Eye-Tracking Insights into the Effects of Type Annotations and Identifier Naming

Nils Alznauer
University of Bern
Switzerland

Norman Peitek
Saarland University
Germany

Youssef Abdelsalam
Saarland University
Germany

Annabelle Bergum
Saarland University, Graduate School of Computer Science
Germany

Marvin Wyrich
Saarland University
Germany

Sven Apel
Saarland University
Germany

Abstract

Understanding and modifying source code is a cognitively demanding task shaped by how the code is structured and presented. Prior work has shown that both identifier naming and type annotations can influence program comprehension, yet findings on their individual effects remain mixed. This study focuses on how these two aspects interact in influencing program comprehension, visual attention, and subjective perceptions of code comprehensibility.

We conducted a controlled eye-tracking experiment with 40 participants who were asked to understand 20 Python code snippets. Our results reveal an interesting dichotomy between objective and subjective findings. Objectively, type annotations slightly slow down developers, but otherwise have almost no statistically significant effect on behavior, visual attention, or subjective perceptions, even when combining them with the meaningfulness of identifier naming. Subjectively, however, nearly all participants *felt* that type annotations helped with program comprehension.

Overall, our findings suggest that type annotations—and their interaction with identifier names—do not generally introduce measurable overhead, but depending on the code snippet and individual participant, may increase developers' sense of certainty and clarity.

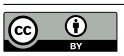
CCS Concepts

• **Human-centered computing** → *Empirical studies in HCI*; *HCI design and evaluation methods*; • **Software and its engineering**;

Keywords

program comprehension, type annotations, identifier naming, eye tracking, controlled experiment

ACM Reference Format:
Nils Alznauer, Norman Peitek, Youssef Abdelsalam, Annabelle Bergum, Marvin Wyrich, and Sven Apel. 2026. Eye-Tracking Insights into the Effects of Type Annotations and Identifier Naming. In *34th IEEE/ACM International Conference on Program Comprehension (ICPC '26), April 12–13, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3794763.3794801



ICPC '26, Rio de Janeiro, Brazil

ACM ISBN 979-8-4007-2482-4/2026/04
https://doi.org/10.1145/3794763.3794801

1 Introduction

Imagine spending over half your working day trying to understand code—this is the reality for most developers, with around 58%–70% of their time dedicated to code comprehension activities [22, 49]. Despite this, and although research into program comprehension has existed for decades [48], we still know surprisingly little about the cognitive processes involved during code comprehension [42].

This lack of understanding is particularly evident regarding which aspects of the code influence a developer's understanding. In the past, the software engineering community suspected type annotations to influence program comprehension [6, 18, 46]. Furthermore, research suggests that meaningful identifier names positively influence program comprehension [11, 13, 19, 35, 39]. What remains unclear is how the interplay of meaningful identifier names and type annotations affects program comprehension.

To shed light on this issue, we conducted an eye-tracking experiment that complements conventional behavioral measures with visual and subjective data, offering a comprehensive view of developers' comprehension processes [40]. We explore how the presence or absence of type annotations influences program comprehension and what role meaningful identifier names play. Our study involved 40 participants whose visual attention and comprehension strategies we analyzed, focusing on reading order and viewing behavior while understanding code snippets.

Our findings reveal that type annotations slightly slow developers, yet have little measurable influence on program comprehension—either alone or in combination with identifier names. Surprisingly, participants nevertheless favored the presence of type annotations, perceiving them as helpful despite their limited effects. An analysis of fixation patterns further reveals that attention to type annotations varies considerably across both snippets and participants, hinting at additional factors shaping their perceived usefulness.
In summary, we make the following contributions:

- We design and conduct a multi-modal experiment that investigates how type annotations and meaningful identifier names individually and jointly influence program comprehension.
- We provide empirical evidence that both factors have limited measurable effects on comprehension, yet are perceived by developers as helpful and important.
- We share a complete online replication package[1] containing the experiment design, raw data, and analysis scripts.

[1] https://github.com/brains-on-code/effect-of-type-annotations

2 Background and Related Work

In this section, we introduce the concepts that frame our study—program comprehension, eye tracking, and type annotations—and discuss prior work that informs our investigation.

2.1 Program Comprehension

Program comprehension is central to software engineering, since it takes up most of the development time spent on code [22, 49]. Program comprehension is essential for various tasks, such as adding new features, testing, and fixing bugs. Measuring program comprehension is inherently difficult, as it is an internal cognitive process [40]. Eye tracking, therefore, has become an established method to study program comprehension [25, 38] providing observable insights into developers' cognitive processes [1].

According to Peitek [27] and based on the work of Exton [16], program comprehension "describes the transition in which a programmer understands an existing implementation formed in source code and constructs an analogous mental model". In other terms, it "describes a person's intentional act and degree of accomplishment in inferring the meaning of source code" [47]. The two main program comprehension strategies developers use are bottom-up comprehension and top-down comprehension. Bottom-up comprehension is the fundamental program comprehension strategy [32]. In bottom-up comprehension, developers read code line by line, forming a mental model from combining and abstracting from the details. This method is thorough but time-consuming and cognitively demanding [43]. In contrast, during top-down comprehension, developers use their domain and contextual knowledge to infer the code's intent [43], allowing them to skip irrelevant parts. This strategy is faster and less cognitively taxing than bottom-up comprehension, but requires semantic cues in the code and domain knowledge [7].

A critical component of whether developers can use top-down comprehension or have to fall back to bottom-up comprehension are identifier names. Casalnuovo et al. [13] observed that programmers prefer predictable and sensible identifier names, indicating better program comprehension with such names. Hofmeister et al. [19] reported that using whole words as identifier names improves program comprehension by 19%, reducing the cognitive load, especially with type annotations. Siegmund et al. [43] observed higher neural efficiency during top-down comprehension when compared with bottom-up comprehension. Scanniello et al. [34] noted that this benefit does not extend to bug fixing, though. In the same vein, Schankin et al. [35] found that descriptive names help identify semantic defects 14% faster but offer no advantage for syntax error detection. In this paper, we aim to trigger both comprehension strategies in separate conditions to understand the effect of type annotations.

2.2 Eye Tracking in Program Comprehension

Eye tracking measures visual attention by observing eye movements and is widely used in many research fields [12]. It is beneficial in program comprehension research since visual attention activates cognitive processes essential for program comprehension [1, 37]. Eye tracking helps measure cognitive load, processes, and strategies developers use to understand code [1, 2, 9, 10, 31].

Eye tracking provides rich data that can be analyzed at multiple abstraction levels [37]. *First-order data* comprise raw measurements such as gaze coordinates, duration, and pupil size. From these, researchers derive *second-order data* like fixations and saccades [12], and *third-order data* such as their counts and durations. These measures are typically aggregated over predefined code regions, referred to as Areas of Interest (AOIs). Finally, *fourth-order data* combine AOI-derived measures into more complex analyses, such as scan paths and transition matrices.

Several eye-tracking studies on program comprehension are related to our work. In particular, Peitek et al. [30] investigated the reading order of programmers based on the linearity of code snippets. They observed that non-linear source code causes unnecessary eye movements, slowing down programmers. Recent research by Andaloussi et al. [1] used eye tracking to enhance cognitive load estimation through machine learning, identifying pupil-based and saccade-based measures as most significant.

2.3 Type Annotations

The type of a variable is fundamental in many traditional programming languages for error checking and optimization. The constraining of variables to specific types may also benefit the developers, as it is hypothesized to aid their understanding and effectively minimize errors [6, 18].

There are two main approaches to type annotations. In statically typed languages (e.g., Java or C++), type annotations are mandatory, and variable types are checked at compile time, allowing errors to be detected before running the program. In dynamically typed languages (e.g., Python or JavaScript), type annotations are not required, and variable types are checked at runtime, providing greater flexibility but potentially risking runtime errors.

Notably, Python allows optional type annotations, which was introduced in version 3.5[2]. These optional type annotations allow developers to add type information where helpful, without sacrificing the benefits of dynamic typing. Type information can be used in Python for variables, function parameters and return types, and classes.

Several studies have investigated the effect of type annotations on program comprehension. Specifically, Hanenberg et al. [18] examined whether static type systems aid in program comprehension. They observed programmers fixing bugs and found mixed results. While type errors were easier to fix with static type systems, there was no significant improvement in fixing semantic errors. Bogner and Merkel [6] compared JavaScript and TypeScript noting better code quality in TypeScript without a clear impact on either bug proneness or bug resolution. Research has been focused on the use of static type systems, which motivated Okon and Hanenberg [26] to designing a study that biased the tasks toward dynamic type systems. Notably, despite the purposefully introduced bias, their results were mixed, with half of the tasks showing static type systems to be beneficial. Spiza and Hanenberg [46] investigated whether type annotations alone improved the usability of APIs without static type checking. They observed that, while type names generally helped,

[2]PEP 484, https://peps.python.org/pep-0484/

incorrect type names significantly hindered comprehension, indicating that type annotations are important for understanding and that inaccurate information can mislead programmers.

In a nutshell, while there are several indicators that type annotations can be helpful for program comprehension, in some contexts on a behavioral level, it is still unclear *why* the effect is different in different scenarios. In this paper, we specifically address this knowledge gap by investigating not only the effect of type annotations on a behavioral level, but also observing how it changes programmers' visual attention in addition to their subjective viewpoints.

3 Study Design

In this section, we describe the design of our study investigating the effect of type annotations and the combination of type annotations and identifier names on program comprehension.

3.1 Research Goals

Our primary focus is on answering the following research question:

> **RQ$_1$** How do type annotations influence program comprehension?

Since type annotations may have less of an influence on program comprehension in a well-documented and well-structured code base, we first explore the extent to which type annotations help with program comprehension. To this end, we create a set of code snippets that forces the participant to understand the code bottom-up, instead of providing much contextual information, such as is the case with meaningful variable names and code comments.

To obtain a more comprehensive understanding, we also investigate how the presence of type annotations and meaningful identifier names jointly affect program comprehension:

> **RQ$_2$** How does the combination of type annotations and identifier names influence program comprehension?

We chose a two-by-two factorial design as illustrated in Figure 1. For both research questions, we measure program comprehension in terms of a participant's behavioral responses (correctness and time), their visual attention through linearity metrics, and their subjective rating of the code's difficulty. We describe the operationalization of all constructs in more detail in the following Section 3.2.

3.2 Study Plan

We opted for a mixed-subject design using both a within-subject and between-subject approach. The within-subject design is chosen for the independent variable *type annotation* to obtain a better understanding of how type annotations influence each participant's program comprehension. A participant, who might have been slower before, might become faster with type annotations and vice versa. The between-subject design is chosen for the independent variable *identifier names*. In simple terms, all participants have seen code with and without type annotations. Additionally, some of the participants always saw obfuscated identifier names, while others always saw meaningful identifier names (randomized group assignment). This design allows us to investigate how the participants' program comprehension is influenced by the type annotations when either the variables are meaningful or not.

3.2.1 Independent Variables. The two independent variables, type annotations and identifier names, each have two levels: type annotations can either be present or not, and the choice of variable name can either be meaningful or obfuscated.

Type Annotations. We limit type annotations to simple data types: `int`, `float`, `str`, `bool`, and `list`. For `list`, all possible combinations with the other simple types are considered (e.g., `list[int]`). Type annotations are used for variables, the function return type, and the parameter input types. In the condition of type annotations being available, we always added type annotations to *all* present variables. We did not include complex or composite types as they typically require more complex code snippets, which would have introduced numerous additional confounding factors.

Identifier Names. In our operationalization, identifier names can either be *meaningful* or *obfuscated*. We define meaningful identifier names as names that consist of words found in a dictionary and properly explain the usage of the identifier while at the same time not giving away the whole function's purpose (e. g., `array_to_sort`). This way, the participant is provided with context about the identifier's intended usage and functionality. When an identifier name did not provide sufficient information, we extended it to add more context by using underscores as suggested by Binkley et al. [5].

We define an obfuscated identifier as consisting only of letters in their alphabetical order without any underscores. To make the 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. Furthermore, we aimed to make all identifiers easily distinguishable to the participant [19]. 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. For example, the first obfuscated variable with a length of 6 characters would be `abcdef`. The next identifier name is taking the next number of letters from the alphabet. When we reached the end of the alphabet, we started again at the beginning. This way, we ensure that no two identifier names are the same or look similar.

3.2.2 Dependent Variables. The dependent variables are the measures used to evaluate the influence of both independent variables. We measure program comprehension with behavioral, visual attention, and subjective measures.

Behavioral Measures. Our basic measures of program comprehension are the response time and the correctness since they can be objectively and quantifiably observed and thus easily compared and evaluated. The *response time* is a general indicator of how difficult the code snippets are. We measure response time from the moment the code snippet appeared on the screen until the question was displayed. Similarly, the correctness is also critically important, since it shows whether the participant fully comprehended the code snippets. *Correct code snippets* measures the number of correctly solved code snippets for each participant.

Even though these measures are high-level, we can already obtain a basic understanding of the individual difficulty of the code snippets and the overall participant behavior in the four conditions.

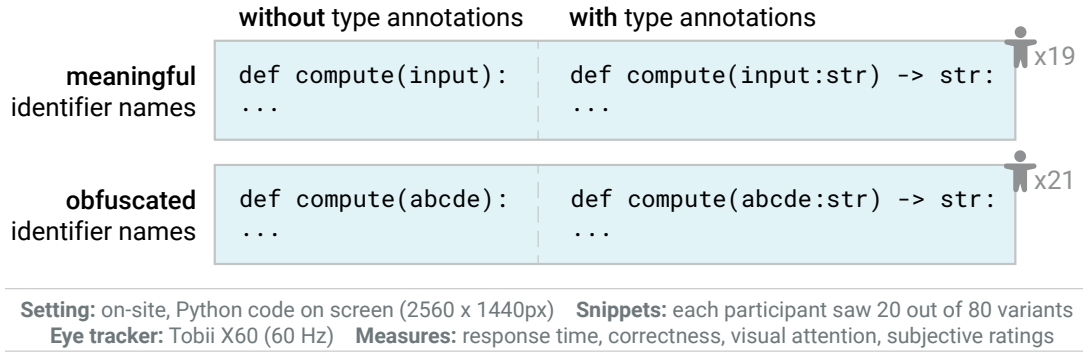


Figure 1: Overview of the main characteristics of our 2x2 factorial, mixed-experiment design

Behavioral measures, however, cannot tell *why* a participant made mistakes in a given task.

Linearity of Reading Order. For our second measure of program comprehension, we measure the participants' visual attention. Specifically, we measure whether they read the code snippets linearly in execution order, or whether their gaze is jumping around within the code snippets. A more linear reading order suggests that participants did not have to backtrack as much as others. This can especially be influenced by adding type annotations to the code snippets, since this creates more information in each line which might lead to less backtracking. To measure the *linearity of reading order*, we employ the eye-tracking measures developed by Busjahn et al. [8], which have been used in several studies [29]. We define all measures in Table 3. In line with Busjahn et al. [8], we also compute the story order with the Needleman-Wunsch alignment score [23] by comparing the fixation order with the linear text reading order.

Complementary to the general visual attention, we aim to measure whether participants paid attention to the type annotations. For this purpose, we measure the number of fixations on the type annotations, both in terms of absolute fixations and also the relative attention in contrast to the entire snippet.

3.2.3 Subjective Measures. Our final measure of program comprehension is how participants subjectively rate the code snippets. This provides us with a complementary view of how difficult the code snippets were individually for the participants. We asked participants at the end of the study to rate each code snippet on a scale from 1 (very easy) to 5 (very hard). We also allowed open-ended comments for each code snippet.

3.3 Code Snippets

To obtain a sizable set of code snippets, we collected code snippets from related eye-tracking studies [4, 28, 44]. We then created additional code snippets that were similar in complexity. To remove as many confounding factors from the code as possible, we removed all comments, applied consistent indentation, and changed all function names to `compute`. We adapted all snippets to Python since Python supports executable code with and without type annotations. Furthermore, we changed all variable names to fit Python's naming convention, and we followed the recommendation to use

```python
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 1: Binary search code snippet with meaningful identifier names and included type annotations.

```python
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 2: Binary search code snippet with obfuscated identifier names and without type annotations.

underscores to separate words in identifier names [5]. These requirements led to an initial pool of 44 code snippets. We then selected code snippets where adding type annotations was a sensible option, which led to a set of 27 code snippets, which we evaluated in a pilot study with 4 participants and selected our final set of code snippets based on the response time and the subjective complexity.

For our final set of 20 code snippets, we created four versions, one for each field in the 2x2 factorial design. Specifically, we created each code snippet with and without type annotations and with meaningful and obfuscated identifier names (i.e., 80 different code snippets). We show examples in Listing 1 (type annotations, meaningful identifier names) and Listing 2 (no type annotations, obfuscated identifier names).

3.4 Pre-Questionnaire

Our pre-questionnaire consisted of demographic questions and programming experience questions (adapted from Siegmund et al. [41]). For most answers, the participants were only able to choose one of the given answer options. For the self-assessment and the programming languages, participants could fill in open fields.

3.5 Experiment Task

During the main experiment, we presented a code snippet to the participants. In line with other program-comprehension studies [4, 28, 44], we operationalized program comprehension by asking participants to select the correct output from four answer options to a given input for each presented code snippet.

3.6 Post-Questionnaire

We conducted the post-questionnaire as a semi-structured interview after the main study. The post-questionnaire consists of general questions regarding the subjective difficulty of the code snippets, the participants' energy level after the study, their subjective experiences with the presented code snippets, and how helpful they consider type annotations for the snippets in our experiment and in general. For each open-ended question, the study conductor noted the answer in a summarized and abbreviated form.

All snippets and the complete pre- and post-questionnaire can be found in the replication package. Our study was approved by the ethical review board of the faculty of Mathematics and Computer Science at Saarland University (approval number 23-11-07).

3.7 Eye-Tracking Software and Hardware

We used the Tobii X60 eye tracker with a sampling rate of 60 Hz. The used monitor has a resolution of 2560 by 1440 pixels. We used a custom script that gathers the responses to the pre-questionnaire, calibrates the eye tracker with the participant's help, and displays the code snippets and tasks. To obtain high-quality data, we seated the participants in a fixed chair and instructed them to keep their heads still.

3.8 Participants

For our study, we invited participants based on three criteria. First, we required participants to be enrolled in a degree related to computer science to ensure a basic level of programming knowledge, programming languages, and the concepts of computer science. We limited our set of participants to students to ensure that all participants have a similar level of experience in both programming and computer science, which is a sensible choice to obtain a more homogeneous participant group [44]. Second, each participant must be able to comprehend small programs in Python, which we tested via a self-assessment.

We recruited 40 participants at Saarland University via e-mail lists and flyers. We incentivized participation with a small number of sweets, a soft drink, and the chance to win one of two Amazon vouchers worth 25 € each. Five participants reported having impaired eyesight, but they all had corrective glasses.

We completely removed five participants who failed to solve more than half of the presented tasks. After this filtering, 35 participants remained in our initial dataset. We provide demographic data in Table 1. Overall, we would classify our participants as intermediate programmers according to Dreyfus' taxonomy of skill acquisition [15, 21].

Table 1: Demographic and experience data of our participants and included data points per group.

	Meaningful	Obfuscated
Male	16 (84%)	17 (81%)
Female	2 (11%)	4 (19%)
Prefer not to say	1 (5%)	—
Age (in Years)	22.74 ± 4.42	22.95 ± 2.80
Semester	5.53 ± 2.57	4.81 ± 3.01
Years of Programming	4.63 ± 2.56	5.33 ± 3.21
Measured Participants	19 Participants	21 Participants
Incl. Eye-Tracking Data	14 Participants	17 Participants
Incl. Eye-Tracking Trials	238 Trials	259 Trials

3.9 Deviations

During the study, some deviations occurred. Due to a problem with the timer, the first participant took a whole hour to finish all the snippets. To avoid any fatigue issues, we removed the snippets completed after 45 minutes from the data analysis. The 27th participant found a small bug in the code snippet `capitalizeFirstLetter` when type annotations are provided. The type annotation in Line 2 should be `list[str]`, rather than `list[int]`. We presume that this did not have a substantial impact on the data, especially, since no other participant detected it.

4 Data Analysis

We analyzed the collected data in three steps: First, we cleaned the data, then we pre-processed it, and finally, we computed the results via descriptive and inferential statistics.

4.1 Data Cleaning

While most of the included 35 participants finished all 20 code snippets within the targeted 45 minutes, 4 did not finish all tasks, missing at most three code snippets.

Eye-Tracking Data. We smoothed the raw eye gaze data using a Savitzky-Golay filter (window size: 5, polynomial order: 3 [24]). Next, we applied a velocity-based algorithm to detect fixations and saccades (threshold of 150 pixels in 100 milliseconds). If the velocity was below this threshold, it was considered a fixation, otherwise it was considered a saccade [20].

For 3 of the 35 participants from the initial dataset we did not obtain any eye-tracking data. This was due to a technical error of the eye tracker. In addition, some participants moved too much for the eye tracker, which led to a significant signal loss. To ensure an eye-tracking dataset of high quality, we removed a total of 86 individual trials (from 583 total trials) based on the following not mutually exclusive criteria:

- *Total fixation count.* Trials were excluded if the total number of fixations was either (i) implausibly low (< 10; 12 trials), or (ii) outside 1.5 IQR from the mean (> 641 fixations; 26 trials).
- *Fixation rate (fixations per second).* Trials were excluded if the fixation rate was (i) below 0.5 fixations/s (18 trials), or (ii) outside 1.5 IQR from the mean (< 0.97 or > 6.3 fixations/s; 47 trials).
- *Spatial validity.* Trials with fewer than 70% of fixations within the code area were excluded (22 trials).

Additionally, within the trial data, we excluded all eye-tracking data points within 140 pixels of the bottom of the screen where the "continue" button was located.

4.2 Eye-Tracking Measures

To quantify the visual attention in terms of specific measures, we defined several AOIs for each code snippet. We created an AOI for each line, for the whole code snippets, and for the type annotations (if they were present). To mitigate potential minor inaccuracies in the eye-tracking data affecting the AOI-based analysis [14], we followed Peitek et al. [31] and added 5 pixels to all sides of each AOI. Furthermore, following Busjahn et al. [8], all fixations that are within 100 pixels horizontally of an AOI are considered to be part of that AOI, otherwise, smaller AOIs could be easily missed. A visualization of all AOIs is part of the replication package.

Based on the preprocessing and the AOI, we computed the linearity measures for each participant and each code snippet, which is in line with Busjahn et al. [8] and Peitek et al. [30]. We additionally computed the fixation statistics in terms of absolute and relative number of fixations for the AOI covering the type annotations.

4.3 Statistical Tests

For the statistical analysis, we first tested for normality with a *Shapiro-Wilk Test* and for homogeneity in the variance with *Levene's Test*. If the data were not normally distributed (p<0.001), we applied the non-parametric *Mann-Whitney U* test, otherwise the *Chi-Square Test of Independence*. For comparing two independent variables in RQ_2, we use a mixed linear regression model to account for the different independent variables and then apply the *Wald Chi-Square* test. We generally consider a significance threshold of $\alpha \leq 0.05$. Due to the number of eye-tracking measures, we apply a false-discovery rate (FDR) correction based on the Benjamini-Hochberg procedure to correct for multiple testing [3]. We computed all statistical tests with `statsmodels` (Version 0.14.2, [36]). The entire executable script is part of the replication package.

5 Results

In this section, we present the results of the study.

5.1 RQ_1: Type Annotations

We analyze the effect of the annotations in regard to 1) behavioral measures, 2) visual attention, and 3) subjective difficulty.

5.1.1 Behavioral Measures. The average response time for snippets with type annotations was longer (67.2 ± 36.0 seconds) than for snippets without type annotations (61.0 ± 37.2 seconds), which is statistically significant (Mann-Whitney U: p = 0.02, U = 27255).

Table 2: The 2x2 matrix for the mean and standard deviation of response time in seconds and correctness.

	Meaningful	Obfuscated	∑
with type annotations	64.4s ± 36.4s 88% (143/163)	70.7s ± 35.0s 85% (165/195)	67.2s ± 36.0s 84% (215/255)
without type annotations	60.7s ± 31.9s 89% (142/160)	62.2s ± 37.2s 88% (133/151)	61.0s ± 37.2s 88% (214/242)
∑	63.0s ± 36.6s 87% (207/238)	65.3s ± 36.8s 86% (222/259)	64.2s ± 36.7s 86% (429/497)

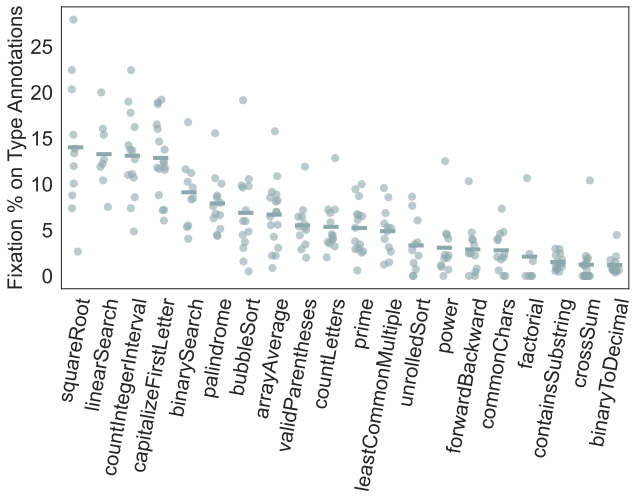


Figure 2: Percentage of fixations on type annotations of overall fixations, separated by snippet. Each dot represents one participant trial.

This means that type annotations actually slow down the measured comprehension process.

Regarding correctness, we found a slight trend towards the non-annotated snippets. The overall correctness of 86% (429 / 497) is divided in 84% for the annotated code snippets and 88% for the non-annotated code snippets, but without a statistically significant difference (*Chi-Square Test of Independence* p = 0.23, χ^2 = 1.45). We provide a detailed overview in Table 2.

5.1.2 Eye-Tracking: Linearity Measures. To obtain a comprehensive overview of the participants' visual attention between annotated and non-annotated code snippets, we used the eye-tracking measures of Busjahn et al. [9] that indicate the linearity of reading order. We find that the presence of type annotations does not substantially change the programmers' reading order, since only two of the measures yielded a statistically significant difference (saccade length and naive story order, cf. Table 3).

5.1.3 Eye-Tracking: AOI-Based Analysis of Type Annotations. To understand whether the type annotations are actually incorporated in the comprehension process, we checked whether they were fixated on based on the number of fixations on the type annotations

Table 3: Description of all linearity measures and the results of the statistical tests after FDR correction.

	Linearity Measure	Definition	Type Annotations*	Combination of Type Annotations and Identifier Names**
Local	*Vertical Next*	% of forward saccades that either stay on the same line or move one line down	$p = 0.57$	$p = 0.57$
	Vertical Later	% of forward saccades that either stay on the same line or move down any number of lines	$p = 0.13$	$p = 0.13$
	Horizontal Later	% of forward saccades within a line	$p = 0.23$	$p = 0.15$
	Regression Rate	% of backward saccades of any length	$p = 0.53$	$p = 0.13$
	Line Regression Rate	% of backward saccades within a line	$p = 0.18$	$p = 0.53$
	Saccade Length	Average Euclidean distance between every successive pair of fixations	$p = 0.02$	$p = 0.20$
Global	*Story Order (Naive)*	Needleman-Wunsch alignment score of fixation order with linear text reading order	$p = 0.04$	$p = 0.63$
	Story Order (Dynamic)	Needleman-Wunsch alignment score of fixation order that tolerates multiple reads	$p = 0.31$	$p = 0.97$

* **Mann-Whitney U Tests** ** **Linear Mixed Model with Wald Chi-Square Test**

in contrast to the total number of fixations when comprehending a snippet. Interestingly, we find that a substantial number of fixations are on type annotations, but it differs across snippets (see Figure 2).

5.1.4 Subjective Difficulty. Finally, we determined the participants' subjective perception of difficulty for each snippet. The majority (70%) of the code snippets were rated as *Very Easy* or *Easy*. Code snippets with type annotations were slightly more often rated as *Very Easy* than the non-annotated ones. However, they were also more often rated as *Difficult*. None of these trends are statistically significant (*Chi-Square Test of Independence*: p = 0.86).

RQ$_1$ We find that type annotations slightly influence program comprehension for our code snippets in terms of behavior (increasing response time), shifting visual attention, but not affecting subjective difficulty.

5.2 RQ$_2$: Type Annotations & Identifier Names

RQ$_2$ is concerned with whether there is an interplay between the meaningfulness of identifier names and type annotations. We present the results of RQ$_2$ in the same structure as RQ$_1$.

5.2.1 Behavioral Measures. The average response time for the obfuscated group (70.7 ± 35.0 seconds) is longer than for the meaningfulgroup (64.4± 36.4 seconds), as presented in Table 2. Thus, the response time is statistically significantly influenced by identifier names (*Wald Chi-Square* test: p = 0.007), but not influenced by type annotations (p = 0.43) or the combination of both (p = 0.67).

Similarly, we find no notable difference in the correctness between the two groups (86% vs. 89%), which reveals no significant influence by the type annotations (*Wald Chi-Square* test: p = 0.86), the identifier names (p = 0.64), or their combination (p = 0.63).

5.2.2 Eye-Tracking: Linearity Measures. When considering the difference in meaningfulness of variable names, we find no significant eye-tracking measure for linearity of reading order (cf. Table 3).

5.2.3 Eye-Tracking: AOI-Based Analysis of Type Annotations. In Section 5.1.3, we established that the fixations on type annotations appear to be different depending on the snippet. When considering the difference in meaningfulness of identifier names, we do not find a clear difference between participant groups. However, there is a difference in how much individual programmers fixate on identifier names (cf. Figure 3).

5.2.4 Subjective Difficulty. The majority (70%) of the meaningful code snippets were rated as *Very Easy* or *Easy*. Interestingly, meaningful code snippets with type annotations have a high number of *Very Easy* ratings and only one *Very Difficult* rating. Contrary to the meaningful and obfuscated group, where the number of *Difficult* and *Very Difficult* ratings is higher. Nevertheless, the subjective difficulty is not significantly influenced by the type annotations (*Wald Chi-Square* test: $p = 0.78$), the identifier names ($p = 0.55$), or their combination ($p = 0.65$).

RQ$_2$ We find that meaningful identifier names significantly reduce response time, but otherwise there is no interplay with type annotations as none of our behavioral, visual attention, or subjective measures reveal statistically significant differences.

5.3 Insights from Post-Questionnaire

At the end of the study, we asked participants about the perceived helpfulness of type annotations in two ways: first specifically for each presented code snippet, and second, regarding type annotations in general. In contrast to our results from RQ$_1$ and RQ$_2$, participants reported type annotations as helpful, both for the experiment (85%) and in general (91%). They explained that type

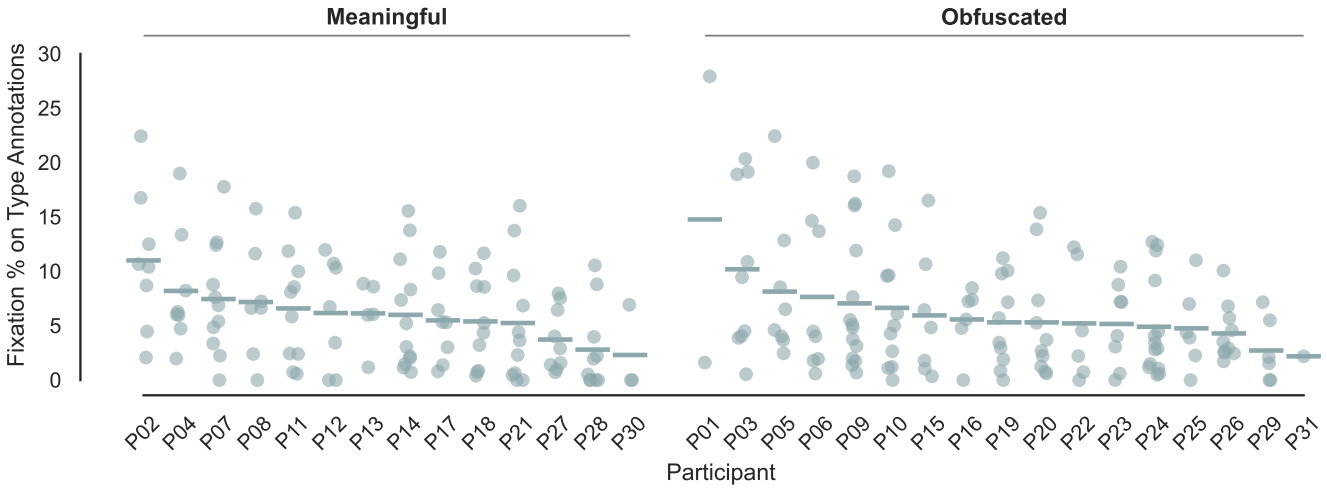


Figure 3: Percentage of fixations on type annotations of overall fixations, separated for each participant and grouped by the meaningful and obfuscated condition.

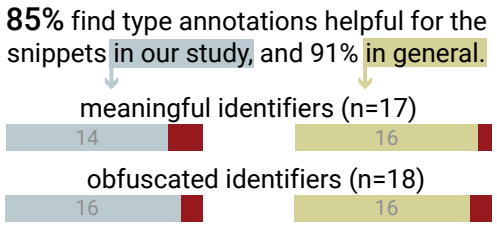


Figure 4: Reported frequency of participants' perceived helpfulness of type annotations in this study and in general

annotations serve as reminders of what the code is intended to do, provide additional information when types are not immediately obvious, and are particularly useful for new, complex, or unfamiliar code. Only a few participants did not find the type annotations helpful for the experiment snippets or in general, often because they considered them unnecessary or distracting for familiar or simple code. We elaborate on this observation in the discussion.

> In stark contrast to our experiment results, the vast majority of our participants reported type annotations to be helpful during program comprehension—within our study and in general programming.

6 Discussion

Our findings provide new insights into how type annotations and identifier names shape program comprehension, revealing how these factors relate to behavioral performance, visual attention patterns, and developers' subjective perceptions, which we discuss next.

6.1 Type Annotations (RQ_1)

Based on the literature, we expected that the participants would be faster and more correct when type annotations are present, irrespective of whether the identifier names were obfuscated or meaningful. However, our study revealed that, on average, the annotated code snippets took longer to comprehend than the non-annotated ones, and the correctness does not change significantly.

Note, that a prior study found that the correctness of program comprehension is not significantly different when changing the difficulty of the code snippets [29]. This suggests that the participants only answered the code snippets' question when they were ready and understood the code independently of the difficulty of the underlying code. This should result in a higher average time taken for the code snippets. While we can see this trend in the data, this is not statistically significant. It is possible that our code snippets or the task design were too easy for our participants. For example, in the study by Peitek et al. [29], the participants were asked to answer each question in a free-text field, which is arguably more difficult than answering a multiple-choice question. We decided against this approach to obtain clear-cut answers that are not open to interpretation, which resulted in high correctness values.

One important consideration is that the annotated snippets contain substantially more text. That is, participants have to process more information when type annotations are present. From this perspective, it is understandable that participants need more time to comprehend the entire snippets. It is possible that the cost of reading additional text is offset by the informative value of the type annotations, as the overall response time increases less than the increase in characters.

We further expected that the presence of type annotations results in a more linear reading order because the type annotations provide additional information supporting program comprehension. Specifically, we expected lower regression scores both within a line and between lines of code. However, our study showed that the presence of type annotations did not substantially result in a

more linear reading order or fewer regressions, but only in shorter saccades. It is possible that, given the complexity of our code snippets, the additional information provided by simple types was less significant than anticipated. We deliberately chose simple types and manageable snippet complexity so that the code would fit on the screen, allowing eye tracking to be conducted without scrolling.

Interestingly, we observed that the share of attention paid to type annotations strongly depends on the snippet (cf. Figure 2), which indicates that our study trying to cover a variety of snippets and type annotations may be too general to differentiate in which context type annotations are helpful for program comprehension.

Finally, we expected that participants would report a greater subjective difficulty when code snippets were not annotated. However, this was not the case. It is important to highlight that the reported difficulty was overwhelmingly (> 70%) considered to be *Easy* or *Very Easy*. This may explain why participants did not report greater difficulty when the code snippets were not annotated, which suggests that the code snippets in our study were too easy to reveal a meaningful difference.

Our behavioral results stand in contrast to the finding that nearly all participants spend a substantial amount of visual attention on the type annotations and reported that the type annotations helped comprehend the code snippets. Most participants said that type annotations were a good reminder of what the code was supposed to do and that they received 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. In contrast, some participants declined that the type annotations were helpful and stated that type annotations do not provide any accurate new information and that they were more of a distraction than a help, yet they still substantially focused on them. This is supported by the observation that 91% of the participants found type annotations generally helpful for understanding code. One of participants who declined explained that they view type annotations as necessary only when learning a new language and not yet fully understanding code. Another participant stated that they find type annotations helpful only when accompanied by a comment, particularly in the case of Python. Other participants who generally found type annotations helpful explained that they are especially helpful when working with code that is not their own or when more complex types are involved, as it is not always possible to infer what an object can do. They also noted that type annotations are helpful when the code is poorly documented or part of a larger code base. This suggests that, despite our quantitative results, type annotations can be a valuable tool for helping developers gain confidence in their program comprehension.

6.2 Type Annotations & Identifier Names (RQ_2)

As in RQ_1, we expected participants to be faster and more correct when working with meaningful identifier names compared to obfuscated ones, and with annotated code snippets compared to non-annotated ones. The combination of both independent variables providing all information should have the largest explanatory potential. However, this is not be supported by the collected data. The response time is significantly longer for obfuscated identifier names, which is in line with prior literature [43]. However, the combination of both independent variables did not result in a significant difference in response time. This suggests that even if the participants are forced to rely more on bottom-up comprehension instead of top-down comprehension and are suspected to receive information through the type annotations, the completion time is not significantly influenced.

Similar to the first research question, we expected for the second a more linear reading flow triggered by the present type annotations and meaningful identifier names. However, we observe no statistically significant differences. A possible reason for this is that the participants were not challenged enough by the code snippets and that the type annotations were not as helpful as they could have been, which was partially by design.

We expected that participants would report a higher difficulty when working with meaningful identifier names compared to obfuscated ones, and with annotated code snippets compared to non-annotated ones. Our data did not confirm these expectations. However, we observed that the reported difficulty was overwhelmingly (> 70%) considered to be *Very Easy* or *Easy* for the meaningful identifier names. At the same time, for obfuscated identifier names, slightly over 60% of participants rated the difficulty as *Very Easy* or *Easy*. The categories *Difficult* and *Very Difficult* accounted for 10% of the reported difficulties in non-annotated code snippets and 20% in annotated code snippets. This indicates that the perceived difficulty of the code snippets is influenced by type annotations, which may make the code snippets harder to comprehend when identifier names are obfuscated; however, this effect is not statistically significant. One possible reason is that, especially in the longer snippets, the type annotations may have been distracting, making comprehension more difficult.

6.3 Overall Findings

From the results for both research questions, we conclude that type annotations are not as influential for developers. For our eye-tracking study, the code snippets had to be relatively simple and short to fit on a single screen. In this constrained setting, type annotations may not be as helpful as suggested in previous literature [6, 18, 26, 46]. This perspective, however, is supported by participants' subjective responses, with the overwhelming majority strongly expressing a strong preference for type annotations, stating that they make code easier to comprehend and help them keep track of what specific parts of the code do.

To ensure the code snippets were not too easy, we conducted a pilot study in which all participants were able to approach the snippets, albeit at different speeds. In the main study, participants' mean completion time was 64.2 seconds (SD = 36.7), indicating a diverse range of difficulty across the snippets. Despite this, participants reported that over 70% of the snippets were *Very Easy* or *Easy*. This suggests that the code snippets were not challenging enough, highlighting the need to replicate the study with more complex snippets in the future.

Considering that the participants were students, the high easy ratings are somewhat surprising. Revisiting the participant's demographics provides a possible explanation: on average, participants were in their 5.15 semester (SD = 3.61) and had been programming

for 5.00 years (SD = 2.91). Additionally, 29% of participants considered themselves more experienced than their classmates, while 82% rated themselves as at least average or above in their experience. Since self-assessment can be more informative than years of experience [29], this suggests that our participants were generally more experienced than the average student.

7 Threats to Validity

In this section, we discuss possible threats to validity and explain how we designed our study to minimize them.

7.1 Construct Validity

Construct validity is concerned with how accurately the study measures what it is supposed to. In this study, one threat to construct validity arises from matching fixations to the correct line of code. This is problematic when participants use peripheral vision and thus do not focus directly on the line of code. Similar to Peitek et al. [30], we mitigated this threat by considering fixations to be on the same line of code if they were less than 100 pixels apart horizontally. Another threat concerns the clarity of the task. To reduce this threat, we used a multiple-choice task. To further prevent misunderstandings, we included a warm-up code snippet with a comparable type of question. Furthermore, the study conductor was present to answer questions, without interfering with the data collection or the execution of the study.

7.2 Internal Validity

Internal validity concerns the extent to which the observed effect can be attributed to the independent variables rather than other factors. To eliminate a bias by the selection of the code snippets, we selected code snippets from a large pool and tested their feasibility with a pilot study. Furthermore, each participant was randomly assigned to one group of identifier names and then provided with all 20 code snippets in a fully randomized order, with each code snippet shown exactly once. Each code snippet was also randomly assigned to either include or omit type annotations, while the task remained the same for all variants.

For our study, we created a controlled environment by using a quiet room without distractions. We closed the blinds and turned on the lights to ensure a consistent environment for the eye tracker. We also minimized fatigue by limiting the study to 45minutes. Additionally, we carefully instructed all participants on how to sit and how to look at the screen.

Participants might have a considerable difference in their programming knowledge. To reduce this threat, we only allowed students with comparable knowledge to participate, ensuring a similar skill level across participants. We assume that all participants had a certain base level of programming knowledge while not yet being at the level of most professional developers.

7.3 External Validity

External validity is concerned with the generalizability of the results. When conducting empirical studies, one must consider the trade-off between enhancing internal or external validity [45]. Our goal for this study was to robustly pinpoint the effect of type annotations on program comprehension and, thus, we targeted internal validity.

Our study exhibits the same threats to external validity as comparable studies using students as participants [17, 33] and small code snippets to test program comprehension. This includes that the code snippets are generally algorithmic and mono-method and do not necessarily compare to more complex real-world projects. Further, our task of understanding code may not be representative for all programming tasks. Furthermore, the code snippets are all written in Python, limiting the generalizability to other programming languages. Our participants were all students enrolled in degrees related to computer science and can thus only to a limited extent be generalized to professional developers.

8 Conclusion

In an eye-tracking study with 35 included participants, we investigated the effect of type annotations and identifier names on program comprehension. We found that for comparatively simple code snippets type annotations of primitive types slow developers down, but otherwise do not strongly affect behavioral measures, visual attention, or subjective difficulty. We obtained the same results when combining type annotations and identifier names. In contrast, the vast majority of participants reported that type annotations helped with program comprehension within this study and in general. This indicates that type annotations might positively affect certain dimensions of program comprehension but that the effect was not fully quantifiable in our study setup. In a nutshell, type information may not be universally useful for program comprehension in all circumstances and is context-dependent.

In future studies, we aim to investigate the more nuanced effects of type annotations on program comprehension in a more ecologically valid setting. This includes opening up the study to a broader population to more easily generalize the study's findings. Furthermore, future studies shall investigate code snippets closer to real-world projects with larger code snippets including complex types and composite types to use the full range of type annotations. This could also entail that researchers create simple code snippets based on a real-life code base with and without type annotations. This would allow us to investigate the effect of type annotations on program comprehension in the context of a complex code base. Ultimately, this paper contributes a reference study for eye-tracking research in software engineering, providing insights into how type annotations influence program comprehension and supporting future investigations into their strategic use in the software development process.

Acknowledgments

We thank all participants of our study. This work has been supported by ERC Advanced Grant 101052182 as well as DFG Grant 389792660 as part of TRR 248 – CPEC.

References

[1] Amine Abbad Andaloussi, Thierry Sorg, and Barbara Weber. 2022. Estimating Developers' Cognitive Load at a Fine-Grained Level Using Eye-Tracking Measures. In *Proc. Int'l Conf. Program Comprehension (ICPC)*. ACM, 111–121.

[2] Roman Bednarik and Markku Tukiainen. 2006. An Eye-Tracking Methodology for Characterizing Program Comprehension Processes. In *Proc. Eye Tracking Research & Application Symposium (ETRA)*. ACM, 125–132.

[3] Yoav Benjamini and Yosef Hochberg. 1995. Controlling the False Discovery Rate: A Practical and Powerful Approach to Multiple Testing. *Journal of the Royal*

Statistical Society: Series B (Methodological) 57, 1 (1995), 289–300.
[4] Annabelle Bergum, Norman Peitek, Maurice Rekrut, Janet Siegmund, and Sven Apel. 2025. On the Influence of the Baseline in Neuroimaging Experiments on Program Comprehension. *ACM Trans. Softw. Eng. Methodol.* (June 2025).
[5] David Binkley, Marcia Davis, Dawn Lawrie, and Christopher Morrell. 2009. To Camelcase or Under_score. In *Int'l Conf. Program Comprehension (ICPC)*. IEEE, 158–167.
[6] Justus Bogner and Manuel Merkel. 2022. 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, 658–669.
[7] Ruven Brooks. 1978. Using a Behavioral Theory of Program Comprehension in Software Engineering. In *Proc. Int'l Conf. Software Engineering (ICSE)*. IEEE, 196–201.
[8] Teresa Busjahn, Roman Bednarik, Andrew Begel, Martha Crosby, James H. Paterson, Carsten Schulte, Bonita Sharif, and Sascha Tamm. 2015. Eye Movements in Code Reading: Relaxing the Linear Order. In *Int'l Conf. Program Comprehension (ICPC)*. IEEE, 255–265.
[9] Teresa Busjahn, Roman Bednarik, and Carsten Schulte. 2014. What Influences Dwell Time During Source Code Reading? Analysis of Element Type and Frequency as Factors. In *Eye Tracking Research and Applications (ETRA)*. ACM, 335–338.
[10] Teresa Busjahn, Carsten Schulte, Bonita Sharif, Simon, Andrew Begel, Michael Hansen, Roman Bednarik, Paul Orlov, Petri Ihantola, Galina Shchekotova, and Maria Antropova. 2014. Eye tracking in computing education. In *Int'l Computing Education Research Conf. (ICER)*. ACM, 3–10.
[11] Simon Butler, Michel Wermelinger, Yijun Yu, and Helen Sharp. 2010. Exploring the Influence of Identifier Names on Code Quality: An Empirical Study. In *Europ. Conf. on Software Maintenance and Reengineering (CSMR)*, Rafael Capilla, Rudolf Ferenc, and Juan C. Dueñas (Eds.). IEEE Computer Society, 156–165.
[12] Benjamin Carter and Steven Luke. 2020. Best Practices in Eye Tracking Research. *Int'l Journal of Psychophysiology* 155 (Sept. 2020), 49–62.
[13] Casey Casalnuovo, Kevin Lee, Hulin Wang, Prem Devanbu, and Emily Morgan. 2020. Do Programmers Prefer Predictable Expressions in Code? *Cogn. Sci.* 44, 12 (2020).
[14] Fabian Deitelhoff, Andreas Harrer, and Andrea Kienle. 2019. The Influence of Different AOI Models in Source Code Comprehension Analysis. In *Int'l Workshop on Eye Movements in Programming (EMIP)*. IEEE, 10–17.
[15] Hubert L. Dreyfus, Stuart E. Drey-fus, and Lotfi A. Zadeh. 1987. Mind over Machine: The Power of Human Intuition and Expertise in the Era of the Computer. *IEEE Expert* 2, 2 (June 1987), 110–111.
[16] Christopher Exton. 2002. Constructivism and Program Comprehension Strategies. In *Int'l Workshop Program Comprehension (IWPC)*. IEEE Computer Society, 281–284.
[17] Robert Feldt, Thomas Zimmermann, Gunnar R Bergersen, Davide Falessi, Andreas Jedlitschka, Natalia Juristo, Jürgen Münch, Markku Oivo, Per Runeson, Martin Shepperd, et al. 2018. Four Commentaries on the Use of Students and Professionals in Empirical Software Engineering Experiments. *Empirical Software Engineering* 23, 6 (2018), 3801–3820.
[18] Stefan Hanenberg, Sebastian Kleinschmager, Romain Robbes, Éric Tanter, and Andreas Stefik. 2014. An Empirical Study on the Impact of Static Typing on Software Maintainability. *Emp. Softw. Eng.* 19, 5 (2014), 1335–1382.
[19] Johannes C. Hofmeister, Janet Siegmund, and Daniel V. Holt. 2019. Shorter Identifier Names Take Longer to Comprehend. *Emp. Softw. Eng.* 24, 1 (Feb 2019), 417–443.
[20] Kenneth Holmqvist. 2011. *Eye Tracking: A Comprehensive Guide to Methods and Measures*. Oxford University Press.
[21] Jerry Mead, Simon Gray, John Hamer, Richard James, Juha Sorva, Caroline St. Clair, and Lynda Thomas. 2006. A Cognitive Approach to Identifying Measurable Milestones for Programming Skill Acquisition. *SIGCSE Bull.* 38, 4 (jun 2006), 182–194.
[22] Roberto Minelli, Andrea Mocci, and Michele Lanza. 2015. I Know What You Did Last Summer-An Investigation of How Developers Spend Their Time. In *Int'l Conf. on Program Comprehension*. IEEE, IEEE, 25–35.
[23] Saul Needleman and Christian Wunsch. 1970. A General Method Applicable to the Search for Similarities in the Amino Acid Sequence of Two Proteins. *Journal of Molecular Biology* 48, 3 (1970), 443–453.
[24] Marcus Nyström and Kenneth Holmqvist. 2010. An Adaptive Algorithm for Fixation, Saccade, and Glissade Detection in Eyetracking Data. *Behavior Research Methods* 42, 1 (2010), 188–204.
[25] Unaizah Obaidellah, Mohammed Al Haek, and Peter C.-H. Cheng. 2018. A Survey on the Usage of Eye-Tracking in Computer Programming. *ACM Comput. Surv.* 51, 1, Article 5 (Jan. 2018), 58 pages.
[26] Sebastian Okon and Stefan Hanenberg. 2016. 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, 1–10.
[27] Norman Peitek. 2022. *A Neuro-Cognitive Perspective of Program Comprehension*. Ph. D. Dissertation. Chemnitz University of Technology, Germany.
[28] Norman Peitek, Sven Apel, Chris Parnin, André Brechmann, and Janet Siegmund. 2021. Program Comprehension and Code Complexity Metrics: An fMRI Study. In *Proc. Int'l Conf. Software Engineering (ICSE)*. IEEE, 524–536.
[29] Norman Peitek, Annabelle Bergum, Maurice Rekrut, Jonas Mucke, Matthias Nadig, Chris Parnin, Janet Siegmund, and Sven Apel. 2022. 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, 120–131.
[30] Norman Peitek, Janet Siegmund, and Sven Apel. 2020. What Drives the Reading Order of Programmers? An Eye Tracking Study. In *Proc. Int'l Conf. Program Comprehension (ICPC)*. ACM, 342–353.
[31] Norman Peitek, Janet Siegmund, Chris Parnin, Sven Apel, Johannes C. Hofmeister, and André Brechmann. 2018. 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, Article 24, 10 pages.
[32] Nancy Pennington. 1987. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology* 19, 3 (1987), 295–341.
[33] Iflaah Salman, Ayse Tosun Misirli, and Natalia Juristo. 2015. Are Students Representatives of Professionals in Software Engineering Experiments?. In *International Conference on Software Engineering*, Vol. 1. 666–676.
[34] Giuseppe Scanniello, Michele Risi, Porfirio Tramontana, and Simone Romano. 2017. Fixing Faults in C and Java Source Code: Abbreviated vs. Full-Word Identifier Names. *ACM Trans. Softw. Eng. Methodol.* 26, 2 (2017), 6:1–6:43.
[35] Andrea Schankin, Annika Berger, Daniel V. Holt, Johannes C. Hofmeister, Till Riedel, and Michael Beigl. 2018. Descriptive Compound Identifier Names Improve Source Code Comprehension. In *Proc. Conf. Program Comprehension (ICPC)*. IEEE, 31–40.
[36] Skipper Seabold and Josef Perktold. 2010. statsmodels: Econometric and Statistical Modeling with Python. In *9th Python in Science Conference*.
[37] Zohreh Sharafi, Bonita Sharif, Yann-Gaël Guéhéneuc, Andrew Begel, Roman Bednarik, and Martha E. Crosby. 2020. A Practical Guide on Conducting Eye Tracking Studies in Software Engineering. *Emp. Softw. Eng.* 25, 5 (2020), 3128–3174.
[38] Zohreh Sharafi, Zéphyrin Soh, and Yann-Gaël Guéhéneuc. 2015. A Systematic Literature Review on the Usage of Eye-Tracking in Software Engineering. *Information and Software Technology* 67 (Nov. 2015), 79–107.
[39] Bonita Sharif and Jonathan I. Maletic. 2010. An Eye Tracking Study on camelCase and under_score Identifier Styles. In *Int'l Conf. Program Comprehension (ICPC)*. IEEE Computer Society, 196–205.
[40] Janet Siegmund. 2016. Program Comprehension: Past, Present, and Future. In *Int'l Conf. Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 5. IEEE, 13–20.
[41] Janet Siegmund, Christian Kästner, Sven Apel, Chris Parnin, Anja Bethmann, Thomas Leich, Gunter Saake, and André Brechmann. 2014. Understanding Understanding Source Code with Functional Magnetic Resonance Imaging. In *Proc. Int'l Conf. Software Engineering (ICSE)*. ACM, 378–389.
[42] Janet Siegmund, Norman Peitek, André Brechmann, Chris Parnin, and Sven Apel. 2020. Studying Programming in the Neuroage: Just a Crazy Idea? *Commun. ACM* 63, 6 (2020), 30–34.
[43] Janet Siegmund, Norman Peitek, Chris Parnin, Sven Apel, Johannes Hofmeister, Christian Kästner, Andrew Begel, Anja Bethmann, and André Brechmann. 2017. Measuring Neural Efficiency of Program Comprehension. In *Proc. Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*. ACM, 140–150.
[44] Janet Siegmund and Jana Schumann. 2015. Confounding Parameters on Program Comprehension: A Literature Survey. *Emp. Softw. Eng.* 20, 4 (Aug 2015), 1159–1192.
[45] Janet Siegmund, Norbert Siegmund, and Sven Apel. 2015. Views on Internal and External Validity in Empirical Software Engineering. In *Proc. Int'l Conf. on Software Engineering*, Vol. 1. IEEE, 9–19.
[46] Samuel Spiza and Stefan Hanenberg. 2014. 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)*. 99–108.
[47] Marvin Wyrich. 2023. Source Code Comprehension: A Contemporary Definition and Conceptual Model for Empirical Investigation. *CoRR* abs/2310.11301 (2023). https://doi.org/10.48550/arXiv.2310.11301
[48] Marvin Wyrich, Justus Bogner, and Stefan Wagner. 2023. 40 Years of Designing Code Comprehension Experiments: A Systematic Mapping Study. *Comput. Surveys* 56, 4 (2023), 1–42.
[49] Xin Xia, Lingfeng Bao, David Lo, Zhenchang Xing, Ahmed E. Hassan, and Shanping Li. 2018. Measuring Program Comprehension: A Large-Scale Field Study with Professionals. In *Proc. Int'l Conf. Software Engineering (ICSE)*. ACM, 584.