Correlates of Programmer Efficacy and Their Link to Experience: A Combined EEG and Eye-Tracking Study

Norman Peitek
Saarland University,
Saarland Informatics Campus
Saarbrücken, Germany

Annabelle Bergum
Saarland University,
Saarland Informatics Campus,
Graduate School of Computer Science
Saarbrücken, Germany

Maurice Rekrut
German Research Center for Artificial Intelligence, Saarland Informatics Campus, Saarbrücken, Germany

Jonas Mucke
Chemnitz University of Technology
Chemnitz, Germany

Matthias Nadig
German Research Center for Artificial Intelligence, Saarland Informatics Campus, Saarbrücken, Germany

Chris Parnin
NC State University
Raleigh, North Carolina, USA

Janet Siegmund
Chemnitz University of Technology
Chemnitz, Germany

Sven Apel
Saarland University,
Saarland Informatics Campus
Saarbrücken, Germany

ABSTRACT

Background: Despite similar education and background, programmers can exhibit vast differences in efficacy. While research has identified some potential factors, such as programming experience and domain knowledge, the effect of these factors on programmers' efficacy is not well understood.
Aims: We aim at unraveling the relationship between efficacy (speed and correctness) and measures of programming experience. We further investigate the correlates of programmer efficacy in terms of reading behavior and cognitive load.
Method: For this purpose, we conducted a controlled experiment with 37 participants using electroencephalography (EEG) and eye tracking. We asked participants to comprehend up to 32 Java source-code snippets and observed their eye gaze and neural correlates of cognitive load. We analyzed the correlation of participants' efficacy with popular programming experience measures.
Results: We found that programmers with high efficacy read source code more targeted and with lower cognitive load. Commonly used experience levels do not predict programmer efficacy well, but self-estimation and indicators of learning eagerness are fairly accurate.
Implications: The identified correlates of programmer efficacy can be used for future research and practice (e.g., hiring). Future research should also consider efficacy as a group sampling method, rather than using simple experience measures.

CCS CONCEPTS

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

ESEC/FSE '22, November 14–18, 2022, Singapore, Singapore

ACM ISBN 978-1-4503-9413-0/22/11.
https://doi.org/10.1145/3540250.3549084

KEYWORDS

Programmer efficacy, program comprehension, cognitive load, electroencephalography, eye tracking

ACM Reference Format:
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 *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '22), November 14–18, 2022, Singapore, Singapore.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3540250.3549084

1 INTRODUCTION

Proficient programmers are essential for providing the critical infrastructure and functioning applications for our modern society [28]. Although the learning strategies, education, and pathways to becoming a programmer may differ, the general expectation is that, with more time and experience, a programmer should generally become more proficient. For example, many research studies of programmers use years of experience [8, 44], education level [17, 38, 57], or employment status [12, 13, 45] as foundational measures for proficiency. While all these choices are sensible, they all implicitly encode the expectation that higher proficiency should correlate with more experience.

However, in practice, reported observations violate this expectation. Hiring managers and technical founders [6] report that they routinely encounter "engineers with years of experience who couldn't competently program" [2], who "struggle with tiny problems" [29], and "*Senior Engineers* who can't write basic code" [1]. The original creator of the infamous *FizzBuzz* interview question [29] only did so after seeing that the "majority of comp sci graduates... and self-proclaimed senior programmers" had difficulty solving simple problems in a timely manner. Research has also found that, in companies, the seniority level showed little correlation to actual programming skill [39], and programmers with

similar education and background can exhibit vast differences in productivity, up to factors of ten [49]. Research is still at a loss when it comes to explaining the cause of these differences, accounting for different trajectories of learning that underlie programming education and training [24, 32], or identifying proficient programmers during a hiring process [46].

In this paper, we explore the idea of unraveling programmer *efficacy*[1]—based on speed and correctness—with the help of programmer *experience*—amount of learning or practice. To improve our understanding of the relationship between efficacy and experience, we conducted a combined electroencephalography (EEG) and eye-tracking study, allowing us to take a close look at how differences in efficacy and experience are related to cognitive differences among programmers. In our study, 37 participants with varying levels of experience performed program-comprehension tasks. We found that programmers with higher efficacy read code more targeted, with shorter fixations, fewer (re)fixations, and skipping more code elements. They also complete their tasks with lower cognitive load, in less time, and make fewer errors than programmers with lower efficacy. Interestingly, we found that commonly used experience measures do not correlate with the observed efficacy, but instead self-estimation and learning indicators have considerable predictive power. To this end, we have identified correlates of high programmer efficacy as well as experience measures that provide a strong link to efficacy.

In summary, we make the following contributions:

- A combined EEG and eye-tracking experiment to investigate programmer efficacy with a diverse participant pool.
- Confirmation of prior results that programmers with high efficacy read source code more efficiently and with lower cognitive load.
- Empirical evidence that conventional experience measures have only poor predictive power for programmer efficacy. Self-estimation and indicators of learning eagerness are better suited.
- An online replication package[2], including experiment design, raw data, and executable analysis scripts.

2 RESEARCH QUESTIONS AND VARIABLES

Our study on programmer efficacy builds on the methodology of previous experiments investigating program comprehension and programming experience. Our aim is to incorporate measures of program comprehension from these experiments into a single coherent study as shown in Table 1. We specifically designed our study to better understand programmer efficacy across a wide range of experience levels in the context of program-comprehension tasks. Programmer efficacy is therefore the independent variable for our experiment. We operationalize programmer efficacy as follows:

$$programmer\ efficacy = \frac{number\ of\ correct\ code\ comprehension\ tasks}{completion\ time\ in\ minutes}$$

Note that programmer efficacy captures both the speed and correctness of a participant's behavior. Due to the strict one-hour time limit of the experiment (see Section 3.1), only fast participants were able to see all snippets before the experiment ended. Our definition of programmer efficacy eliminates effects arising from a difference in the number of attempted tasks and is in line with prior work [34, 69, 78].

To guide our experiment design regarding dependent variables, we defined several research questions, which we introduce next.

2.1 Reading Behavior (RQ_1)

Several studies have shown that experienced programmers show a different reading behavior than novices based on eye-gaze measures. The reading behavior describes a programmer's eye movements while they are comprehending a source-code snippet. Therefore, we state the following research question:

> **RQ_1** Do different levels of programmer efficacy exhibit differences in reading behavior (in terms of navigation strategy and code element coverage)?

Operationalization. For RQ_1, we adopt 7 measures that have been identified in the literature: In a longitudinal study, Al Madi et al. observed differences in the navigation strategy at the token level [4]. Specifically, they analyzed fixation duration and how likely it was that a participant (re)fixated on a token. Experts showed significantly shorter fixation durations, a lower chance of refixating on a token, and a higher chance to skip tokens. Likewise, Aljehane et al. found differences between novices and experts in terms of *code element coverage* [5]. It refers to the number of elements on which a participant fixates during a task in contrast to the total number of code elements. They found that novices read, in particular, more method signatures, variable declarations, identifiers, and keywords. Finally, Busjahn et al. have also identified a difference in code element coverage, in that novices fixate more code elements than experts [13]. They also identified a difference in the *reading order*. However, our replication study revealed that this effect is principally driven by the execution order of the snippets [60]. As our snippets are not balanced for this aspect, we will not consider reading order as a measure; these studies used different measures to pinpoint the participants' reading behavior. In our study, we aim at answering how programmer efficacy affects reading behavior across all described measures to increase comparability.

2.2 Cognitive Load (RQ_2)

Similar to reading behavior, previous studies found that different levels of programming experience can be distinguished by the observed neural correlates of cognitive load, which leads us to our next research question:

> **RQ_2** Do different levels of programmer efficacy exhibit differences in neural correlates of cognitive load?

Operationalization. Related to RQ_2, Crk et al. used event-related desynchronization along alpha and theta powerbands measured by an EEG device to classify participants into two programming experience levels [16]. Similarly, Lee et al. found a similar effect also using EEG, but using a more comprehensive analysis across

[1] Efficacy specifically refers to the ability to quickly produce the intended result. Notably, this differs from expertise, which involves additional facets, such as deep knowledge, effort, and mastery of skills.

[2] https://github.com/brains-on-code/NoviceVsExpert

Table 1: Overview of our research questions and chosen measures.

RQ	Literature	Selected Measure(s)
RQ_1	Al Madi et al. [4]	Fixation durations (singe fixation duration, total fixation duration) and fixation probabilities (skipping probability, single fixation probability, multiple fixation probability)
	Aljehane et al. [5], Busjahn et al. [13]	Code element coverage, code element coverage over time
RQ_2	Lee et al. [45]	Alpha, beta, theta, and gamma power
	Medeiros et al. [52], Holm et al. [35]	Ratio of theta power at frontal brain region and alpha power at parietal brain region
RQ_3		Correlation between efficacy and experience measures

more EEG bands [45]. Ishida and Uwano found an increase in the alpha frequency band for programmers who successfully finished tasks [36]. However, Madeiros et al. compared several EEG measures to distinguish different experience levels and suggested the ratio between theta and alpha power as a measure of cognitive load [52]. Notably, this ratio is commonly used as a cognitive load measure in other fields [35, 40]. We analyze both powerbands and the ratio measure in our study to better understand the link between programmer efficacy and cognitive load.

2.3 Experience Measures (RQ_3)

Finally, we focus on programmer efficacy as a distinguishing factor between our participants. Prior research most commonly uses rather simple experience measures to separate participants (e.g., years of programming). In this vein, we pose our last research question:

> **RQ_3** Do different levels of programmer efficacy correlate with common measures of programming experience?

Operationalization. Many commonly used experience measures can be inconsistent in their predictive power, which led us to use programmer efficacy as the distinguishing factor. In RQ_3, we analyze widely used experience measures[3] and distill the relevant measures that correlate best with the observed programmer efficacy.

3 STUDY DESIGN AND CONDUCT

To answer our research questions, we designed and conducted a study, which we describe in this section. All materials including snippets, tasks, and the experiment script, are available on the project's Web site.[2]

3.1 Experiment Plan

We opted for a within-subject experiment design [15], which we illustrate in Figure 1. We presented up to 32 source-code snippets with a program-comprehension task (cf. Section 3.3), which were pseudo-randomized to avoid learning and fatigue effects. The experiment ended after the 32 program-comprehension tasks were completed or after 60 minutes, whichever happened first. The tasks were presented in three runs of 20 minutes and a voluntary break of 5 minutes between runs. In addition to the program-comprehension tasks, we included a search task, in which participants had to count

[3]We share the full list of experience measures on the project's Web site.

```java
public boolean containsSubstring(String word, String substring) {
    boolean containsSubstring = false;

    for (int i = 0; i < word.length(); i++) {
        for (int j = 0; j < substring.length(); j++) {
            if (i + j > word.length()) {
                break;
            }
            if (word.charAt(i + j) != substring.charAt(j)) {
                break;
            } else {
                if (j == substring.length() - 1) {
                    containsSubstring = true;
                    break;
                }
            }
        }
    }

    return containsSubstring;
}
```

Listing 1: Example source-code snippet with intermediate complexity that checks a string for the existence of a substring. An example input provided to the participant would be `containsSubstring("Example","Sample")` with answer options "`Always False`", "`False`" (correct), "`True`", and "`3`".

brackets. This serves as a baseline for neural activation and is common in neuroscience studies of program comprehension [59, 73]. We chose a 4:1 design, so a participant completes four program-comprehension tasks and one search task. Between tasks, we included a 30 second rest condition, in which participants were instructed to focus their eyes on a fixation cross and relax.

For the comprehension tasks, the participants could choose among four answer options (cf. Listing 1) as well as the option "next" to skip a task. Participants used the left hand to press space as "submit" and "continue" buttons, and the right hand to navigate with the arrow keys between answer options. After a short training session on the experiment flow (i.e., presentation of an example snippet, example input/output task, answering possible clarification questions), participants could use the keyboard without constantly looking at their hands, which minimizes motion artifacts.

Our study was approved by the ethical review board of the faculty of Mathematics and Computer Science at Saarland University.

3.2 Snippet Selection

A crucial element of our experiment are the source-code snippets. We aimed at selecting snippets covering a variety of complexities. This ranges from simple snippets, with only a few lines that can be understood within seconds, to complex source-code snippets that

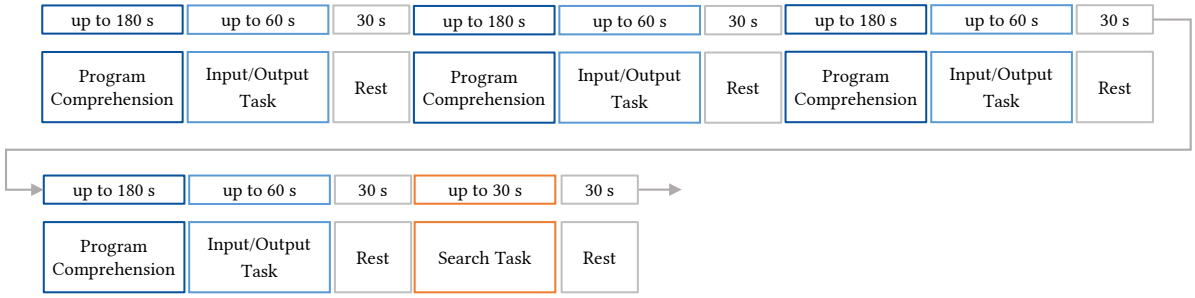

Figure 1: Visualization of the experiment design, which shows the first 4 comprehension tasks. The experiment ended after all 32 comprehension tasks were completed, or, at latest, after 60 minutes. Participants could take a voluntary break after 20 minutes and 40 minutes.

Table 2: Overview of the source-code snippets used in the experiment and the outcomes in terms of correctness and response time. The number of times a snippet was shown is unbalanced due to the randomized presentation and outlier removal (cf. Section 4.1).

Snippet	Correctness (in %)	Response Time (in sec, Mean ± SD)
Ackermann	5/36 (14%)	83.51 ± 53.16
ArrayAverage	30/35 (86%)	31.29 ± 17.03
BinarySearch	13/35 (37%)	55.57 ± 31.16
BinomialCoefficient	10/32 (31%)	77.70 ± 39.83
BinaryToDecimal	13/35 (37%)	62.18 ± 25.27
BogoSort	24/35 (69%)	79.64 ± 42.77
CheckIfLettersOnly	32/33 (97%)	36.81 ± 22.69
ContainsSubstring	31/32 (97%)	56.36 ± 29.74
DropNumber	15/36 (42%)	55.19 ± 25.67
GreatestCommonDivisor	29/31 (94%)	70.86 ± 32.40
HeightOfTree	26/34 (76%)	41.81 ± 23.63
hIndex	19/33 (58%)	100.60 ± 44.14
InsertionSort	34/35 (97%)	67.40 ± 38.21
IsAnagram	26/30 (87%)	101.70 ± 44.67
IsPrime	30/35 (86%)	21.67 ± 12.91
LengthOfLastWord	33/36 (92%)	64.52 ± 21.13
MedianOnSorted	23/30 (77%)	45.44 ± 22.32
Palindrome	35/35 (100%)	26.26 ± 14.57
PermuteString	21/35 (60%)	129.24 ± 50.06
Power	29/33 (88%)	34.08 ± 15.59
RabbitTortoise	11/33 (33%)	103.54 ± 46.45
RecursivePower	29/31 (94%)	23.68 ± 9.18
Rectangle	21/29 (72%)	26.28 ± 18.81
RemoveDoubleChar	30/33 (91%)	40.98 ± 17.68
ReverseArray	30/33 (91%)	40.73 ± 23.21
ReverseQueue	26/33 (79%)	41.26 ± 26.64
SieveOfEratosthenes	19/33 (58%)	105.38 ± 48.16
SignChecker	31/33 (94%)	28.53 ± 12.04
SmallGauss	18/36 (50%)	27.24 ± 16.68
SumArray	32/33 (97%)	13.70 ± 7.58
UnrolledSort	30/34 (88%)	61.85 ± 32.32
Vehicle	34/35 (97%)	36.13 ± 19.43
Overall	789/1072 (74%)	56.02 ± 41.63

require substantial mental effort to comprehend. Thus, our snippets require different levels of cognitive effort to comprehend them. This helps us to comprehensively capture program comprehension in relation to programmer efficacy.

We started our search by selecting Java snippets from a variety of previous studies of program comprehension [13, 59, 73]. We then searched for further snippets implementing algorithms of comparable complexity (in terms of size, nesting depth, and execution length), which lead to a pool of 38 snippets. Three of the authors independently assessed each snippet's complexity and suitability for the study (e.g., whether prior knowledge is necessary to comprehend it). While the snippets are in Java, we aimed at selecting snippets that did not require deep knowledge of the language. This resulted in 32 snippets. We show a sample snippet that checks for the existence of a substring in Listing 1. We list all included snippets in Table 2 and provide them in the replication package.

3.3 Program-Comprehension Task

We subdivided the program-comprehension task into two distinct steps to isolate the underlying cognitive processes of program comprehension and mental calculation of the result. To this end, we first presented the source-code snippet and instructed participants to comprehend it. Once a participant confirmed they comprehended its functionality (by pressing a button), we presented a sample input. Then, the participant had to mentally calculate the resulting output. The sample input was not shown until after the participant fully comprehended the code snippet, to prevent premature mental calculation without fully understanding the snippet. We informed participants of this multiple-step process beforehand and ensured their understanding with two training snippets.

3.4 Experiment Execution and Data Collection

All participants provided their informed consent and completed our experience questionnaire (see Section 3.5). We put the EEG cap on the participant's head, calibrated the eye-tracker, and started the experiment. After the experiment, we conducted a semi-structured interview, including questions on their subjective views about the experiment as well as each snippet.

The EEG laboratory is in a dim-light room with minimized distractions, such as external sounds or mobile devices. Participants sat in a comfortable chair to prevent unnecessary muscle movements to reduce noise and artifacts in the EEG signal. EEG signals were

Table 3: Overview of our participant demographics and 8 out of 49 overall measures from our experience questionnaire. $^{\alpha}$ denotes self-estimated measures on a 1–5 Likert scale [47]. $^{\beta}$ denotes measures that were only collected from (at least part-time) professional programmers.

Participant Demographic/Experience Measure	No./Mean ± SD
Number of (Included) Participants	37
Female	5
Male	31
Non-Binary	1
Age (in Years)	25.95 ± 6.76
Undergraduate/graduate students	27 of 37
… of which work (at least part time)	14 of 27
Full-time professionals	10 of 37
Years of Learning Programming	7.93 ± 6.14
Years of Professional Programming	3.55 ± 4.30
Years of Java Programming	4.54 ± 4.31
Number of Known Programming Languages	5.11 ± 2.02
Comparison to Peers$^{\alpha}$	3.67 ± 0.76
Comparison to 10-Year Professional$^{\alpha}$	2.25 ± 0.94
Hours per Week Spent in Software Engineering$^{\beta}$	24.76 ± 21.08
Hours per Week Spent Programming$^{\beta}$	10.78 ± 11.36

recorded using *LiveAmp 64*[4], which is a 64-channel EEG device. The sampling rate was set to 500 Hz, and the international 10–20 system of electrode placement [37] was used to cover the entire scalp and obtain spatial information from the brain recordings. For simultaneous eye tracking, we used the Tobii Pro Fusion eye-tracker[5] attached to the screen. The eye gaze was recorded with a sampling rate of 250 Hz. The experiment was run with a PsychoPy [58] script (available in the replication package).

3.5 Participants

We recruited participants at Saarland University via e-mail lists and online bulletin boards. Participants received 10 Euro compensation each. The prerequisite for participation was, at least, one year of experience with Java or, at least, three years of experience with a related programming language, such as `C#`. It was important to recruit a diverse set of programmers with a wide range of experience levels to explore the differences and commonalities across different efficacy levels. In Table 3, we provide an overview of our participants' demographics. Based on the conventionally used experience measures, our participant sample exhibits a wide range of programming experience, from programmers in their first year of programming at the university to 30 years of experience in industry.

We invited 39 participants to start the experiment. 38 out of the 39 measured participants completed the experiment.[6] For one participant, the eye-tracker could not be calibrated, but all other modalities are available and the data are included in the analysis. We later excluded one complete participant data set due to their behavioral data consisting of a majority of outliers (cf. Section 4.1), leaving us with 37 participants included in the data set. We based our programming experience questions on a validated questionnaire [72], but extended it to cover more topics for further investigation (e.g., programming-content consumption and production and daily work; the questionnaire is available in the replication package).

4 DATA ANALYSIS

In this section, we describe our data-analysis procedure for eye-tracking data, EEG data, and experience measures.

4.1 Outlier Removal

We started with removing outliers in response time for comprehending a source-code snippet. Specifically, we discarded the slowest 5% (i.e., a response time of over 2 min 32 s) as well as the fastest 5% (under 11 s), but only if a participant chose the option "Next", since some tasks can be rapidly answered by proficient programmers (e.g., the `SignChecker` snippet). This way, data points were only removed when participants did not thoroughly attempt to understand the snippet and program comprehension may not have occurred. With these rules, we removed 9 data points[7] in the lower 5% and 55 in the upper 5%, leaving 1072 data points for further analysis. This led to the removal of more than half of the data points for one (slow) participant, so we excluded the participant's entire data set from the study.

4.2 Eye-Tracking Data

We defined areas-of-interests (AOIs) for each source-code snippet to relate eye-gaze behavior with particular regions and elements of code. For this purpose, we first obtained the abstract syntax tree of each source-code snippet to identify each token and code element. Then, we manually identified higher-order syntactical structures such as the head and body of loops or if-else-statements for each snippet and thereupon defined AOIs based on categories defined in previous work [4, 5].

We preprocessed the eye-tracking data to separate fixations and saccades from the raw stream of `(x,y)` coordinates. For this purpose, we used `I2CM`, which is a noise-robust algorithm [33]. Then, we computed the token-level and element-coverage measures described in Section 2.1.

4.3 EEG Data

Our cognitive load measures are based on a spectral analysis of EEG data. To improve the robustness of our analysis, we calculate cognitive load using two different approaches: theta/alpha power ratio and relative power analysis.

Data Cleaning. First, we removed noise from the EEG data, especially movement artifacts, using established preprocessing methods: We filtered the data using Hamming-windowed finite impulse response (FIR) filters. Power line noise was removed by a notch filter with a lower cutoff frequency of 49 Hz and an upper cutoff frequency of 51 Hz. Second, to obtain a more robust performance in

[4] Brain Products GmbH, https://brainvision.com/products/liveamp-64/
[5] Tobii AB, https://www.tobiipro.com/product-listing/fusion/
[6] One participant aborted the experiment early due to personal time constraints.
[7] One data point here is the response time for one participant to one snippet

the subsequent analysis, we also removed baseline drifts and high-frequent noise with a bandpass filter ranging from 4 to 200 Hz [63].

Due to the nature of the experiment, where participants have to look at different points of the screen, eye movements and other muscle activity cannot be fully avoided. Therefore, we performed blind source separation as a second step to remove corresponding artifacts. The used EEGLAB toolbox [19][8] provides an automated implementation of the independent component analysis (ICA). For each component, it yields a classification with a confidence level, which we can use to automatically reject noisy components. Rejection thresholds for eye and muscle artifacts were set to the default values of 0.9, while components without a clear assignment to a group were rejected at 0.95.

Theta/Alpha Power Ratio. As a first technique, we computed the cognitive load based on the work of Holm et al. [35] and Kartali et al. [40] as the ratio of the relative power of the theta and alpha band of the EEG. We calculated the power spectral density on a sliding window obtaining the mean power within each frequency band of interest. The sliding window had a size of 3 s and was shifted by 0.1 s over each task. The cognitive load measure is obtained by dividing the mean power within a window. By using the sliding window, we can observe the time course of the cognitive load during the tasks.

Relative Power Analysis. As a second technique, we also computed a per-band and per-channel relative power analysis along alpha, beta, gamma, and theta band based on the technique established by Lee et al. [45], which is sensitive to the spatial location of brain activation by considering the channel of the electrode.[9]

4.4 Efficacy and Experience

For RQ_3, we correlated the observed efficacy with all collected numerical and categorical experience measures. To this end, we used Spearman's ρ correlation due to the mix of continuous and ordinal nature of our data.

5 RESULTS AND DISCUSSION

In this section, we present the results of our EEG and eye-tracking data analysis, along with their subsequent interpretation.

Applying our programmer efficacy definition to the data, we find that, on average, 1 task was correctly solved around every 3 minutes (i.e., mean programmer efficacy of 0.33 ± 0.10). We visualize the distribution of participant efficacy in Figure 2, which is a non-normal distribution according to a Shapiro-Wilk test ($W = 0.94, p = 0.046$). It could be normalized by removing the upper tail (2 participants)—neither which were the most experienced programmers—but since we already removed outliers, this appears to be the real distribution in our sample.

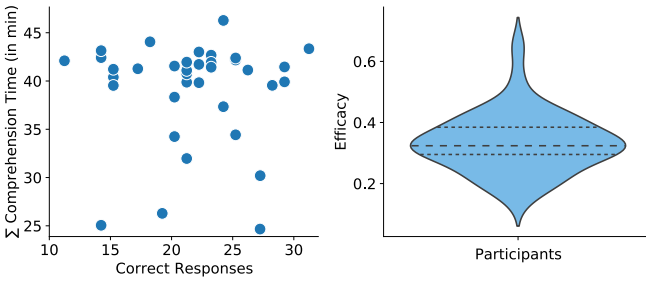

Figure 2: Illustration of behavioral data and the resulting distribution of programmer efficacy among our participants.

5.1 Reading Behavior (RQ_1)

Our eye-tracking data show a change in reading behavior with increasing efficacy levels. Based on the token-level measures from Al Madi et al., we found that increased efficacy leads to:

- shorter (first) fixations ($\rho = -0.14$),
- shorter gaze duration ($\rho = -0.19$),
- a much lower chance that a token is revisited ($\rho = -0.37$), and
- a slightly higher probability that a token is skipped ($\rho = 0.08$).

We visualize these relationships in Figure 3.

Regarding code element coverage, based on Aljehane et al. and Busjahn et al., we find a similar result, such that higher efficacy leads to a lower code element coverage ($\rho = -0.35$). Notably, this difference increases throughout the task, as we illustrate in Figure 4. To underline the difference between programmer with low and high efficacy, we show an example scanpath in Figure 5 (Page 8). Clearly, the programmer with high efficacy requires fewer fixations and refixations to comprehend the source-code snippet.

Overall, this leads to the following answer to our research question:

RQ_1 We can confirm prior results that programmers with higher efficacy read code more efficiently in terms of shorter fixations on fewer code elements. However, some measures show only weak correlations.

Discussion. Our results for RQ_1 confirm prior results and corroborate the theory that proficient programmers read source code more efficiently [55] and are actively looking for an efficient way to solve a task [64]. This supports the view that their knowledge guides their eyes [77]. However, the established measures that we use capture this effect to different degrees. Fixation duration measures show only weak correlations ($\rho = -0.14$), while fixation probabilities show, at best, medium correlations with programmer efficacy ($\rho = -0.37$). These results support that proficient programmers read individual elements faster, and, in particular, focus on the important

[8] Version 2021.1, https://sccn.ucsd.edu/eeglab/

[9] To facilitate comparison with Lee et al.'s work [45], which used two groups, we formed two groups post-hoc based on experimental scores of efficacy. We separated our participants by performance into thirds. For increased potential, we contrasted the high-performing group (i.e., efficacy >= 0.35, n=12) and the low-performing group (i.e., efficacy <= 0.29, n=13), leaving out the middle group.

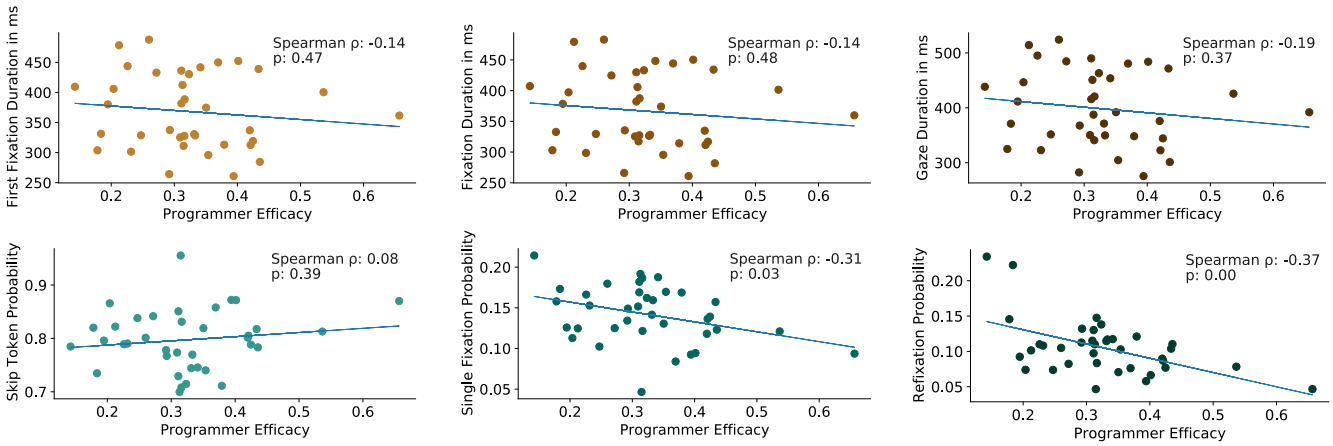

Figure 3: Spearman's ρ correlation between programmer efficacy and different eye-tracking measures: Programmers with high efficacy show an efficient reading strategy in terms of shorter fixations, skipping more code elements, and fewer (re)fixations.

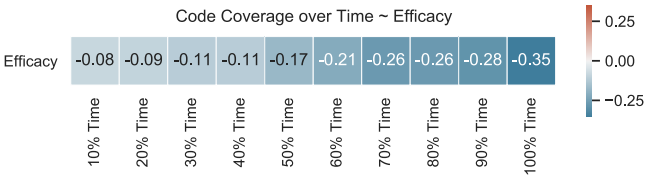

Figure 4: Spearman's ρ correlation between programmer efficacy and code element coverage over time. In the beginning of a task, the difference in code element coverage for different efficacy levels is not substantial. It grows throughout the task in that programmer with high efficacy skip more code elements.

elements—skipping several parts that they deem unimportant. This narrow focus on important elements is further highlighted in the code element coverage measure. Programmers with high efficacy read fewer code elements throughout the task, increasing over time as shown in Figure 4. Even at the end of a task, they still only fixated on 69 % of the tokens, on average—skipping tokens they rate as irrelevant.

While a narrow focus enables high performers to quickly solve a task, they might overlook relevant elements. For example, research in psychology has shown that experts' internal filtering heuristics can lead them to miss relevant information and make mistakes [65]. In programming, this effect was observed with experts missing obvious syntax errors that novices consistently notice [60].

5.2 Cognitive Load (RQ_2)

The observed ratio of alpha and theta band as an indicator of cognitive load can be analyzed regarding different features, which we summarize in Table 4. While generally the cognitive load appears to be lower for programmers with higher efficacy ($\rho = -0.09$), the correlations are weak. Still, it has to be taken into account that programmers with higher efficacy tend to complete their tasks faster. In Figure 6, we visualize the cognitive load over time depending on efficacy, which shows that the level of cognitive load generally

Table 4: Spearman's ρ correlation between cognitive-load measures and programmer efficacy.

Alpha/Theta Measures of Cognitive Load	Correlation with Programmer Efficacy
$\text{Ratio}_{\text{mean}}$	−0.09
$\text{Ratio}_{\text{median}}$	−0.08
$\text{Ratio}_{\text{min}}$	0.19
$\text{Ratio}_{\text{max}}$	−0.15

stays lower, and with fewer spikes, for programmers with higher efficacy.

Results from our powerband analysis confirm results from Lee et al. [45]. Like Lee et al., we separated our participants into a high-performing and low-performing group (cf. Section 4.3). We found a lower beta power for programmers with higher efficacy, and higher alpha and gamma power, which is illustrated in Figure 7.

RQ_2 The cognitive load is slightly lower for programmers with higher efficacy, despite faster completion times. Programmers with higher efficacy further experience fewer high spikes of cognitive load. This is in line with previous findings of lower beta and higher alpha and gamma power for programmers with higher efficacy.

Discussion. The results of RQ_2 indicate that programmers with higher efficacy can not only comprehend source code faster, but also with less mental effort. This could be explained by an increased *neural efficiency*, which has been shown in other fields [54] or as an effect of source code structures serving as beacons for program comprehension [73], but not as a factor of proficiency in programming. One explanation would be a difference in underlying cognitive processing: Experts may see the presented snippets as a task, that is, they have a strategy to solve the problem, and simply need to implement the solution. This is unlike novice programmers, who also

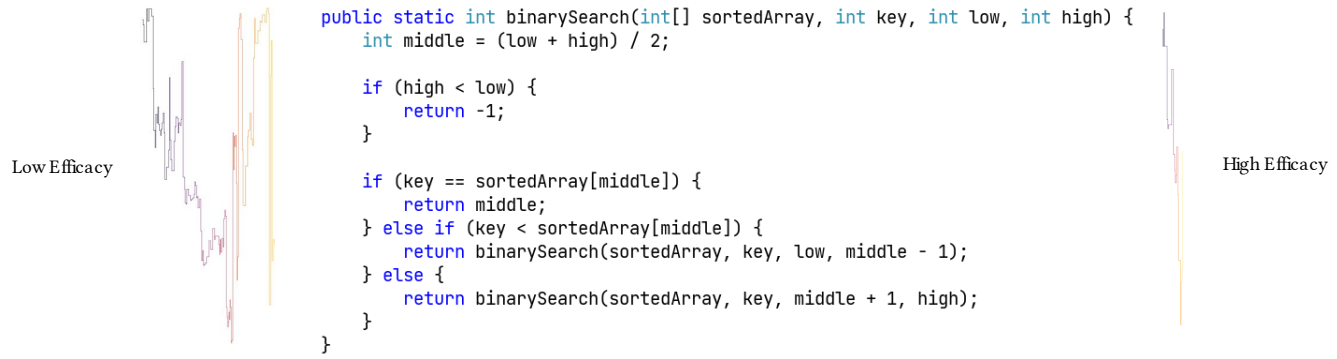

Figure 5: Example scanpath of a programmer with low efficacy (left) and with high efficacy (right). The lines indicate the vertical position in the snippet during comprehension of the code snippet. Clearly, the programmer with high efficacy displays a more targeted, efficient reading strategy.

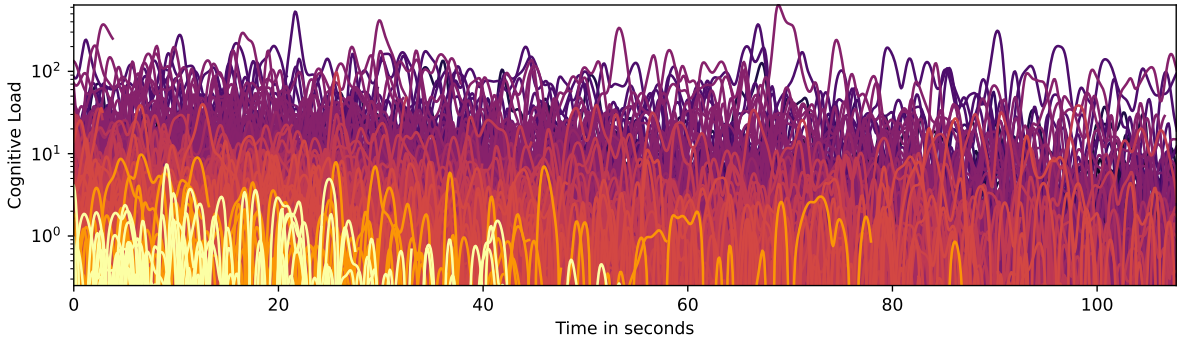

Figure 6: Visualization of cognitive-load measure (i.e., ratio of alpha and theta band) over comprehension-task time. Each line is one participant with the line color indicating their efficacy level.

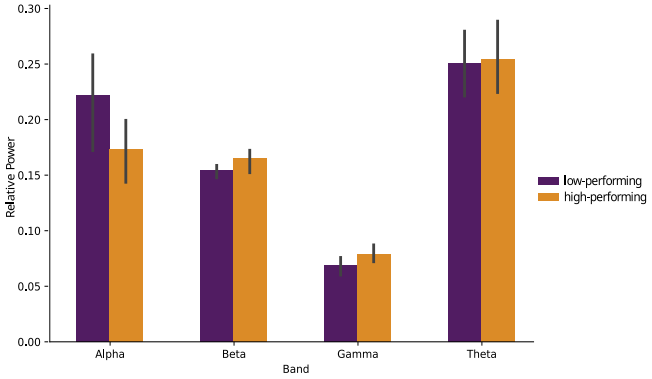

Figure 7: Visualization of relative power along alpha, beta, theta, and gamma bands for the high-performing (efficacy top 33%) and low-performing group (efficacy bottom 33%).

need to find a strategy first, then implement the solution, which leads to higher cognitive load and longer response times [50]. Due to the lower cognitive-load levels, programmers with high efficacy likely can sustain longer periods of work. By contrast, programmers with lower efficacy are more likely to be overwhelmed by constant spikes of cognitive load (Figure 6).

Synthesis of RQs$_{1-2}$. Our study was inspired by several prior experiments investigating program comprehension with various operationalizations and different definitions of experience. We aimed at finding a common ground across their measures, specifically when focusing on programmer efficacy. Overall, our study confirmed the accuracy of some of these measures, but to different degrees. Programmers with higher efficacy can be particularly identified due to their efficient reading strategy and lower spikes in cognitive load. This is not only important for future research, but also has practical implications. For example, a lot of hiring processes use technical interviews in front of a whiteboard, which artificially introduce stress and high cognitive load [10]. An alternative solution to evaluate potential talent with measures that allow for accurate and quick responses, without inducing unnecessary stress and cognitive load, would be private interviews, comprehension tasks, or other alternative interview methods, for example, as proposed by Behroozi et al. [9].

Regarding programming language design, the combination of eye tracking and EEG may have potential. Future work shall explore whether we can use eye-tracking data to pinpoint spikes in

cognitive load to specific code elements, similar to Fakhoury et al.'s work with functional near-infrared spectroscopy (fNIRS) [26] or our prior work on combining eye tracking and fMRI [61]. Such combination may reveal code structures that are particularly difficult for programmers to comprehend.

5.3 Experience Measures (RQ_3)

Table 5: Spearman's ρ correlation between various experience measures and programmer efficacy. Bold text highlights medium and strong correlations ($\rho \geq 0.3$). All measures, including a full correlation matrix, raw data, and results are available on the project's Web site.

Measure of Programming Experience		ρ Corr. with Efficacy
Time	Years of Programming	0.15
	Years of Professional Programming	0.14
	Years of Java	0.04
	Years at Work	−0.06
Self-Estimation	Experience Logical Paradigm	0.07
	Experience Functional Paradigm	0.22
	Experience Object-Oriented Paradigm	0.17
	Experience Imperative Paradigm	**0.32**
	Comparison to Peers	**0.59**
	Comparison to 10-Year Programmer	**0.38**
Work Hours per Week (Professionals Only, n=26)	**Overall**	**0.43**
	Programming	0.06
	Code Review	**0.47**
	Meeting	0.16
	Tests	**0.35**
	Deploy	0.10
	Mentoring	**0.32**
	Learning	**0.32**
	Other	0.27
	Number of Programming Languages	**0.42**

In Table 5, we show the strength of correlations between programmer efficacy and a subset of popular measures of programming experience. Notably, some commonly used experience measures (e.g., years of (professional) programming [8]) show little predictive power to our participants' efficacy. But, several other dimensions of experience show, at least, medium-strength correlations. Specifically, self-estimation in comparison to peers ($\rho = 0.59$) and a 10-year professional programmer ($\rho = 0.38$) show that our participants seem to be keenly aware of their proficiency level. The number of known programming languages also shows a medium-strength correlation ($\rho = 0.42$). For the subset of professional programmers ($n = 26$), several questions that capture learning eagerness (e.g., learning, mentoring, code review) show medium correlations. However, strikingly, the time professionals spend on pure programming does not correlate to their performance in our experiment ($\rho = 0.06$).

Overall, these results allow us to answer this research question:

RQ_3 Programmer efficacy does not correlate with commonly used experience measures, such as years of programming. Self-estimation and indicators of learning eagerness show, at least, medium correlations with observed programmer efficacy.

Discussion. Our experiment highlights two fundamental problems for studying programmers. One issue is that programming is such a diverse field with different technologies that require different skill sets, so it is incredibly difficult to accurately measure a programmer's experience. Therefore, many researchers rely on simple measures, such as years of programming. However, this can become problematic if the difference between the selected measure and the *actual* proficiency level becomes a significant confounding factor [74]. Our experiment, in line with prior research [20], underlines how limited common experience measures are in predicting programmers' efficacy. Thus, future research must carefully consider collecting more comprehensive experience data, in particular, when using it to separate the participants into groups. While this may not be completely new insights, our experiment further corroborates this critical point. The use of too simplistic measures can potentially weaken empirical studies of programmers and their conclusions.

6 THREATS TO VALIDITY

In this section, we discuss threats to construct, internal, and external validity of our study.

6.1 Construct Validity

Our programmer efficacy measure may deviate from a participant's *true* efficacy due to the nature of our selected snippets and programming language. We mitigated this threat by selecting a variety of source-code snippets such that prior knowledge plays a minuscule role, and we ensured sufficient Java knowledge for each participant.

6.2 Internal Validity

We have operationalized program comprehension in a multiple-step design, in which participants first had to comprehend a source-code snippet, then compute an input/output task, and finally select the output from four answers. Clearly, program comprehension is a multi-faceted phenomenon for which a variety of operationalizations are possible [22], but our approach ensures that participants genuinely comprehended each snippet.

Regarding data collection, we limited the experiment to a maximum of 1 hour (with two breaks) to avoid fatigue effects. Depending on each participant's speed, this led to an unequal number of collected data points (17 of 37 participants completed all 32 snippets). While we could have ended the experiment after, for example, 25 instead of 32 snippets for everyone, we would have lost around 10% of data. Therefore, we chose a small potential bias as a trade-off for a notably larger data set (to gain more statistical power and external validity). We mitigated the threat of an unequal number of snippets by randomizing the presentation order of snippets.

Our results indicate a strong influence regarding measures of several programming activities, including the amount of code review and testing, on observed efficacy. This relationship between observed efficacy and time spent with these identified programming activities may be overly emphasized by our experiment design since they are close in nature to our program-comprehension task.

6.3 External Validity

Due to our focus on internal validity, our presented study is limited regarding its generalizability. While our participant sample covers a wide range of experience and proficiency levels, the presented tasks were limited to algorithms in an object-oriented programming language. Thus, our study on program comprehension may not generalize to all software-engineering activities, including comprehension of large code bases and other programming paradigms. Still, our study pinned down an effect with high internal validity; future work shall replicate, vary, and extend the setup toward external validity.

7 RELATED WORK

Our study is at the intersection between the fields of program comprehension, eye tracking, neuroimaging, and programming expertise, which we each discuss below.

Program Comprehension. There is a multitude of program-comprehension studies [3, 13, 25, 45, 71, 75], on which we build by reusing source-code snippets as a foundation for our experiment. In the early years, behavioral or reflective studies were popular [70]. Our study shares a similar design with them in terms of snippets and tasks, but our observed measures are more in line with the recent movement toward more objective measurement methods, such as eye tracking or EEG [70], which we discuss next.

Eye Tracking. Eye tracking enables researchers to objectively observe attention during complex tasks and has increased in popularity, including program comprehension [67]. Again, there is a multitude of studies that use various eye-gaze measures [4, 5, 7, 8, 11, 13, 18, 55, 57, 60, 68]. We have used recently established eye-gaze measures for differentiation of programmer expertise [4, 5, 13], but we applied these measures to a larger pool of diverse programmers than before and included insights from neural correlates via EEG data. With efficacy, we also arguably used a better way to separate participants (cf. Section 5.3).

Neural Correlates of Program Comprehension. In addition to eye tracking, some studies have employed methods to observe neural correlates of program comprehension. Some have used fMRI [14, 27] or fNIRS [26, 53], others have used EEG [43, 45, 48, 51, 52, 80]. Our study adopted the measures of cognitive load of Lee et al. (i.e., alpha and beta power) [45] and Medeiros et al. (i.e., ratio of theta and alpha) [52], but it differs in terms of the goal and the participant pool. Lee et al. focused on classification between two distinct groups, while Medeiros et al. focused on software metrics [52]. By contrast, we focused on the programmer and investigated their efficacy on a continuous distribution, without the need to create two groups.

Programming Experience and Expertise. There have been multiple waves of program-comprehension research regarding expertise. In the early years, several studies have devised theories of expertise, in particular related to plans [31], mental models [62, 79], strategy [42] and knowledge [30]. More recently, researchers have focused on visual attention [8] and implementation style [56]. Our study differs in that we operationalize on a more narrow scope of expertise based on programmer efficacy, rather than measures of knowledge, mental representations, or other intangible factors. We argue our operationalization offers a higher relevance to practical and educational problems as well as a clearer definition over an obfuscated experience measure, because of its better link to task performance.

In summary, our experiment is a fusion of established research designs from different fields and is novel by integrating several measures in one experiment. Furthermore, we focused on inviting a diverse participant pool as well as on efficacy as a separating factor (while still collecting a wide array of experience measures).

8 PERSPECTIVES AND CONCLUSION

Measuring Programming Experience. For future studies, one major obstacle is to select the correct measure of programming experience. Cognitive psychology has investigated the relationship between experience and expertise in many fields beyond programming [66]. One consistent finding is that the length of experience can be part of expertise, as people continue to acquire skill [21], but is not everything [65]. In programming, studies from the 1990s identified a similar effect, in that there are two elements at play: the time and learning [76]. In our questionnaire, we aimed to capture many aspects of programming experience, including measures of learning, but also regarding content creation and work-time distribution. One theme that consistently showed a strong connection with observed efficacy is the eagerness of learning and inquisitiveness. The number of known programming languages, how much time is spent on mentoring, and on code review are all highly relevant factors. Psychology describes this conscious effort to keep learning as *deliberate practice*. It also highlights that not necessarily the length of practice, but the intensity and goal-orientation is critical to achieve expertise in a topic [23]. The effect of deliberate practice has been observed in programming education and practice, as well. For example, students show higher ability when participating in programming competitions [81]. In an industry survey, software engineers shared the notion that a great software engineer is shown by their open-mindedness and eagerness to learn. They considered the ability to learn as more important than technical skill [46]. This was later confirmed in an experimental setting, in which company seniority level showed little correlation to actual programming skill [39]. Overall, the results of our experiment underlined that the learning component is a highly relevant measure to capture efficacy, which leads us to suggest it as important measure(s) for future experiments.

Implications for Research. Beyond learning measures, our results indicate that future research could maximize its potential when collecting self-estimation measures. This result is in line with a similar finding by Siegmund et al., which also highlighted self-estimated comparison as a differentiating factor among students [72]. In the same vein, Kleinschmager and Hanenberg found self-estimated comparison, at least, as effective as pretests or university grades in

its predictive power of efficacy [41]. Our shared experience questionnaire provides a proven basis for others to use to capture several dimensions of programming experience.

Implications for Industry. From a practical perspective, our empirical results provide scientific evidence that technical interviews may be necessary for establishing efficacy, since common measures selected for decisions (e.g., years of experience), have limited to no predictive power. Our questionnaire provides a few promising directions for operationalizing a more effective method for assessing efficacy in industry, such as peer review and mentoring experience, peer comparison, and learning behaviors. However, in high-stake decisions, such as promotion or hiring, the degree to which self-estimation can be reliable remains an open question.

Conclusion. Experience, expertise, and efficacy are three dimensions of characteristics of programmers that are not well understood. In particular, the relationship between them is unclear. In this paper, we have presented correlates of efficacy in terms of reading order and cognitive load across a wide range of programmers.

Commonly used experience measures do not correlate well to observed efficacy. Instead, we underline to use self-estimation and learning eagerness as more accurate measures for programming experience.

Despite encouraging results, future work shall explore programmer efficacy in more detail. For example, the link between reading behavior and cognitive load could be explored to understand causalities. Is the more efficient reading strategy of programmers with high efficacy the cause for lower cognitive load, or vice versa?

ACKNOWLEDGEMENTS

We thank all participants of our study. Furthermore, we thank Julia Hess, Tobias Jungbluth, and Johannes Ihl for their support during data acquisition.

Rekrut's work is supported by the German Federal Ministry of Education and Research (01IS12050). Parnin's work is supported by the National Science Foundation under grant number 2045272. Siegmund's work is supported by DFG grant SI 2045/2⁄2. Apel's work is supported by ERC Advanced Grant 101052182 as well as DFG Grant 389792660 as part of TRR 248 – CPEC.

REFERENCES

[1] 2020. The Myth of the Developer That Can't Code. https://news.ycombinator.com/item?id=22862871
[2] 2021. Facebook Senior Software Engineer Interview. https://news.ycombinator.com/item?id=25658098
[3] Shulamyt Ajami, Yonatan Woodbridge, and Dror Feitelson. 2017. Syntax, Predicates, Idioms - What Really Affects Code Complexity?. In *Proc. Int'l Conf. Program Comprehension (ICPC)*. Springer, 66–76.
[4] Naser Al Madi, Cole Peterson, Bonita Sharif, and Jonathan Maletic. 2021. From Novice to Expert: Analysis of Token Level Effects in a Longitudinal Eye Tracking Study. In *Proc. Int'l Conf. Program Comprehension (ICPC)*. IEEE, 172–183.
[5] Salwa Aljehane, Bonita Sharif, and Jonathan Maletic. 2021. Determining Differences in Reading Behavior Between Experts and Novices by Investigating Eye Movement on Source Code Constructs During a Bug Fixing Task. In *Proc. Symposium on Eye Tracking Research & Applications (ETRA)*. ACM, Article 30, 6 pages.
[6] Jeff Atwood. 2007. Why Can't Programmers... Program? https://blog.codinghorror.com/why-cant-programmers-program/
[7] Jennifer Bauer, Janet Siegmund, Norman Peitek, Johannes Hofmeister, and Sven Apel. 2019. Indentation: Simply a Matter of Style or Support for Program Comprehension? In *Proc. Int'l Conf. Program Comprehension (ICPC)*. ACM, 11.
[8] Roman Bednarik. 2012. Expertise-Dependent Visual Attention Strategies Develop over Time during Debugging with Multiple Code Representations. *Int'l Journal of Human-Computer Studies* 70, 2 (2012), 143–155.
[9] Mahnaz Behroozi, Chris Parnin, and Titus Barik. 2019. Hiring Is Broken: What Do Developers Say About Technical Interviews?. In *Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 1–9.
[10] Mahnaz Behroozi, Shivani Shirolkar, Titus Barik, and Chris Parnin. 2020. Debugging Hiring: What Went Right and What Went Wrong in the Technical Interview Process. In *Proc. Int'l Conf. Software Engineering: Software Engineering in Society (ICSE-SEIS)*. IEEE, 71–80.
[11] Tanja Blascheck and Bonita Sharif. 2019. Visually Analyzing Eye Movements on Natural Language Texts and Source Code Snippets. In *Proc. Symposium on Eye Tracking Research & Applications (ETRA)*. ACM, 14:1–14:9.
[12] Jean-Marie Burkhardt, Françoise Détienne, and Susan Wiedenbeck. 2002. Object-Oriented Program Comprehension: Effect of Expertise, Task and Phase. *Empirical Softw. Eng.* 7, 2 (2002), 115–156.
[13] Teresa Busjahn, Roman Bednarik, Andrew Begel, Martha Crosby, James Paterson, Carsten Schulte, Bonita Sharif, and Sascha Tamm. 2015. Eye Movements in Code Reading: Relaxing the Linear Order. In *Proc. Int'l Conf. Program Comprehension (ICPC)*. IEEE, 255–265.
[14] João Castelhano, Isabel Duarte, Carlos Ferreira, João Durães, Henrique Madeira, and Miguel Castelo-Branco. 2019. The Role of the Insula in Intuitive Expert Bug Detection in Computer Code: An fMRI Study. *Brain Imaging and Behavior* 13, 3 (2019), 623–637.
[15] Gary Charness, Uri Gneezy, and Michael Kuhn. 2012. Experimental Methods: Between-Subject and Within-Subject Design. *Journal of Economic Behavior & Organization* 81, 1 (2012), 1–8.
[16] Igor Crk and Timothy Kluthe. 2014. Toward Using Alpha and Theta Brain Waves to Quantify Programmer Expertise. In *Int'l Conf. Engineering in Medicine and Biology Society*. IEEE, 5373–5376.
[17] Martha Crosby, Jean Scholtz, and Susan Wiedenbeck. 2002. The Roles Beacons Play in Comprehension for Novice and Expert Programmers. In *Annual Conf. Psychology of Programming Interest Group (PPIG)*. 58–73.
[18] Martha Crosby and Jan Stelovsky. 1990. How Do We Read Algorithms? A Case Study. *Computer* 23, 1 (1990), 25–35.
[19] Arnaud Delorme and Scott Makeig. 2004. EEGLAB: An Open Source Toolbox for Analysis of Single-Trial EEG Dynamics Including Independent Component Analysis. *Journal of Neuroscience Methods* 134, 1 (2004), 9–21.
[20] Oscar Dieste, Alejandrina Aranda, Fernando Uyaguari, Burak Turhan, Ayse Tosun, Davide Fucci, Markku Oivo, and Natalia Juristo. 2017. Empirical Evaluation of the Effects of Experience on Code Quality and Programmer Productivity: An Exploratory Study. *Empirical Softw. Eng.* 22, 5 (2017), 2457–2542.
[21] Hubert Dreyfus and Stuart Dreyfus. 1986. *Mind over Machine: The Power of Human Intuition and Expertise in the Era of the Computer.*
[22] Alastair Dunsmore and Marc Roper. 2000. *A Comparative Evaluation of Program Comprehension Measures.* Technical Report EFoCS 35-2000. Department of Computer Science, University of Strathclyde.
[23] Anders Ericsson and Andreas Lehmann. 1996. Expert and Exceptional Performance: Evidence of Maximal Adaptation to Task Constraints. *Annual Review of Psychology* 47, 1 (1996), 273–305.
[24] Marisa Exter, Secil Caskurlu, and Todd Fernandez. 2018. Comparing Computing Professionals' Perceptions of Importance of Skills and Knowledge on the Job and Coverage in Undergraduate Experiences. *ACM Transactions on Computing Education (TOCE)* 18, 4 (2018), 1–29.
[25] Sarah Fakhoury, Yuzhan Ma, Venera Arnaoudova, and Olusola Adesope. 2018. The Effect of Poor Source Code Lexicon and Readability on Developers' Cognitive Load. In *Proc. Int'l Conf. Program Comprehension (ICPC)*. IEEE, 286–28610.
[26] Sarah Fakhoury, Devjeet Roy, Yuzhan Ma, Venera Arnaoudova, and Olusola Adesope. 2020. Measuring the Impact of Lexical and Structural Inconsistencies on Developers? Cognitive Load during Bug Localization. *Empirical Softw. Eng.* 3 (2020), 2140–2178. Issue 25.
[27] Benjamin Floyd, Tyler Santander, and Westley Weimer. 2017. Decoding the Representation of Code in the Brain: An fMRI Study of Code Review and Expertise. In *Proc. Int'l Conf. Software Engineering (ICSE)*. IEEE, 175–186.
[28] Vahid Garousi, Gorkem Giray, Eray Tuzun, Cagatay Catal, and Michael Felderer. 2019. Closing the Gap Between Software Engineering Education and Industrial Needs. *IEEE Software* 37, 2 (2019), 68–77.
[29] Imran Ghory. 2007. Using FizzBuzz to Find Developers who Grok Coding. https://imranontech.com/2007/01/24/using-fizzbuzz-to-find-developers-who-grok-coding/
[30] David Gilmore. 1990. Expert Programming Knowledge: A Strategic Approach. *Psychology of Programming* (1990), 223–234.
[31] David Gilmore and Thomas Green. 1988. Programming Plans and Programming Expertise. *The Quarterly Journal of Experimental Psychology* 40, 3 (1988), 423–442.
[32] Wouter Groeneveld, Joost Vennekens, and Kris Aerts. 2021. Identifying Non-Technical Skill Gaps in Software Engineering Education: What Experts Expect But Students Don't Learn. *ACM Transactions on Computing Education (TOCE)* 22, 1 (2021), 1–21.

[33] Roy Hessels, Diederick Niehorster, Chantal Kemner, and Ignace Hooge. 2017. Noise-Robust Fixation Detection in Eye Movement Data: Identification by Two-Means Clustering (I2MC). *Behavior Research Methods* 49, 5 (2017), 1802–1823.

[34] Johannes Hofmeister, Janet Siegmund, and Daniel Holt. 2019. Shorter Identifier Names Take Longer to Comprehend. *Empirical Softw. Eng.* 24, 1 (2019), 417–443.

[35] Anu Holm, Kristian Lukander, Jussi Korpela, Mikael Sallinen, and Kiti Müller. 2009. Estimating Brain Load from the EEG. *TheScientificWorldJOURNAL* 9 (2009), 639–651.

[36] Toyomi Ishida and Hidetake Uwano. 2019. Synchronized analysis of eye movement and EEG during program comprehension. In *Proc. Int'l Workshop on Eye Movements in Programming (EMIP)*. IEEE, 26–32.

[37] Herbert Jasper. 1958. Report of the Committee on Methods of Clinical Examination in Electroencephalography. *Electroencephalogr. Clin. Neurophysiol.* 10 (1958), 370–375.

[38] Ahmad Jbara and Dror Feitelson. 2017. How Programmers Read Regular Code: A Controlled Experiment Using Eye Tracking. *Empirical Softw. Eng.* 22, 3 (2017), 1440–1477.

[39] Magne Jorgensen, Gunnar Bergersen, and Knut Liestol. 2020. Relations Between Effort Estimates, Skill Indicators, and Measured Programming Skill. *IEEE Trans. Softw. Eng.* 47, 12 (2020), 2892–2906.

[40] Aneta Kartali, Milica M. Janković, Ivan Gligorijević, Pavle Mijović, Bogdan Mijović, and Maria Chiara Leva. 2019. Real-Time Mental Workload Estimation Using EEG. In *Human Mental Workload: Models and Applications*. Springer International Publishing, 20–34.

[41] Sebastian Kleinschmager and Stefan Hanenberg. 2011. How to Rate Programming Skills in Programming Experiments? A Preliminary, Exploratory, Study Based on University Marks, Pretests, and Self-Estimation. In *Proc. Workshop on Evaluation and Usability of Programming Languages and Tools*. ACM, 15–24.

[42] Jürgen Koenemann and Scott Robertson. 1991. Expert Problem Solving Strategies for Program Comprehension. In *Proc. Conf. Human Factors in Computing Systems (CHI)*. ACM, 125–130.

[43] Makrina Kosti, Kostas Georgiadis, Dimitrios Adamos, Nikos Laskaris, Diomidis Spinellis, and Lefteris Angelis. 2018. Towards an Affordable Brain Computer Interface for the Assessment of Programmers' Mental Workload. *Int.Ĵ J. Human-Computer Studies* 115 (2018), 52–66.

[44] Thomas LaToza, David Garlan, James Herbsleb, and Brad Myers. 2007. Program Comprehension as Fact Finding. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*. ACM, 361–370.

[45] Seolhwa Lee, Andrew Matteson, Danial Hooshyar, SongHyun Kim, JaeBum Jung, GiChun Nam, and Heuiseok Lim. 2016. Comparing Programming Language Comprehension between Novice and Expert Programmers Using EEG Analysis. In *Proc. Int'l Conf. on Bioinformatics and Bioengineering (BIBE)*. IEEE, 350–355.

[46] Paul Luo Li, Amy Ko, and Jiamin Zhu. 2015. What Makes a Great Software Engineer?. In *Proc. Int'l Conf. Software Engineering (ICSE)*, Vol. 1. IEEE, 700–710.

[47] Rensis Likert. 1932. A Technique for the Measurement of Attitudes. *Archives of Psychology* 22, 140 (1932), 1–55.

[48] Yu-Tzu Lin, Yi-Zhi Liao, Xiao Hu, and Cheng-Chih Wu. 2021. EEG Activities During Program Comprehension: An Exploration of Cognition. *IEEE Access* 9 (2021), 120407–120421.

[49] Steve McConnell. 2011. What Does 10x Mean? Measuring Variations in Programmer Productivity. In *Making Software*. O'Reilly & Associates, Inc., 567–574.

[50] Jerry Mead, Simon Gray, John Hamer, Richard James, Juha Sorva, Caroline Clair, and Lynda Thomas. 2006. A Cognitive Approach to Identifying Measurable Milestones for Programming Skill Acquisition. *ACM SIGCSE Bulletin* 38, 4 (2006), 182–194.

[51] Julio Medeiros, Ricardo Couceiro, João Castelhano, Castelo Branco, Gonçalo Duarte, Catarina Duarte, João Durães, Henrique Madeira, Paulo Carvalho, and César Teixeira. 2019. Software Code Complexity Assessment Using EEG Features. In *Proc. Int'l Conf. Engineering in Medicine and Biology Society*. IEEE, 1413–1416.

[52] Júlio Medeiros, Ricardo Couceiro, Gonçalo Duarte, João Durães, João Castelhano, Catarina Duarte, Miguel Castelo-Branco, Henrique Madeira, Paulo de Carvalho, and César Teixeira. 2021. Can EEG Be Adopted as a Neuroscience Reference for Assessing Software Programmers' Cognitive Load? *Sensors* 21, 7 (2021), 2338.

[53] Takao Nakagawa, Yasutaka Kamei, Hidetake Uwano, Akito Monden, Kenichi Matsumoto, and Daniel German. 2014. Quantifying Programmers' Mental Workload During Program Comprehension Based on Cerebral Blood Flow Measurement: A Controlled Experiment. In *Proc. Int'l Conf. Software Engineering (ICSE)*. ACM, 448–451.

[54] Aljoscha Neubauer and Andreas Fink. 2009. Intelligence and Neural Efficiency. *Neuroscience & Biobehavioral Reviews* 33, 7 (2009), 1004 – 1023.

[55] Markus Nivala, Florian Hauser, Jürgen Mottok, and Hans Gruber. 2016. Developing Visual Expertise in Software Engineering: An Eye Tracking Study. In *Global Engineering Education Conference (EDUCON)*. IEEE, 613–620.

[56] Francisco Ortin, Oscar Rodriguez-Prieto, Nicolas Pascual, and Miguel Garcia. 2020. Heterogeneous Tree Structure Classification to Label Java Programmers According to Their Expertise Level. *Future Generation Computer Systems* 105 (2020), 380–394.

[57] Patrick Peachock, Nicholas Iovino, and Bonita Sharif. 2017. Investigating Eye Movements in Natural Language and C++ Source Code - A Replication Experiment. In *Augmented Cognition. Neurocognition and Machine Learning*. Springer, 206–218.

[58] Jonathan Peirce, Jeremy Gray, Sol Simpson, Michael MacAskill, Richard Höchenberger, Hiroyuki Sogo, Erik Kastman, and Jonas Lindeløv. 2019. PsychoPy2: Experiments in Behavior Made Easy. *Behavior Research Methods* 51, 1 (2019), 195–203.

[59] 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)*. ACM, 524–536.

[60] 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.

[61] Norman Peitek, Janet Siegmund, Chris Parnin, Sven Apel, and André Brechmann. 2018. Toward Conjoint Analysis of Simultaneous Eye-Tracking and fMRI Data for Program-Comprehension Studies. In *Proc. Int'l Workshop on Eye Movements in Programming (EMIP)*. ACM, 1:1–1:5.

[62] Nancy Pennington. 1987. Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs. *Cognitive Psychology* 19, 3 (1987), 295–341.

[63] Maurice Rekrut, Mansi Sharma, Matthias Schmitt, Jan Alexandersson, and Antonio Krüger. 2020. Decoding Semantic Categories from EEG Activity in Object-Based Decision Tasks. In *Proc. Int'l Winter Conf. Brain-Computer Interface (BCI)*. IEEE, 1–7.

[64] Tobias Roehm, Rebecca Tiarks, Rainer Koschke, and Walid Maalej. 2012. How Do Professional Developers Comprehend Software?. In *Proc. Int'l Conf. Software Engineering (ICSE)*. IEEE, 255–265.

[65] James Shanteau. 1992. Competence in Experts: The Role of Task Characteristics. *Organizational Behavior and Human Decision Processes* 53, 2 (1992), 252–266.

[66] James Shanteau and Thomas Stewart. 1992. Why Study Expert Decision Making? Some Historical Perspectives and Comments. *Organizational Behavior and Human Decision Processes* 53, 2 (1992), 95 – 106.

[67] 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 (2015), 79–107.

[68] Bonita Sharif, Michael Falcone, and Jonathan Maletic. 2012. An Eye-Tracking Study on the Role of Scan Time in Finding Source Code Defects. In *Proc. Symposium on Eye Tracking Research & Applications (ETRA)*. ACM, 381–384.

[69] Ben Shneiderman and Richard Mayer. 1979. Syntactic/Semantic Interactions in Programmer Behavior: A Model and Experimental Results. *Int'l J. Parallel Programming* 8, 3 (1979), 219–238.

[70] Janet Siegmund. 2016. Program Comprehension: Past, Present, and Future. In *Int'l Conf. Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 13–20.

[71] 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.

[72] Janet Siegmund, Christian Kästner, Jörg Liebig, Sven Apel, and Stefan Hanenberg. 2014. Measuring and Modeling Programming Experience. *Empirical Softw. Eng.* 19, 5 (2014), 1299–1334.

[73] 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. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*. ACM, 140–150.

[74] Janet Siegmund and Jana Schumann. 2015. Confounding Parameters on Program Comprehension: A Literature Survey. *Empirical Softw. Eng.* 20, 4 (2015), 1159–1192.

[75] Elliot Soloway and Kate Ehrlich. 1984. Empirical Studies of Programming Knowledge. *IEEE Trans. Softw. Eng.* 10, 5 (1984), 595–609.

[76] Harold Stanislaw, Beryl Hesketh, Sylvia Kanavaros, Tim Hesketh, and Ken Robinson. 1994. A Note on the Quantification of Computer Programming Skill. *International Journal of Human-Computer Studies* 41, 3 (1994), 351–362.

[77] Marie Vans, Anneliese von Mayrhauser, and Gabriel Somlo. 1999. Program Understanding Behavior During Corrective Maintenance of Large-Scale Software. *Int.'l Journal of Human-Computer Studies* 51, 1 (1999), 31–70.

[78] Susan Wiedenbeck. 1985. Novice/Expert Differences in Programming Skills. *Int.'l Journal of Man-Machine Studies* 23, 4 (1985), 383–390.

[79] Susan Wiedenbeck, Vikki Fix, and Jean Scholtz. 1993. Characteristics of the Mental Representations of Novice and Expert Programmers: An Empirical Study. *Int.'l Journal of Man-Machine Studies* 39, 5 (1993), 793–812.

[80] Martin Yeh, Dan Gopstein, Yu Yan, and Yanyan Zhuang. 2017. Detecting and Comparing Brain Activity in Short Program Comprehension Using EEG. In *Frontiers in Education Conference*. IEEE, 1–5.

[81] Fanghui Zha and Yong Wang. 2021. Correlation Analysis of Subject Competition and Programming Ability for Novice Programmers. In *Proc. Int'l Symp. System and Software Reliability (ISSSR)*. IEEE, 25–31.