UNIVERSITY of York

This is a repository copy of *Exploring the Impact of Source Code Linearity on the Programmer's Comprehension of API Code Examples*.

White Rose Research Online URL for this paper:
https://eprints.whiterose.ac.uk/218768/

Version: Accepted Version

Proceedings Paper:

Alharbi, Seham and Kolovos, Dimitris orcid.org/0000-0002-1724-6563 (2024) Exploring the Impact of Source Code Linearity on the Programmer's Comprehension of API Code Examples. In: Proceedings - 2024 32nd IEEE/ACM International Conference on Program Comprehension, ICPC 2024. 32nd IEEE/ACM International Conference on Program Comprehension, ICPC 2024, 15-16 Apr 2024 IEEE International Conference on Program Comprehension . IEEE Computer Society , PRT , pp. 236-240.

https://doi.org/10.1145/3643916.3644395

Reuse

This article is distributed under the terms of the Creative Commons Attribution (CC BY) licence. This licence allows you to distribute, remix, tweak, and build upon the work, even commercially, as long as you credit the authors for the original work. More information and the full terms of the licence here: https://creativecommons.org/licenses/

Takedown

If you consider content in White Rose Research Online to be in breach of UK law, please notify us by emailing eprints@whiterose.ac.uk including the URL of the record and the reason for the withdrawal request.

White Rose
university consortium
Universities of Leeds, Sheffield & York

eprints@whiterose.ac.uk
https://eprints.whiterose.ac.uk/

Exploring the Impact of Source Code Linearity on the Programmers' Comprehension of API Code Examples

Seham Alharbi*
saaa528@york.ac.uk
University of York
York, United Kingdom

Dimitris Kolovos
dimitris.kolovos@york.ac.uk
University of York
York, United Kingdom

ABSTRACT

Context: Application Programming Interface (API) code examples are an essential knowledge resource for learning APIs. However, a few user studies have explored how the structural characteristics of the source code in code examples impact their comprehensibility and reusability.
Objectives: We investigated whether the (a) linearity and (b) length of the source code in API code examples affect users' performance in terms of correctness and time spent. We also collected subjective ratings.
Methods: We conducted an online controlled code comprehension experiment with 61 Java developers. As a case study, we used the API code examples from the Joda-Time Java library. We had participants perform code comprehension and reuse tasks on variants of the example with different lengths and degrees of linearity.
Findings: Participants demonstrated faster reaction times when exposed to linear code examples. However, no substantial differences in correctness or subjective ratings were observed.
Implications: Our findings suggest that the linear presentation of a source code may enhance initial example understanding and reusability. This, in turn, may provide API developers with some insights into the effective structuring of their API code examples. However, we highlight the need for further investigation.

CCS CONCEPTS

• **Software and its engineering** → **Documentation**; *Programming by example*; • **Human-centered computing** → *Empirical studies in HCI*.

KEYWORDS

API comprehension, API code examples, source code linearity, human factors in software engineering, controlled experiment

ACM Reference Format:
Seham Alharbi and Dimitris Kolovos. 2024. Exploring the Impact of Source Code Linearity on the Programmers' Comprehension of API Code Examples. In *32nd IEEE/ACM International Conference on Program Comprehension (ICPC '24), April 15–16, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3643916.3644395

*Seham Alharbi is also affiliated with the College of Computer, Qassim University, Buraydah, Saudi Arabia.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICPC '24, April 15–16, 2024, Lisbon, Portugal
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0586-1/24/04…$15.00
https://doi.org/10.1145/3643916.3644395

1 INTRODUCTION

Working through Application Programming Interface (API) code examples has been proven to be the most preferred learning strategy for both beginner and experienced API users [16]. Surprisingly, little is known about how the different source code structures in these examples affect their comprehensibility and reusability. Furthermore, existing work (discussed in Section 5) appears to be focused on examining the impact of several source code characteristics on comprehension only in the context of generic software. Therefore, to fill this gap, the study presented in this paper focuses specifically on API code examples. Moreover, unlike existing studies, we examine different source code constructs that illustrate the same API usage and functionality, narrowing the evaluation focus to the constructs' impact on comprehension. We also assess an additional concept, which is the impact of the examined source code structures on the reusability of API code examples.

In this study, we are particularly interested in exploring the impact of two source code aspects: the degree of linearity and length. Linear source code refers to code that can be read primarily in a sequential order without interference from interdependent methods or classes. Considering the absence of jumps between method definitions in such a code, we hypothesise it may be easier to comprehend. In addition, linear API code examples may be easier to reuse and adapt into one's codebase, as they typically contain a single self-contained method that can be copied and edited, as opposed to non-linear code examples that involve multiple methods.

Through our study, we aim to help API developers understand how to structure their API code examples more effectively, thereby enhancing the examples' comprehensibility and reusability. This, in turn, would promote the learnability of their APIs.

This work is a part of larger research [3] in which we develop tools and techniques to enhance the maintainability and comprehensibility of API code examples. All data collected or used in this study is available in our replication package.[1]

2 METHODOLOGY

2.1 Research Questions

We intended to answer the following research questions:

- **RQ1:** In terms of correctness and time spent, how does the linearity of an API code example impact a programmer's performance in tasks that require code comprehension?

[1]Replication package: https://figshare.com/s/52e11ece2f39bac64bcb

RQ2: What effects does the length of a linear API code example[2] have on its comprehensibility and reusability?
RQ3: Does the degree of linearity in a non-linear API example affect its comprehensibility and reusability?

2.2 Study Design

This study was conducted online to allow access to a large and more diverse pool of participants. We utilised Gorilla [1], a widely used online experiment builder, which provided all the features we needed in our study (randomisation, counterbalancing presentation of formatted source code, accurate reaction times and integration with participant recruitment platforms). We recruited Java developers from Prolific[3] – a participant recruitment platform for online research – and assigned them to two main groups: linear vs non-linear (between-subjects). Each group consisted of two sub-groups that correspond to the treatment categories shown in Table 1. Participant assignment to groups was fully randomised and balanced, with a 1:1 ratio. Each participant completed two code comprehension and reuse tasks from the same treatment category (highlighted in the same colour in Table 1). The order in which each participant received tasks was also randomised to eliminate any potential order effects. Ethical approval was obtained before the study was conducted.

2.3 Independent Variables

We considered a single independent variable: the source code structure of API code examples. Two primary source code factors were systematically varied: code linearity, which is manipulated using the source code linearity metric (i) proposed by Peitek et al. [17]; and code length, which is varied by adjusting the number of lines of code (LOC). As shown in Table 1, we combined these two factors and generated four treatment categories: linear-short, linear-long, and non-linear with varying levels of linearity (i), ranging from $(10.00 < i \leq 15.00)$ to $(15.00 < i \leq 20.00)$. These selected values reflect a diverse spectrum of code linearity.

2.4 Dependent Variables

We measured three dependent variables: reaction time, correctness and subjective rating.

Time Duration Marking. For the comprehension phase, reaction time was defined as the amount of time that elapsed between a participant's initial view of an API code example and submission of their overall comprehension rating Similarly, for the code-reuse phase, reaction time was defined as the time between a participant's first view of the required code-reuse task and the submission of their solution code.

Judging Correctness. We ensured marking consistency by defining a set of correctness categories and criteria[4]: correct (A), almost correct (B), partially correct (C), incorrect (D) and absent (F).

[2]Please refer to our (replication package → examples) for some sample code illustrating linear and non-linear API code examples.
[3]https://www.prolific.com
[4]Detailed criteria are available in our replication package.

2.5 Participants

Pilot. To validate the study design, we conducted a pilot with four participants (average age 28.5 ± 9.5; average years of Java programming 3.2 ± 1.5). Based on the results, we reduced the number of tasks assigned to each participant from four to two to minimise the experiment's overall duration. We also changed the online IDE[5] used due to its slow execution time and enhanced the wording of the code optimisation question. The data collected in the pilot study was not used in the final analysis.

Pre-screening survey. In addition to the pre-screeners provided by Prolific, we created a separate programming knowledge survey to assess participants' programming knowledge before they participated in the study. This was essential since recent research revealed that, while recruitment platforms, such as Prolific, greatly mitigate self-selection bias and some security issues, their pre-screeners may not always be reliable [7, 18, 19]. In this survey, we used the basic knowledge questions and time limit recommended by Danilova et al. [7]. Participants who correctly answered all the questions within the time limit were manually invited to participate in the study.

Study. We recruited 61 Java developers from 14 countries with varying levels of programming experience. Only 19 (31%) participants stated that they previously used the Joda-Time Java library. Among them, only eight (42%) said that they used it more than once, and none of them reported regular usage. The participants' prior programming experience was assessed using a validated questionnaire that is based on self-estimates [8, 22]. Each participant was compensated £10 for their time and effort. Additional information about the participant demographics is shown in Table 2.

2.6 Material

API code examples. We chose the Joda-Time[6] Java library because it addresses a well-known concept (i.e. date and time handling). Joda-Time met our selection criteria of: 1) not requiring prior domain knowledge that would pose an unnecessary challenge to participants; 2) being well-documented and 3) not being too popular so that an average Java developer would not necessarily be familiar with it. Furthermore, we intended to utilise the code examples available on the Joda-Time documentation page.[6] However, we found that these examples were not complex enough. Thus, we decided to create our own examples.

As shown in Table 1, we developed four API code examples, each of which demonstrated a distinct usage of Joda-Time. This variation of examples was important to minimise the risk of any potential learning effect arising from within the examples. We then created linear and non-linear versions of each example. We strove to make these examples look as natural as possible by: 1) properly documenting them and 2) letting them depict real-world scenarios such as data manipulation or meeting scheduling. For the non-linear versions of the examples, we refactored the linear version by extracting some functionalities into a set of utility methods and replacing the extracted code with method calls using the extract method refactoring technique. Examples within the same treatment category were relatively comparable in terms of length, complexity, degree of linearity and the number of utility method calls.

[5]https://replit.com
[6]https://www.joda.org/joda-time/

Table 1: API code examples and their variants, metric values and study results. Variants sharing the same colour belong to the same treatment category. The imbalance in the number of responses (N) between two variants of the same example is due to our exclusion of responses with inaccurately reported break times.

API Code Example	Variant	Metrics				N	Correctness	Comprehension		Reuse	
		LOC	Complexity	Linearity *(i)*	# Method Calls			Median Reaction Time	p-value*	Median Reaction Time	p-value*
Date Example	Linear	31	7	0.00	0	17	15 (88%)	44s	**0.01**	6m 11s	0.36
	Non-linear	14	1	12.95	5	13	8 (62%)	1m 38s		7m 14s	
Chronology Example	Linear	25	4	0.00	0	18	10 (56%)	43s	**0.03**	3m 37s	**0.10**
	Non-linear	11	1	11.41	4	14	13 (93%)	1m 49s		6m 15s	
Duration Example	Linear	35	7	0.00	0	16	9 (56%)	3m 31s	0.60	16m 18s	0.40
	Non-linear	21	2	19.47	6	12	5 (42%)	2m 17s		24m 59s	
Interval Example	Linear	46	7	0.00	0	14	10 (71%)	1m 14s	0.33	9m 21s	**0.06**
	Non-linear	33	5	19.43	6	12	9 (75%)	2m 49s		12m 56s	
Overall	**Linear**					65	44 (68%)	58s		7m 17s	
	Non-linear					51	35 (69%)	1m 50s		10m 13s	

* Mann–Whitney U test

Table 2: Participant demographics.

Category	n=61
Student	43 (70%)
Professional Developer	18 (30%)
Programming Experience (in Years)	5.9 ± 3.3
Java Programming Experience (in Years)	3.2 ± 2.2
Familiarity with Joda-Time	19 (31%)
Male	54 (89%)
Female	7 (11%)
Age (in Years)	25.4 ± 6.1

Tasks. Oftentimes, when API users turn to code examples to learn a new API, they typically have a specific problem in mind. They are hoping that the code in the example they are reviewing will be reusable. If this is possible, they copy and paste the example, then modify its source code by adding or deleting statements to match their needs [10, 16]. In our study, we wanted to simulate this behaviour. Therefore, each code-reuse task had two parts: 1) code modification, in which participants were required to make changes to address a specific problem; and 2) code optimisation, in which we asked them to remove any unnecessary code that did not directly contribute to their task solution. The tasks were generally easy and designed to be solved with a few edits. Each API code example had a unique task that remained the same for both versions of the example.

2.7 Experiment Procedure

After obtaining their consent, we asked the participants to complete a demographics questionnaire. Subsequently, each of them was randomly assigned two code examples from the same treatment category. This means that each participant completed two distinct code-reuse tasks.

We designed each task to be completed in four sequential parts. The first was the comprehension part, in which participants were asked to review the example and rate their own understanding. The next part pertained to instructions, in which participants were given a link to an online IDE[7] that contained a Java project of the example, with Joda-Time imported and ready to use. Participants were also instructed on how to download the example if they preferred using their own IDE. The third part was the code-reuse task, in which participants answered a two-part question (as explained in Section 2.6) and pasted their solution code in a given text box. The final part involved post-task questions, in which participants were asked to rate how difficult it was to reuse the code, whether they employed the provided online IDE or their own, and report any break time (if any was taken).

We only measured the time spent on two of the four parts: comprehension time (part 1) and reuse time (part 3). The rationale for separating the comprehension and reuse of the same code example was to reduce participants' use of the 'as-needed' program comprehension strategy [9, 15, 24].

[7]https://www.jdoodle.com

2.8 Data Analysis

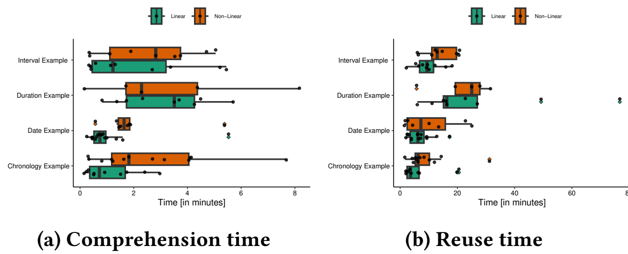
We manually analysed the correctness of responses for each task. First, we converted the categories mentioned in Section 2.4 to numerical values (correct (A) = 100%, almost correct (B) = 70%, partially correct (C) = 40%; both incorrect (D) and absent (F) = 0%). We applied the same scale to both parts of the code-reuse task (i.e. code modification and code optimisation). However, when calculating the overall task score, we assigned more weight (90%) to the first part of the task, as it required greater effort than the second part, which accounted for only 10% of the total weight. Responses with an overall score of 60% or higher were considered correct. This overall correctness threshold ensures that participants achieve at least 70% in the first part of the task.

When analysing reaction times automatically captured by Gorilla [1], we only considered correct responses. We used the Shapiro-Wilk test [20] to assess the normality of reaction times and correctness as well as Levene's test [13] to evaluate variance homogeneity. The findings indicated non-normality and unequal variances for both correctness and reaction times. Therefore, to test for statistically significant differences, we used a non-parametric test, the Mann–Whitney U test (Wilcoxon rank-sum test), with a significance level of $\alpha = 0.05$.

3 RESULTS AND DISCUSSION

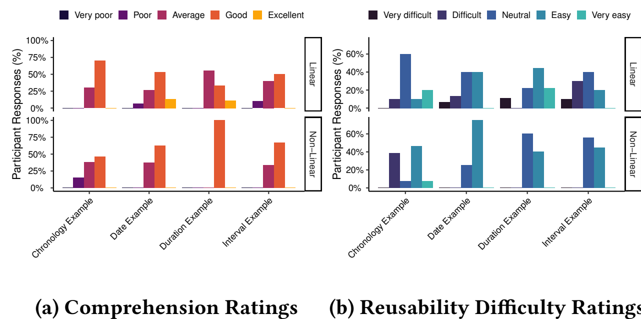
As shown in Table 1 and Figure 1, participants generally spent less time comprehending and reusing linear code examples (both in mean and median reaction times). This observation suggests that the source code linearity in an API code example may affect a programmer's performance. This influence has a greater impact on comprehension and is statistically significant when the linear API code example is also short (e.g. date and chronology examples). Moreover, in terms of reusability, there appears to be a trend towards significance in two of the API code examples (chronology and interval examples), as reflected by their moderate p-values of 0.10 and 0.06, respectively. However, the impact on correctness (Mann–Whitney U test, $W = 1690, p = 0.428$) and subjective rating (as shown in Figure 2) was not substantial **(RQ1)**.

The Mann–Whitney U test revealed a significant difference in both comprehension ($W = 125, p = 0.004$) and reusability ($W = 99, p = 0.000$) between the groups that received linear-short and linear-long API code examples **(RQ2)**. Similarly, participants spent less time reusing the non-linear code examples when the linearity value (i) was lower ($i < 15.00$, Mann–Whitney U test: $W = 68, p = 0.003$). Notably, unlike the comparison in RQ1, this comparison is based on code examples illustrating different API usage; thus, the significant differences in participants' performance could be due to variations in the implemented API functionality and required tasks **(RQ3)**.



(a) Comprehension time **(b) Reuse time**

Figure 1: The time spent on (a) comprehending, and (b) reusing the API code examples used in the study. Each boxplot represents the responses for one version (linear or non-linear) of a single example.



(a) Comprehension Ratings **(b) Reusability Difficulty Ratings**

Figure 2: Participants' subjective ratings of API code example comprehension (a) and reusability difficulty (b).

4 THREATS TO VALIDITY

Construct Validity. To mitigate construct validity threats, we used specific metrics to manipulate our independent variables. We created code examples and tasks that represented real-world scenarios and ensured a consistent correctness evaluation. However, since the experiment was conducted online, we had limited control over participants' activities. To address this, we asked them to self-report break times, which were subtracted from the time spent on solving the task, and disclose whether they used their own IDEs. While this approach provided insights into participants' behaviour, it was not entirely conclusive.

Internal Validity. We reduced internal validity threats by randomly assigning participants to treatment groups and randomising the order in which they viewed tasks. We also administered a validated programming experience questionnaire and pre-screened participants for their programming knowledge.

External Validity. One potential threat to external validity was the study's limited scope. It focused solely on the API of one Java library and included only a few code examples. Also, 70% of the participants were students, which limited the generalisability of our findings. The study's virtual setting and the use of an online IDE, which lacks features such as auto-completion and error checking, may not fully reflect the conditions of a traditional coding environment.

5 RELATED WORK

A large number of studies investigated the impact of various source code characteristics on programmers' code comprehension. These characteristics include the use of intermediate variables [6], certain syntactic structures such as `ifs` and `for` loops [2], different identifier names [4, 12] as well as naming conventions [5, 21]. Moreover, some studies explored the effects of more global factors such as the order of methods [11], code regularity [14], and the linearity of source code and reading order [17].

6 CONCLUSION AND FUTURE WORK

In this paper, we investigated the impact of source code linearity and length on the comprehensibility and reusability of API code examples. We chose code examples from the Joda-Time Java library and manipulated their structure. Furthermore, we recruited 61 Java developers, assigned each one of them two code examples, and asked them to complete code-reuse tasks. This study found that participants demonstrated relatively faster reaction times when working with linear API examples.

For future work, we intend to expand this study by incorporating a broader range of APIs from diverse domains. This will involve utilising a larger set of code examples with varying levels of linearity and increasing the number of participants. Also, to better capture participants' activities, we plan to conduct this experiment in a laboratory setting. Additionally, we are interested in determining whether the activities of participants, as they work with code examples of different linearity, still align with the activities reported in existing studies on the COIL[8] model [10, 23].

[8]Collection and Organization of Information for Learning.

REFERENCES

[1] 2023. Gorilla Experiment Builder. https://gorilla.sc

[2] Shulamyt Ajami, Yonatan Woodbridge, and Dror G. Feitelson. 2019. Syntax, predicates, idioms — what really affects code complexity? *Empirical Software Engineering* 24, 1 (2 2019), 287–328. https://doi.org/10.1007/s10664-018-9628-3

[3] Seham Alharbi, Dimitris Kolovos, and Nicholas Matragkas. 2022. Synthesising Linear API Usage Examples for API Documentation. In *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 607–611.

[4] Eran Avidan and Dror G. Feitelson. 2017. Effects of Variable Names on Comprehension: An Empirical Study. In *IEEE 25th International Conference on Program Comprehension*. IEEE Computer Society, 55–65. https://doi.org/10.1109/ICPC.2017.27

[5] Dave Binkley, Marcia Davis, Dawn Lawrie, Jonathan I. Maletic, Christopher Morrell, and Bonita Sharif. 2013. The impact of identifier style on effort and comprehension. *Empirical Software Engineering* 18, 2 (4 2013), 219–276. https://doi.org/10.1007/s10664-012-9201-4

[6] Roee Cates, Nadav Yunik, and Dror G. Feitelson. 2021. Does Code Structure Affect Comprehension? On Using and Naming Intermediate Variables. In *IEEE International Conference on Program Comprehension*, Vol. 2021-May. IEEE Computer Society, 118–126. https://doi.org/10.1109/ICPC52881.2021.00020

[7] Anastasia Danilova, Alena Naiakshina, Stefan Horstmann, and Matthew Smith. 2021. Do you really code? Designing and evaluating screening questions for online surveys with programmers. In *Proceedings - International Conference on Software Engineering*. IEEE Computer Society, 537–548.

[8] Janet Feigenspan, Christian K¨astner, J¨org Liebig, Sven Apel, and Stefan Hanenberg. 2012. Measuring Programming Experience. In *2012 20th IEEE International Conference on Program Comprehension*. IEEE, 73–82. https://doi.org/10.1109/ICPC.2012.6240511

[9] Dror G. Feitelson. 2021. Considerations and Pitfalls in Controlled Experiments on Code Comprehension. In *IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*. 106–117. https://doi.org/10.1109/ICPC52881.2021.00019

[10] Gao Gao, Finn Vpichick, Michelle Ichinco, and Caitlin Kelleher. 2020. Exploring Programmers' API Learning Processes: Collecting Web Resources as External Memory. In *2020 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 1–10.

[11] Yorai Geffen and Shahar Maoz. 2016. On method ordering. In *IEEE International Conference on Program Comprehension*, Vol. 2016-July. IEEE Computer Society, 1–10. https://doi.org/10.1109/ICPC.2016.7503711

[12] Johannes C. Hofmeister, Janet Siegmund, and Daniel V. Holt. 2019. Shorter identifier names take longer to comprehend. *Empirical Software Engineering* 24, 1 (2 2019), 417–443. https://doi.org/10.1007/s10664-018-9621-x

[13] Levene Howard. 1960. Robust tests for equality of variances. *Contributions to probability and statistics* (1960), 278–292.

[14] Ahmad Jbara and Dror G. Feitelson. 2014. On the effect of code regularity on comprehension. In *22nd International Conference on Program Comprehension (ICPC 2014)*. Association for Computing Machinery, 189–200. https://doi.org/10.1145/2597008.2597140

[15] David Littman, Jeannine Pinto, Stanley Letovsky, and Elliot Soloway. 1987. Mental models and software maintenance. *Journal of Systems and Software* 7, 4 (1987), 341–355. https://www.sciencedirect.com/science/article/pii/0164121287900331

[16] Michael Meng, Stephanie Steinhardt, and Andreas Schubert. 2018. Application Programming Interface Documentation: What Do Software Developers Want? *Journal of Technical Writing and Communication* 48, 3 (2018), 295–330. https://doi.org/10.1177/0047281617721853

[17] Norman Peitek, Janet Siegmund, and Sven Apel. 2020. What drives the reading order of programmers? an eye tracking study. In *28th International Conference on Program Comprehension (ICPC '20)*. ACM, 342–353. https://doi.org/10.1145/3387904.3389279

[18] Brittany Reid, Markus Wagner, Marcelo d'Amorim, and Christoph Treude. 2022. Software Engineering User Study Recruitment on Prolific: An Experience Report. In *International Workshop on Recruiting Participants for Empirical Software Engineering (RoPES'22)*, Vol. 5. Association for Computing Machinery.

[19] Daniel Russo. 2022. Recruiting Software Engineers on Prolific. In *the1st International Workshop on Recruiting Participants for Empirical Software Engineering (RoPES 2022)*, Vol. 5. Association for Computing Machinery.

[20] Samuel Shapiro and Martin Wilk. 1965. An analysis of variance test for normality (complete samples). *Biometrika* 52, 3/4 (1965), 591–611.

[21] Bonita Sharif and Jonathan I. Maletic. 2010. An eye tracking study on camelcase and under-score identifier styles. In *IEEE 18th International Conference on Program Comprehension*. 196–205. https://doi.org/10.1109/ICPC.2010.41

[22] Janet Siegmund, Christian Kästner, Jörg Liebig, Sven Apel, and Stefan Hanenberg. 2014. Measuring and modeling programming experience. *Empirical Software Engineering* 19, 5 (2014), 1299–1334. https://doi.org/10.1007/s10664-013-9286-4

[23] Sören Sparman and Carsten Schulte. 2023. Analysing the API learning process through the use of eye tracking. In *the 2023 Symposium on Eye Tracking Research and Applications (ETRA)*. 1–6.

[24] Anneliese Von Mayrhauser and A. Marie Vans. 1998. Program Understanding Behavior During Adaptation of Large Scale Software. In *the 6th International Workshop on Program Comprehension (IWPC'98)*. 164–172.