**Empirical Software Engineering manuscript No.**
(will be inserted by the editor)

# Measuring the Impact of Lexical and Structural Inconsistencies on Developers' Cognitive Load during Bug Localization[*]

**Sarah Fakhoury · Devjeet Roy · Yuzhan Ma · Venera Arnaoudova · Olusola Adesope**

Received: November 15, 2018 / Accepted: date

**Abstract** 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. Measuring program comprehension is not a trivial task. In fact, different studies use self-reported and various psycho-physiological measures as proxies.

In this research, we propose a methodology using functional Near Infrared Spectroscopy (fNIRS) and eye tracking devices as an objective measure of program comprehension that allows to conduct studies in environments close to real world settings at identifier level of granularity. We validate our methodology and apply it to study the impact of lexical, structural, and readability issues on developers' cognitive load during bug localization tasks. Our study involves 25 undergraduate and graduate students and 21 metrics. Results show that the existence of lexical inconsistencies in the source code significantly increases the cognitive load experienced by participants not only on identifiers involved in the inconsis-

[*] This work is an extension of our previous paper *Fakhoury et al.* [2018].

Sarah Fakhoury
SEL Lab, School of EECS, Washington State University
355 NE Spokane St., Sloan 326
Pullman, WA, 99164-2752 U.S.A.
Tel.: +1 (509) 335-8665
Fax: +1 (509) 335-3818
E-mail: sarah.fakhoury@wsu.edu

Devjeet Roy
SEL Lab, School of EECS, Washington State University
E-mail: devjeet.roy@wsu.edu

Yuzhan Ma
Amazon
E-mail: yuzhanm@amazon.com

Venera Arnaoudova
SEL Lab, School of EECS, Washington State University
E-mail: venera.arnaoudova@wsu.edu

Olusola Adesope
College of Education, Washington State University
E-mail: olusola.adesope@wsu.edu

tencies but also throughout the entire code snippet. We did not find statistical evidence that structural inconsistencies increase the average cognitive load that participants experience, however, both types of inconsistencies result in lower performance in terms of time and success rate. Finally, we observe that self-reported task difficulty, cognitive load, and fixation duration do not correlate and appear to be measuring different aspects of task difficulty.

**Keywords** Program Comprehension · Cognitive Load · fNIRS · Biometrics · Linguistic Antipatterns · Readability

## 1 Introduction

One of the most cognitively intensive, yet fundamental activities in the software development life cycle is the act of program comprehension. Before developers can make any changes to a piece of source code, they spend a considerable amount of time reading through existing source code, using a variety of different comprehension strategies. Therefore, program comprehension plays a key role in the pursuit of improving the overall costs and processes involved for the creation of any software system. Which is why over the past few decades program comprehension has been studied extensively by researchers trying to understand more about how developers comprehend source code, the different source code aspects that effect comprehension, and ways to improve this process.

The resulting research has shown that an important contributor to software comprehension has to do with the quality of the lexicon, i.e., the identifiers and comments that are used by developers to embed domain concepts and to communicate with their teammates. Although recent work by *Scanniello and Risi* [2013] suggests that identifier length has no significant effect on identifying and fixing faults in the source code, many other studies show evidence of a correlation between the quality of identifiers (measured using various metrics) and the quality of a software project [*Abebe et al.*, 2012; *Buse and Weimer*, 2010; *Marcus et al.*, 2008; *Poshyvanyk et al.*, 2006]. Additionally, the readability and the structural complexity of the code can have a significant impact on program comprehension.

Before improving program comprehension, we first need reliable methods to measure it. However, this is a non trivial task because program comprehension involves a multitude of complex cognitive processes. The most commonly used metrics of program comprehension are conventional research methods based on self-reported verification. These conventional methods are mostly indirect measures of comprehension, where subjects report on their own comprehension levels or summarize part of an artifact so that researchers can instead deduce the level of comprehension. Some of these methods include think aloud protocols, surveys, and comprehension summaries. For example, *Binkley et al.* [2009a] studied the impact of identifier style on code readability. *Lawrie et al.* [2006] use source code summaries and self reported confidence levels to assess comprehension levels of participants reading source code snippets containing single letter, abbreviated, and full length identifiers. However, there are several potential issues that can result from using these indirect measures because they are prone to participant biases [*Hochstein et al.*, 2005]. For example, participants may report they understood a piece of source code, but their perceived understanding does not necessarily mean they correctly understood the source code.

This is why in recent years researchers have begun exploring how to use physiological data to supplement our perspective on comprehension with direct, empirical measures that can provide a more objective understanding of the cognitive process behind program comprehension. For example, *Lee et al.* [2017] use a combination of EEG and eye tracking metrics to predict task difficulty and programmer expertise. *Fritz et al.* [2014] combined EEG, eye tracking, and electro dermal activity (EDA) to investigate task difficulty during code comprehension. Recent efforts to investigate how the human brain processes program comprehension tasks involve the use of functional magnetic resonance imaging (fMRI). For example, fMRI is used to study program comprehension in the brain [*Siegmund et al.*, 2014], different comprehension strategies [*Siegmund et al.*, 2017] and areas of brain activation between source code and natural language tasks [*Floyd et al.*, 2017]. Most recently, *Peitek et al.* [2018] explore the early stages of using both fMRI and eye tracking together for program comprehension tasks. Despite the success of fMRI studies in the domain, fMRI machines remain a costly and restrictive approach, with which it is hard to reproduce the real life working conditions of software developers.

We aim to expand the knowledge on human cognition by introducing functional near infrared spectroscopy (fNIRS) as a more practical tool to empirically investigate the effects of source code on brain activity through the hemodynamic response within physical structures of the brain. FNIRS is a brain imaging technique comparable to fMRI [*Fishburn et al.*, 2014] as both rely on blood-oxygen-level dependent (BOLD) response and show highly correlated results for cognitive tasks. The low cost and minimally restrictive nature of fNIRS makes it particularly well suited to the task of uncovering a deeper understanding of how developers comprehend source code. Existing research involving the use of fNIRS by *Nakagawa et al.* [2014] investigates the hemodynamic response during mental code execution tasks of varying difficulty. The only other fNIRS study in the domain by Ikutani and Uwano, uses fNIRS to investigate the effects of variables and control flow statements on blood oxygenation changes in the prefrontal cortex [*Ikutani and Uwano*, 2014].

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 researchers to pinpoint problematic elements within the source code at a very fine level of granularity.

In this research, we propose a methodology using functional Near Infrared Spectroscopy (fNIRS) and eye tracking devices as an objective measure of program comprehension that allows to conduct studies in environments close to real world settings at identifier level of granularity. We validate our methodology and apply it to study the impact of 21 distinct lexical, structural, and readability metrics on developers' cognitive load during bug localization tasks. This work is an extension of our previous work [*Fakhoury et al.*, 2018]. We follow the same experiment methodology in this paper, expanding our participant pool from 15 to 25, and answering additional research questions, specifically RQ5–RQ8, as outlined in our contributions.

**The contributions of this work are as follows:**

1. A methodology to accurately measure developers' cognitive load at a low level of granularity.
2. A study with 25 undergraduate and graduate students investigating the impact of lexical and structural inconsistencies on cognitive load.
3. Confirming previous results showing that lexical inconsistencies significantly increase developers' cognitive load and that both lexical and structural inconsistencies decrease developers' performance during bug localization tasks (RQ1–RQ4).
4. Evidence that cognitive load significantly increases over identifiers containing linguistic antipatterns (RQ5).
5. A comparison of self-reported measures with cognitive load and eye tracking data showing that the three types of measures capture different aspects of task difficulty (RQ8).
6. A replication package [*Fakhoury*, 2018], 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 and Section 5 discusses the implications of these results. Section 6 discusses the threats to validity of this work. Section 7 discusses related work and Section 8 concludes the study.

## 2 Background

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

### 2.1 Linguistic Antipatterns (LAs)

Several studies have investigated identifier naming patterns [*Blackwell*, 2006] and the effects of identifier naming on developer cognition [*Liblit et al.*, 2006] [*Binkley et al.*, 2009b] [*Takang et al.*, 1996]. There also exists literature aiming to provide guidelines for proper identifier naming [*Deissenboeck and Pizka*, 2006]. Here, we focus on a specific class of linguistic smells that can hinder program comprehension. Linguistic Antipatterns (LAs), are recurring poor practices in the naming, documentation, and choice of identifiers in the implementation of program entities [*Arnaoudova et al.*, 2013]. LAs are perceived negatively by developers as they could impact program understanding [*Arnaoudova et al.*, 2016]. 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 *Buse and Weimer* [2010] 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 [*McCabe*, 1976], nesting depth, the number of arguments, Halstead's complexity measures [*Halstead*, 1977], and overall number of lines of code have also been shown to impact code readability [*Posnett et al.*, 2011].

Table 1 lists method level metrics that have been shown to correlate with readability and comprehensibility [*Buse and Weimer*, 2010; *Halstead*, 1977; *McCabe*, 1976; *Posnett et al.*, 2011; *Scalabrino et al.*, 2016]. 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 the fNIR100, a stand-alone functional brain imaging system, in the shape of a headband, produced by *BIOPAC* [2018a]. It measures blood oxygenation levels in the prefrontal cortex, which is sufficient as studies have shown

Table 1: Method level metrics. 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.

| 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 | − | | |

that mentally demanding tasks require resources in the prefrontal cortex [*Causse et al.*, 2017]. 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 optnodes 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 [*Delpy et al.*, 1988]. 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 [*Treacy Solovey et al.*, 2015]. For example, fNIRS has been successfully used to detect task difficulty in real-time on path planning for Unmanned Air Vehicle tasks [*Afergan et al.*, 2014] and tasks designed to invoke working memory [*Fishburn et al.*, 2014]. The hemodynamic response has been show to have a delay of around six seconds, from the time of exposure to a specific stimulus. [*Kruggel and von Cramon*, 1999] Therefore, when analyzing fNIRs data, researchers must account for this delay.

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 [*Fishburn et al.*, 2014; *Girouard et al.*, 2009; *Herff et al.*, 2014] 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

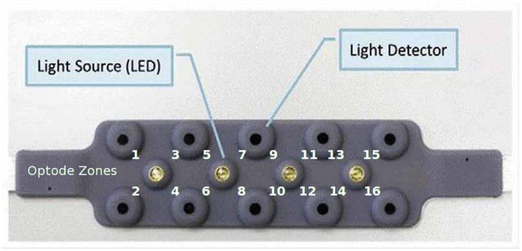


Fig. 1: The fNIRS sensor has 4 light sources and 10 detectors. The 16 optnode locations are mapped on the device.

remove any motion artifacts and noise, as well as to exclude data collected by individual optnodes that may not have been fit properly against the forehead. These optnodes are usually optnodes 1 and 15, which are located on the outer edge of the device, near the user's hairline. These optnodes are easily identifiable as they show patterns of either sharp peaks and dips or remain flat. optnode configuration can be seen in Figure 1. The exclusion of an optnode does not effect the data collected by other optnodes. 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 [*BIOPAC*, 2018b]. 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.

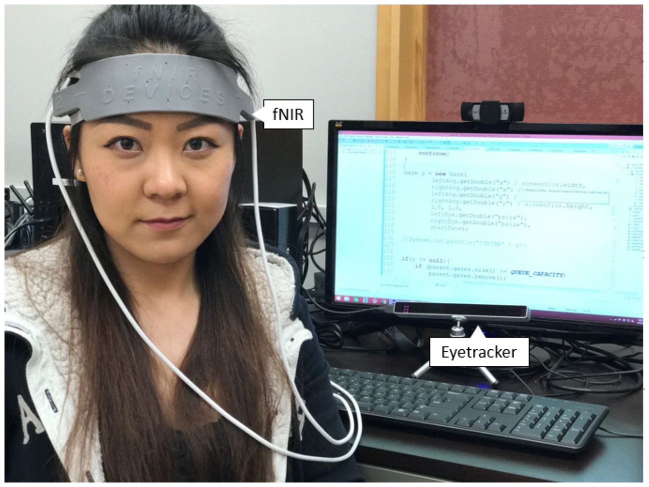


Fig. 2: The actual experimental setup. The image shows one of our authors wearing the fNIRS on her forehead, with the eyetracker placed at the base of the computer monitor.

### 2.4 Eye tracking

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 [*Sharafi et al.*, 2015b] [*Sharafi et al.*, 2015a]. 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 [*Rayner*, 1998]. Therefore, we will use fixation duration to determine areas participants spent a substantial amount of time reading.

We use the EyeTribe eye tracker [*EyeTribe*, 2018] 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 eye tracking studies that study saccades, however it is appropriate for our purpose of investigating fixations within the source code [*Ooms et al.*, 2015]. We calibrate the eye tracker using 16 gaze points (as opposed to 9 or 12 points) to cover the screen with higher accuracy. To ensure the integrity of the eye tracking 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 (less than 19–30 pixels) respectively.

Participants use the Eclipse IDE [*Eclipse*, 2018] as their environment during the experimental tasks. We will be using iTrace [*Shaffer et al.*, 2015], a plugin for Eclipse that interfaces with the eye tracker 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 eye tracker. 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).

Figure 2 shows one of the authors with the complete experimental setup, including the eyetracker, fNIRS device and computer setup that was used by the participants during the study.

## 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 lexical 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.

## 3.1 Research Questions

More precisely, the study aims at answering the following research questions:

1. **RQ1:** *Can developers' cognitive load be accurately associated with identifiers' terms using fNIRS and eye tracking devices?*
   *Why?*: fNIRS and eye tracking devices have not previously been used to assess cognitive load at an identifier level of granularity. Therefore, we must first determine if our methodology is capable of automatically determining areas of high cognitive load correctly at low level of granularity.
   *How?*: We ask participants to perform a comprehension tasks and then explore the similarity between fixations on text highlighted by participants as difficult to comprehend and fixations that are automatically classified as having high cognitive load.
2. **RQ2:** *Do inconsistencies in the source code lexicon cause a measurable increase in developers' cognitive load during program comprehension?*
   *Why?*: Linguistic antipatterns are perceived negatively by developers as they could impact program understanding [*Arnaoudova et al.*, 2016]. We introduce linguistic antipatterns to source code snippets to determine if they also cause an increase the cognitive load of developers.
   *How?*: We ask participants to perform bug localization tasks on a snippet that does not contain lexical inconsistencies and on 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.
3. **RQ3:** *Do structural inconsistencies related to the readability of the source code cause a measurable increase in developers' cognitive load during program comprehension?*
   *Why?*: Various structural aspects of the source code can affect its readability. We want determine if these factors also affect the cognitive load of developers.
   *How?*: We ask participants to perform bug localization tasks on a snippet that contains structural inconsistencies and on one that does not. We then explore the average cognitive load experienced on the two snippets.
4. **RQ4:** *Do both structural and lexical inconsistencies combined cause a measurable increase in developers' cognitive load during program comprehension?*
   *Why?*: We explore a combination of lexical and structural inconsistencies to determine if, when both factors are combined, they cause an increase in the cognitive load of developers.
   *How?*: We ask participants to perform bug localization tasks on a snippet that contains both structural and lexical inconsistencies and on one that does not. We then explore the average cognitive load experienced on the two snippets.
5. **RQ5:** *Does the presence of inconsistencies in the source code lexicon affect the cognitive load of developers over an entire source code snippet or only over the identifiers that are involved in the inconsistencies?*
   *Why?*: When answering research questions 2-4 we use the average cognitive load experienced over an entire snippet to determine if the treatment effected participants. However, participants could initially struggle to understand source code due to the various treatments, but eventually get past the inconsistencies and figure out an alternative way to solve the problem. Therefore, we want to use temporal eyetracking and fNIRS data to observe cognitive load

fluctuations more precisely over the course of the task. This is the primary motivation behind RQs 5 – 7. For RQ5 in particular, if we observe overall increased cognitive load for source code snippets that contain linguistic antipatterns, we want to determine what areas of the source code the increase cognitive load comes from.
*How?*: We compare the normalized Oxy distributions over identifiers from control snippets and those from LA treatment snippets. Identifiers from LA treatments that contain LAs are separated from those that do not contain LAs. This allows us to determine if cognitive load is spread between these two distributions, or if it stems only from identifiers that contain LAs. We also rank identifiers in control snippets and LA snippets by the average normalized Oxy to determine which identifiers have high cognitive load in both control and LA treatments.

6. **RQ6**: *Is participant performance, in terms of success rate and task duration, affected by the presence of lexical and structural inconsistencies?*
   *Why?*: Difficult tasks are expected to cause an increase in cognitive load of developers but also they could have a lower success rate and higher task duration. We want to determine how structural and lexical inconsistencies affect these performance metrics.
   *How?*: We calculate the task duration of each participant completed task and assess the answers from the post analysis survey to determine if a bug localization task was successfully completed or not.
7. **RQ7**: *Does fixation duration significantly increase over identifiers containing lexical inconsistencies?*
   *Why?*: Fixation duration has been shown to be positively correlated with cognitive effort [*Rayner*, 1998] and has been used in software engineering to measure the visual effort experienced by participants [*Binkley et al.*, 2013; *Sharafi et al.*, 2012; *Sharif et al.*, 2012]. In this research question we want to determine if fixation duration increases over identifiers that contain linguistic antipatterns.
   *How?*: We calculate the fixation duration over unique identifiers in control and LA versions of each treatment snippet. We then test for significant between the duration distributions for each method.
8. **RQ8**: *Are self-reported measures consistent with cognitive load and fixation duration data?*
   *Why?*: Different approaches can be used to measure task difficulty. Self-reported measures are the easiest to collect during experiments but they might not be always accurate [*Hochstein et al.*, 2005]. Fixation duration has been shown to be positively correlated with cognitive effort [*Rayner*, 1998]. In this work, we measure Oxy as it has been previously shown that it is a function of task difficulty [*Fishburn et al.*, 2014; *Girouard et al.*, 2009; *Herff et al.*, 2014]. This research question investigates whether these three ways of measuring task difficulty are consistent.
   *How?*: We use answers from the post analysis survey to determine the difficulty rating of each task. We also calculate the average Oxy and fixation duration per participant task, and perform a pairwise test for correlation between the three metrics.

## 3.2 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:

- Participants must be able to understand the snippet on its own, without too many 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 JFree-Chart—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.

### *3.2.1 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 [*Fakhoury*, 2018].

#### *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.* [2013]. 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.

For example, listings 1 and 2 show the control and LA treatments respectively of the same snippet. In this example, we modify the method signature to introduce LA **A.3** (*"Set" method returns*). We also add documentation to describe an if statement that is not implemented in the source code to introduce LA **B.1** (*Not implemented condition*). We intoduce LA **C.2** (*Method signature and comment are opposite*) by indicating in the comment that the 'last index' is found instead of the 'first' index. LA **D.1** (*Says one but contains many*) relates to renaming the array 'searchStrings' to 'potentialString' which both suggests a singular instance instead of plural and reflects a more ambiguous meaning than the original term.

```
/** <p>Find the first index of any of a set of potential substrings.</p>
* <p/>
* <p><code>null</code> String will return <code>-1</code>.</p>
*
* @param string the String to check
* @param searchStrings the Strings to search for
* @return the first index of any of the searchStrings in string
* @throws NullPointerException if any of searchStrings[i] is
    <code>null</code> */
public static int firstIndexOfAny( String string, String [] searchStrings)
{
   if ( ( string == null ) || ( searchStrings == null ) )
   {
      return -1;
   }
   // String's can't have a MAX_VALUEth index. So begin by initilizing
       resultIndex to max int value.
   int resultIndex = Integer.MAX_VALUE;

   int temp;
   for ( String searchString : searchStrings )
   {
   ...
}
```

Listing 1: Part of source code snippet with control treatment.

```
/**
* <p>Find the last index of any of a set of potential substrings.</p>
* <p/>
* <p><code>null</code> String will return <code>-1</code>.</p>
* if the case of the substring matches that of the case within the string
* return this value first.
*
* @param string the String to check
* @param potentialString the Strings to search for
* @return the first index of any of the potentialString in string
* @throws NullPointerException if any of potentialString[i] is
    <code>null</code> */
public static int setFirstIndexOfAny(String string, String [] potentialString)
{
   .....
}
```

Listing 2: Comment and method signature for snippet with LA treatment.

*Structural and Readability Metrics*

We alter the code snippets to change the values of 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.

Recall the example presented in the previous section. Code listing 3 contains a part of the snippet with the structural treatment that corresponds to the control snippet shown in code listing 1. Examples of changes here include modifying identifier terms so that they do not follow camelCase typesetting, reducing the number of comment lines and spaces, increasing the number of parameters, variable declarations, expressions, if statements, and lines of code. We also format the source code against typical java conventions, for example, indentation and brackets are misaligned.

```
 /**
 * <p>Find the first index of any of a set of potential substrings.</p>
 * <p/>
 * <p><code>null</code> String will return <code>-1</code>.</p>
 */
public static int firstindexofany( String string, String [] searchstrings,
     int numbOfStrings)
 {
   int notfound =-1;

     if  ( searchstrings == null ){ return notfound;
     }
   if(string ==null) return notfound;

     int resultindex
     = Integer.MAX_VALUE;
     int temp; int i;
     for ( i=0; i< numbOfStrings; i++)
     {
     ...
  }
```

Listing 3: Part of source code snippet with structural treatment.

## 3.3 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 by filling out an eligibility survey. 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 make sure the participants will be able to

Table 2: Participants' demographic data.

| Demographic Variables | | # Partic. | # New |
|---|---|---|---|
| Programming Languages | C++ | 5 | 2 |
| | Java | 1 | 3 |
| | Both | 9 | 5 |
| Degree Pursuing or Completed | Bachelor | 8 | 6 |
| | Master | 2 | 2 |
| | PhD | 5 | 2 |

Table 3: Study design.

| | | Comprehension Task | Bug Localization Task | | |
|---|---|---|---|---|---|
| | # Partic. | | Snippet 1 | Snippet 2 | Snippet 3 |
| Group 1 | 7 | Prose | Control | LA | Structural |
| Group 2 | 5 | German Code | LA & Structural | Control | LA |
| Group 3 | 8 | Prose | LA | Structural | LA & Structural |
| Group 4 | 5 | German Code | Structural | LA & Structural | Control |

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 eye tracker 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 eye tracking data may be significantly impacted. Twenty five participants satisfied the eligibility criteria and participated in the experiments. The remaining students who indicated their interest either did not satisfy the eligibility requirements or did not complete the experiment scheduling process. Table 2 summarizes the programming language in which participants describe themselves as more proficient and their educational background.

## 3.4 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 summarizes the design of the experiment.

Different groups contain different number of participants to account for data collection issues that would otherwise cause an imbalance in the number of tasks included in data analysis. The table contains the number of participants in each group. Participants have between 1-15 years of programming experience, with an average of around 3.5 years of experience, first quartile at 2 and the third quartile at 4 years.

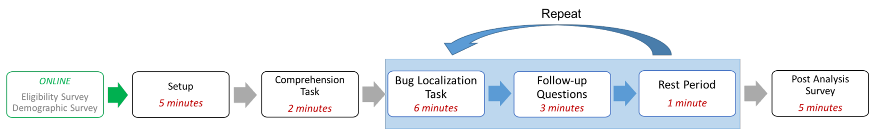


Fig. 3: Overview of the experimental procedure.

## 3.5 Procedure

Figure 3 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.

### *3.5.1 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.

### *3.5.2 Comprehension Task*

To answer **RQ1**, 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.

### *3.5.3 Bug Localization Task*

The bug finding task allows us to answer **RQ2**– **RQ8**. During this task participants are shown a relatively short code snippet on the computer screen. They are told

[1] The experiment was approved through a full board review for human subject research from the Institutional Review Board (IRB) at Washington State University (IRB #16113).

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).

#### *3.5.4 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.

#### *3.5.5 Rest Period*

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

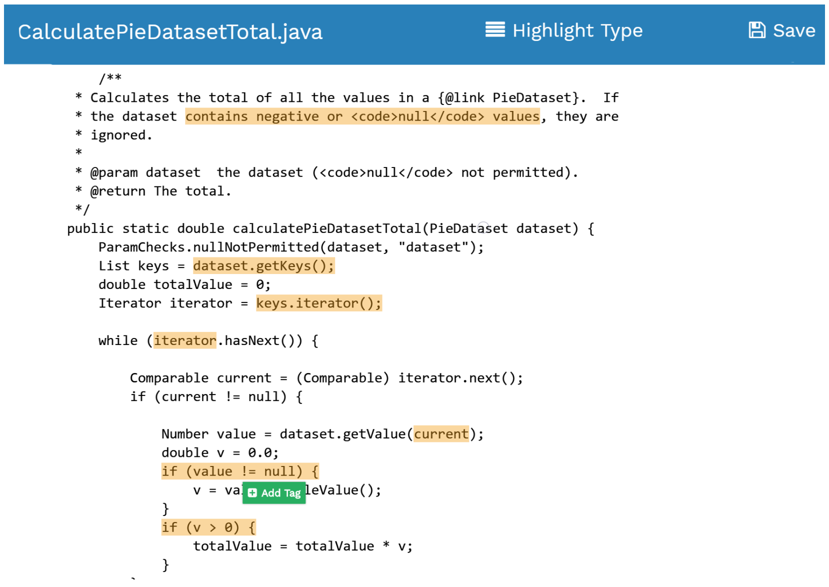
#### *3.5.6 Post Analysis*

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.

### 3.6 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 localization tasks, and decided to



```
    /**
  * Calculates the total of all the values in a {@link PieDataset}.  If
  * the dataset contains negative or <code>null</code> values, they are
  * ignored.
  *
  * @param dataset  the dataset (<code>null</code> not permitted).
  * @return The total.
  */
 public static double calculatePieDatasetTotal(PieDataset dataset) {
     ParamChecks.nullNotPermitted(dataset, "dataset");
     List keys = dataset.getKeys();
     double totalValue = 0;
     Iterator iterator = keys.iterator();

     while (iterator.hasNext()) {

         Comparable current = (Comparable) iterator.next();
         if (current != null) {

             Number value = dataset.getValue(current);
             double v = 0.0;
             if (value != null) {
                 v = va         leValue();
             }
             if (v > 0) {
                 totalValue = totalValue * v;
             }
```

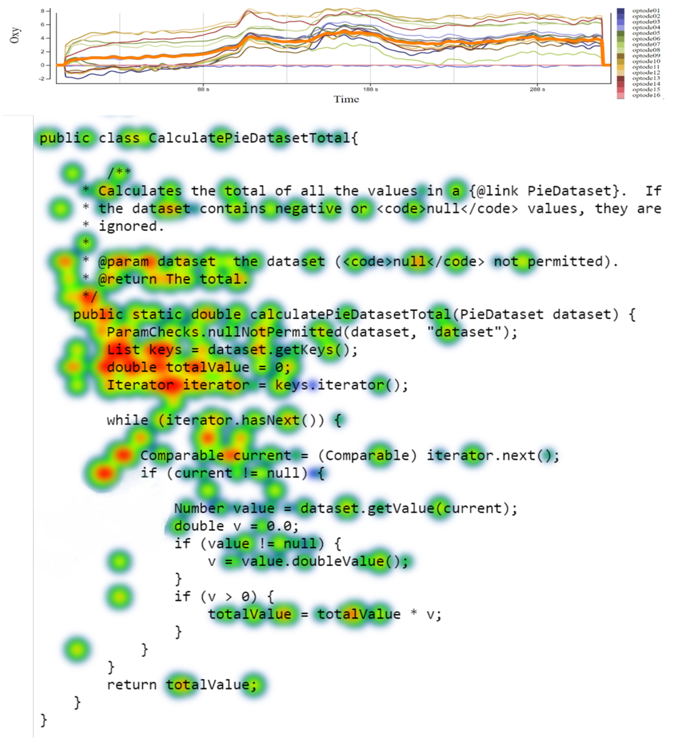
Fig. 4: Highlighting Tool Interface, used to tag source code snippets. The figure shows the tool in as it is displayed to the user in the participant highlight mode.

reduce this to three. One of the snippets that was initially chosen makes references to external methods; it was discarded after the pilot study.

## 3.7 Analysis Method

### *3.7.1 Tagging Source Code Snippets and Gaze Data*

Before answering the research questions presented in Section 3.1 we must tag the source code snippets with specific information needed to carry out the analysis. For example, which identifier contain Linguistic Antipatterns, which areas of the code are affected by structural elements, and which areas of code were highlighted/commented on by participants. To facilitate the analysis, we built a tool that takes source code files and synchronized eye tracking and fNIRS data files and acts as an interface to highlight and tag source code snippets. We then map these tags to the corresponding gazes in the eye tracking data. We can then analyze gazes by looking at their corresponding fNIRS data, in addition to any structural/LA data and highlights/comments left by participants. Figure 4 shows the UI for the highlight tool, where participants can highlight specific areas of source code and add comments to explain highlighted areas. The tool also allows researchers to highlight identifiers that contain linguistic antipatterns and structural defects. Figure 5 shows the UI for tool that generates the vizualization of Oxy and eye tracking data. The tool creates a heatmap using color to indicate Oxy levels for each fixation; we use the tool during post analysis surveys.



```
public class CalculatePieDatasetTotal{

        /**
     * Calculates the total of all the values in a {@link PieDataset}.  If
     * the dataset contains negative or <code>null</code> values, they are
     * ignored.
     *
     * @param dataset  the dataset (<code>null</code> not permitted).
     * @return The total.
     */
    public static double calculatePieDatasetTotal(PieDataset dataset) {
        ParamChecks.nullNotPermitted(dataset, "dataset");
        List keys = dataset.getKeys();
        double totalValue = 0;
        Iterator iterator = keys.iterator();

        while (iterator.hasNext()) {

            Comparable current = (Comparable) iterator.next();
            if (current != null) {

                Number value = dataset.getValue(current);
                double v = 0.0;
                if (value != null) {
                    v = value.doubleValue();
                }
                if (v > 0) {
                    totalValue = totalValue * v;
                }
            }
        }
        return totalValue;
    }
}
```

Fig. 5: Visualization Tool Interface, used to generate Oxy heatmaps using fixation data.

### 3.7.2 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 of 20% for overall Oxy and 10% for delta values. 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 7 out of 25 participants.

### *3.7.3 Optnodes*

FNIRS data is susceptible to noise thus proper placement of the device on a participant's forehead is crucial to data acquisition. Sometimes however, certain optnodes do not record data or are incredibly noisy. For example, if the device is too large for a participant's forehead an optnode might not be in contact with the skin, or could be obstructed by hair. In these cases, the data from the noisy optnodes must be filtered out. We have made two observations: 1) generally, data from uncompromised optnodes follows a similar trend and 2) optnodes that are compromised either significantly diverge from the trend and contain sharp changes in HBO, or do not have any data at all. The Spearman correlation evaluates a monotonic relationship between any two continuous variables that change together but not necessarily at a constant rate. Therefore, in order to determine which optnodes should be rejected we calculate the Spearman correlation matrix for each optnode pair using raw HBO data. An average correlation is calculated for each column in the matrix. We then experiment with rejection thresholds using data from 5 participants. We choose to retain optnodes with average correlation coefficient greater than 0.3. We find that this threshold only excludes optnodes that are clearly noisy while not assuming all optnodes should follow the same trend too closely.

### *3.7.4 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:

$$normalizedOxy = \frac{Oxy_{raw} - Oxy_{min}}{Oxy_{max} - Oxy_{min}} \tag{1}$$

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 *Ikutani and Uwano* [2014].

### *3.7.5 fNIRS and Eye Tracking Data*

Raw data collected from the fNIR device is first preprocessed using fnirSoft, software provided by Biopac, the manufacturers of the device. This preprocessing involves appling the modified Beer-Lamber Law [*Baker et al.*, 2014] and a low pass FIR filter of order 20. We do not apply any preprocessing to the eyetracker 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. Figure 6 outlines the data synchronization process between the fNIRS and eye tracking data. To accommodate for the natural delay in the hemodynamic response in the brain to an external stimulus, such as reading source code, the fNIRS fixation data is synchronized to the eye tracking data with a delay of 6 seconds [*Kruggel and von Cramon*, 1999]. 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 of omission 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.

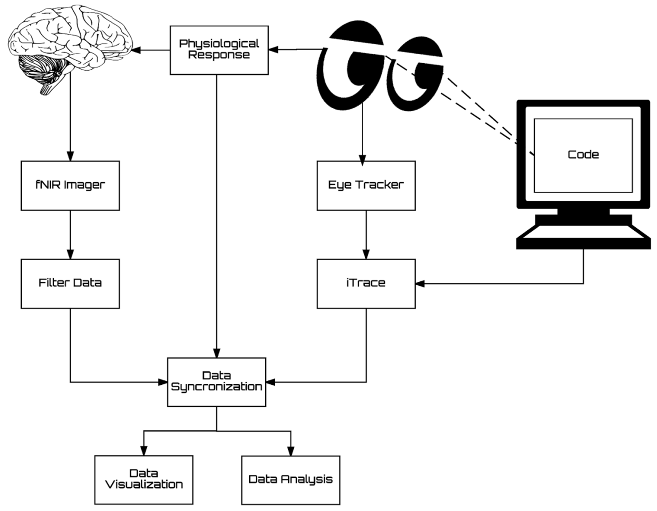


Fig. 6: This figure illustrates how the flow of data from fNIRS and eye tracking devices is collected and synchronized.

### *3.7.6 Simple Matching Coefficient (SMC)*

To answer **RQ1**, we use the Simple Matching Coefficient [*Sokal*, 1958]—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.

### *3.7.7 Kruskal Wallis Hypothesis 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. For the above research questions, we use the Kruskal-Wallis hypothesis test, a non-parametric statistical test used to compare two or more independent samples, to assess whether the population medians differ. The Kruskal-Wallis test requires 5 or more observations, which makes it more suitable as compared to other non-parametric tests, considering our population sample size. 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.

### *3.7.8 Mann Whitney U Test*

To answer **RQ5** we consider the normalized Oxy values of all fixations from control and LA treatment tasks. LA task fixations are further filtered by fixations over identifiers that contain linguistic antipatterns and those that do not. There are three distinct distributions of normalized Oxy values that we compare. We use the Mann Whitney U test, a non-parametric statistical test used to determine if two independent samples are from the same distribution. Our null hypothesis for **RQ5** is that there is no difference between the normalized Oxy values for the control snippets and fixations that contain LAs and those that do not contain LAs from the LA 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 LA and non-LA fixation groups.

To answer **RQ7** we investigate the duration of fixations over identifiers from LA treatment snippets that contain linguistic antipatterns and identifiers from control treatment snippets from the three different bug localization source code snippets. To take into account the frequency of identifier fixations, we aggregate the duration of fixations over the same identifiers together, within the same participant and task. Thus, for all identifiers, we calculate the sum of duration times. Our null hypothesis is that there is no difference between the duration of fixations in control snippets and fixations in LA treatment snippets. Our alternative hypothesis is that the duration of fixations in control snippets are lower than those in LA treatment snippets.

### *3.7.9 Cliff's Delta (d) Effect Size*

After performing the Kruskal-Wallis 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 [*Grissom and Kim*, 2005] 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 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.

### *3.7.10 Kendall's Rank Correlation Coefficient*

Kendall's Rank Correlation Coefficient is a non-parametric test to measure the strength of dependence between two variables based on ranks of the data. The correlation coefficient can range between -1 (perfect negative correlation) to +1 (perfect positive correlation). To answer $\mathbf{RQ_8}$ we use Kendall's Rank Correlation Coefficient to determine the relationship between self-reported task difficulty, cognitive load, and eye tracking data.

### *3.7.11 Identifier Ranking*

For $\mathbf{RQ_5}$, to investigate which identifiers contain the highest normalized Oxy we generate a ranking of identifiers. Fixations over the same identifiers from control snippets are grouped within file, across participants and the normalized Oxy mean is generated. Similarly, a mean Oxy value is calculated for each distinct identifier in LA snippets. We calculate the mean Oxy so that identifiers are ranked by cognitive load, independent of the number of fixations, and duration over the identifier.

Similarly, for $\mathbf{RQ_7}$ we generate a ranking with the identifiers that were looked at the longest, across all participants.

## 4 Results

We conducted experiments with 10 new participants, for a total of 25 experiments including the initial 15 participants from our previous work [*Fakhoury et al.*, 2018]. We collected data for 100 tasks; each participant completes 4 tasks, one comprehension task and three bug localization tasks. We discarded 8 out of the 100 tasks. For 3 of the 8 tasks, the participants appeared to be in a hurry and did not spend the time to try and understand the source code snippet, in two cases stating that the structure of the code was off putting. In 2 tasks there was an issue with the generated eye tracking data files, in 2 other tasks the participant clicked outside of the IDE, which stops iTrace from collecting gaze data, and in the last task the fNIRS baseline was improperly conducted.

**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. SMC values in the table are reported for each participant, according the

the task treatment type they received. Data for the task of one participant was discarded due to data collection issues. The average SMC for the two comprehension snippets, German code and English prose, is 0.73 and 0.76 respectively, with a total average of 0.74. This means that 74% 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.

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.

With respect to our previous work [*Fakhoury et al.*, 2018], we observe an 8% drop (from 81% to 73%) for the German code and a slight increase for the prose (from 74% to 76%).

When exploring the nature of the discrepancy over the remaining 27% 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. One participant exhibits high cognitive load over the line of code: `if(pos < 0)`, when asked if this statement indeed caused any confusion, the participant explained that it is not a confusing statement, but that it requires some 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 five participants exhibit high cognitive load on words that are in sentences that contain other highlighted words. This could be due to cognitive load carried out from difficult words to other parts of the sentence, or participants might be more inclined to highlight the most problematic words rather than all words that posed difficulty.

**$\mathbf{RQ_1}$** Summary: Using fNIRS and eye tracking devices, developers' cognitive load can be accurately associated with identifiers in source code and text, with a similarity of 74% 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 7 contains the distribution of normalized Oxy averages calculated per participant and task, for all treatment types.

There are a total of seventeen participants that completed tasks with the control treatment and fifteen participants that completed tasks with LA treatment. Thirteen participants were able to complete bug localization in the control snippets successfully, ten participants were able to complete bug localization in the lexical snippets. Performing the Kruskal-Wallis test we obtain a significant p-value

Table 4: Similarity between fixations with high cognitive load and highlighted fixations. SMC values are reported for each participant, according to the type of treatment task they received.

| Treatment | SMC | Treatment | SMC |
|---|---|---|---|
| German Code | 0.61 | Prose | 0.81 |
| | 0.63 | | 0.76 |
| | 0.84 | | 0.91 |
| | 0.70 | | 0.93 |
| | 0.75 | | 0.75 |
| | 0.72 | | 0.87 |
| | 0.67 | | 0.56 |
| | 0.72 | | 0.77 |
| | 0.58 | | 0.65 |
| | 0.81 | | 0.60 |
| | 0.92 | | 0.74 |
| | 0.82 | | 0.80 |
| Average | 0.73 | Average | 0.76 |
| **Total Average** | | | **0.74** |

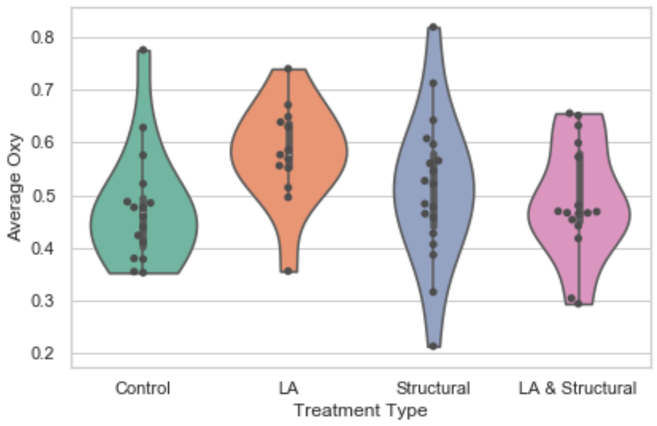


Fig. 7: Normalized Oxy Averages Calculated Per Snippet for all Treatments.

($p - value = 0.005$), with a large effect size ($d = -0.992$), which indicates that the presence of linguistic antipatterns in the source code significantly increases the average Oxy a participant experiences. Results are consistent with results of our previous work [*Fakhoury et al.*, 2018], which was conducted on a smaller population.

From the post analysis survey, we observe that participants made comments on source code containing linguistic antipatterns in 14 out of 20 tasks. Six participants highlighted linguistic antipattern **B.1** (*Not implemented condition*), where a condition in a method comment is not implemented. Two of these two participants showed high cognitive load when reading the line of comment that was not implemented and all six participants explicitly stated that they spent time searching the code for the unimplemented condition but could not find it. Also, five participants highlighted 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. Only three participants commented on identifiers that contained linguistic antipattern **E.1** (*Says many but contains one*), stating that they were poorly named. One participant only noticed the linguistic antipattern after it was pointed out to them, after which they immediately realized it had actually confused them during the task because were "thinking of the identifier as multiple values".

**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?*

In Figure 7 we can observe the distribution of normalized average Oxy for snippets with control and structural treatments. We include data for a total of nineteen participants that completed a structural treatment. Eleven participants were able to complete bug localization in the structural snippets. Results from the Kruskal-Wallis test are not statistically significant ($p-value = 0.246$), with a small effect size ($d = -0.281$), 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. These results are also consistent with results found in our previous work [*Fakhoury et al.*, 2018] on a smaller population.

From the post analysis survey, we observe that participants made comments on source code containing poor structure in 18 out of 20 tasks. 16 participants found that poor structure, including incorrect indentation and breaking one line of code into multiple lines, creates frustration and slows down their performance in bug localization tasks. One participant commented that *"terrible formatting severely increases readers burden"*. Only two participants commented that the structure was not confusing since they were able to click on opening brackets to find the associated closing brackets in the Eclipse IDE. However despite this frustration, there is no evidence that structural inconsistencies increase the average cognitive load that participants experience.

**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?*

In Figure 7 we can observe the distribution of normalized average Oxy for snippets with both control and LA & structural treatments. We include data for 17 participants who completed a task containing both structural and lexical inconsistencies. Seven of these participants successfully completed the bug localization task. Performing the Kruskal-Wallis test did not show statistically significant results ($p - value = 0.624$), with a small effect size ($d = -0.120$), meaning that there is no evidence that structural inconsistencies combined with linguistic antipatterns significantly increase the cognitive load that participants experience. These results are also consistent with results found in our previous work [*Fakhoury et al.*, 2018] on a smaller population.

Interestingly, for four 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 in the code, which indicates that the treatment did negatively affect their comprehension of the code. For the 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 for two of the participants).

From the post analysis survey, two participants highlighted 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 participants who identified structural inconsistencies in source code highlighted that such inconsistencies caused distractions and prolonged the bug localization task. Only one participant commented that although the indentations was frustrating, it did not hinder bug localization.

**RQ$_4$** Summary: There is no evidence that structural inconsistencies combined with linguistic antipatterns significantly increase the cognitive load that participants experience. However, source code containing both lexical and structural inconsistencies mislead more than 55% of the participants. Participants who successfully completed the bug localization tasks experienced higher cognitive load on code containing both inconsistencies compared to the control snippets.

So far, RQ1 – RQ4 have used average cognitive load over an entire snippet to determine if treatments effected participants. Next, we will use temporal data from eye tracking and fNIRS devices to observe cognitive load fluctuations more precisely over the course of the task.

**RQ5:** *Does the presence of inconsistencies in the source code lexicon affect the cognitive load of developers over an entire source code snippet or only over the identifiers that are involved in the inconsistencies?*

Table 5 contains the p-values from the Mann-Whitney U tests. The amount of cognitive load over fixations that contain linguistic antipatterns is significantly different compared to the cognitive load of fixations from control snippets, with, large effect size (d = 1). Higher cognitive load is also experienced by participants over fixations that do not contain linguistic antipatterns in the LA treatments compared to fixations that do not contain linguistic antipatterns in the control treatments,

Table 5: Mann Whitney U test p-values for Normalized average Oxy of fixations in LA and Control treatments.

| Treatment/Fixation | Control / non-LA Fixations | LA / LA Fixation |
|---|---|---|
| LA / LA Fixations | 0.00047 | - |
| LA / non-LA Fixations | 0.038 | 0.00011 |

with large effect size (d = 0.83). These results suggest that the cognitive load due to the presence of linguistic antipatterns is not experienced in isolation but instead, the overall understanding of a piece of source code is affected.

However, when testing between fixations that contain LAs and those that do not contain LAs within the LA treatment snippets, we obtain a $p-value$ of 0.00011 with large effect size (d = 1). Meaning that identifiers that do contain linguistic antipatterns cause a significantly higher increase in cognitive load as compared to those that do not contain LAs within the same treatment.

Next, to investigate which identifiers contain the highest normalized Oxy we generate a ranking of identifiers. In the LA treatment version of the `calculatePie-DatasetTotal` snippet, the top ranked identifier is `int currentValues` which is altered to contain linguistic antipattern **E.1** (*Says many but contains one*). This identifier is used multiple times throughout the snippet and is part of the bug. However, the corresponding identifier in the control treatment `value` is not among the top ranking identifiers. The majority of the top ranked identifiers for the LA snippet are on comments whereas for the control version, the highest ranked identifiers are mostly on `if` statements and external method calls. Identifiers relating to the bug and the `iterator` class are highly ranked in both treatments.

For the LA treatment version of `indexOfAny`, the top ranked identifier is from the comment `begin by initializing resultIndex to min double value`, which was altered to contain the linguistic antipattern **F.2** (*Attribute signature and comment are opposite*). However, the corresponding comment in the control version of the method is ranked as the 4th highest. This comment precedes the initialization of `resultIndex` and it is directly involved in the bug for the snippet, which could explain the relatively high ranking in both treatments.

The top identifier rankings for the LA and control treatments of the method `replace` are very different. Similar to the other two snippets, identifiers with linguistic antipatterns are among the top ranked identifiers, however identifiers related to the bug are also highly ranked in both snippets. Several identifiers that do not contain linguistic antipatterns or bugs are also highly ranked. This suggests that Oxy is spread out over the entire source code snippets and is not only concentrated on specific identifiers.

**RQ$_5$** Summary: The presence of LAs significantly increases the cognitive load of developers both throughout the entire source code snippet as well as over the specific areas in the source code that contain linguistic antipatterns. However, identifiers involved in LAs are associated with significantly higher cognitive load compared to other identifiers in the same code snippet.

Table 6: Bug localization task results: success rate and time.

| Treatment | # Participants | # Bugs Found (%) | Avg. Time (min:sec) |
|---|---|---|---|
| Control | 17 | 13 (76.4%) | 3:21 |
| LA | 20 | 13 (65.0%) | 5:13 |
| Structural | 20 | 11 (55.8%) | 4:03 |
| LA & Structural | 18 | 8 (44.4%) | 4:57 |

**RQ6:** *Is participant performance, in terms of success rate and task duration, affected by the presence of lexical and structural inconsistencies?*

Overall, 44 out of 75 bug localization tasks were completed successfully. The distribution of successfully completed tasks for the four treatment groups is shown in Table 6. In the control group, 76.4% of the bugs were found with average time of 3.35 minutes. The success rate decreases as linguistic antipatterns and poor structure characteristics are added (65% and 55.8%, respectively). At the same time, the average time spent on bug localization increases as linguistic antipatterns and poor structure characteristics are added (5:13 min/sec and 4:03 min/sec, respectively). When both linguistic antipatterns and poor structure are present in the code snippets, only 44.4% of the bugs were found successfully with an average time of 4:57 min/sec. The outcome shows that the presence of structural and lexical inconsistencies slows down participants and even hinders bug localization.

It is interesting to note that the success rate for snippets with structural treatments are the lowest. This could be due to the fact that at an initial glance, in structural treatments, participants see that there are obviously problems with the source code. This could cause participants to be less careful about trying to fully comprehend the code, and prematurely report a bug, even though they have not finished understanding the code.

**RQ$_6$** Summary: Participants spend more time reading snippets that contain lexical or structural inconsistencies. We observe the lowest success rate in treatments that contain both lexical and structural inconsistencies.

**RQ7:** *Does fixation duration significantly increase over identifiers containing lexical inconsistencies?*

Table 7 contains the number of fixations and participants per task, for each code snippet as well as the results from the Mann Whitney U test for the difference between the fixation duration in control and LA snippets.

For `CalculatePieDatasetTotal` and `indexOfAny` we obtain a $p - value$ of 0.480 (Median$_C$=592, Median$_L$=610, U=294151) and 0.102 (Median$_C$=593, Median$_L$=610, U=860865.5), respectively, which indicates that the amount of time spent by participants reading identifiers in control and LA snippets is not significantly different. For `replace`, we obtain a $p - value$ of 0.026 (Median$_C$=562, Median$_L$=594, U=1160845.5), which indicates the distributions are significantly different.

In order to further understand on which identifiers participants focus the most, we rank them by duration in both treatments, for each snippet, from the highest to the lowest value. In the control treatment version of `calculatePieDatasetTotal`,

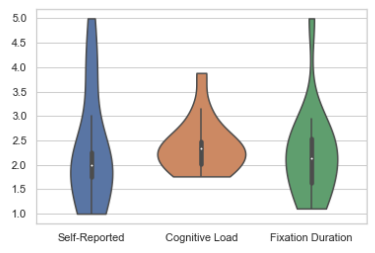
Table 7: The total number of fixations, the number of participants, and the p-values when comparing fixation duration between control and LA snippets.

| Source code snippet | | Control Treatment | Lexical Treatment | Duration p-value |
|---|---|---|---|---|
| `CalculatePieDatasetTotal` | # Fixations | 1259 | 335 | 0.48 |
| | # Participants | 6 | 5 | |
| `replace` | # Fixations | 1271 | 301 | 0.026 |
| | # Participants | 4 | 5 | |
| `indexOfAny` | # Fixations | 1685 | 490 | 0.102 |
| | # Participants | 5 | 5 | |

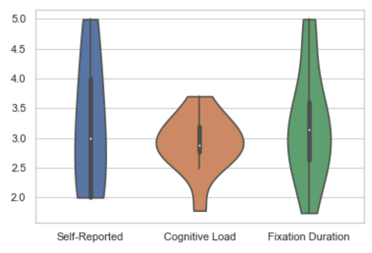
the top identifier is from the line of code `ParamChecks.nullNotPermitted(dataset, "dataset");` which is an external Java class. In the post analysis survey, participants that highlighted this line of code explained that they weren't familiar with the class thus they either read the javadoc or guessed the functionality of the class based on the preceding comment. Indeed, the next highest ranked identifier is from the comment *"if the dataset contains negative or null values they are ignored"* which directly explains the functionality of the `ParamChecks` class. The next highest ranked identifier is `totalValue` which is used multiple times throughout the source code snippet. The bug localization task for this file involves this identifier directly which could explain why it is so highly ranked. If we look at the highest ranked identifiers from the LA treatment version of the `calculatePieDatasetTotal` snippet we find a very different ranking. The top identifier for the LA treatment of this snippet is `double currentValues`. This identifier corresponds to the identifier `double v` from the control snippet, but was altered to contain linguistic antipattern **E.1** (*Says many but contains one*), where the identifier name (`currentValues`) suggests a collection, but the type (`double`) does not. The next highest ranked identifier is from the statement `iterator.next()` which is in a `while` loop. Participants who highlighted this line of code in the post analysis survey said that they looked at this identifier multiple times while mentally executing the while loop. The identifier `ParamChecks` from `ParamChecks.nullNotPermitted(dataset, "dataset");` is also highly ranked in the LA snippet.

We perform the same analysis for `indexOfAny`. The highest ranked identifier for the control snippet is `temp` which is used multiple times throughout the method and is directly related to the bug in the code. The next highest rated identifier is `indexOf` from the statement `temp = string.indexOf(searchString)`. Both `temp` and `indexOf` are highly rated in the LA treatment version of the snippet as well, which could mean that executing the `indexOf` statement mentally is inherently more complex as compared to the rest of the code. We find that the top rated identifier for the LA snippet is from statement `resultIndex = Integer.MAX_VALUE`. `resultIndex` is also highly ranked in the control snippet, and is related to the bug. Both identifiers `searchStrings` and the equivalent identifier in the LA treatment `potentialString`, which was altered to introduce linguistic antipattern **D.1** (*Says one but contains many*), are similarly ranked in both treatments.

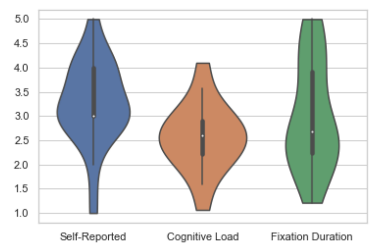
When ranking identifiers for the control and LA treatment versions of snippet `replace`, we notice very similar ranking between the two. The top two identifiers in both treatments are from `buf.append()` and `text.indexOf()` statements.



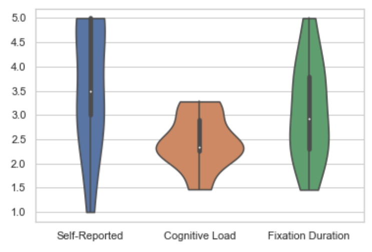
(a) Control treatment



(c) LA treatment



(b) Structural treatment



(d) LA & Structural treatment

Fig. 8: Task difficulty as per self-reported, cognitive load, and fixation duration data.

`StringBuilder, text.length(), replacementCount` are all among the top 7 identifiers with the highest duration in both treatments.

**RQ$_7$** Summary: Fixation duration increases significantly over identifiers involved in LAs for only one of the three code snippets.

**RQ8:** *Are self-reported measures consistent with cognitive load and fixation duration data?*

To answer this research question we look at participant rated task difficulty per treatment type. We asked participants to rate the difficulty of each bug localization task on a scale from 1 to 5, 1 being the lowest and 5 being the highest. Figure 8 contains the distributions of their answers per treatment type. We also plot cognitive load and fixation duration data on the same graph, with the values normalized to the 5 point scale. The median difficulty rating for control snippets is 2 (see Figure 8a). Two participants that rated two control treatment of snippet `indexOfAny` as 4 and 5 were not able to find the bug. LA treatment snippets and structural treatment snippets have a median difficulty rating of 3 (see Figure 8b and 8c, respectively), whereas snippets containing both LA and structural inconsistencies are rated as 3.5 (see Figure 8d). We calculate Kendall's Tau correlation coefficient between self-reported task difficulty, average normalized Oxy, and fixation duration to determine if the three metrics are consistent with each other. Table 8 contains the correlation coefficients by treatment type. The correlation coefficients between self-reported task difficulty and cognitive load are weak across all treatments, ranging from 0.060 to -0.007. Similarly, for fixation duration and cognitive load correlation values range between 0.15 and -0.23. The correlation coefficients for self-reported task difficulty and fixation duration are relatively higher, with a moderate correlation for control and LA & structural treatment but still with a weak correlation for LA and structural treatments.

Overall, our data shows that there is very little correlation between these three metrics, which suggests that they are capturing different aspects of difficulty. For instance, fixation duration captures the visual effort of participants over different areas of the source code. Oxy captures the cognitive load experienced at different points in the source code. And self-reported task difficulty is an overall subjective assessment of how difficult the task was as perceived by participants.

In fact, we find that in 7 out of 9 cases where we observe high Oxy but low self-reported difficulty, the bug localization tasks were successfully completed. Thus, while participants are to some extent positively affected by the satisfaction of successfully completing the task and rate it as relatively easy, Oxy values capture the cognitive load associated with the entire process of mentally executing the source code and searching for the bug. Moreover, we notice that when participants do not have an idea of what the bug could be, they consistently rate the task as difficult, yet their average Oxy ranges from 0.37 to 0.56 which is relatively low to medium cognitive load. If participants thought they knew what the bug was, they were more likely to rate the task as easier, even if it was incorrect. Participant's cognitive load data is a measurement of how hard their brain is working to understand a piece of source code. To successfully complete a bug localization task, a participant must mentally execute the code in their mind and have a deep understanding of how the code works. If the participant cannot do this, their cognitive load will not increase, and thus it will not be captured by the fNIRS. Tasks that are not fully understood by participants may not increase their cognitive load, but they are inclined to rate them as more difficult because they did not have an idea where the bug could be. This could explain the discrepancy between the distribution of cognitive load data and the self-reported task difficulty.

**RQ$_8$** Summary: Self-reported task difficulty, cognitive load, and fixation duration largely do not correlate and appear to be measuring different aspects of task difficulty. Fixation duration captures the visual effort of participants over different areas of the source code. Oxy captures the cognitive load experienced at different points in the source code. Self-reported task difficult appears to be influenced by participants' confidence on their performance.

## 5 Discussion

Previous work used fNIRS to measure cognitive load during program comprehension tasks [*Ikutani and Uwano*, 2014; *Nakagawa et al.*, 2014] . In this work, we confirm the feasibility of using fNIRS as a tool for measuring cognitive load in the context of bug localization tasks performed by developers. Furthermore, we establish the feasibility of combining fNIRS and eyetracking data to allow analysis at a very fine level of granularity within the source code, i.e., program identifiers. Thus, we establish a methodology and a framework that allows future research to

Table 8: Kendall Correlation coefficient between Self-Reported (SR), Cognitive Load (Oxy), and Eye Tracking Fixation Duration (ET)

| Treatment | SR & Oxy | SR & ET | Oxy & ET |
|---|---|---|---|
| Control | 0.069 | 0.52 | 0.15 |
| LA | -0.17 | 0.028 | -0.23 |
| Structural | -0.007 | 0.32 | 0.09 |
| LA & Structural | -0.009 | 0.61 | 0.1 |

tackle new research questions in the domain of software engineering that pertain to program comprehension in a less restrictive environment compared to fMRI. The implication of this methodology on the software engineering research community is not only the availability of empirical and objective evaluations of cognitive load, but also the opportunity to compare and contrast cognitive load, self reported, and various psycho-physiological metrics that capture different aspects of program comprehension.

At a high level, our results show that inconsistencies in the source code have significant effects on the cognitive load, success, and time spent to comprehend source code. We provide empirical evidence that confirm the importance of clean code, both in terms of its lexical and structural aspects.

Our results also show that three metrics for comprehension, namely: fixation duration, cognitive load, and self reported task difficulty, show very little correlation. Future work should identify in which context each of those metrics must be used and whether a combination of them is needed to fully understand different aspects of program comprehension.

Future studies need to design experiments with developers with varying experience levels, as experience might play a significant role in the amount of cognitive load that developers experience during a program comprehension task. This will allow us to answer questions about how different inconsistencies or code smells in the source code effect different groups. For example, developers with more experience could have certain expectations about how source code should look, which may cause them to be more confused than novices with no expectations when such inconsistencies are encountered. On the other hand, we might also be able to identify which factors are the most detrimental to novice programmers, which will enable a better understanding about how to effectively educate students.

## 6 Threats to Validity

This section discusses the threats to validity that can affect our study. A common classification [*Wohlin et al.*, 2000; *Yin*, 1994] 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. This reduces the heterogeneity, however participants have a diverse range of experience. We plan to investigate this in our future work. 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.

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 18 out of 25 participants consider that they are more proficient in C++ and 5 participants have no previous experience in Java. This might cause an increase in 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. Similarly, the order in which participants complete tasks could affect the results of our study. Participants could potentially find later tasks easier to complete as they learn from earlier tasks. To mitigate this threat, we randomly shuffle the order the code snippets and treatment types received by 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. Studies have shown that there exist differences in cortical oxygenation between young individuals (age below 50) as compared to elderly individuals (age above 50) [*Ehlis et al.*, 2014]. However, by design, all of the participants are students, with the majority of them working towards a bachelor's degree. We did not collect their age but all of them are definitely below 50. Thus, our study is not impacted by this threat. If the study was to be replicated with professional developers, age would be a confounding factor that needs to be considered.

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 subjectivity 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 [*Salman et al.*, 2015]. 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 [*Fakhoury*, 2018]. Moreover, we are currently working on publishing the extension of iTrace and our visualization tool online.

## 7 Related Work

### 7.1 Proxies of Source Code Comprehensibility

Several research studies investigate the relationship between low quality code, characterized by code smells and antipatterns, and proxies for code maintainability, comprehensibility, and readability.

For example, *Butler et al.* [2009] evaluated the quality of identifier names in 8 established open source Java application libraries using a set of 12 identifier naming guidelines. They found statistically significant associations between flawed identifiers and code quality issues as reported by FindBugs, a static analysis tool.

*Khomh et al.* [2012] investigated the relationships between classes affected by antipatterns and their susceptibility to changes, issues, and unhandled exceptions in 13 releases of the Eclipse project. They found that these classes were negatively impacted on all three aspects i.e., they were changed more often, and were more susceptible to both issues and throwing unhandled exceptions. They concluded that the cost of ownership of systems containing such classes would be higher due to the time spent repairing issues caused by these classes. Similarly, *Jaafar et al.* [2013] analyzed static and temporal relationships between classes containing antipatterns and those that do not, from three open source projects. They found that in all releases of the projects, classes having static relationships with antipatterns and those that co-changed with antipatterns were more fault prone than those that didn't have these relationships.

More recently, *Aghajani et al.* [2018] investigated the effects of linguistic antipatterns on a large scale dataset of 1.6k releases of Maven libraries, 14k open source Java projects using these libraries and 4.4k questions related to the use of these libraries on StackOverflow. They found that statistically, linguistic antipatterns had some effect on the likelihood of introducing bugs (29% higher) and of triggering StackOverflow questions, although their qualitative analysis did not allow them to reveal any explanation for the phenomenon. Their findings call for additional controlled experiments to allow for a better isolation of the effect of linguistic antipatterns.

## 7.2 Psycho-Physiological Measures in Software Engineering

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.* [2014] 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 and Fritz* [2016] conducted a study using heart rate variability and EDA. 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.* [2017] 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.* [2014] 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* [2014], 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.* [2014] 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.* [2017] 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. Most recently, *Peitek et al.* [2018] have begun exploring the use of eye tracking and fMRI together, to map brain activation to specific areas in the source code. Results show that it is feasible to use eye tracking in fMRI machines, however, there are considerable advancements that still need to be made to improve the reliability and accuracy of these tools. *Floyd et al.* [2017] 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.

FMRI machines are able to capture brain activity deep within the brain and across all regions of the brain, whereas most fNIRs models only capture activity in certain lobes, and closer to the surface of the brain. This means that fMRI may be able to capture brain activation relevant to software engineering tasks that the fNIRs cannot. *Duraes et al.* [2016] used fMRI to observe 13 professional developers as they solved a bug fixing task and confirmed that brain areas associated with language processing and mathematics were highly active during the code review process, and were able to identify activity in the anterior insula region positively correlated to the precision of the bug fixing. *Castelhano et al.* [2018] studied professional developers with high expertise and confirmed the importance of the insula region in source code comprehension, bug detection and decision-making.

However, although fMRI does 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 restrictive 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.

## 8 Conclusion

This work provides a methodology to measure developers' cognitive load at a low level of granularity, i.e., terms composing program identifiers. To validate our methodology we conduct a study with 25 participants to measure the effect of lexical and structural inconsistencies on their cognitive load while performing bug localization tasks. Our findings show that 1) the methodology is accurate, 2) the studied inconsistencies have a negative impact on developers' cognitive load and their performance (in terms of time and success rate), and 3) developers' cognitive load as measured here using fNIRS and Eye Tracking devices captures a different aspect of task difficulty compared to self-reported measures and fixation duration.

As part of our future work, we plan to explore how structural and linguistic inconsistencies effect novice and professional developers during software engineering tasks. We hope to identify factors that are most detrimental to program comprehension for novices, so that we can learn how to effectively educate students. Moreover, we plan to further investigate the type of information captured by different types of metrics that characterize task difficulty to allow for more accurate task difficulty prediction.

**Acknowledgements** This work is supported by the NSF (award number CCF-1755995). The authors thank Thom Hemenway, Keon Sadatian, Nehemiah Salo, and Kyle Tilton for their help in developing tools for the environment in which we conducted the experiment. We also thank all students that participated in the experiment for their time and effort.

## References

Abebe, S. L., V. Arnaoudova, P. Tonella, G. Antoniol, and Y. G. Guéhéneuc, Can lexicon bad smells improve fault prediction?, in *Proceedings of the Working Conference on Reverse Engineering (WCRE)*, pp. 235–244, 2012.

Afergan, D., E. M. Peck, E. T. Solovey, A. Jenkins, S. W. Hincks, E. T. Brown, R. Chang, and R. J. Jacob, Dynamic difficulty using brain metrics of workload, in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pp. 3797–3806, ACM, 2014.

Aghajani, E., C. Nagy, G. Bavota, and M. Lanza, A large-scale empirical study on linguistic antipatterns affecting apis, in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 25–35, IEEE, 2018.

Arnaoudova, V., M. Di Penta, G. Antoniol, and Y.-G. Guéhéneuc, A new family of software anti-patterns: Linguistic anti-patterns, in *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*, pp. 187–196, 2013.

Arnaoudova, V., M. Di Penta, and G. Antoniol, Linguistic antipatterns: What they are and how developers perceive them, *Empirical Software Engineering (EMSE)*, *21*(1), 104–158, 2016.

Baker, W. B., A. B. Parthasarathy, D. R. Busch, R. C. Mesquita, J. H. Greenberg, and A. Yodh, Modified beer-lambert law for blood flow, *Biomedical optics express*, *5*(11), 4053–4075, 2014.

Binkley, D., M. Davis, D. Lawrie, and C. Morrell, To CamelCase or Under_score, in *Proceedings of the International Conference on Program Comprehension (ICPC)*, pp. 158–167, 2009a.

Binkley, D., D. Lawrie, S. Maex, and C. Morrell, Identifier length and limited programmer memory, *Science of Computer Programming*, *74*(7), 430–445, 2009b.

Binkley, D., M. Davis, D. Lawrie, J. I. Maletic, C. Morrell, and B. Sharif, The impact of identifier style on effort and comprehension, *Empirical Software Engineering (EMSE)*, *18*(2), 219–276, 2013.

BIOPAC, Biopac homepage, `https://www.biopac.com`, 2018a.

BIOPAC, fnirsoft user manual, `https://www.biopac.com/wp-content/uploads/fnirsoft-user-manual.pdf`, 2018b.

Blackwell, A. F., Metaphors we program by: Space, action and society in java., in *PPIG*, p. 8, 2006.

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

Butler, S., M. Wermelinger, Y. Yu, and H. Sharp, Relating identifier naming flaws and code quality: An empirical study, in *2009 16th Working Conference on Reverse Engineering*, pp. 31–35, IEEE, 2009.

Castelhano, J., I. C. Duarte, C. Ferreira, J. Duraes, H. Madeira, and M. Castelo-Branco, The role of the insula in intuitive expert bug detection in computer code: an fmri study, *Brain imaging and behavior*, pp. 1–15, 2018.

Causse, M., Z. Chua, V. Peysakhovich, N. Del Campo, and N. Matton, Mental workload and neural efficiency quantified in the prefrontal cortex using fnirs, *Scientific reports*, *7*(1), 5222, 2017.

Deissenboeck, F., and M. Pizka, Concise and consistent naming, *Software Quality Journal*, *14*(3), 261–282, 2006.

Delpy, D. T., M. Cope, P. van der Zee, S. Arridge, S. Wray, and J. Wyatt, Estimation of optical pathlength through tissue from direct time of flight measurement, *Physics in Medicine & Biology*, *33*(12), 1433, 1988.

Duraes, J., H. Madeira, J. Castelhano, C. Duarte, and M. C. Branco, Wap: Understanding the brain at software debugging, in *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*, pp. 87–92, IEEE, 2016.

Eclipse, Eclipse ide, `https://www.eclipse.org/ide`, 2018.

Ehlis, A.-C., S. Schneider, T. Dresler, and A. J. Fallgatter, Application of functional near-infrared spectroscopy in psychiatry, *Neuroimage*, *85*, 478–488, 2014.

EyeTribe, The eye tribe homepage, `https://theeyetribe.com`, 2018.

Fakhoury, S., Online replication package, `https://github.com/smfakhoury/fNIRS-and-Cognitive-Load`, 2018.

Fakhoury, S., Y. Ma, V. Arnaoudova, and O. Adesope, The effect of poor source code lexicon and readability on developers' cognitive load, 2018.

Fishburn, F. A., M. E. Norr, A. V. Medvedev, and C. J. Vaidya, Sensitivity of fnirs to cognitive state and load, *Frontiers in human neuroscience*, *8*, 76, 2014.

Floyd, B., T. Santander, and W. Weimer, 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)*, pp. 175–186, 2017.

Fritz, T., A. Begel, S. C. Müller, S. Yigit-Elliott, and M. Züger, Using psychophysiological measures to assess task difficulty in software development, in *Proceedings of the International Conference on Software Engineering (ICSE)*, pp. 402–413, 2014.

Girouard, A., E. T. Solovey, L. M. Hirshfield, K. Chauncey, A. Sassaroli, S. Fantini, and R. J. Jacob, Distinguishing difficulty levels with non-invasive brain activity measurements, in *IFIP Conference on Human-Computer Interaction*, pp. 440–452, Springer, 2009.

Grissom, R. J., and J. J. Kim, *Effect sizes for research: A broad practical approach*, 2nd edition ed., Lawrence Earlbaum Associates, 2005.

Halstead, M. H., Elements of software science, 1977.

Herff, C., D. Heger, O. Fortmann, J. Hennrich, F. Putze, and T. Schultz, Mental workload during n-back task-quantified in the prefrontal cortex using fnirs, *Frontiers in Human Neuroscience*, *7*, 935, 2014.

Hochstein, L., V. R. Basili, M. V. Zelkowitz, J. K. Hollingsworth, and J. Carver, Combining self-reported and automatic data to improve programming effort measurement, *SIGSOFT Softw. Eng. Notes*, *30*(5), 356–365, 2005.

Ikutani, Y., and H. Uwano, 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)*, pp. 1–6, 2014.

Jaafar, F., Y.-G. Guéhéneuc, S. Hamel, and F. Khomh, Mining the relationship between anti-patterns dependencies and fault-proneness, in *2013 20th Working Conference on Reverse Engineering (WCRE)*, pp. 351–360, IEEE, 2013.

Khomh, F., M. D. Penta, Y.-G. Guéhéneuc, and G. Antoniol, An exploratory study of the impact of antipatterns on class change- and fault-proneness, *Empirical Softw. Engg.*, *17*(3), 243–275, 2012.

Kruggel, F., and D. Y. von Cramon, Temporal properties of the hemodynamic response in functional mri, *Human brain mapping*, *8*(4), 259–271, 1999.

Lawrie, D., C. Morrell, H. Feild, and D. Binkley, What's in a name? A study of identifiers, in *Proceedings of International Conference on Program Comprehension (ICPC)*, pp. 3–12, 2006.

Lee, S., D. Hooshyar, H. Ji, K. Nam, and H. Lim, Mining biometric data to predict programmer expertise and task difficulty, *Cluster Computing*, pp. 1–11, 2017.

Liblit, B., A. Begel, and E. Sweetser, Cognitive perspectives on the role of naming in computer programs., in *PPIG*, p. 11, Citeseer, 2006.

Marcus, A., D. Poshyvanyk, and R. Ferenc, Using the conceptual cohesion of classes for fault prediction in object-oriented systems, *IEEE Transactions on Software Engineering (TSE)*, *34*(2), 287–30, 2008.

McCabe, T. J., A complexity measure, *IEEE Transactions on Software Engineering (TSE)*, *SE-2*(4), 308–320, 1976.

Müller, S. C., and T. Fritz, Using (bio)metrics to predict code quality online, in *Proceedings of the International Conference on Software Engineering (ICSE)*, pp. 452–463, 2016.

Nakagawa, T., Y. Kamei, H. Uwano, A. Monden, K. Matsumoto, and D. M. German, 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)*, pp. 448–451, 2014.

Ooms, K., L. Dupont, L. Lapon, and S. Popelka, 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.

Peitek, N., J. Siegmund, C. Parnin, S. Apel, J. Hofmeister, and A. Brechmann, Simultaneous measurement of program comprehension with fmri and eye tracking: a case study, 2018.

Poshyvanyk, D., Y.-G. Guéhéneuc, A. Marcus, G. Antoniol, and V. Rajlich, Combining probabilistic ranking and latent semantic indexing for feature identification, in *Proceedings of the International Conference on Program Comprehension (ICPC)*, pp. 137–148, 2006.

Posnett, D., A. Hindle, and P. Devanbu, A simpler model of software readability, in *Proceedings of the Working Conference on Mining Software Repositories (MSR)*, pp. 73–82, 2011.

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

Salman, I., A. T. Misirli, and N. Juristo, Are students representatives of professionals in software engineering experiments?, in *Proceedings of the International Conference on Software Engineering (ICSE)*, pp. 666–676, 2015.

Scalabrino, S., M. Linares-Vasquez, D. Poshyvanyk, and R. Oliveto, Improving code readability models with textual features, in *Proceedings of the International Conference on Program Comprehension (ICPC)*, pp. 1–10, 2016.

Scanniello, G., and M. Risi, Dealing with faults in source code: Abbreviated vs. full-word identifier names, in *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, pp. 190–199, IEEE, 2013.

Shaffer, T. R., J. L. Wise, B. M. Walters, S. C. Müller, M. Falcone, and B. Sharif, 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)*, pp. 954–957, 2015.

Sharafi, Z., Z. Soh, Y.-G. Guhneuc, and G. Antoniol, Women and men  different but equal: On the impact of identifier style on source code reading, in *IEEE International Conference on Program Comprehension*, pp. 27–36, 2012.

Sharafi, Z., T. Shaffer, B. Sharif, and Y.-G. Guéhéneuc, Eye-tracking metrics in software engineering, in *2015 Asia-Pacific Software Engineering Conference (APSEC)*, pp. 96–103, IEEE, 2015a.

Sharafi, Z., Z. Soh, and Y.-G. Guéhéneuc, A systematic literature review on the usage of eye-tracking in software engineering, *Information and Software Technology*, *67*, 79–107, 2015b.

Sharif, B., M. Falcone, and J. I. Maletic, An eye-tracking study on the role of scan time in finding source code defects, *In Proceedings of the Symposium on Eye Tracking Research and Applications (ETRA)*, pp. 381–384, 2012.

Siegmund, J., C. Kästner, S. Apel, C. Parnin, A. Bethmann, T. Leich, G. Saake, and A. Brechmann, Understanding understanding source code with functional magnetic resonance imaging, in *Proceedings of the International Conference on Software Engineering (ICSE)*, pp. 378–389, 2014.

Siegmund, J., N. Peitek, C. Parnin, S. Apel, J. Hofmeister, C. Kästner, A. Begel, A. Bethmann, and A. Brechmann, Measuring neural efficiency of program comprehension, in *Proceedings of the Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, pp. 140–150, 2017.

Sokal, R. R., A statistical method for evaluating systematic relationship, *University of Kansas science bulletin*, *28*, 1409–1438, 1958.

Takang, A. A., P. A. Grubb, and R. D. Macredie, The effects of comments and identifier names on program comprehensibility: an experimental investigation, *J. Prog. Lang.*, *4*(3), 143–167, 1996.

Treacy Solovey, E., D. Afergan, E. M. Peck, S. W. Hincks, and R. J. K. Jacob, Designing Implicit Interfaces for Physiological Computing, *ACM Transactions on Computer-Human Interaction*, *21*(6), 1–27, 2015.

Wohlin, C., P. Runeson, H. Martin, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering - An Introduction*, Kluwer Academic Publishers, 2000.

Yin, R. K., *Case Study Research: Design and Methods*, 2nd ed., Sage Publications, 1994.

**Sarah Fakhoury** is a Ph.D. candidate at Washington State University, School of Computer Science and Electrical Engineering. She performs research in the Software Engineering Lab, in conjunction with the NeuroCognitive Science Lab under the supervision of Dr. Venera Arnaoudova. In 2016, Sarah received her B.S. in computer science from the American University of Beirut. Her research is focused on improving the way developers write, comprehend, and review source code. She is currently exploring the use of eyetracking and brain imaging technology as measures of program comprehension and is specifically interested in how various aspects of the source code lexicon and readability affect developers cognitive load during software engineering tasks. Sarah has received a Distinguished Paper Award and served as reviewer and an external reviewer for multiple IEEE and ACM conferences. More information at https://www.sarahfakhoury.com.

**Devjeet Roy** Devjeet is a Masters student at Washington State University. His advisor is Dr. Venera Arnaoudova. His research interests include empiricial software engineering, program comprehension and software testing.

**Yuzhan Ma** joined Amazon.com Services, Inc. in 2018 as a Software Development Engineer. She primarily builds, maintains, and vends base map data, road network data, and map tiles for Amazon delivery service. Yuzhan received her Master of Science degree in Computer Science from Washington State University, School of Computer Science and Electrical Engineering. During her study, she performed research in the Software Engineering Lab under the supervision of Dr. Venera Arnaoudova. Yuzhan's research interests include Empirical Software Engineering, Software Quality and Maintenance, Information Retrieval, and Natural Language Processing.

**Venera Arnaoudova**is an Assistant Professor at Washington State University. She received her Ph.D. degree in 2014 from Polytechnique Montral under the supervision of Dr. Giuliano Antoniol and Dr. Yann-Gal Guhneuc. Her research interest is in the domain of software evolution and particularly, empirical software engineering, program comprehension and the analysis of source code lexicon and documentation. Arnaoudova has published in several international SE conferences and journals. She received two Distinguished Paper Awards and a Distinguished Reviewer Award. She is a Digital Learning Co-Chair for ACM SIGSOFT. She is part of the review board of the Empirical Software Engineering Journal (EMSE) and IEEE Transactions on Software Engineering (TSE). She has served as a program committee member for ICPC, ICSME, MSR, SANER. More information available at http://www.veneraarnaoudova.com.

**Olusola O. Adesope**is a Boeing Distinguished Professor of STEM Education and a Professor of Educational Psychology in the College of Education at Washington State University-Pullman. His research is positioned at the intersection of educational psychology, STEM education, learning sciences, and instructional design and technology. Dr. Adesopes current research focuses on the use of systematic reviews and meta-analyses for evidence-based practices, cognitive and pedagogical underpinnings of learning with computer-based multimedia resources, and investigation of instructional principles and assessments in STEM education. Dr. Adesopes research is mostly funded by the National Science Foundation and published in top peer-reviewed journals. Dr. Adesope has over 100 published journal papers, book chapters and proceedings. He is a recipient of several awards including the American Educational Research Associations early career research award and Washington State University's College of Education ”Excellence in Research” award. His meta-analysis of practice testing published in the Review of Educational Research was rated as one of the top 10 most read education research articles of 2017.