The Effect of Poor Source Code Lexicon and Readability on Developers' Cognitive Load

Sarah Fakhoury, Yuzhan Ma
Venera Arnaoudova
Washington State University
School of Electrical Engineering and Computer Science
{sfakhour,yma1,varnaoud}@eecs.wsu.edu

Olusola Adesope
Washington State University
Educational Psychology Program, College of Education
olusola.adesope@wsu.edu

ABSTRACT

It has been well documented that a large portion of the cost of any software lies in the time spent by developers in understanding a program's source code before any changes can be undertaken. One of the main contributors to software comprehension, by subsequent developers or by the authors themselves, has to do with the quality of the lexicon, (i.e., the identifiers and comments) that is used by developers to embed domain concepts and to communicate with their teammates. In fact, previous research shows that there is a positive correlation between the quality of identifiers and the quality of a software project. Results suggest that poor quality lexicon impairs program comprehension and consequently increases the effort that developers must spend to maintain the software. However, we do not yet know or have any empirical evidence, of the relationship between the quality of the lexicon and the cognitive load that developers experience when trying to understand a piece of software. Given the associated costs, there is a critical need to empirically characterize the impact of the quality of the lexicon on developers' ability to comprehend a program.

In this study, we explore the effect of poor source code lexicon and readability on developers' cognitive load as measured by a cutting-edge and minimally invasive functional brain imaging technique called functional Near Infrared Spectroscopy (fNIRS). Additionally, while developers perform software comprehension tasks, we map cognitive load data to source code identifiers using an eye tracking device. Our results show that the presence of linguistic antipatterns in source code significantly increases the developers' cognitive load.

CCS CONCEPTS

• **Social and professional topics → Software maintenance**; • **Human-centered computing → Empirical studies in HCI**;

KEYWORDS

Source code lexicon, biometrics, fNIRS, cognitive load, eyetracking, program comprehension.

ICPC '18, May 27–28, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5714-2/18/05…$15.00
https://doi.org/10.1145/3196321.3196347

ACM Reference Format:
Sarah Fakhoury, Yuzhan Ma, Venera Arnaoudova, and Olusola Adesope. 2018. The Effect of Poor Source Code Lexicon and Readability on Developers' Cognitive Load. In *ICPC '18: 26th IEEE/ACM International Conference on Program Comprehension, May 27–28, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3196321.3196347

1 INTRODUCTION

Program comprehension is a fundamental activity within the software development life cycle. An important contributor to software comprehension has to do with the quality of the lexicon, i.e., the identifiers (names of programming entities such as classes or variables) and comments that are used by developers to embed domain concepts and to communicate with their teammates. Previous studies show that source code contains 42% of the domain terms [19] meaning that the lexicon is a way to express understanding of the problem domain and solution, and comment upon the ideas that underlie developers' work. Previous research also shows that there is a correlation between the quality of identifiers and the quality of a software project [1, 7–9, 24, 29].

However, despite considerable advancement in software engineering research in recent years, very little is known about how the human brain processes program comprehension tasks. Cutting edge research conducted by Siegmund et al. involves the use of functional magnetic resonance imaging (fMRI) to study program comprehension in the brain [35] and to understand the cognitive processes related to bottom-up and top-down comprehension strategies [36]. Similarly, Floyd et al. use fMRI to compare areas of brain activation between source code and natural language tasks [15]. Despite the success of fMRI studies in the domain, fMRI machines remain a costly and invasive approach, with which it is hard to reproduce the real life working conditions of software developers.

Functional Near Infrared Spectroscopy (fNIRS) is a brain imaging technique comparable to fMRI [14] that can provide a minimally invasive way to empirically investigate the effects of source code on human cognition and the hemodynamic response within physical structures of the brain. To our knowledge, only two studies explore the use of fNIRS in the domain. Nakagawa et al. [27] investigate the hemodynamic response during mental code execution tasks of varying difficulty and Ikutani and Uwano investigate the effects of variables and control flow statements on blood oxygenation changes in the prefrontal cortex [22].

However, the effect of lexicon and readability of source code on developers' cognitive load during software comprehension tasks

remains unexplored. The low cost and minimally invasive nature of fNIRS makes it particularly well suited for this task. FNIRS data can be related to specific aspects of source code in real time through the use of modern eye tracking devices. This would allow to pinpoint problematic elements within the source code at a very fine level of granularity.

We present an fNIRS study focused on investigating how the human brain processes source code comprehension tasks, in particular, the hemodynamic response of the prefrontal cortex to instances of poor programming practices pertaining to the lexicon and readability of the source code. At a high level, we aim at answering whether we can use fNIRS and eyetracking technology to associate identifiers in source code to cognitive load experienced by developers. Furthermore, we aim to understand how and if poor linguistic, structural, and readability characteristics of source code affect developers' cognitive load.

Our results show that the presence of linguistic antipatterns in source code significantly increases the participants' cognitive load. Overall, when both linguistic antipatterns and structural inconsistencies are introduced to the source code, we do not observe an increase in cognitive load, but, the number of participants that are unable to complete the tasks increases to 60%.

The contributions of this work are as follows:

(1) We provide a methodology to relate terms that compose source code identifiers to direct and objective measures to assess developers' cognitive load.
(2) We provide empirical evidence on the significant negative impact of poor source code lexicon on developers' cognitive load during program comprehension.
(3) We provide a replication package [13], which includes the source code snippets used for our experiment, to allow reproducibility of our results.

Paper organization. The rest of the paper is organized as follows. Section 2 discusses the background, in particular metrics and technologies used throughout the study. Section 3 defines our research questions and presents the experimental set up and methodology used to answer those research questions. Section 4 presents the results and analysis of our findings. Section 5 discusses the threats to validity of this work. Section 6 discusses related work and Section 7 concludes the study.

2 BACKGROUND

In this section, we provide a brief background on Linguistic Antipatterns (Section 2.1), structural and readability metrics (Section 2.2), Functional Near Infrared Spectroscopy (Sections 2.3), and Eyetracking (Section 2.4).

2.1 Linguistic Antipatterns (LAs)

Linguistic Antipatterns (LAs), are recurring poor practices in the naming, documentation, and choice of identifiers in the implementation of program entities [4]. LAs are perceived negatively by developers as they could impact program understanding [3]. In this section, we briefly summarize a subset of the catalog of Linguistic Antipatterns used in our study.

A.1 *"Get" - more than an accessor*: A getter that performs actions other than returning the corresponding attribute without documenting it.
A.3 *"Set" method returns*: A set method having a return type different than `void` and not documenting the return type/values with an appropriate comment.
B.1 *Not implemented condition*: The method' comments suggest a conditional behavior that is not implemented in the code. When the implementation is default this should be documented.
B.6 *Expecting but not getting a collection*: The method name suggests that a collection should be returned, but a single object or nothing is returned.
B.7 *Get method does not return corresponding attribute*: A get method does not return the attribute suggested by its name.
C.2 *Method signature and comment are opposite*: The documentation of a method is in contradiction with its declaration.
D.1 *Says one but contains many*: An attribute name suggests a single instance, while its type suggests that the attribute stores a collection of objects.
D.2 *Name suggests Boolean but type does not*: The name of an attribute suggests that its value is true or false, but its declaring type is not Boolean.
E.1 *Says many but contains one*: Attribute name suggests multiple objects, but its type suggests a single one.
F.2 *Attribute signature and comment are opposite*: Attribute declaration is in contradiction with its documentation.

2.2 Structural and Readability Metrics

There exists a depth of research about how various structural aspects of source code can affect both the readability of the source code and impede the comprehension of developers. Buse and Weimer [7] conduct a large scale study investigating code readability metrics and find that structural metrics such as the number of branching and control statements, line length, the number of assignments, and the number of spaces negatively affect readability. They also show that metrics such as the number of blank lines, the number of comments, and adherence to proper indentation practices positively impact readability. Metrics such as McCabe's Cyclomatic Complexity [25], nesting depth, the number of arguments, Halstead's complexity measures [20], and overall number of lines of code have also been shown to impact code readability [30].

Table 1 lists method level metrics that have been shown to correlate with readability and comprehensibility [7, 20, 25, 30, 33]. The '+' symbol indicates that a feature is positively correlated with high readability and comprehensibility of the code, and the '-' symbol indicates the opposite. The number of symbols indicate how strongly correlated each feature is. Three is high, two is medium, and one is low. A subset of these metrics, which are **bold** in the table, are used in our study.

2.3 Functional Near Infrared Spectroscopy (fNIRS)

Functional Near Infrared Spectroscopy is an optical brain imaging technique that detects changes in oxygenated and deoxygenated hemoglobin in the brain by using optical fibers to emit near-infrared light and measure blood oxygenation levels. The device we use is

Table 1: Metrics shown to correlate positively (+) or negatively (-) with source code readability.

Feature	Corr.	Feature	Corr.
Cyclomatic Complexity	– – –	Halstead vocabulary	–
Number of Arguments	– – –	Halstead length	–
Number of operands	– – –	Number of casts	–
Class References	– – –	**Number of loops**	–
Local Method References	– – –	**Number of expressions**	–
Lines of Code	– – –	**Number of statements**	–
Halstead effort	– –	**Variable Declarations**	–
Halstead bugs	– –	**Number of Comments**	+ +
Max depth of nesting	– –	**Number of Comment Lines**	+ +
External Method References	– –	**Number of Spaces**	+ +
Halstead volume	–	Number of operators	+
Halstead difficulty	–		

the fNIR100, a stand-alone functional brain imaging system, in the shape of a headband, produced by BIOPAC [5]. Overall, the device is light weight, portable, and easy to set up.

Light sources are arranged on the headband along with light detectors. The light sources send two wavelengths of near-infrared light into the forehead, where it continues through the skin and bone 1 to 3cm deep into the prefrontal cortex. These light sources and detectors form 16 distinct optodes which allow the fNIR100 to collect data from 16 distinct points across the prefrontal cortex. Biological tissues in the prefrontal cortex are relatively transparent to these wavelengths, but the oxygenated and deoxygenated hemoglobin are the main absorbers of this light. After the light scatters in the brain, some reaches the light detector on the surface. By determining the amount of light sensed by the detector, the amount of oxygenated and deoxygenated hemoglobin in the area can be calculated using the modified Beer-Lambert Law [10]. Because these hemodynamic and metabolic changes are associated with neural activity in the brain, fNIRS measurements can be used to detect changes in a person's cognitive state while performing tasks [38]. For example, fNIRS have been successfully used to detect task difficulty in real-time on path planning for Unmanned Air Vehicle tasks [2] and tasks designed to invoke working memory [14].

From the measured oxygenated hemoglobin (HbO) and deoxygenated hemoglobin (HbR) concentration levels we are able to calculate HbT, which is the total hemoglobin HbO + HbR, as well as Oxy, which is the difference between HbO and HbR and reflects the total oxygenation concentration changes. In this study, we use Oxy, which has been shown in a wide variety of studies [14, 17, 21] to be a function of task difficulty, as a measure of cognitive load during the various code reading tasks. Due to the fact that fNIRS devices are highly sensitive to motion artifacts and light, users should remain relatively still and not touch the device during recording. Before any analysis can take place, fNIRS data must be refined and filtered to remove any motion artifacts and noise, as well as to exclude data collected by individual optodes that may not have been fit properly against the forehead. These optodes are usually optodes 1 and 5, which are located on the outer edge of the device, near the user's hairline. These optodes are easily identifiable as they show patterns of either sharp peaks and dips or remain flat. The exclusion of an optnode does not effect the data collected by other optodes. To remove noise, all data is filtered using a linear phase, low pass filter that attenuates high frequency components of the signal. We use the filtering provided by Biopac's fNIRSoft [6]. If a user has any unexpected movement, such as sneezing or coughing, we place a marker in the data and such peaks are excluded during the data analysis process.

2.4 Eyetracking

There are an ample amount of studies within the eye tracking research domain that give insight into visual attention patterns and behavior during reading tasks. For example, fixations, which are defined as a relatively steady state between eye movements, and fixation duration, which is the amount of time spent in one location. Research suggests that processing of visual information only occurs during a fixation and that fixation duration is positively correlated with cognitive effort [31]. Therefore, we will use fixation and fixation duration to determine areas participants spent a substantial amount of time reading.

We use the EyeTribe eyetracker [12] throughout this experiment. The Eyetribe offers a sampling rate of 60 Hz and an accuracy of around 0.5–1 degrees of visual angle which translates to an average error of 0.5 to 1 cm on a screen (19–38 pixels). To mitigate the effects of this error we set the font size of the source code to 18 pt which translates to an average error of one to three characters. The 60 Hz sampling rate of the Eyetribe is not suitable for eyetracking studies that study saccades, however it is appropriate for our purpose of investigating fixations within the source code [28]. We calibrate the eyetracker using 16 gaze points (as opposed to 9 or 12 points) to cover the screen with higher accuracy. To ensure the integrity of the eyetracking data collected, only calibration quality that is rated as 4 out of 5 stars or higher is accepted for use in the experiment. Calibration quality at these levels indicate an error of < 0.7 and 0.5 degrees respectively.

Participants use the Eclipse IDE [11] as their environment during the experimental tasks. We will be using iTrace [34], a plugin for Eclipse that interfaces with the eyetracker to determine what source code elements the participants are looking at. We extend the iTrace plugin to identify source code elements at a lower level of granularity, which is terms that compose identifiers. iTrace has a fixation filter to filter out noisy data that may arise due to errors from the eyetracker. This filter estimates fixations on source code elements using the median and joins fixations that are spatially closer together within a threshold radius of 35 pixels (3 characters).

3 METHODOLOGY

The *goal* of this study is two-fold: First, to determine if fNIRS and eye tracking devices can be used to successfully capture high cognitive load within text or source code, at a word level of granularity. Second, to determine if structural or linguistic inconsistencies within the source code increase developers' cognitive load during software comprehension tasks. The *perspective* is that of researchers interested in collecting and evaluating empirical evidence about the effect of poor lexicon and readability of source code on developers' cognitive load during software comprehension. Specifically, the study aims at answering the following research questions:

- **RQ$_1$**: *Can developers' cognitive load be accurately associated with identifiers' terms using fNIRS and eye tracking devices?* We ask participants to perform a comprehension task and then

explore the similarity between fixations on text highlighted as difficult to comprehend by participants and fixations that are automatically classified as having high cognitive load.
- **RQ2**: *Do inconsistencies in the source code lexicon cause a measurable increase in developers' cognitive load during program comprehension?* We ask participants to perform bug localization tasks on a snippet that does not contain lexical inconsistencies and one that does. We then explore the average cognitive load experienced on the two snippets as well as the percentage of fixations that contain high cognitive load in each snippet.
- **RQ3**: *Do structural inconsistencies related to the readability of the source code cause a measurable increase in developers' cognitive load during program comprehension?* We ask participants to perform bug localization tasks on a snippet that contains structural inconsistencies and one that does not. We then explore the average cognitive load experienced on the two snippets.
- **RQ4**: *Do both structural and lexical inconsistencies combined cause a measurable increase in developers' cognitive load during program comprehension?* We ask participants to perform bug localization tasks on a snippet that contains both structural and lexical inconsistencies and one that does not. We then explore the average cognitive load experienced on the two snippets.

3.1 Source Code Snippets

In an effort to replicate real life development environment as close as possible we aim at identifying four code snippets from open-source projects to use in our experiment. Snippets had to meet the following criteria:

- The snippet should be able to be understood on its own without too much external references.
- The snippets must be around 30-40 lines of code including comments so that all chosen snippets take similar time to comprehend without interference due to length.
- The snippets should be able to be altered in such a way that a reasonably difficult to detect semantic defect can be inserted.
- The snippets should be able to be altered to contain Linguistic Antipatterns.

The snippets were chosen from JFreeChart, JEdit, and Apache Maven projects. Two snippets were chosen Apache Maven—methods `replace` and `indexOfAny` (from StringUtils.java), one from JEdit—method `LoadRuleSets` (from SelectedRules.java), and one from JFreeChart—method `calculatePieDatasetTotal` (from DatasetUtilities.java). After conducting a pilot study to assess the suitability of each snippet we discarded method `LoadRuleSets` from JEdit as it required a good understanding of surrounding source code and domain knowledge. Thus, the experiment is performed with the remaining three code snippets.

3.1.1 Altering snippets. In this section we first describe how original snippets are altered to contain bugs to become control snippets. Then, we describe how control snippets are altered to contain either linguistic antipatterns, structural inconsistencies, or both. All snippets and treatments can be found online in our replication package.

Table 2: Participants' demographic data.

Demographic Variables		# of Participants
Programming Languages	C++	5
	Java	1
	Both	9
Degree Pursuing or Completed	Bachelor	8
	Master	2
	PhD	5

Bugs. Source code snippets are altered to contain a semantic fault. Participants are asked to locate the fault as a way to trigger program comprehension. Semantic defects are inserted in the code snippets as opposed to syntactic defects, which can be found without deep understanding of source code snippets. All bugs inserted are one line defects, inserted at around the same location in the code snippets to control for any unwanted location-based effect (i.e., finding a defect earlier if it located higher up in the code).

Linguistic Antipatterns. Section 2.1 describes a subset of the catalog of LAs defined by Arnaoudova et al. [4]. We alter the snippets to contain the listed LAs. Due to the limited number of code snippets it is impossible to include all seventeen LAs, which is why a subset is selected. We aimed at including a variety of antipatterns that arise in method signatures, documentation, and attribute names.

Structural and Readability metrics. We alter the code snippets by introducing a subset of the metrics described in Section 2.2 that have been shown to correlate with the readability and comprehensibility of code snippets. Snippets are formatted in a way that is against conventional Java formatting standards in order to reduce readability. This implies opening and closing brackets are not on their own lines and are not indented properly. Metrics that are described as having negative correlation to readability, such as number of loops, are increased in the snippet. Metrics that are shown to have positive correlation to readability, such as number of comments, are decreased in the snippet.

3.2 Participants

The participants were recruited from a pool of undergraduate and graduate Computer Science students at the authors' institution. A total of 70 participants indicated their interest. Participants were asked to complete an online eligibility survey to ensure that they have some programming experience, thus we require that they must have taken at least one introductory course in C++ or Java. This is to ensure the participants will be able to navigate the source code for the tasks and provide legitimately informed input. Participants receive a $15 giftcard as compensation for participation.

Due to constraints with the eyetracker device used, participants who require the use of bi-focal or tri-focal glasses, or are diagnosed with persistent exotropia or esotropia, are considered ineligible to participate as the eyetracking data may be significantly impacted. Fifteen participants satisfied the eligibility criteria and participated in the experiments. Table 2 summarizes the programming language in which participants declare themselves as more proficient and their educational background.

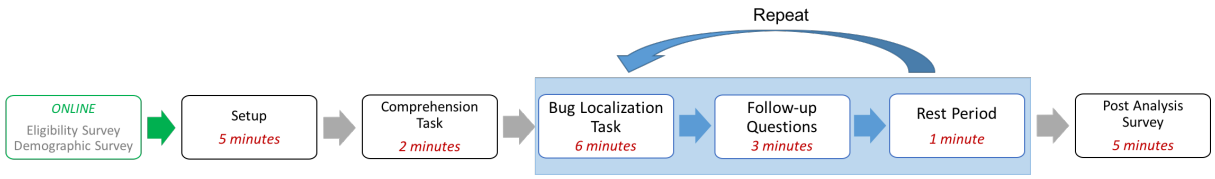

Figure 1: Overview of the experimental procedure.

Table 3: Study design.

Task:	Comprehension	Bug Localization		
		Snippet 1	Snippet 2	Snippet 3
Group 1	Prose	Control	LA	Structural
Group 2	German Code	LA & Structural	Control	LA
Group 3	Prose	LA	Structural	LA & Structural
Group 4	German Code	Structural	LA & Structural	Control

3.3 Study Design

Participants are randomly assigned to one of four different groups, following a balanced design. Each group is shown one comprehension task snippet and three bug localization code snippets. The order of the type of treatment received is randomized to ensure the order of which the tasks are completed does not affect the data. Table 3.3 summarizes the design of the experiment.

Group 1 contains 5 participants, groups 2 and 4 contain 3 participants, and group 3 contains 4 participants. Participants have between 1-15 years of programming experience, with an average of around 4 years of experience, first quartile at 2.25 and the third quartile at 4 years.

3.4 Procedure

Figure 1 illustrates the steps of experimental procedure[1]. Each step has an estimated time for completion, determined through a pilot study. Overall, the experiment is planed to take no longer than one hour. Each step is described in the following sections.

3.4.1 Setup. The researcher explains every step of the experiment to the participants beforehand to ensure that they understand the experimental procedure and what is expected of them. Participants are given a consent form to read and sign if they agree. Next, participants are fit to the fNIRS device and positioned in front of the eye tracking device, computer screen, and keyboard. After this, the participant is asked to relax and a baseline for the fNIRS is conducted. Participants are then asked to calibrate the eye tracker by using their eyes to follow a sequence of dots on the screen in front of them. Anytime a baseline is conducted throughout the experiment, participants are shown a video of fish swimming for one minute. This has been used in similar fNIRS research studies to provide a controlled way to relax participants.

3.4.2 Comprehension Task. To answer **RQ**$_1$, participants are shown either a short code snippet or a comment containing a paragraph of an English prose. The code snippet contains easy to comprehend identifiers in English as well as difficult to comprehend identifiers in a language that the participant is not familiar with (i.e., German). The prose task was taken from an appendix of GRE questions used to test reading comprehension, and we used one reading comprehension question related to the text to assess comprehension. For both prose and code snippets, participants are asked to carefully go through the task, reading the text carefully. Upon completion, participants are asked describe the functionality of the code snippet or answer the comprehension question to ensure that they have properly understood the text and thus engaged throughout the task.

3.4.3 Bug Localization Task. The bug finding task allows us to answer **RQ**$_2$, **RQ**$_3$, and **RQ**$_4$. During this task participants are shown a relatively short code snippet on the computer screen. They are told that the code contains a semantic bug and that they should locate the fault in the code. Participants are asked to continue the task until they find the bug but they are also given the option to end the task if no bug could be found.

We create four versions for each code snippet. Thus, a code snippet shown to a participant will be from one of the following categories:

(1) Code snippet containing a bug and lexical inconsistencies as described in Section 2.1.
(2) Code snippet containing a bug and poor structural/readability characteristics as measured by the metrics described in Section 2.2.
(3) Code snippet containing a bug and both lexical inconsistencies and poor structural/readability characteristics, (i.e., categories (1) and (2)).
(4) Code snippet containing a bug and no lexical inconsistencies or poor structural/readability characteristics, (i.e., the control snippet).

3.4.4 Follow-up questions. In this step participants fill out a questionnaire about the snippet they have read. They are asked to explain if the code snippet provided in the bug localization task had any features that impeded their task of finding the bug, and if yes to describe the feature of interest and highlight it. They are also asked to rate, on a scale of 1 to 5 the effort taken to find the bug (1 being 'little to no effort' and 5 being 'considerable effort'). These follow-up questions are used to add another level of validation to our results.

3.4.5 Rest Period. Participant are asked to relax for a minute so that a new fNIR baseline is recorded to ensure that the measured cognitive load is not impacted by the strain of the previous task.

[1]The experiment was approved through a full board review for human subject research from the Institutional Review Board (IRB) at the author's university (IRB #16113-001).

3.4.6 Post Analysis. The features of interest for each code snippet shown to the participant will be revealed and the participant will be asked questions about comprehension and the impact of the features.

Eye tracking and fNIRS data is only collected during the comprehension and bug finding tasks. Steps outlined in blue are repeated three times, sequentially, per participant before moving onto the post analysis survey.

3.5 Pilot Study

A pilot study is conducted with four participants so that every snippet/treatment combination can be assessed. During the pilot study we make sure that bugs can be found within a reasonable amount of time and that they are not too difficult or too simple. We also determine if the experiment layout can be done within a reasonable amount of time (1 hour) and does not induce unneeded fatigue for the participants. Initially, we included four bug localizations tasks, and decided to reduce this to three. One of the snippets that was initially chosen makes references to external methods; it was discarded after the pilot study.

3.6 Analysis Method

3.6.1 High Cognitive Load. In order to determine fixations that contain high cognitive load, we analyze the Oxy values over the entire source code snippet. We classify fixations containing Oxy values in the highest 20% to be indicative of high cognitive load. Additionally, we calculate fixations that cause a peak, or a sharp increase in cognitive load, as causing high cognitive load. We refer to both of these high cognitive load points as 'points of interest'. A sharp increase is defined as a delta between two immediate fixations that is in the highest 10% of delta values. In order to obtain the most accurate classification of high cognitive load data points, we use participants' highlighted identifiers as a ground truth to determine the percentage thresholds. Therefore, it is important that participants accurately highlight areas of code and identifiers during the follow up question portion of the experiment. We choose the thresholds that balance between classifying the maximum number of highlighted identifiers as high cognitive load, while still not over classifying fixations that are not highlighted. Thresholds are optimized using a subset of 5 out of 15 participants.

3.6.2 Feature Scaling. Due to natural biological differentiation between participants and inherent HbO and HbR concentration differences in the prefrontal cortex, raw Oxy values cannot be reliably compared across subjects. Within subject comparisons can also be problematic. For example, if the baseline values for the fNIRS are sampled while the participant is not properly relaxed for one snippet, and then again while the participant is relaxed for another snippet, raw Oxy data will be skewed. To mitigate this, we normalize all raw Oxy data using feature scaling before comparing within participant. Feature scaling is a method used to standardize a range of independent variables within the dataset. To normalize Oxy values to a range between 0 and 1 inclusive, we use the following formula: $normalizedOxy = \frac{Oxy_{raw} - Oxy_{min}}{Oxy_{max} - Oxy_{min}}$ where Oxy_{raw} is the raw Oxy value, Oxy_{min} is the minimum Oxy value recorded over the snippet and Oxy_{max} is the maximum Oxy value recorded over the snippet. Similar normalization on fNIRS data was performed by Ikutani and Uwano [22].

3.6.3 fNIRS and Eyetracking Data. To map fNIRS data to fixation points we use the output from our modified version of iTrace using system time as our reference point. Fixation data may not always be consistent with the areas of code that participants highlighted during the post analysis questions. This is due to participants error during the follow-up questions phase. In such cases, participants are asked to verify fixation data at the end of their experiment session. We use our visualization tool to identify areas of high cognitive load and peaks during the post analysis step of the procedure. These are then shown to the participants and they are asked about specific areas of code where we identify fixations with high cognitive load and are not highlighted by the participants. If the participants agree with the data, they are given the choice to highlight additional sections.

3.6.4 Simple Matching Coefficient (SMC). To answer **RQ1**, we use the Simple Matching Coefficient [37]—a statistic used to compare similarity between two or more datasets. SMC is similar to the Jaccard index but counts mutual presence (when an attribute is present in both sets) and mutual absence (when an attribute is absent in both sets). The Jaccard index only counts mutual presence. We use SMC to calculate the similarity between the fixations on identifiers that are highlighted by participants and the set of fixations that are flagged as having high cognitive load. This way we count mutual absence (no high cognitive load, and not highlighted code) as part of the similarity to assess the algorithm used to determine high cognitive load.

3.6.5 Wilcoxon Signed-Rank Test. To answer **RQ2**, **RQ3**, and **RQ4** we need to determine if there is a significant increase between the average normalized Oxy on treatment snippets compared to the average normalized Oxy on control snippets. To this end, we use the paired Wilcoxon signed-rank test, a non-parametric statistical test used to compare two related samples, to assess whether the population mean ranks differ. Our null hypothesis is that there is no difference between the normalized average Oxy values for the control snippets and treatment snippets. Our alternative hypothesis is that the normalized average Oxy values for the control snippets are lower than the normalized average Oxy values for the treatment snippets.

3.6.6 Cliff's Delta (d) Effect Size. After performing a Wilcoxon signed-rank test, we measure the strength of the difference between the average normalized Oxy on treatment snippets and the average normalized Oxy on control snippets. Cliff's delta (d) effect size [18] is a non-parametric statistic estimating whether the probability that a randomly chosen value from one group is higher than a randomly chosen value from another group, minus the reverse probability. Possible values for effect size range from -1 to 1, with 1 indicating there is no overlap between the two groups and all values from group 1 are greater than the values from group 2, -1 indicating there is no overlap between the two groups but all values from group 1 are lower than the values from group 2, and 0 indicating there is a complete overlap between the two groups and thus there is no effect size. The guideline for interpreting effect size between 0

Table 4: Similarity between fixations with high cognitive load and highlighted fixations.

Treatment	SMC	Treatment	SMC
German Code	0.87	Prose	0.81
	0.76		0.81
	0.82		0.70
	0.79		0.73
	0.81		0.65
	0.82		0.75
Average	0.81	Average	0.74
Total Average			**0.78**

and 1 is as follows: $0 \leq |d| < 0.147$: negligible, $0.147 \leq |d| < 0.33$: small, $0.33 \leq |d| < 0.474$: medium, $0.474 \leq |d| \leq 1$: large.

4 RESULTS

RQ1: *Can developers' cognitive load be accurately associated with identifiers' terms using fNIRS and eye tracking devices?*

Table 4 contains the SMC values calculated between fixation data containing identifiers highlighted by participants and fixations that have high cognitive load values. Each SMC value is calculated per participant. The average SMC for the two comprehension snippets, German code and English prose, is 0.81 and 0.74 respectively, with a total average of 0.78. This means that 78% of the fixations are correctly identified as having high cognitive load and are highlighted by the participant, or do not have high cognitive load and are not highlighted by the participant. Interestingly, for code snippets that contain German code, we observe a higher average similarity.

Achieving 100% similarity is probably too optimistic. For code snippets that contain German code, for example, participants cannot be expected to reliably highlight all parts of the code that may have caused confusion or that caused them difficulties. For instance, some parts of source code may cause an initial increase in cognitive load, such as a computational statement, and is picked up by the fNIRS. However, this statement might not be registered as something the participant deems as confusing or difficult to understand and is therefore not highlighted. When exploring the nature of the discrepancy over the remaining 19% of the data points—i.e., analyzing the fixations that are not highlighted by participants—we find that three participants exhibit high cognitive load for fixations over "if statements" containing computations, two participants exhibit cognitive load over statements that contain return statements, one participant exhibits high cognitive load on a German comment, and one participant exhibits high cognitive load initially, at the very beginning of the code snippet. For example, one participant exhibits high cognitive load over the line of code: `if(pos < 0)`, when asked if this statement indeed caused them any confusion the participant explains that it is not a confusing statement, but that it requires effort to understand and recall the variable `pos`.

When analyzing the English prose treatment regarding the fixations recorded as containing high cognitive load and not highlighted by participants, we see that three participants exhibit high cognitive load over the comprehension questions and three participants exhibit high cognitive load on words that are in sentences that contain other highlighted words.

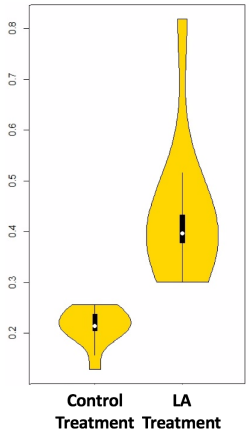

Figure 2: Percentage of points of interest for fixations in control treatment and fixations containing LAs in LA treatment.

RQ$_1$ Summary: using fNIRS and eyetracking devices, developers' cognitive load can be accurately associated with identifiers in source code and text, with a similarity of 78% compared to self-reported high cognitive load.

RQ2: *Do inconsistencies in the source code lexicon cause a measurable increase in developers' cognitive load during program comprehension?*

Figure 2 shows the distribution of the percentage of high cognitive load data points identified automatically, i.e., the points of interest. The percentage of points of interest are calculated over fixations that do not contain linguistic antipatterns in snippets with the control treatment (i.e., all fixations) and over fixations that do contain linguistic antipatterns in snippets with the LA treatment. There are a total of 10 participants who completed both a control snippet and a snippet with the LA treatment. Performing a paired Wilcoxon signed-rank test we obtain a significant p-value (0.0009 p-value), with a large effect size (d = -1), which indicates that fixations over identifiers that contain linguistic antipatterns contain a significantly higher percentage of points of interest, as compared to fixations in the control snippets.

Figure 3 contains the distribution of normalized Oxy averages per participant, over both snippets that receive the LA treatment and control snippets. This data is taken from participants belonging to groups 1 and 2 as described in Section 3.3. There are a total of eight participants that completed both a task with a control treatment and a task with an LA treatment. Five participants were able to complete the bug localization in both code snippets successfully and three participants were only able to complete the bug localization in control snippets. One of the three participants failed the task in both code snippets and was excluded from the analysis, therefore the analysis is carried out on data from seven participants total. Performing a paired Wilcoxon signed-rank test we obtain a significant p-value (0.003), with a large effect size (d = -0.81), which indicates that the presence of linguistic anitpatterns in the source code significantly increases the average Oxy a participant experiences.

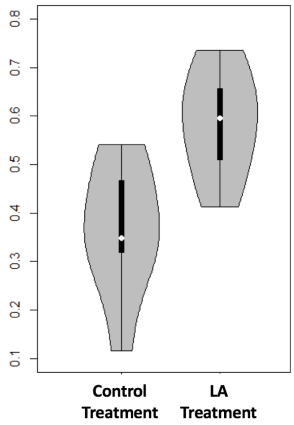

Figure 3: Normalized average Oxy on control vs. LA treatments.

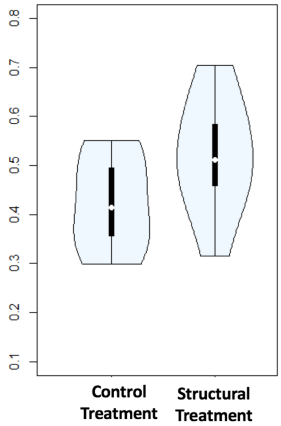

Figure 4: Normalized average Oxy on control vs. structural treatments.

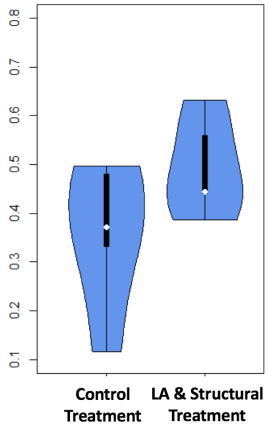

Figure 5: Normalized average Oxy on control vs. LA & structural treatments.

From the post analysis survey, we observe that participants made comments on source code containing linguistic antipatterns in 9 out of 12 tasks. Two participants noted linguistic antipattern **B.1** (*Not implemented condition*), where a condition in a method comment is not implemented. One of these two participants showed high cognitive load when reading the line of comment that was not implemented and both participants explicitly stated that they spent time searching the code for the unimplemented condition but could not find it. Also, five participants noted linguistic antipattern **C.2** (*Method signature and comment are opposite*). One such example in the code snippets is that method signature is `getPieDatasetTotals` while the comment states `sets the calculated total of all the values in a PieDataset`. One participant highlighted that such linguistic antipattern is confusing. This participant as well as two other participants who stated that they were able to filter out the inconsistency between method names and comments showed high cognitive load on fixations over the source code containing the LA.

RQ_2 Summary: The existence of linguistic antipatterns in the source code significantly increases the cognitive load experienced by participants.

RQ3: *Do structural inconsistencies related to the readability of the source code cause a measurable increase in developers' cognitive load during program comprehension?*

Figure 4 contains the distribution of normalized Oxy averages per participant, over both snippets that receive the structural treatment and control snippets. There are a total of eight participants that completed both a task with a control treatment and a task with a structural treatment. This data is taken from participants belonging to groups 1 and 4. Four participants were able to complete bug localization in both code snippets successfully, four participants were only able to complete bug localization in the control snippets. Results from a paired Wilcoxon signed-rank test are not statistically significant ($p - value$=0.14), with a medium effect size (d=-0.47), which indicates that there is no evidence that structural inconsistencies alone increase the average cognitive load that participants experience during program comprehension in the context of a bug localization task.

From the post analysis survey, we observe that participants made comments on source code containing poor structure in 10 out of 12 tasks. 9 participants found poor structure, including incorrect indentation and breaking one line of code into multiple of lines, created frustration and slowed down their performance in bug localization tasks. One participant commented that *"terrible formatting severely increases readers burden"*. Only one participant commented that the structure was not confusing since she/he was able to click on opening brackets to find the associated closing brackets.

RQ_3 Summary: Although participants found structural inconsistencies to be frustrating, there is no statistical evidence that structural inconsistencies increase the average cognitive load that participants experience.

RQ4: *Do both structural and lexical inconsistencies combined cause a measurable increase in developers' cognitive load during program comprehension?*

Figure 5 contains the distribution of normalized Oxy averages per participant, over snippets with a control treatment and LA & Structural treatment. There are a total of six participants that completed tasks with both a control treatment and a treatment with linguistic anitpatterns and structural inconsistencies. This data is taken from participants belonging to groups 2 and 4. All six participants successfully completed the task with the control treatment but only two participants successfully completed the task in the treatment snippet. Performing a paired Wilcoxon signed-rank test did not show statistically significant results ($p - value$=0.48), with a small effect size (d=-0.28), meaning that there is no evidence that structural inconsistencies affecting the readability and comprehensibility of the source code combined with linguistic antipatterns significantly increase the cognitive load that participants experience.

Interestingly, in four out of six participants, the average Oxy over control snippets is higher than over snippets containing the LA & structural treatment. Using the post analysis survey, as well as the snippets questionnaire we observe that all four participants were mislead by the structural and linguistic elements when they are part of the same treatment. All four participants failed at locating the bug

Table 5: Bug localization task results: success rate and time.

Treatment	# Bugs	# Bugs Found (%)	Avg. Time (min.)
Control	11	10 (90.9%)	3.7
LA	12	9 (75.0%)	5.0
Structural	12	7 (58.0%)	4.7
LA & Structural	10	4 (40.0%)	6.3

in the code, which indicates that the treatment did in fact negatively affect their comprehension of the code. For the two participants that did correctly locate the bug, their average cognitive load is considerably higher compared to the control snippets (i.e, 0.19 and 0.51 difference between treatments per participant).

From the post analysis survey, we observe that participants made comments on linguistic antipatterns in 7 out of 10 of the tasks and structural inconsistencies in 9 out of 10 of tasks with both the LA and structural treatments. Two participants noted linguistic antipattern **E.1** (*Says many but contains one*), where variable name `replacementValues` actually contains a single value. Both participants commented that they were able to understand from the from context and the naming did not hinder bug finding. They showed low cognitive load for the identifiers containing the LA. Also, two participants noted LA **F.2** (*Attribute signature and comment are opposite*), where the comment states `min double value` while the attribute is assigned with value `Integer.MAX_VALUE`. Both participants found this linguistic antipattern misleading, prolonged their task, and showed high cognitive load. All nine participants who identified structural inconsistencies in source code highlighted that such inconsistencies caused distractions and prolonged the bug localization task. One participant commented that although the indentations was frustrating, it did not hinder bug localization.

Overall, 30 out of 45 bug localization tasks were completed successfully. The distribution of successfully completed tasks amongst four treatment groups is shown in Table 5. Over 90% of the bugs were found in the control group with average time of 3.7 minutes. Performance decreases as linguistic antipatterns and poor structure characteristics are added (75% and 58%, respectively). At the same time, the average time spent on bug localization increases as linguistic antipatterns and poor structure characteristics are added (5 minutes and 4.71 minutes, respectively). When both linguistic antipatterns and poor structure are present in the code snippets, only 40% of the bugs were localized successfully with average time of 6.25 minutes. The outcome shows that the presence of structural and lexical inconsistencies slows down and even hinders bug localization.

RQ$_4$ Summary: When analyzing the within group participant data, source code containing both lexical and structural inconsistencies mislead more than 60% of the participants. The remaining participants who successfully completed the bug localization tasks experienced higher cognitive load on code containing both inconsistencies compared to the control snippets.

5 THREATS TO VALIDITY

This section discusses the threats to validity that can affect our study. A common classification [39, 40] involves five categories, namely threats to conclusion, internal, construct, external, and reliability threats.

Threats to *conclusion validity* relate to issues that could affect the ability to draw correct conclusions about relations between the treatment and the outcome of an experiment. There is always heterogeneity in a study group. If the group is very heterogeneous, there is a risk that the variation due to individual differences is larger than the one due to the treatment. Our experiment is conducted with only undergraduate and graduate students instead of a general population of developers, which reduces the heterogeneity. Another threat to conclusion validity may come from the statistical tests used to draw conclusions. Since the collected data cannot be assumed to be normal, we use non-parametric statistical tests, specifically the Wilcoxon signed-rank test and the Cliff's delta effect size.

Threats to *internal validity* concern the relation between the independent and dependent variables and factors that could have influenced the relation with respect to the causality. One potential confounding factor is the programming experience of participants. The code snippets used in our study are written in Java but 12 out of 15 participants consider that they are more proficient in C++ and 5 participants have no previous experience in Java. This might cause an increased cognitive load. However, that would impact the results for all treatments equally and thus does not invalidate our comparison of different treatment groups. Another threat here might be that as participants perform bug localization tasks, they can become tired or less motivated as time passes. To mitigate this threat, we asked feedback from students in the pilot study regarding the length and difficulty of the snippets to ensure that the experiment is designed with an appropriate length, which is around 1 hour. To minimize the effect of the order, in which participants use the treatments, the order is assigned randomly to each participant. Another threat could come from the calibration of thresholds to define high cognitive load. Indeed, different calibrations could have produced different results, and also indirectly affected the assessment of the proposed approach. The threshold is experimentally determined, however, this does not guarantee that the choice is optimal for every single human subject.

Threats to *construct validity* concern the relation between theory and observation. In this study, construct validity threats are mainly due to measurement errors. As for bug localization tasks, all code snippets within the same treatment groups are designed to be with the same difficulty level, which can be affected by subjectiveness of the researchers. If we conduct the experiment with a different set of code snippets, the results might not be the same. To mitigate this threat, performed a pilot study to ensure that the code snippets are at a similar level of difficulty.

Threats to *external validity* concern the generalizability of the findings outside the experimental settings. A potential threat to external validity in this study might come from the use of students as participants in the experiment rather than professional developers, which can raise doubts about how transferable the results are to the software industry. However, research has shown that given a carefully scoped experiment on a development approach that is new to both students and professionals, similar performances are observed [32]. We believe that students are expected to show similar performance as professionals when asked to perform bug localization on an open-source application that they are not familiar

with. Another potential threat is the selection of the code snippets, which may not be representative of the studied population. To mitigate this threat, we extracted code snippets from 2 different open-source applications from GitHub. We selected code snippets between 30 and 40 lines of code to ensure that participants will finish the bug localization tasks within an hour. However, results might be different on snippets with different length and complexity.

Threats to *reliability validity* concern the ability to replicate a study with the same data and to obtain the same results. We provide details on the selected code snippets and their altered versions in our replication package [13]. Moreover, we are currently working on publishing the extension of iTrace and our visualization tool online.

6 RELATED WORK

A broad range of studies have explored the use of psycho-physiological measures to investigate cognitive processes and states in software engineering. Eye tracking metrics such as pupil size, saccades, and fixation duration have been used in combination with other biometric measures to investigate cognitive processes during software engineering tasks. For example, Fritz et al. [16] combined EEG, eye tracking, and electro dermal activity (EDA) to investigate task difficulty during code comprehension. Participants performed mental execution of code and the authors were able to successfully predict the perceived task difficulty.

Similarly, Müller et al. conducted a study using heart rate variability and EDA [26]. They associated biometric data to specific areas of code changed by developers through the use of interaction logs in Eclipse and were able to use the data to predict code quality concerns within areas of changed source code. Lee et al. [23] used a combination of EEG and eye tracking metrics to predict task difficulty and programmer expertise. They found that both metrics could accurately predict expertise and task difficulty.

Although various psycho-physiological measures have proven to be successful measures of cognitive processes within the domain, brain imaging techniques as measures of cognitive states remain a relatively new trajectory of research in empirical software engineering. The first fNIRS study within the domain of software engineering was conducted by Nakagawa et al. [27] in which they investigated oxygenation changes in the prefrontal cortex as a response to mental code execution tasks of varying difficulty. They discovered a correlation between increased blood flow in the prefrontal cortex and difficulty of the task. The experiment was conducted with 10 subjects and involved showing them code snippets on a sheet paper.

To the best of our knowledge the only other fNIRS study conducted within the domain was by Ikutani and Uwano [22], who used fNIRS to investigate the effects of variables and control flow statements on blood oxygenation changes in the prefrontal cortex. They were able to conclude that oxygenation changes in the prefrontal cortex reflect working-memory intensive tasks. Their experiment involved 11 participants reading code on a screen that consisted of arithmetic and control flow statements.

The first fMRI study within the domain was conducted by Siegmund et al. [35] where participants were asked to read short source code snippets and find syntax errors in an effort to measure program comprehension in the brain. They discovered a network of brain areas activated that are related to natural language comprehension, problem solving, and working memory. Another fMRI study conducted by Siegmund et al. [36] was conducted with the aim of isolating specific cognitive processes related to bottom up and top down comprehension strategies. 11 participants were asked to find syntax and semantic bugs in code that was altered to either remove semantic cues or obfuscate code through formatting and indentation changes. They found evidence of semantic chunking during bottom-up comprehension and lower activation of brain areas during comprehension based on semantic cues. Floyd et al. [15] also conducted an fMRI study, inspired by the work of Siegmund et al., which aimed to compare areas of brain activation between source code and natural language tasks. They use activation patterns to successfully predict which tasks were being completed.

Although fMRI provides increased spatial resolution over fNIRS imaging techniques, participants in fMRI studies are asked to read code from a mirror placed within the fMRI machine. This significantly impacts the type and length of the code snippets that can be used. Moreover, it is difficult to simulate real life working conditions that developers are used to with studies using fMRI. The portability and minimally invasive nature of the fNIRS device allows a more realistic simulation of a real working environment. Moreover, to the best of our knowledge, no previous studies map and analyze biometric data at such fine level of granularity that is terms that compose identifiers. Instead, conclusions are made about the entire source code snippets. Finally, our work is the first to empirically investigate the effect of source code lexicon and readability on developers' cognitive load.

7 CONCLUSION

In this paper we present an fNIRS study focused on investigating how the human brain processes source code comprehension tasks, in particular, whether we can use fNIRS and eyetracking technology to associate identifiers in source code to cognitive load experienced by developers. Furthermore, we investigate how poor linguistic, structural, and readability characteristics of source code affect developers' cognitive load. Results show that fNIRS and eyetracking technology are suitable for measuring and associating cognitive load to source code at the identifier level of granularity. In addition, we conclude that the presence of linguistic antipatterns in source code significantly increases the cognitive load of developers during program comprehension tasks. We do not find any evidence to conclude the same for structural and readability characteristics. However, when a snippet contains both the structural and linguistic antipattern treatments, program comprehension is significantly impacted and 60% of participants are unable to complete the task successfully; we do not observe an increase in cognitive load over the treatment snippet for those participants. However, for the remaining 40% of participants who do complete the tasks successfully, we observe an increase in cognitive load.

Future work includes replicating the experiment with open-source/industrial developers as well as evaluating other poor practices and source code characteristics.

REFERENCES

[1] Surafel Lemma Abebe, Venera Arnaoudova, Paolo Tonella, Giuliano Antoniol, and Yann Gaël Guéhéneuc. 2012. Can Lexicon Bad Smells improve fault prediction?. In *Proceedings of the Working Conference on Reverse Engineering (WCRE)*. 235–244.

[2] Daniel Afergan, Evan M Peck, Erin T Solovey, Andrew Jenkins, Samuel W Hincks, Eli T Brown, Remco Chang, and Robert JK Jacob. 2014. Dynamic difficulty using brain metrics of workload. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 3797–3806.

[3] Venera Arnaoudova, Massimiliano Di Penta, and Giuliano Antoniol. 2016. Linguistic Antipatterns: What They are and How Developers Perceive Them. *Empirical Software Engineering (EMSE)* 21, 1 (2016), 104–158.

[4] Venera Arnaoudova, Massimiliano Di Penta, Giuliano Antoniol, and Yann-Gaël Guéhéneuc. 2013. A New Family of Software Anti-Patterns: Linguistic Anti-Patterns. In *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*. 187–196.

[5] BIOPAC. 2018. BIOPAC Homepage. (March 2018). https://www.biopac.com

[6] BIOPAC. 2018. fNIRSoft User Manual. (March 2018). https://www.biopac.com/wp-content/uploads/fnirsoft-user-manual.pdf

[7] Raymond P.L. Buse and Westley R. Weimer. 2010. Learning a metric for code readability. *IEEE Transactions on Software Engineering (TSE)* 36, 4 (2010), 546–558.

[8] Simon Butler, Michel Wermelinger, Yijun Yu, and Helen Sharp. 2009. Relating Identifier Naming Flaws and Code Quality: An Empirical Study. In *Proceedings of the Working Conference on Reverse Engineering (WCRE)*. 31–35.

[9] Simon Butler, Michel Wermelinger, Yijun Yu, and Helen Sharp. 2010. Exploring the Influence of Identifier Names on Code Quality: An empirical study. In *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*. 156–165.

[10] David T Delpy, Mark Cope, Pieter van der Zee, SR Arridge, Susan Wray, and JS Wyatt. 1988. Estimation of optical pathlength through tissue from direct time of flight measurement. *Physics in Medicine & Biology* 33, 12 (1988), 1433.

[11] Eclipse. 2018. Eclipse IDE. (March 2018). https://www.eclipse.org/ide

[12] EyeTribe. 2018. The Eye Tribe Homepage. (March 2018). https://theeyetribe.com

[13] Sarah Fakhoury. 2018. Online Replication Package. (March 2018). https://github.com/smfakhoury/fNIRS-and-Cognitive-Load

[14] Frank A. Fishburn, Megan E. Norr, Andrei V. Medvedev, and Chandan J. Vaidya. 2014. Sensitivity of fNIRS to cognitive state and load. *Frontiers in human neuroscience* 8 (2014), 76.

[15] 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 *Proceedings of the International Conference on Software Engineering (ICSE)*. 175–186.

[16] Thomas Fritz, Andrew Begel, Sebastian C Müller, Serap Yigit-Elliott, and Manuela Züger. 2014. Using Psycho-physiological Measures to Assess Task Difficulty in Software Development. In *Proceedings of the International Conference on Software Engineering (ICSE)*. 402–413.

[17] Audrey Girouard, Erin Treacy Solovey, Leanne M. Hirshfield, Krysta Chauncey, Angelo Sassaroli, Sergio Fantini, and Robert J.K. Jacob. 2009. Distinguishing difficulty levels with non-invasive brain activity measurements. In *IFIP Conference on Human-Computer Interaction*. Springer, 440–452.

[18] Robert J. Grissom and John J. Kim. 2005. *Effect sizes for research: A broad practical approach* (2nd edition ed.). Lawrence Earlbaum Associates.

[19] Sonia Haiduc and Andrian Marcus. 2008. On the Use of Domain Terms in Source Code. In *Proceedings of the International Conference on Program Comprehension (ICPC)*. 113–122.

[20] Maurice H Halstead. 1977. Elements of software science. (1977).

[21] Christian Herff, Dominic Heger, Ole Fortmann, Johannes Hennrich, Felix Putze, and Tanja Schultz. 2014. Mental workload during n-back task-quantified in the prefrontal cortex using fNIRS. *Frontiers in Human Neuroscience* 7 (2014), 935.

[22] Yoshiharu Ikutani and Hidetake Uwano. 2014. Brain activity measurement during program comprehension with NIRS. In *Proceedings of the International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*. 1–6.

[23] Seolhwa Lee, Danial Hooshyar, Hyesung Ji, Kichun Nam, and Heuiseok Lim. 2017. Mining biometric data to predict programmer expertise and task difficulty. *Cluster Computing* (2017), 1–11.

[24] Andrian Marcus, Denys Poshyvanyk, and Rudolf Ferenc. 2008. Using the Conceptual Cohesion of Classes for Fault Prediction in Object-Oriented Systems. *IEEE Transactions on Software Engineering (TSE)* 34, 2 (2008), 287–30.

[25] Thomas J. McCabe. 1976. A Complexity Measure. *IEEE Transactions on Software Engineering (TSE)* SE-2, 4 (1976), 308–320.

[26] Sebastian C Müller and Thomas Fritz. 2016. Using (bio)metrics to predict code quality online. In *Proceedings of the International Conference on Software Engineering (ICSE)*. 452–463.

[27] Takao Nakagawa, Yasutaka Kamei, Hidetake Uwano, Akito Monden, Kenichi Matsumoto, and Daniel M. German. 2014. Quantifying programmers' mental workload during program comprehension based on cerebral blood flow measurement: a controlled experiment. In *Proceedings of the International Conference on Software Engineering (ICSE)*. 448–451.

[28] Kristien Ooms, Lien Dupont, Lieselot Lapon, and Stanislav Popelka. 2015. Accuracy and precision of fixation locations recorded with the low-cost Eye Tribe tracker in different experimental setups. *Journal of eye movement research* 8, 1 (2015).

[29] Denys Poshyvanyk, Yann-Gaël Guéhéneuc, Andrian Marcus, Giuliano Antoniol, and Vaclav Rajlich. 2006. Combining Probabilistic Ranking and Latent Semantic Indexing for Feature Identification. In *Proceedings of the International Conference on Program Comprehension (ICPC)*. 137–148.

[30] Daryl Posnett, Abram Hindle, and Premkumar Devanbu. 2011. A Simpler Model of Software Readability. In *Proceedings of the Working Conference on Mining Software Repositories (MSR)*. 73–82.

[31] Keith Rayner. 1998. Eye movements in reading and information processing: 20 years of research. *Psychological Bulletin* 124, 3 (1998), 372–422.

[32] Iflaah Salman, Ayse Tosun Misirli, and Natalia Juristo. 2015. Are students representatives of professionals in software engineering experiments?. In *Proceedings of the International Conference on Software Engineering (ICSE)*. 666–676.

[33] Simone Scalabrino, Mario Linares-Vasquez, Denys Poshyvanyk, and Rocco Oliveto. 2016. Improving code readability models with textual features. In *Proceedings of the International Conference on Program Comprehension (ICPC)*. 1–10.

[34] Timothy R Shaffer, Jenna L Wise, Braden M Walters, Sebastian C Müller, Michael Falcone, and Bonita Sharif. 2015. iTrace: Enabling eye tracking on software artifacts within the IDE to support software engineering tasks. In *Proceedings of the Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. 954–957.

[35] 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 *Proceedings of the International Conference on Software Engineering (ICSE)*. 378–389.

[36] 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 *Proceedings of the Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. 140–150.

[37] Robert R Sokal. 1958. A statistical method for evaluating systematic relationship. *University of Kansas science bulletin* 28 (1958), 1409–1438.

[38] Erin Treacy Solovey, Daniel Afergan, Evan M. Peck, Samuel W. Hincks, and Robert J. K. Jacob. 2015. Designing Implicit Interfaces for Physiological Computing. *ACM Transactions on Computer-Human Interaction* 21, 6 (2015), 1–27.

[39] Claes Wohlin, Per Runeson, Höst Martin, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. 2000. *Experimentation in Software Engineering - An Introduction*. Kluwer Academic Publishers.

[40] Robert K. Yin. 1994. *Case Study Research: Design and Methods* (2nd ed.). Sage Publications.