Moving towards Objective Measures of Program Comprehension

Sarah Fakhoury
Washington State University, USA
School of Electrical Engineering and Computer Science
sarah.fakhoury@wsu.edu

ABSTRACT

Traditionally, program comprehension research relies heavily on 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. However, there are several potential issues that can result from using these indirect measures because they are prone to participant biases and implicitly deduce comprehension based on various factors.

The proposed research presents a framework to move towards more objective measures of program comprehension through the use of brain imaging and eye tracking technology. We aim to shed light on how the human brain processes comprehension tasks, specifically what aspects of the source code cause measurable increases in the cognitive load of developers in both bug localization tasks, as well as code reviews. We discuss the proposed methodology, preliminary results, and overall contributions of the work.

CCS CONCEPTS

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

KEYWORDS

Program Comprehension, Cognitive Load, Source Code Lexicon, fNIRS, Eyetracking, Biometrics

ACM Reference Format:
Sarah Fakhoury. 2018. Moving towards Objective Measures of Program Comprehension. In *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '18), November 4–9, 2018, Lake Buena Vista, FL, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3236024.3275426

1 THE RESEARCH PROBLEM

The process of program comprehension is a time-intensive and fundamental activity for every developer during the software development life cycle [22][23][24]. Thus, over the past few decades, program comprehension has been studied extensively by researchers who aim to understand more about how developers comprehend source code, and look for ways to improve the process.

ESEC/FSE '18, November 4–9, 2018, Lake Buena Vista, FL, USA

ACM ISBN 978-1-4503-5573-5/18/11…$15.00
https://doi.org/10.1145/3236024.3275426

Research has shown than 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. There is also evidence of a correlation between the quality of identifiers and the quality of a software project [1, 5–7, 15, 17].

However, until recently, understanding the role of source code lexicon on developers' comprehension involved measuring the internal cognitive process using indirect measures of comprehension, such as self-reported verification or summarization tasks designed to implicitly deduce comprehension. For example, Binkley et al. [4] studied the impact of identifier style on code readability. The authors recorded the time and accuracy of participants during comprehension tasks to deduce which identifier style written leads to more accurate results. Lawrie et al. [13] 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.

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. [14] 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. Fritz et al. [11] combined EEG, eye tracking, and electro dermal activity (EDA) to investigate task difficulty during code comprehension.

Despite considerable advancement in biometric research in recent years, very little is known about how the human brain processes program comprehension tasks. Recent efforts to investigate this involve the use of functional magnetic resonance imaging (fMRI). For example Siegmund et al. conduct a study involving the use of fMRI to study program comprehension in the brain [20] and to understand the cognitive processes related to bottom-up and top-down comprehension strategies [21]. Similarly, Floyd et al. use fMRI to compare areas of brain activation between source code and natural language tasks [10]. Despite the success of fMRI studies in the domain, fMRI machines remain a costly and invasive approach, with which it is hard to reproduce the real life working conditions of software developers.

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 [9] as both rely on blood-oxygen-level dependent

(BOLD) response and show highly correlated results for cognitive tasks. The low cost and minimally invasive 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. [16] 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 [12].

Our research aims to improve on the existing research by relating fNIRS data to specific terms in the source code in real time through the use of modern eye tracking devices. This allows researchers to pinpoint problematic elements within the source code at a very fine level of granularity. 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.

By using technology that can provide direct and objective measures of comprehension we aim to understand how human factors impact the cost and quality of the software they develop and define guidelines and tools needed to help those developers create software that both requires less mental effort to be understood, and is easier to maintain.

The rest of the paper is organized as follows: Section 2 discusses the overall goals of our research and how we plan to contribute to the existing body of program comprehension research throughout the thesis. In section 3 we propose a methodology to answer the research problems presented in this section. Section 4 presents the preliminary results we have obtained for a subset the research questions proposed in section 3.

2 EXPECTED CONTRIBUTIONS

The goal of this research is to establish fNIRS and eyetracking as objective measures of program comprehension in software engineering research as a means to improve software development practices.

We establish an fNIRS study framework with which we can tackle advanced research questions and allow fNIRS to be adopted as a practical measure for program comprehension research. We expect that by doing this we can provide a methodology to facilitate the acceptance of fNIRS and eyetracking as objective measures of comprehension in the research community. Once we have assessed the suitability of the technology, we obtain objective evidence of the impact of poor quality lexicon and readability on the cognitive load of developers during bug localization tasks.

We also expect to shed light on difficult aspects of code reviews and how lexical, readability, and design antipatterns affect developers cognitive load during code review tasks. Finally, we expect to uncover a deeper understand on the aspects of source code that less experienced developers find more challenging as compared to more experienced developers.

Ultimately, our contributions will be consolidated and presented to the research community in the form of guidelines and recommendations for software naming and documentation based on the insights gained from our investigation of the neuro-cognitive perspective of program comprehension.

3 METHODOLOGY

In order to explore the effects of various aspects of the source code on developers' cognitive load, we break down our methodology into four main steps. The first step is to assess whether fNIRS and eyetracking technology can be established together as suitable measurements of comprehension, which to the best of our knowledge has never been explored before. We then investigate the role of the source code lexicon, source code readability and developer experience on program comprehension during bug localization tasks. Next, we aim to uncover difficult aspects of the code review process using different lexical, structural, readability and design metrics within the source code. Finally, using the insights gained through the studies conducted we will develop new software naming and documentation guidelines.

3.1 Using fNIRS and Eyetracking as Objective Measures of Comprehension

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

Previous work by Nakagawa et al. [16] and Ikutani et al. [12] map fNIRS data to overall task difficulty or specific methods. We improve upon this by mapping fNIRS data to an identifier level of granularity. Our primary goal is to determine if fNIRS and eye tracking devices can be used to successfully together to capture high cognitive load within text or source code, at a word level of granularity. We develop a systematic methodology to relate cognitive load data to eyetracking data over the course of a comprehension task, as well as, an approach to characterize high cognitive load across participants. Measuring cognitive load at word level of granularity allows us to make a distinction between parts of the source code that are understood and those that are confusing to the developer, which cannot be achieved if we only consider the cognitive load over an entire source code snippet.

In order to achieve this we first modify the iTrace plugin [19], which is a plugin for Eclipse that interfaces with an eyetracker to determine what source code elements a participant is looking at. We extend the iTrace plugin to identify source code elements at identifier level. Next, we come up with a systematic methodology to synchronize fNIRS and eyetracking data, by mapping fixation data points from the eyetracker to consolidated data points from the fNIRS device using system time as our reference point. We also create an algorithm to determine fixations that contain high cognitive load using average normalized blood oxygenation values over an entire snippet. Initial thresholds indicating high cognitive load will be determined with developer input, in the form of highlighted areas of code that the developers found difficult to understand.

3.2 The Impact of Inconsistencies on Cognitive Load During Bug Localization Tasks

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

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

Previous work by Arnaoudova et al. [2] investigates the perception of Linguistic Antipatterns, defined as recurring poor practices in the naming, documentation, and choice of identifiers in the implementation of program entities, on internal and external developers. They find that the majority of developers perceive these inconsistencies as poor practices and should be avoided. Similarly, 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 [5][18]. Inspired by these works we will empirically investigate the effect of poor source code lexicon and poor readability on developers cognitive load.

In order to investigate both the impact of poor source code lexicon and readability on developer's cognitive load we will conduct an empirical study in which developers are asked to perform bug localization tasks on source code snippets that have been altered to either contain linguistic antipatterns, poor readability, or both. Using fNIRS and eyetracking technology we will be able to map instances of high cognitive load of developers to identifiers in the source code throughout the tasks. We will then be able to compare this data about areas of code that developers found difficult to that of control tasks that do not contain poor source code lexicon or readability and draw conclusions about the effects that both treatments have on cognitive load.

3.3 The Impact of Inconsistencies on Cognitive Load During Code Review Tasks

RQ4: What aspects of the code review process cause an increase in the cognitive load of developers?

RQ5: Do inconsistencies in the source code lexicon cause an increase in the cognitive load of developers during the code review process?

RQ6: Do structural inconsistencies related to the readability of the source code cause an increase in the cognitive load of developers during the code review process?

RQ7: Do design anti-patterns in the source code cause an increase in the cognitive load of developers during the code review process?

Code reviews are conducted to improve several aspects of software maintenance and development. Work by Bacchelli and Bird [3] suggests that one of the main motivations of modern code reviews is to improve the quality of a change to the software. During code reviews developers assess various aspects of the source code, including the source code lexicon and it's overall readability. By investigating developers' cognitive load during code review tasks we are able to gain a more in depth idea of what aspects of the source code developers are looking at and what aspects of the code review processes are most difficult. This augments insights learned from answering **RQ2** and **RQ3** because developers are now tasked with reviewing the quality of an entire source code snippet, as opposed to only finding a bug in the source code.

To answer the above research questions we will ask participants to review code snippets and use fNIRS and eye tracking devices to capture data during code review tasks. We will analyze collected data and determine which aspects of the code review process incurred the highest amount of cognitive load. We will then investigate various source code metrics of the reviewed code snippets to determine which aspects may have contributed to increased cognitive load. The code review tasks will consist of reading various source code snippets, determining if the code is appropriate, and leaving comments on the code for the original authors.

3.4 Role of Experience on Cognitive Load During Program Comprehension Tasks

RQ8: Does developer experience impact the cognitive load of developers during program comprehension tasks?

Developer experience is a significant factor in the amount of cognitive load experienced during a program comprehension task. Using data from the empirical studies conducted in sections 3.2 and 3.3, we aim to take a closer look at how developer experience affects cognitive load by analyzing the data collected from groups of expert and novice programmers. We will observe the impact of linguistic antipatterns and poor readability on both groups, as well as determine what aspects of the code review processes were more difficult to one group as compared to another. This will allow us to answer questions about what factors are the most detrimental to novice programmers and enable a better understanding about how to more effectively educate students.

3.5 Improving Existing Software Naming and Documentation Guidelines

Using insights gained from answering the research questions presented above, we will augment existing software naming and documentation guidelines by developing new antipatterns and cataloging a list of practices that have been shown to increase cogntive load in developers. Our ultimate goal is to learn how to best help educate inexperienced students on how to write source code that is easy to understand and does not cause unneeded increase in cognitive load by other developers who maintain their code.

4 CURRENT RESULTS

We have obtained preliminary results relating to **RQ1, RQ2, and RQ3** as defined in Section 3. To answer our research questions we conduct an empirical study where we include different tasks, such as a comprehension task and a bug localization task. Figure 1 illustrates the experimental procedure used. The comprehension task is used to answer **RQ1** and determine whether developer's cognitive load can be accurately associated to identifier terms. After the comprehension task, participants are asked to carefully highlight areas of the code that were difficult or took a lot of time to understand. This data is used as a means to evaluate the accuracy of our results. We find that using fNIRS and eyetracking devices, developers' cognitive load can be accurately associated with identifiers in source code and text, with a similarity of 78% compared to self-reported high cognitive load.

Next, as described in Section 3.2, we answer **RQ2 and RQ3** through a bug localization task. Participants perform the task on source code that has been altered to either contain poor lexicon or poor readability. Follow up questions and a post analysis survey are used for qualitative analysis of our results. In summary, Results show that the existence of linguistic antipatterns in the source

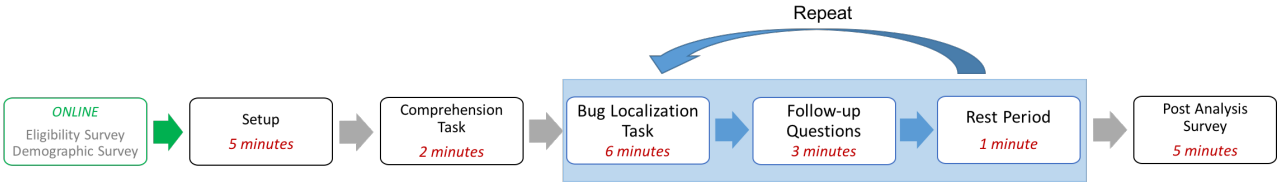

Figure 1: Overview of the experimental procedure.

code significantly increases the cognitive load experienced by participants. Additionally, we are able to pinpoint instances of high cognitive load over identifiers containing linguistic antipatterns in multiple participants.

However, we do not find any evidence to conclude that cognitive load is increased for snippets that contain poor structural and readability characteristics. When a snippet contains both poor readability and linguistic antipatterns, program comprehension is significantly impacted and 60% of participants are unable to complete the task successfully. Although we do not observe an increase in cognitive load over the treatment snippet for those participants, we do observe and increase in the remaining 40% of participants who do complete the tasks successfully. More details on the empirical study and results may be found in our paper [8].

5 CONCLUSION

Through the proposed research presented in this paper, our goals are twofold, first we aim to encourage the research community to adopt more objective measures of program comprehension as a way of more accurately capturing the nature of developer comprehension in research studies and gaining a deeper understanding of how the human brain processes comprehension tasks. Second, we aim to improve the current guidelines for the naming and documentation of source code using evidence from these objective measures about what aspects of source code cause significant increases in the cognitive load of developer's trying to understand them.

ACKNOWLEDGMENTS

I would like to thank my advisor, Dr. Venera Arnaoudova, for her invaluable feedback and constant support. This work is supported by the NSF (award number CCF-1755995).

REFERENCES

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

[2] Venera Arnaoudova, Massimiliano Di Penta, and Giuliano Antoniol. 2016. Linguistic antipatterns: what they are and how developers perceive them. *Empirical Software Engineering* 21, 1 (2016), 104–158.

[3] Alberto Bacchelli and Christian Bird. 2013. Expectations, Outcomes, and Challenges of Modern Code Review. In *Proceedings of the International Conference on Software Engineering (ICSE)*. 712–721.

[4] David Binkley, Marcia Davis, Dawn Lawrie, and Christopher Morrell. 2009. To CamelCase or Under_score. In *Proceedings of the International Conference on Program Comprehension (ICPC)*. 158–167.

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

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

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

[8] Sarah Fakhoury, Yuzhan Ma, Venera Arnaoudova, and Olusola Adesope. 2018. The Effect of Poor Source Code Lexicon and Readability on Developers' Cognitive Load. *IEEE International Conference on Program Comprehension (ICPC)* (2018).

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

[10] Benjamin Floyd, Tyler Santander, and Westley Weimer. 2017. Decoding the Representation of Code in the Brain: An fMRI Study of Code Review and Expertise. In *Proceedings of the International Conference on Software Engineering (ICSE)*. 175–186.

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

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

[13] Dawn Lawrie, Christopher Morrell, Henry Feild, and David Binkley. 2006. What's in a Name? A Study of Identifiers. In *Proceedings of International Conference on Program Comprehension (ICPC)*. 3–12.

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

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

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

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

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

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

[20] Janet Siegmund, Christian Kästner, Sven Apel, Chris Parnin, Anja Bethmann, Thomas Leich, Gunter Saake, and André Brechmann. 2014. Understanding understanding source code with functional magnetic resonance imaging. In *Proceedings of the International Conference on Software Engineering (ICSE)*. 378–389.

[21] Janet Siegmund, Norman Peitek, Chris Parnin, Sven Apel, Johannes Hofmeister, Christian Kästner, Andrew Begel, Anja Bethmann, and André Brechmann. 2017. Measuring neural efficiency of program comprehension. In *Proceedings of the Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. 140–150.

[22] Thomas A. Standish. 1984. An Essay on Software Reuse. *IEEE Transactions on Software Engineering (TSE)* 10, 5 (September 1984), 494–497.

[23] Rebecca Tiarks. 2011. What Maintenance Programmers Really Do: An Observational Study. In *Proceedings of the Workshop Software Reengineering (WSR)*. 36–37.

[24] A. von Mayrhauser and A.M. Vans. 1995. Program comprehension during software maintenance and evolution. *IEEE Computer* 28, 8 (August 1995), 44–55.