Snippets unrolledSort annotated and meaningful identifier names

```python
def compute(a: int, b: int, c: int, d: int) -> list[int]:
    if a > b:
        b, a = a, b
    if c > d:
        d, c = c, d
    if a > c:
        c, a = a, c
    if b > d:
        d, b = b, d
    if b > c:
        c, b = b, c
    return [a, b, c, d]
```

Listing A.39: Source Code Snippets validParentheses non-annotated and obfuscated identifier names

```python
def compute(abcdefgh):
    ijklm = []
    for n in abcdefgh:
        if n == '(':
            ijklm.append(')')
        elif n == '{':
            ijklm.append('}')
        elif n == '[':
            ijklm.append(']')
        elif len(ijklm) == 0 or ijklm.pop() != n:
            return False
    return len(ijklm) == 0
```

Listing A.40: Source Code Snippets validParentheses annotated and meaningful identifier names

```python
def compute(sentence: str) -> bool:
    stack: list[str] = []
    for i in sentence:
        if i == '(':
            stack.append(')')
        elif i == '{':
            stack.append('}')
        elif i == '[':
            stack.append(']')
        elif len(stack) == 0 or stack.pop() != i:
            return False
    return len(stack) == 0
```

BIBLIOGRAPHY

[1] Amine Abbad Andaloussi, Thierry Sorg, and Barbara Weber. "Estimating developers' cognitive load at a fine-grained level using eye-tracking measures." In: *Proc. Int'l Conf. Program Comprehension (ICPC)*. IEEE/ACM, 2022, pp. 111–121.

[2] Jackson Beatty. "Task-evoked pupillary responses, processing load, and the structure of processing resources." In: *Psychological bulletin* 91 2 (1982), pp. 276–292.

[3] Roman Bednarik and Markku Tukiainen. "An eye-tracking methodology for characterizing program comprehension processes." In: *Proc. Eye Tracking Research & Application Symposium (ETRA)*. ACM, 2006, pp. 125–132.

[4] Yoav Benjamini and Yosef Hochberg. "Controlling the False Discovery Rate: A Practical and Powerful Approach to Multiple Testing." In: *Journal of the Royal Statistical Society: Series B (Methodological)* 57.1 (1995), pp. 289–300.

[5] Annabelle Bergum. "On Baselines for Program Comprehension Studies with FMRI." MA thesis. Saarland Informatics Campus, Saarland University, Germany, 2021.

[6] David W. Binkley, Marcia Davis, Dawn J. Lawrie, and Christopher Morrell. "To camelcase or under_score." In: *Int'l Conf. Program Comprehension (ICPC)*. IEEE, 2009, pp. 158–167.

[7] Justus Bogner and Manuel Merkel. "To Type or Not to Type? A Systematic Comparison of the Software Quality of JavaScript and TypeScript Applications on GitHub." In: *Int'l Conf. Mining Software Repositories (MSR)*. IEEE/ACM, 2022, pp. 658–669.

[8] Ruven E. Brooks. "Using a Behavioral Theory of Program Comprehension in Software Engineering." In: *Proc. Int'l Conf. Software Engineering (ICSE)*. Ed. by Maurice V. Wilkes, Laszlo A. Belady, Y. H. Su, Harry Hayman, and Philip H. Enslow Jr. IEEE Computer Society, 1978, pp. 196–201.

[9] Teresa Busjahn, Roman Bednarik, Andrew Begel, Martha Crosby, James H. Paterson, Carsten Schulte, Bonita Sharif, and Sascha Tamm. "Eye Movements in Code Reading: Relaxing the Linear Order." In: *Int'l Conf. Program Comprehension (ICPC)*. IEEE, 2015, pp. 255–265.

[10] Teresa Busjahn, Roman Bednarik, and Carsten Schulte. "What influences dwell time during source code reading?: analysis of element type and frequency as factors." In: *Eye Tracking Research and Applications (ETRA)*. Ed. by Pernilla Qvarfordt and Dan Witzner Hansen. ACM, 2014, pp. 335–338.

[11] Teresa Busjahn, Carsten Schulte, Bonita Sharif, Simon, Andrew Begel, Michael Hansen, Roman Bednarik, Paul Orlov, Petri Ihantola, Galina Shchekotova, and Maria Antropova. "Eye tracking in computing education." In: *Int'l Computing Education Research Conference (ICER)*. ACM, 2014, pp. 3–10.

[12] Benjamin T. Carter and Steven G. Luke. "Best practices in eye tracking research." In: *Int'l Journal of Psychophysiology* 155 (Sept. 2020), pp. 49–62.

[13] Casey Casalnuovo, Kevin Lee, Hulin Wang, Prem Devanbu, and Emily Morgan. "Do Programmers Prefer Predictable Expressions in Code?" In: *Cogn. Sci.* 44.12 (2020).

[14] Christopher Exton. "Constructivism and Program Comprehension Strategies." In: *Int'l Workshop Program Comprehension (IWPC)*. IEEE Computer Society, 2002, pp. 281–284.

[15] Stefan Hanenberg, Sebastian Kleinschmager, Romain Robbes, Éric Tanter, and Andreas Stefik. "An empirical study on the impact of static typing on software maintainability." In: *Empir. Softw. Eng.* 19.5 (2014), pp. 1335–1382.

[16] Johannes C. Hofmeister, Janet Siegmund, and Daniel V. Holt. "Shorter Identifier Names Take Longer to Comprehend." In: *Empir. Softw. Eng.* 24.1 (2019), pp. 417–443.

[17] Phillip Johnston and R. L. Harris. "The Boeing 737 MAX Saga: Lessons for Software Organizations." In: *Software Quality Professional* 21.3 (2019), pp. 4–12.

[18] N. G. Leveson and C. S. Turner. "An investigation of the Therac-25 accidents." In: *Computer* 26.7 (1993), pp. 18–41.

[19] Sebastian Okon and Stefan Hanenberg. "Can we enforce a benefit for dynamically typed languages in comparison to statically typed ones? A controlled experiment." In: *Int'l Conf. Program Comprehension (ICPC)*. IEEE, 2016, pp. 1–10.

[20] Norman Peitek. "A Neuro-Cognitive Perspective of Program Comprehension." PhD thesis. Chemnitz University of Technology, Germany, 2022.

[21] Norman Peitek, Sven Apel, Chris Parnin, André Brechmann, and Janet Siegmund. "Program Comprehension and Code Complexity Metrics: An FMRI Study." In: *Proc. Int'l Conf. Software Engineering*. IEEE Press, 2021, pp. 524–536.

[22] Norman Peitek, Annabelle Bergum, Maurice Rekrut, Jonas Mucke, Matthias Nadig, Chris Parnin, Janet Siegmund, and Sven Apel. "Correlates of programmer efficacy and their link to experience: a combined EEG and eye-tracking study." In: *Proc. Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 2022, pp. 120–131.

[23] Norman Peitek, Janet Siegmund, and Sven Apel. "What Drives the Reading Order of Programmers? An Eye Tracking Study." In: *Proc. Int'l Conf. Program Comprehension (ICPC)*. Seoul, Republic of Korea: ACM, 2020, pp. 342–353.

[24] Norman Peitek, Janet Siegmund, Chris Parnin, Sven Apel, Johannes C. Hofmeister, and André Brechmann. "Simultaneous Measurement of Program Comprehension with FMRI and Eye Tracking: A Case Study." In: *Proc. Int'l Symposium on Empirical Software Engineering and Measurement (ESEM)*. ACM, 2018.

[25] Nancy Pennington. "Stimulus structures and mental representations in expert comprehension of computer programs." In: *Cognitive Psychology* 19.3 (1987), pp. 295–341.

[26] A Poole and Linden Ball. "Eye tracking in human-computer interaction and usability research: Current status and future prospects." In: Jan. 2006, pp. 211–219.

[27] Giuseppe Scanniello, Michele Risi, Porfirio Tramontana, and Simone Romano. "Fixing Faults in C and Java Source Code: Abbreviated vs. Full-Word Identifier Names." In: *ACM Trans. Softw. Eng. Methodol.* 26.2 (2017), 6:1–6:43.

[28] Andrea Schankin, Annika Berger, Daniel V. Holt, Johannes C. Hofmeister, Till Riedel, and Michael Beigl. "Descriptive compound identifier names improve source code comprehension." In: *Proc. Conf. Program Comprehension (ICPC)*. IEEE, 2018, pp. 31–40.

[29] Zohreh Sharafi, Bonita Sharif, Yann-Gaël Guéhéneuc, Andrew Begel, Roman Bednarik, and Martha E. Crosby. "A practical guide on conducting eye tracking studies in software engineering." In: *Empir. Softw. Eng.* 25.5 (2020), pp. 3128–3174.

[30] Bonita Sharif and Jonathan I. Maletic. "An Eye Tracking Study on camelCase and under_score Identifier Styles." In: *Int'l Conference Program Comprehension (ICPC)*. IEEE Computer Society, 2010, pp. 196–205.

[31] Janet Siegmund, Christian Kästner, Sven Apel, Chris Parnin, Anja Bethmann, Thomas Leich, Gunter Saake, and André Brechmann. "Understanding Understanding Source Code with Functional Magnetic Resonance Imaging." In: *Proc. Int'l Conf. Software Engineering (ICSE)*. ACM, 2014, pp. 378–389.

[32] Janet Siegmund, Norman Peitek, André Brechmann, Chris Parnin, and Sven Apel. "Studying programming in the neuroage: just a crazy idea?" In: *Commun. ACM* 63.6 (2020), pp. 30–34.

[33] Janet Siegmund, Norman Peitek, Chris Parnin, Sven Apel, Johannes Hofmeister, Christian Kästner, Andrew Begel, Anja Bethmann, and André Brechmann. "Measuring Neural Efficiency of Program Comprehension." In: *Proc. Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2017. Paderborn, Germany: ACM, 2017, pp. 140–150.

[34] Janet Siegmund and Jana Schumann. "Confounding Parameters on Program Comprehension: A Literature Survey." In: *Empir. Softw. Eng.* 20.4 (2015), pp. 1159–1192.

[35] Benoît De Smet, Lorent Lempereur, Zohreh Sharafi, Yann-Gaël Guéhéneuc, Giuliano Antoniol, and Naji Habra. "Taupe: Visualizing and analyzing eye-tracking data." In: *Sci. Comput. Program.* 79 (2014), pp. 260–278.

[36] Samuel Spiza and Stefan Hanenberg. "Type names without static type checking already improve the usability of APIs (as long as the type names are correct): an empirical study." In: *Int'l Conf. Modularity (MODULARITY)*. 2014, pp. 99–108.

[37] Xin Xia, Lingfeng Bao, David Lo, Zhenchang Xing, Ahmed E. Hassan, and Shanping Li. "Measuring program comprehension: a large-scale field study with professionals." In: *Proc. Int'l Conf. Software Engineering (ICSE)*. ACM, 2018, p. 584.