Eye Movements in Programming: Models to Data

Proceedings of the Third International Workshop

ROMAN BEDNARIK, TERESA BUSJAHN, CARSTEN SCHULTE, SASCHA TAMM

Eye Movements in Programming: Models to Data

Proceedings of the Third International Workshop

Publications of the University of Eastern Finland
Reports and Studies in Forestry and Natural Sciences
No 23

University of Eastern Finland
Faculty of Science and Forestry, School of Computing
Joensuu, Finland
2016

Grano Oy
Joensuu, 2016
Editor Prof. Pertti Pasanen, Prof. Pekka Kilpeläinen,
Prof. Kai Peiponen, Prof. Matti Vornanen
Julkaisujen myynti
Itä-Suomen yliopiston kirjasto(julkaisumyynti@uef.fi)
ja tiedekirjakauppa Granum
ISSN (nid): 978-952-61-2039-3
ISBN (nid): 1798-5684
ISSN-L: 1798-5684
ISSN (PDF): 1798-5692
ISBN (PDF): 978-952-61-2040-9

Bednarik, Roman; Busjahn, Teresa; Schulte, Carsten; Tamm, Sascha (Eds.)
Eye Movements in Programming: Models to Data. Proceedings of the Third International Workshop
Itä-Suomen yliopisto, School of Computing, 2016
Publications of the University of Eastern Finland. Reports and Studies in Forestry and Natural Sciences, no 23
ISSN (nid): 978-952-61-2039-3
ISBN (nid): 1798-5684
ISSN-L: 1798-5684
ISSN (PDF): 1798-5692
ISBN (PDF): 978-952-61-2040-9

Welcome to the proceedings of the third international workshop on "Eye Movements in Programming: From Models to Data"

While it has been for some time clear that eye movements can provide important insights into the processes involved in computer programming, the previous efforts were isolated and not systematic. With that mindset, we established a series of workshops to bring together researchers interested in various aspects of eye movements in programming.

The third international workshop on Eye Movements in Programming was held at the University of Eastern Finland, Joensuu, November 22-24, 2015. It focused on advancing the methodological, theoretical and applied aspects of eye movements in programming. The event had 20 participants and was kindly supported by SensoMotoric Instruments and UEF's Joensuu University Foundation.

Previously we focused on understanding expert (EMIP'2013) and novice (EMIP'2014) source code reading behaviors, and we launched the first distributed data collection in this discipline. By analyzing the pre-shared datasets and discussion of bottom-up data-to-models approaches, the workshops produced new methods, tools, and understanding of source code reading patterns.

The third edition, from which this proceedings were collected, aimed at investigating the opposite direction. We invited contributions departing from theoretical perspectives and grounds to present new hypotheses about gaze behaviour in comprehension, debugging, and other tasks of programming. Contributions were expected to present implications to industrial programming practice or programming education.
All submissions were reviewed by at least two members of the program committee:

Roman Bednarik - University of Eastern Finland
Andrew Begel - Microsoft Research, USA
Sven Buchholz - FH Brandenburg, Germany
Teresa Busjahn - Freie Universität Berlin, Germany
Martha Crosby - University of Hawaiʻi at Mānoa, USA
Carsten Schulte - Freie Universität Berlin, Germany
Bonita Sharif - Youngstown State University, USA
Jozef Tvarozek - Slovak University of Technology, Slovakia

Ten submissions were accepted to the workshop. We hope readers will find the collected papers interesting. We also thank all participants for their active work and creativity.

Roman Bednarik, Teresa Busjahn, Carsten Schulte and Sascha Tamm

Contents

Eye Movements and Cognitive Models

Dario Salvucci
Drexel University
dds26@drexel.edu

Abstract: This talk will discuss how computational models can be used to better understand both eye movements and cognitive processes. In particular, the talk will give a short overview of the ACT-R cognitive architecture and its visual system, and discuss how these models can help to explain behavior in task domains such as driving and multitasking.

Eye Movements and Reading Development

Tuomo Häikiö
University of Turku
tuilha@utu.fi

Abstract: Eye movement registration has proved a valuable tool to examine reading behavior and has been used extensively to study skilled reading. However, much less is known about how reading development progresses from early to skilled reading. This is surprising, given how important reading skill is for education, for example. As this lack of research can partly be attributed to the ease of use of equipment with young children, it should not come as a surprise that with the recent development of state-of-the-art eye movement registration equipment, in last few years there has been a surge of new research into this particular field of interest. In my talk, I will go over some basic findings about skilled reading behavior and reading development as well as describe several techniques that can be used to study different aspects of reading.

Eye Movements During Code Review

Andrew Begel
Microsoft Research
Redmond, WA, USA
andrew.begel@microsoft.com

ABSTRACT

Programmers read through their source code for many reasons: comprehension, investigation, even curiosity. During code review, programmers read over other people's code, a process vitally important to ensuring that poor quality code does not get committed into their product's code repository. Code reviewing is a form of reading that consists of skimming the text to identify beacons (e.g. patterns of bad code) that trigger the reader to stop and communicate some feedback to the code author. Research on code review has indicated differing levels of understanding required to properly evaluate code quality, often affecting the time it takes to finish a code review. We believe that it is possible to identify different qualities of code review feedback based on gaze data recorded by eye trackers of developers doing code review. This proposal will sketch out how we plan to do it.

Categories and Subject Descriptors

D.2.5 [**Testing and Debugging**]: Code inspections and walk-throughs

General Terms

Measurement, Human Factors

Keywords

Eye tracking, Code review

1. INTRODUCTION

Code review is a common part of the software development process in which relevant software developers read code that was created or edited by a colleague to comment on its suitability for including it in a source code repository, and thus in the final software product. Code reviews can be in-person or electronic, but typically involve a developer skimming the desired changes in the code and making comments about any parts that she sees that should be changed and/or improved. Rounds of iteration ensue, with a final result that the code is approved for checkin, or scrapped.

Within a code review, the reviewer does not just blindly look through the code, but often follows a set of guidelines. Some organizations enshrine these guidelines in a checklist to help reviewers remember what to look for in the code. Some of these guidelines include bugs in the implementation, common coding mistakes, an encouraged or acceptable coding style, a test of reasonableness of whether the code solves the problem in an efficient manner, and a list of serious security or performance problems that have caused problems for the team in the past. Often the reviewer will spot something that is not clearly wrong, but could affect the code's maintainability in the future (e.g. code that is too clever and undocumented), and suggest changes to improve it.

Each item in the checklist requires a different level of code understanding. A study of code review practices at Microsoft reported that finding bugs required much more complete understanding of the program than merely improving the code or the development process [1]. Since every programmer has a different amount of knowledge and understanding of code in their product, they will demonstrate a different degree of skill in spotting and “fixing” the problems in the code. [4].

A lab study of specification review found that English majors were actually much better at spotting ambiguities and bugs in specs than computer science majors [3]. Computer science majors likely presumed too much correctness on the part of the spec author rather than relying on skepticism and a careful reading of the spec. Reviewers also have trouble when there is too much code to review and little guidance from the code author as to what they should be looking for [1]. Reviews in many open source projects were found to be smaller, but more frequent, lessening the effects of session fatigue on the reviewer, but perhaps reducing the reviewer's patience with particular code authors as the number of reviews increases [5]. Uwano et al. measured the effectiveness of code reviewers using eye tracking metrics and discovered that subjects who scanned through the entire program were more efficient at finding defects in that code [6].

2. OPERATIONALIZATION

We believe that eye tracking can be used to identify how good reviewers are at identifying problems in their code. In this section, we will list measurements that may help us identify the behaviors.

- **Coverage** (Words read, time per word, area covered by gaze): There is often a lot of code to review, and much of it is boiler-plate text without interesting meaning or importance. This would impact a reader's word coverage during the reviewing task. A bored reader may skip more words than an attentive one. However, we know that as they read, experienced readers skip more words than inexperienced readers, so the skip amount should be mediated by development experience and/or experience with the code [2].

- **Reading Speed** (Words per minute, places where speed drops, word revisit rate, and correlation with affect measures): Code review is a primarily a skimming process where the reader moves through the code quickly until triggered by something "interesting." Places where reading speed slows dramatically, or halts entirely, could indicate places where the reader was cognitively challenged or triggered by what was seen. Then again, it could mean the reader investigated that part of the program more carefully. Revisiting the same text again and again may indicate lack of comprehension or confusion, which also may identify places where code should be changed or improved. Some code review tools record time stamps whenever the reader comments on the code; these timestamps can be used to index into the gaze data stream and look for reading speed changes that could indicate an upcoming code review comment is forthcoming.
- **Structural Scanpath** (Order in which reviewer reads the files, classes, and methods): These metrics are not guided entirely by theory but more from an empirical curiosity about code review. In what order do reviewers look through the code? How regular is this order across files or across code reviews? What influences it? Is there any effect related to development experience, fatigue (how many other reviewers were already done today), familiarity with the review (many reviews have several back and forth iterations before the code is finally accepted), or personal familiarity with the code author or social distance in the work hierarchy? How often does this order violate linear scanning (i.e. reading a file or diff from top to bottom)?
- **Intraprocedural Scanpath** (Order in which a method is read): Code review appears to be a skimming process which ends when the reader is triggered by a beacon to pay closer attention. When this beacon is spotted, does the reader switch their reading order from linear (like a book) to forward or reverse control flow or forward or reverse data flow? How does this change depend on (or help us indicate) the kind of beacon that triggered the closer inspection? Does the beacon cause the same kind of scanpath change over several points in the same code review or across code reviews?

3. PROGRESS

This summer, Hana Vrzáková, a Ph.D. student at University of Eastern Finland working as a research intern with me at Microsoft, and I conducted a study of Microsoft engineers doing code reviews, and recorded them with several biometric sensors, including a Tobii EyeX eye tracker. We are currently analyzing our data and will investigate the measures we list above to see if eye tracking can reveal useful insights into how software engineers conduct code reviews.

4. REFERENCES

[1] A. Bacchelli and C. Bird. Expectations, outcomes, and challenges of modern code review. In *Proceedings of the International Conference on Software Engineering*. IEEE, 2013.

[2] T. Busjahn, R. Bednarik, A. Begel, M. Crosby, J. H. Paterson, C. Schulte, B. Sharif, and S. Tamm. Eye movements in code reading: Relaxing the linear order. In *Proceedings of the 23rd International Conference on Program Comprehension*, pages 255–265, Piscataway, NJ, USA, 2015. IEEE Press.

[3] J. C. Carver, N. Nagappan, and A. Page. The impact of educational background on the effectiveness of requirements inspections: An empirical study. *IEEE Transactions on Software Engineering*, 34(6):800–812, 2008.

[4] T. Fritz, G. C. Murphy, E. Murphy-Hill, J. Ou, and E. Hill. Degree-of-knowledge: Modeling a developer's knowledge of code. *ACM Transactions on Software Engineering Methodology*, 23(2):14:1–14:42, Apr. 2014.

[5] P. C. Rigby, D. M. German, and M.-A. Storey. Open source software peer review practices: A case study of the apache server. In *Proceedings of the International Conference on Software Engineering*, pages 541–550, New York, NY, USA, 2008. ACM.

[6] H. Uwano, M. Nakamura, A. Monden, and K. Matsumoto. Exploiting eye movements for evaluating reviewer's performance in software review. *IEICE Transactions on Fundamentals of Electronics, Communications, and Computer Sciences*, E90-A(10):2290–2300, Oct. 2007.

Studying Sustained Attention and Cognitive States with Eye Tracking in Remote Technical Interviews

Denae Ford
NC State University
Raleigh, North Carolina, USA
denae_ford@ncsu.edu

Titus Barik
ABB Corporate Research
Raleigh, North Carolina, USA
titus.barik@us.abb.com

Chris Parnin
NC State University
Raleigh, North Carolina, USA
cjparnin@ncsu.edu

ABSTRACT

In this position paper, we argue that eye tracking can be used to understand the underlying cognitive states of a programmer during remote technical interviews, specifically programming interviews. We describe a mock-interview experiment that applies eye tracking to identify these cognitive states, and propose two computational interventions that support an interviewer and a candidate. We posit that these interventions will increase the effectiveness of remote technical interviews.

Keywords

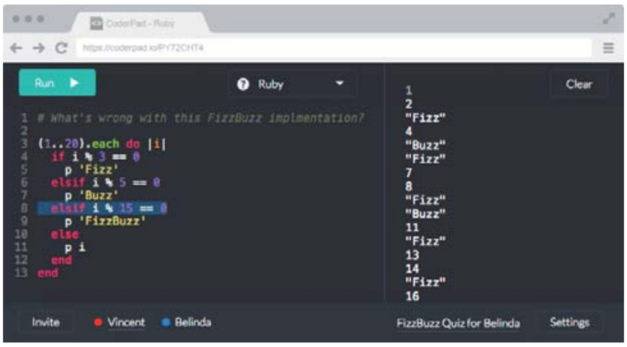
eye tracking, technical interview, remote programming, interventions, anxiety, attention

1. INTRODUCTION

A technical interview is a stage of a job interview where recruiters ask candidates technical questions pertaining to a specific field of work for a job position. For software developers, these technical interviews often include a programming portion where the applicant is asked to work through a series of programming scenarios. Technical interviews can be an expensive process. Some companies have reported that they will spend at least $100,000 per position to have current employees recruit candidates, fly candidates out for interviews, and support these candidates as they go through the onboarding process.[1]

One cost saving mechanism is to conduct interviews remotely, using commodity webcams [12] and a shared editing environment, such as CoderPad[2] (Fig. 1). However, remote technical interviews have challenges of their own. Determining what is going on during a period of silence is difficult for an interviewer and uncomfortable for a candidate. Candidates must balance the time devoted to solving problems and expressing their thought process. Interviewers may perceive extended silence as negative or assume that a candidate is stalling for time or that a candidate simply does not know how to solve a problem. However, programming is a cognitively demanding activity, and like other deep thinking tasks [4], periods of silence are absolutely necessary.

In this position paper, we propose a model for comprehending the cognitive state and attention of a programmer during a remote technical interview. The unique affordances of this mode of interviewing can add to the misinterpretation of what is going on during the deep thinking silence.



```
# What's wrong with this FizzBuzz implmentation?

(1..20).each do |i|
  if i % 3 == 0
    p 'Fizz'
  elsif i % 5 == 0
    p 'Buzz'
  elsif i % 15 == 0
    p 'FizzBuzz'
  else
    p i
  end
end
```

```
1
2
"Fizz"
4
"Buzz"
"Fizz"
7
8
"Fizz"
"Buzz"
11
"Fizz"
13
14
"Fizz"
16
```

Figure 1: Online technical interview conducted using CoderPad. Vincent remotely interviews Belinda, who is debugging a FizzBuzz implementation. Figure from https://CoderPad.io/.

Performing an analysis of the different cognitive states of interview candidates can help define the challenges and benefits of remote technical interviews and how it affects their cognitive state and attention. As a result of this research, we can use these models of attention to design better interview procedures that minimize disruption to candidates while allowing interviewers to assess a candidate's thought process and problem solving speed.

2. BACKGROUND

2.1 Nonverbal Cues and Think-Aloud

Hollandsworth et al. found that interviewers place a high importance on nonverbal cues when making hiring decisions [8]. Although the content of discussion is ranked highest, nonverbal cues, such as composure and eye contact, were highly ranked as well. In remote interviews, nonverbal communication is limited when a candidate cannot be fully seen behind a phone or a computer. Not only are these cues important for assessing performance, but they help regulate disruption to a candidate's cognitive processes, including sustained attention.

Borrowing from concurrent think-aloud protocols, where eye tracking has been used, we can hypothesize that analogous techniques might apply to remote interviews. For example, Guan et al. found that retrospective think-aloud can provide information about a participant's strategies to solving a problem using eye movements [7]. One challenge with retrospective think-aloud is that a candidate may be

[1] http://www.entrepreneur.com/article/242613

[2] http://coderpad.io

subject to forgetting if they are asked too late [11]. In the aforementioned study, participant strategies were determined by the "eye mind hypothesis". Just and Carpenter's eye-mind hypothesis provides support that people are paying attention to what they are looking at [9]. As eye movements are representative of what a candidate can be attending to, we can review measures of these movements to determine the cognitive load during sustained attention tasks such as programming.

2.2 Sustained Attention in Programming

Cognitive load refers to the total information demand placed on an individual [10]. Several studies have found multiple ways to measure it. One technique, pupillometry, measures changes in pupil diameter in response to changing mental workloads. With task-evoked pupillary responses such as mean pupil dilation, peak dilation, and latency to the peak, the intensity of cognitive load can be monitored[2]. Large pupil dilation is associated with high cognitive load and small pupil dilation is associated with low cognitive load. It's not ideal to interrupt a person when they have a high cognitive load as they are attempting to make sense of a lot of information. When this processing is deterred during peak processing, it can break concentration and candidates may not be able to resume processing where they stopped. Determining the intensity of cognitive load can cue the appropriate time to deter the attention of a candidate. It is ideal to switch concentration during a low cognitive load task.

Studies have shown there is a relationship between shifts of attention and saccades [5]. Saccades are involuntary actions that occur during information processing; when the information is being processed. With this information we can get a better understanding of how sustained attention shifts by saccades. In addition, studies have shown that anxious people have difficulty with attentional control through saccadic movement [13]. If saccades begin to stray from the information being processed, we hypothesize that we have a stray in sustained attention.

In addition, blink rates can demonstrate the interest in a task. Chan et al. found that blink rates were highly associated with disinhibition signs during sustained attention tasks [3]. Graham et al. found that blinks become larger, starting from a full open eye to completely close, and faster when the reflex-eliciting stimulus is attended than when attention is directed elsewhere [6]. In this context, blinks can monitor a candidate's sustained attention and aid in understanding the underlying cognitive mechanisms of a programming interview.

With these features of pupil movement we can determine the sustained attention of individuals during programming experiences and determine when the appropriate time to deter their attention.

3. INTERVENTIONS

To assist in our investigation of remote technical interviews, we propose two interventions that if added to remote programming tools, might reduce disruption to a candidate, while allowing interviewers to have a better comprehension of a candidate's mental state.

Blackouts: A good interviewer might allow some time for a candidate to reflect on a problem in isolation, without worrying about the presence of a interviewer pressuring a candidate. For example, an interviewer might say, "now that I have explained the problem, I will put the phone down and walk out for about 4 minutes to allow you to digest the problem." For coding activities, having your live state exposed to an interviewer can cause constant anxiety about making mistakes in front of others. This blackout intervention captures the benefits of a "walk-out" during remote interviews by only refreshing a candidate's screen in predetermined time intervals. Refreshing the screen allows an interviewer to monitor the progress of a candidate's code while allowing a candidate to have moments to reflect. During the blackout period a candidate can complete short programming tasks in a time-boxed manner, without the constant pressure of being scruitnized on every character typed on the shared screen of an interviewer.

Remote Focus Lights: When driving in a car, passengers are better able to sense when a driver is busy than someone having a phone conversation with the driver. In remote interviews, a candidate may be in a mental state with high cognitive load but is not easily observable by an interviewer. For example, when editing a document, an interviewer may use typing as a cue for when not to interrupt. However, if a candidate is reflecting on a problem or reading code while deep in thought, that information is not easily accessible to an interviewer, but still reflects a high cognitive load. The remote focus light intervention indicates when a candidate is currently involved in a high mental workload to indicate when not to interrupt (🔴 — red light) and when a candidate is accessible for questions (🟢 — green light). The measurements of high and low cogntivie load will be collected through pupillary movements and reflected through the focus light visible to an interviewer.

4. RESEARCH QUESTIONS

Through studying cognitive states, we can identify the circumstances under which some interviews are more effective than others. As a starting point for model building, we ask three research questions:

RQ1 *How is the performance of a candidate reflected in their cognitive states during a technical interviews?*
In order to confirm variations of how interviews vary with introducing interventions, we must first confirm how cognitive states are demonstrated in a standard technical interview.

RQ2 *Is a candidate able to sustain their attention to the programming issue less when an interviewer has increased view of their actions?*
There is a limit on how much an interviewer can see a candidate with the blackout feature. With this research question we investigate how candidate-managed visibility affects the candidates performance.

RQ3 *How does an interviewer's access to a candidates's cognitive state affect a candidate's performance?*
The focus light intervention will provide an interviewer with knowledge of the varying cognitive states of a candidate. With this research question we investigate how this feature affects a candidate's performance as they are studied by an interviewer.

5. PROPOSED STUDY

To understand the impact and effectiveness of our interventions on the remote technical interview process, we propose an experiment involving an interviewer and a candidate that simulates a typical remote interview setting (Fig. 1). During the experiment, we instrument an eye tracker to collect gaze data on pupil dilation, saccadic movement, fixations, which correlate with measures on attention and cognitive load.

We envision four experimental conditions:

C1 No interventions. A control condition in which no interventions are present. This provides a baseline that is representative of how remote interviews are conducted today.

E2 Blackout-only. The candidate has access to the blackout feature, but an interviewer has no interventions available to them.

E3 Focus light-only. The interviewer has a remote indicator of the focus light state of a candidate, but a candidate has no interventions available to them.

E4 Both blackout and focus light. The candidate has access to the blackout feature, and an interviewer has a remote indicator of the focus light state of a candidate.

This experimental design allows us to measure the anxiety of candidates with and without blackout capability, and, through the focus light capability, measure the effect of asking questions to a candidate under high and low periods of cognitive loads. We hypothesize that in all intervention conditions, a candidate will have reduced anxiety and increased sustained attention. A candidate will be more comfortable knowing an interviewer has more knowledge of when to solicit interview questions and power to control when an interviewer can see their work. Furthermore, these conditions enable us to assess the ability of a candidate and interviewer in regulating their activities, both as independent interventions and finally as a co-intervention.

In conducting this experiment, we may encounter challenges with recreating the intensity of an interview environment which may play a role in the success of candidates. However, we can combat this challenge by having candidates use their own personal computer as they would during standard remote technical interview; placing them in a familiar setting. This would help recreate an atmosphere similar to a real remote technical interview. This will also reduce the costs of having to send a candidate an eye tracking device as studies have shown ways to leverage the use of web cameras to conduct eye tracking experiments[1].

6. CONCLUSION

Remote technical interviews allow candidates to be evaluated at a lower cost to the company. But what else is lost when removing the fully visible aspect of the interview process? Furthermore, programming is a cognitive intensive task that defies expectations of constant feedback that today's interview processes follow. This has left a gap in understanding what goes on during the programming interview process and how to properly assess programming skills of candidates to succeed at these interviews. With the proposed study, we will be begin to comprehend the cognitive state and sustained attention of candidates during remote technical interviews to refine the interview process.

7. REFERENCES

[1] M. Andiel, S. Hentschke, T. Elle, and E. Fuchs. Eye tracking for autostereoscopic displays using web cams, 2002.

[2] J. Beatty and B. Lucero-Wagoner. The pupillary system. *Handbook of Psychophysiology*, 2:142–162, 2000.

[3] R. C. Chan and E. Y. Chen. Blink rate does matter: a study of blink rate, sustained attention, and neurological signs in schizophrenia. *The Journal of Nervous and Mental Disease*, 192(11):781–783, 2004.

[4] J. Dawson. Reflectivity, creativity, and the space for silence. *Reflective Practice*, 4(1):33–39, 2003.

[5] R. Godijn and J. Theeuwes. The relationship between exogenous and endogenous saccades and attention. *The Mind's Eye: Cognitive and Applied Aspects of Eye Movement Research*, pages 3–26, 2003.

[6] F. K. Graham. Attention: The heartbeat, the blink, and the brain. *Attention and Information Processing in Infants and Adults: Perspectives from Human and Animal Research*, 8:3–29, 1992.

[7] Z. Guan, S. Lee, E. Cuddihy, and J. Ramey. The validity of the stimulated retrospective think-aloud method as measured by eye tracking. In *CHI '06*, pages 1253–1262, 2006.

[8] J. G. Hollandsworth, R. Kazelskis, J. Stevens, and M. E. Dressel. Relative contributions of verbal, articulative, and nonverbal communication to employment decisions in the job interview setting. *Personnel Psychology*, 32(2):359–367, 1979.

[9] M. A. Just and P. A. Carpenter. A theory of reading: from eye fixations to comprehension. *Psychological Review*, 87(4):329, 1980.

[10] F. G. W. C. Paas and J. J. G. Van Merriënboer. Instructional control of cognitive load in the training of complex cognitive tasks. *Educational Psychology Review*, 6(4):351–371, Dec. 1994.

[11] J. Russo, E. Johnson, and D. Stephens. The validity of verbal protocols. *Memory & Cognition*, 17(6):759–769, 1989.

[12] W. Sewell and O. Komogortsev. Real-time eye gaze tracking with an unmodified commodity webcam employing a neural network. In *CHI '10*, pages 3739–3744, 2010.

[13] M. J. Wieser, P. Pauli, and A. Mühlberger. Probing the attentional control theory in social anxiety: An emotional saccade task. *Cognitive, Affective, & Behavioral Neuroscience*, 9(3):314–322, 2009.

Using focused attention to improve programming comprehension for novice programmers.

Emlyn Hegarty-Kelly
Dept. of Computer Science
Maynooth University
Maynooth, Ireland
emlyn.hegartykelly@nuim.ie

Susan Bergin*
Dept. of Computer Science
Maynooth University
Maynooth, Ireland
susan.bergin@nuim.ie

Aidan Mooney*
Dept. of Computer Science
Maynooth University
Maynooth, Ireland
aidan.mooney@nuim.ie

ABSTRACT

Learning to program is difficult for many novice programmers. This study will attempt to improve programming comprehension by actively focusing the gaze of novice programmers to important parts of programming code. The study will design a set of programming comprehension tasks and include two dependent experiments. In the first experiment important code segments, referred to as beacons, will be identified by tracking eye gaze data of skilful programmers as they successfully solve the comprehension tasks. The second experiment will involve novice programmers of comparable programming ability, divided into three groups. The first group will be asked to solve the tasks without any beacons provided, the second group will be asked to solve the tasks with the beacons visually highlighted and the third group will be asked to solve the tasks with unimportant code segments visually highlighted. It is hypothesized that the group with the correct beacons highlighted will perform better.

Keywords

Eye tracking, focused attention, programming comprehension

1. INTRODUCTION

It is well established that learning to program is difficult. [1][2][3][4] . Focused attention is the ability to respond discreetly to specific visual, auditory or tactile stimuli. The goal of this study is to focus the attention of novice programmers while performing programming tasks to facilitate improved comprehension. By visually highlighting critical areas of code blocks a novice programmer will be able to solve programming comprehension tasks that they would not be able to solve ordinarily.

The Duncker Radiation problem, describes where a tumour exists under the skin that cannot be attacked directly with lasers as it will damage the healthy tissue around the tumour. In a study at Cornell University ninety-five participants were asked how to nullify the tumour without damaging the healthy tissue around it. Two experiments were carried out as part of the study. In the first experiment, participant's eye gaze was tracked. They were given a diagram of the problem, that clearly marked the tumour, the healthy tissue and the skin, and participants were asked how they would nullify the tumour. Thirty-three percent of participants successfully solved the problem. When the eye gaze of each participant was analysed the skin was identified as the critical area that the successful participants fixated on most when nullifying the tumour. The second experiment was divided into three further groups. The first group had the same experimental setup as the first experiment, the second group were given an animated diagram with the critical area of the skin animated and the third group had a diagram with a non-critical area animated. The first and third groups had similar results to the first experiment with 63% unable to solve the problem. The second group showed virtually opposing performance with 67% of participants successfully solving the problem [5].

The critical areas of code blocks for comprehension can be referred to as beacons. Beacons are lines of code which can be found to exist in code structures. These beacons point out the function or operation of a given code block [6]. It has been shown that beacons found by expert programmers were not identified by novice programmers and that the novice programmers made little or no distinction between different segments of a given code block [7] . This study will track the eye gaze of expert programmers to determine the beacons they focus on and then these beacons will be highlighted to novice programmers in a subsequent experiment to determine if this results in improved comprehension.

2. METHODOLOGY

The study will attempt to address the following question:

- By highlighting beacons in code blocks will novice programmers be able to solve programming comprehension tasks that they would not be able to solve without beacon highlighting?

Both novice and expert programmers will be invited to participate in the study. Two experiments will be set up. The first experiment will involve both novice and expert programmers. The novice and expert programmers will be asked to complete the same programming comprehension tasks based on introductory programming material, including nested loops, recursion, and sorting algorithms.

During the experiment a remote eye tracker will be used to monitor eye movement data of the participants. The locations of fixations, duration of fixations and attention switching between critical areas will be recorded and analysed. The data from the eyetracker will be used to identify beacons in the code for successful solvers. The beacons identified will be used for the second experiment.

*Senior / Corresponding Author

The second experiment will then only consist of novice programmers. They will be split into three groups where the first group will complete the challenge like the programmers from the first experiment. The second group will be presented with code segments in which the beacons are highlighted. The third group will take the same programming comprehension task except incorrect beacons will be highlighted. Each group will have their response time and answers tracked.
At the end of both experiments it is hoped that a number of findings can be made. From the first experiment it is hoped that expert and novice programmers will find different beacons in the code and the expert programmers will be quicker and more accurate in the comprehension task. In the second experiments it is hoped that a number of results would be verified as follows. The first and third groups will achieve the same success rate as the novice programmers from experiment one. The second group will achieve a much higher success rate than both of the other groups.

3. CONCLUSIONS

The study will make two contributions to the computer science education community. The first experiment could provide additional evidence that expert and novice programmers do not identify the same beacons while performing programming comprehension tasks [7]. The second experiment could show that when provided with the correct beacons novice programmers could more successfully complete programming comprehension tasks, as was the case in the Dunckers Radiation Problem [5] .

Providing the beacons to novice programmers would reduce the difficulty in learning how to code, and not only focus them on individual code blocks but also provide focus to the concepts they should learn as a whole.

4. REFERENCES

[1] Susan Bergin and Ronan Reilly. Programming: factors that influence success. In *ACM SIGCSE Bulletin*, volume 37, pages 411–415. ACM, 2005.

[2] Susan Bergin and Ronan Reilly. Predicting introductory programming performance: A multi-institutional multivariate study. *Computer Science Education*, 16(4):303–323, 2006.

[3] Anabela Gomes and António José Mendes. Learning to program-difficulties and solutions. In *International Conference on Engineering Education–ICEE*, volume 2007, 2007.

[4] Jackie O'Kelly, Susan Bergin, S Dunne, Peter Gaughran, John Ghent, and Aidan Mooney. Initial findings on the impact of an alternative approach to problem based learning in computer science. *Pleasure by Learning*, 2004.

[5] Spivey M J Grant, E R. Eye movements and problem solving. *Psychological Science-Cambridge*, 14(5):462–466, 2003.

[6] Susan Wiedenbeck. Beacons in computer program comprehension. *International Journal of Man-Machine Studies*, 25(6):697–709, 1986.

[7] Martha E Crosby, Jean Scholtz, and Susan Wiedenbeck. The roles beacons play in comprehension for novice and expert programmers. In *14th Workshop of the Psychology of Programming Interest Group*, pages 58–73, 2002.

Towards identifying programming expertise with the use of physiological measures

Dimosthenis Kontogiorgos
Department of Computer Science
University of Copenhagen, Denmark
dimo@di.ku.dk

Konstantinos Manikas
Department of Computer Science
University of Copenhagen, Denmark
kmanikas@di.ku.dk

ABSTRACT

In this position paper we propose means of measuring programming expertise on novice and expert programmers. Our approach is to measure the cognitive load of programmers while they assess Java/Python code in accordance with their experience in programming. Our hypothesis is that expert programmers encounter smaller pupillary dilation within programming problem solving tasks. We aim to evaluate our hypothesis using the EMIP Distributed Data Collection in order to confirm or reject our approach.

Keywords

code comprehension, programming expertise, pupillometry

1. INTRODUCTION

Recently, neuroscientists used functional magnetic resonance imaging (fMRI) to identify and measure intelligence, but also to predict individuals' cognitive behaviour during tasks [4]. Considering the ethical implications, researchers have tried numerous times to extract information from subjects' cognitive activity in order to classify them according to their intelligence or expertise.

Programmers today work in several programming languages and often develop expertise in a number of them. In the industry, however, seniority is usually dependent on knowledge in a specific domain (i.e. data structures, algorithms), rather than a specific programming language. Therefore, programming experience is not necessarily related to expertise. A novice programmer might experience different challenges than an expert, while comprehending code [6].

In software engineering experience is an important parameter within programming comprehension. It is defined as "the amount of acquired knowledge regarding the development of programs, so that the ability to analyse and create programs is improved" [11]. There has not been, however, an agreed way of measuring programming experience.

Psychologists in the 60s found a correlation between cognitive activity and pupillary dilation. There can be various reasons on why our pupils can encounter dilation, and one of them is our cognitive workload. Using pupillary response we can therefore measure our cognitive activity.

In this position paper, we examine how programming expertise influences the programmer's cognitive activity. We are interested in identifying connections between expertise in a specific technology (i.e. Java, Python) and physiological measures. Several studies have measured cognitive activity using physiological means such as pupillometry. We propose experimenting with novice and experts in Java/Python programming to investigate how experience in a technology differentiates within physiological measures.

2. RELATED WORK

2.1 Programming expertise

Independently of acquired knowledge, intelligence is a parameter used to measure one's capacity in reasoning and problem solving [4]. However, in domain-oriented tasks, the amount of acquired knowledge is an important ability to analyse and solve problems. Knowledge is a combination of our long-term memory and our working memory. Within cognitive load theory, working memory is described as the number of elements we can hold while processing a task. A large number of material elements may be a single element for one with expertise in a particular task [12].

Sweller et al. derived three fundamental types of measuring cognitive load: (1) subjective (individuals' self-reporting), (2) physiological (heart rate, brain activity, pupillary dilation) and (3) task performance [12]. Within program comprehension, different means of measuring cognitive load or programming experience have been utilised, mainly including self reporting or subjective techniques, but also physiological measures (fNIRS) [11, 13].

To the extent of our knowledge, there has not been a correlation between cognitive effort and programming experience while using pupillary response, and several studies have shown its precision as means of measurement. Whether we should use such measures to identify individuals' expertise independently of their self reporting is posed as an open question.

2.2 Pupillary dilation

Psychology. Beatty found that task-evolved pupillary response is a good indicator of cognitive load and cognitive activity [2]. It has been used in several disciplines, as pupillary dilation is a very precise and non-invasive manner of measuring cognitive activity and can be easily measured [9]. In terms of expertise, Ahern and Beatty showed a correlation between intelligence and pupillary response; more intelligent subjects showed smaller task-evolved pupillary dilation in arithmetic problems [1].

Computer science. Klinger used pupillometry to assess visual interfaces by the amount of cognitive load they require from a user [8]. Maier et al. investigated how particular UML diagram layouts affect cognitive load [10]. Within programming tasks, Iqbal investigated how well pupil size correlates with mental workload [7]. Fritz et al. assessed and predicted task difficulty on expert programmers [5].

3. METHOD

In this proposed exploratory study, we intend to measure programmers' cognitive load and mental effort during programming tasks, in order to identify and measure programming expertise. We argue that measures such as self reporting [11] might pose a potential threat to the validity of measuring experience and attempt to correlate physiological measures to programming expertise.

Previous studies in this workshop have experimented with patterns of reading behaviour within novice and expert programmers. To investigate individual behaviour and detect experts from novices, we apply pupillometry measures in order to identify cognitive effort during programming tasks.

3.1 Research design

Hypothesis. Our hypothesis is that expert programmers experience significantly less cognitive load within programming tasks than novice programmers. In order to measure cognitive activity we use pupillary response to various types of task difficulty. We therefore form two research questions:

RQ1. Can we use pupilometry to identify/measure programming expertise?

RQ2. Does programming expertise influence pupillary dilation within programming tasks?

3.2 Distributed data

Using the Distributed Data Collection from the workshop, we attempt to validate or reject our hypothesis within program comprehension. The data includes gaze interactions from novice and expert programmers while assessing code in Java/Python. Our approach is to combine raw sensor data such as fixation, pupil size and gaze interaction (dependent variables) with demographics including age, gender, years of experience and education (independent variables).

4. EXPECTED RESULTS

In programming (similarly to reading) processing of visual information only occurs during eye fixations [3], where we also expect to encounter pupillary dilation in terms of cognitive processing. We attempt to assess the ability to read and understand code, as it differentiates from the ability to immediately recognise its flaws or various imperfections.

Researchers have tried to identify criteria that can be used to measure experience on novice or expert programmers, but to our knowledge no attempts were made to do so using pupillary response. To address our research questions, we experiment with pupil dilation in programming tasks and propose the measurement of programming expertise by measuring cognitive activity on novice and expert programmers.

5. REFERENCES

[1] S. Ahern and J. Beatty. Pupillary responses during information processing vary with scholastic aptitude test scores. *Science*, 205(4412):1289–1292, 1979.

[2] J. Beatty. Task-evoked pupillary responses, processing load, and the structure of processing resources. *Psychological bulletin*, 91(2):276, 1982.

[3] T. Busjahn, C. Schulte, B. Sharif, Simon, A. Begel, M. Hansen, R. Bednarik, P. Orlov, P. Ihantola, G. Shchekotova, and M. Antropova. Eye tracking in computing education. In *Proceedings of the tenth annual conference on International computing education research*, pages 3–10. ACM, 2014.

[4] E. S. Finn, X. Shen, D. Scheinost, M. D. Rosenberg, J. Huang, M. M. Chun, X. Papademetris, and R. T. Constable. Functional connectome fingerprinting: identifying individuals using patterns of brain connectivity. *Nature Neuroscience*, (October):1–11, 2015.

[5] T. Fritz, A. Begel, S. C. Müller, S. Yigit-Elliott, and M. Züger. Using psycho-physiological measures to assess task difficulty in software development. In *Proceedings of the 36th International Conference on Software Engineering*, pages 402–413. ACM, 2014.

[6] M. Hansen, A. Lumsdaine, and R. L. Goldstone. An experiment on the cognitive complexity of code. *In Proceedings of the Thirty-Fifth Annual Conference of the Cognitive Science Society, Berlin, Germany*, 2013.

[7] S. T. Iqbal, X. S. Zheng, and B. P. Bailey. Task-evoked pupillary response to mental workload in human-computer interaction. In *CHI'04 extended abstracts on Human factors in computing systems*, pages 1477–1480. ACM, 2004.

[8] J. M. Klingner. *Measuring cognitive load during visual tasks by combining pupillometry and eye tracking.* PhD thesis, Stanford University, 2010.

[9] B. Laeng, S. Sirois, and G. Gredebäck. Pupillometry a window to the preconscious? *Perspectives on psychological science*, 7(1):18–27, 2012.

[10] A. Maier, N. Baltsen, H. Christoffersen, and H. Störrle. Towards diagram understanding: A pilot study measuring cognitive workload through eye-tracking. In *International Conference on Human Behaviour in Design 2014*.

[11] J. Siegmund, C. Kästner, J. Liebig, S. Apel, and S. Hanenberg. Measuring and modeling programming experience. *Empirical Software Engineering*, 19(5):1299–1334, 2014.

[12] J. Sweller, J. J. Van Merrienboer, and F. G. Paas. Cognitive architecture and instructional design. *Educational psychology review*, 10(3):251–296, 1998.

[13] M. P. Uysal. Evaluation of learning environments for object-oriented programming: measuring cognitive load with a novel measurement technique. *Interactive Learning Environments*, pages 1–20, 2015.

Eye-tracking to trace anxieties of programmers

David C. Moffat
Glasgow Caledonian University
Cowcaddens Road
Glasgow
G4 0BA
UK
D.C.Moffat@gcu.ac.uk

James H. Paterson
Glasgow Caledonian University
Cowcaddens Road
Glasgow
G4 0BA
UK
James.Paterson@gcu.ac.uk

ABSTRACT

Students of programming can experience anxiety and other emotions while they are learning, and writing code. Professionals might get anxious while debugging. It could be helpful to record their eye-movements while working at the computer screen, if it can indicate where and when they experience more anxiety. This could possibly show which parts of the subject matter they are less sure about, and even whether they are getting dismayed at their lack of progress. It might also be possible to reveal where and when programmers experience a heavier cognitive load, as we suggest in some initial ideas for further experiments.

Categories and Subject Descriptors

K.3.4 [**Computer and Information Science Education**]: Programming

General Terms

Experimentation, Human Factors

Keywords

Code reading, eye-tracking, computing education

1. EMOTIONS OF PROGRAMMERS

It is possible that students of programming start with some misconceptions about the nature of software engineering. They could assume that code needs to be correct at the first attempt, for example, and that bugs in their code is evidence of lack of aptitude. In some earlier work, we explored such early attitudes [4].

Furthermore, it is not only novice programmers who are affected by emotion. Professional programmers work under time constraints, and their work is often critically important to other people. Added to that, they have to learn new programming languages and software methods, so that they are similar in some ways to novices. A lot of time is also spent in finding and fixing bugs in programs, and that is an open-ended and unpredictable task that can be frustrating. All these pressures mean that we should expect professional programmers to experience a range of emotions, even if not exactly the same as novices do.

2. EMOTION THROUGH THE EYES

In a small study on the effects of different mood music while watching video, we recorded viewers with an eye-tracker [5]. This was to explore the possibilities of measuring emotion using eye-tracking. The field of eye-tracking research was, and still is, broadly skeptical of this possibility, but we had some minor success with it. Different pieces of music were played in the background, over the same video clip, in order to evoke different emotions or mood states in the viewer. Any changes in the eye-movements could then be attributed to the effects of the music and mood alone.

Table 1: Pupil dilation ranges for different mood music

emotion of music	mean range of pupil (mm)
happy	1.205
sad	1.363
fear	1.454
anger	1.677
none	1.743

In Table 1 the average pupil ranges (from minimum to maximum pupil dilations) are shown, for the different conditions of background music. The ranges are shown with increasing size, so that the happy music induces the least variation in pupil dilation while watching the video clip. The more negative emotions, of fear and anger or aggression (for which the music was an up-tempo rock music track), produced a wider variation in pupil dilation. Lastly, the silent condition produced the widest variation of all.

These data suggest a connection between different emotion states and the ways in which eye pupils can dilate and contract over time, while watching a video clip. In other data from the experiment we found that eye-movement patterns, saccades in particular, can also vary across different emotion conditions. However, it is more challenging to infer a person's emotional or cognitive thought processes from saccades, and research is barely begun in this direction.

Other researchers in computer games have also found some evidence that data from the eyes can indicate emotional arousal, in the context of first-person or third-person perspective [3]. Outwith the context of games, correlations have been found between pupil dilation and affective processing [6, 8]. This is mediated by the connection between pupil dilation and autonomic activation [1]. Because emotions usually cause some activity in the autonomic nervous system, we may therefore detect emotion by measuring pupil dilation.

There is also potential for eye-tracking to be able to detect more specifically cognitive processing. For example, pupils have recently been found to dilate when the memory system is accessed, both in recording new memories, but also on occasion when recalling past memories to consciousness [2]. Blink data can also be useful, in helping to detect windows of effortful cognitive activity or "thinking" [7].

In all these techniques research is at an early stage; and they are only able to detect presence of emotional arousal (or cognitive load). It is much harder to understand more about the *nature* of that mental activity, whether it be the specific type of emotion, or what the viewer might be thinking about.

However, it is already a significant step forward to be able to use pupil dilation to indicate affective arousal. It is also encouraging that some of the work referred to was able to achieve results with viewers watching complex images that had changing light levels, such a dramatic video clips. This means that the observed effects on pupil dilation are large enough and strong enough to show through, even against changing light conditions, which of course would also have a large effect on the pupils. That in turn means that the techniques could be even more effective in a context such as computer programming, where we can maintain more stable light levels.

These techniques could thus lead to insights into the programmer's mind if the experimental environment is controlled well enough.

3. POSSIBLE EXPERIMENTS

In order to use an eye-tracker on a programming task, and to tightly control the computer screen environment, and to be able to collate results across numbers of programmers, it would be sensible to start with an IDE (integrated development environment) in a standard setup. The menus and panels should all be the same for everybody, and the code should also be the same. This includes any help windows with supporting documentation.

Sections or snippets of the necessary program code should ideally be small enough to fit into the windows or panels entirely, without needing to scroll to see more. That would avoid distractions for the programmer; and it would also be easier to analyse the data from the eye-tracker.

Different tasks might be set to the participants, such as answering a question about what a code snippet would do, or debugging it. If some snippets involve a concept that the programmer is unfamiliar with, then we could learn about the anxieties of novice programmers and any effects on their performance. Any bugs in the code could be controlled for difficulty, and that would help us to understand how programmers, including professionals, respond when the task gets harder.

Measures in these experiments should certainly include pupil dilation, but fixation patterns and even eye-blinks could be useful data. As well as emotional arousal, then, we could correlate these data with the actions of the programmer (e.g. fixing a bug), to try and infer the type of emotion (e.g. joy). It might even be feasible to localise time windows around when the cognitive effort reached a peak, and so infer when the programmer realised the problem, and what it was that suggested the solution.

However, it must be said that it is very difficult to make sense out of what a person looks at from moment to moment. Do we look away at a thing because it is of no significance; or because it is, but we have understood and remembered the information we needed from it? For the time being we suggest that experiments on global mood reponses, unrelated to specific informational content that is strongly differentiated across the scene, are appropriate at this stage of the research field. Such experiments are like the ones proposed above.

4. CONCLUSION

We propose that it might be feasible to measure some aspects of emotional experience of programmers, by the use of eye-tracking technology.

There are considerable difficulties, especially in coming to understand the mental models needed, such as the micro-decisions made by viewers when deciding what to look at next, and for how long. This is not easy to determine, even in relatively simpler tasks like normal text reading, where eye-tracking was first used with great success.

However, some solid research results have been obtained with measures of pupil dilation, showing that it is possible to infer at least the presence of emotion, or some types of cognitive activity. This kind of measure could be useful in programming experiments. We have suggested some possibilities for such experiments that could be done to investigate emotion and cognitive activity such as problem-solving in both novice and professional programmers.

The possibilities here are potentially very valuable, and we feel that this research is therefore well worth pursuing.

5. REFERENCES

[1] M. M. Bradley, L. Miccoli, M. A. Escrig, and P. J. Lang. The pupil as a measure of emotional arousal and autonomic activation. *Psychophysiology*, 45(4):602–607, July 2008. doi:10.1111/j.1469-8986.2008.00654.x.

[2] S. D. Goldinger and M. H. Papesh. Pupil dilation reflects the creation and retrieval of memories. *Current Directions in Psychological Science*, 21(2):90–95, 2012.

[3] K. Kallinen, M. Salminen, N. Ravaja, R. Kedzior, and M. Sääksjärvi. Presence and emotion in computer game players during 1st person vs. 3rd person playing view: evidence from self-report, eye-tracking, and facial muscle activity data. In *Proceedings of the 10th Annual International Workshop on Presence , Barcelona, Spain*, 2007.

[4] D. C. Moffat. Students' early attitudes and possible misconceptions about programming. In *22nd Annual Psychology of Programming Interest Group: PPIG-2010*, 2010.

[5] D. C. Moffat and K. Kiegler. Can we track the emotions via the eyes? In *CogSci'06 Workshop: What have eye movements told us so far, and what is next?*, Vancouver, Canada, July 2006.

[6] T. Partala and V. Surakka. Pupil size variation as an indication of affective processing. *Psychophysiology*, 59(1–2):185–198, 2003. doi:10.1016/s1071-5819(03)00017-x.

[7] G. J. Siegle, N. Ichikawa, and S. Steinhauer. Blink before and after you think: blinks occur prior to and following cognitive load indexed by pupillary responses. *Psychophysiology*, 45(5):679–687, 2008.

[8] M. L.-H. Võ, A. M. Jacobs, L. Kuchinke, M. Hofmann, M. Conrad, A. Schacht, and F. Hutzler. The coupling of emotion and cognition in the eye: Introducing the pupil old/new effect. *Psychophysiology*, 45(1):130–140, 2008.

Examining the role of cognitive load when learning to program.

Keith Nolan
Department of Computer Science
Eolas Building
Maynooth University
keith.nolan@nuim.ie

Aidan Mooney*
Department of Computer Science
Eolas Building
Maynooth University
aidan.mooney@nuim.ie

Susan Bergin*
Department of Computer Science
Eolas Building
Maynooth University
susan.bergin@nuim.ie

ABSTRACT

Cognitive load is concerned with the amount of mental effort imposed on working memory at an instant of time. Changes in cognitive load cause very small dilations of the pupils. The aim of this paper is to examine the role of cognitive load while learning to program through the use of remote eye tracking. Although numerous studies have been carried out to evaluate cognitive load using this approach, very few can be found that have focused on programming comprehension especially with students learning to program for the first time. This study will develop a suite of programming tasks, designed to induce different levels of cognitive load (low to high). The programming tasks will be completed by novice programmers whilst a remote eye tracking system monitors pupil dilation. It is hypothesised that, once environmental factors (ambient light etc) have been controlled, programming tasks designed to induce a high level of cognitive load will result in dilations of the pupils, whilst easier tasks will not result in such a change.

Keywords

Eye tracking, cognitive load, pupillometry, programming comprehension

1. INTRODUCTION

Cognitive science is concerned with how processes of the mind work including the mental processes behind learning, memory and problem-solving [1]. Cognitive load is the total amount of mental activity (processing effort) imposed on working memory at any given time.
Working memory is one of three types of memory that humans have. Long term memory holds a permanent and large body of knowledge and skill. This would include many everyday things that we take for granted such as knowing how to walk or ride a bicycle, how to perform mathematical operations like addition and subtraction, and even recalling simple things such as where we live [1]. Sensory memory is a short term memory which acts as a buffer for the stimuli received through the five senses. However, of most importance to cognitive load is working memory. Working memory provides an interface between long term memory and the senses [1]. It is the section of memory responsible for our creative and logical thinking and allows us to solve problems including programming comprehension. The goal of this study is to investigate the implications of cognitive load on learning to program.

1.1 Measuring cognitive load

Cognitive load can be measured by performance on primary and secondary cognitive tasks with targeted questionnaires. For example, the NASA-TLX survey attempts to gauge the participants perceived cognitive load [2].Physiological measurements can also be used to determine cognitive load with eye tracking showing considerable value as an inexpensive and effective method of measurement. Of particular importance is the relationship between pupil dilation and cognitive load. Changes in cognitive load cause very small dilations of the pupils and in controlled settings, high-precision pupil measurements can be used to detect small differences in cognitive load at time scales shorter than one second [3]. Studies have focused on four main tasks to elicit cognitive load - object manipulation, reading comprehension, mathematical reasoning and searching [4]. These areas are quite similar to the types of tasks a programmer must do, that is a programmer must set up the program in a logical structure, read previous code written and complete their program. The above four tasks are all included in programming comprehension.

1.2 Programming comprehension and cognitive load

It is well established that learning to program is difficult for many students [5]. Research largely indicates that although syntax can be problematic the most significant problem is being able to break down a problem into its component parts and express the component parts in programming code [6] [7] . Programming, like problem solving, relies heavily on working memory, where only a few items can be stored and even fewer can be processed and thus this can lead to the high levels of overload. However, very few studies have attempted to examine this. The most closely related work in this area found was a study that attempted to reduce cognitive load while learning to program by breaking down the program into a concept map to allow the programmer to visualise the code in a more graphical manner. The method was found to reduce cognitive load by administrating the NASA-TLX survey to the student [6].

1.3 Task-evoked pupillary response

When someone performs a task such as recallinginforma-

*Senior / Corresponding Author

tion from memory, paying close attention or just thinking hard their pupils tend to dilate slightly [8].After a few seconds of completing the task, the persons pupils' return to their normal state [8] [9] [10]. This response is called Task-Evoked pupillary response (TERP). This response is involuntary and associated with a broad set of cognitive functions [10] (doing mental arithmetic [11], sentence comprehension [12] and letter matching [13]). As novice programmers will do all of these cognitive functions while programming, TERP would appear to be an appropriate measure.

2. PROPOSED METHODOLOGY

This paper proposes to develop a model to describe the relationship between cognitive load and programming comprehension.
The goal of this experiment is to answer the following research question:

- How well does pupil size correlate with the mental workload that programming imposes on a student?

As the focus of this study is on novice programmer's key topics within a first year third level introductory to programming course will be used. Students undertaking their first year of study in a computing degree (that involves programming) will be invited to participate in the study. A set of tasks will be prepared based on core concepts from the first year course. The student will be asked to determine the outcome of a given code snippet. Each code snippet will contain only one core concept of programming (if statements, while loops, for loops etc). An equal number of easy and difficult programming tasks will be created. At the start of the experiment the participant will be asked to fixate on a particular spot on the screen. This will give a baseline reading for the participant. After the participant has read and understands the instructions they will be begin by completing practice trials. Then the participant will begin the real trials. After completing the trial, a NASA-TLX survey will be given in order for the students to gauge their own perceived cognitive load.

2.1 Measurements

Along with the NASA-TLX survey, the participants pupil data, on-screen activities and time completion will be recorded. The percentage change in pupil size(PCPS) will be computed for each instance. The average PCPS will then be computed over the entire experiment. Inter-and intra- participant eye gaze patterns will be examined and evaluated.

3. CONCLUSIONS

The basis of this model is heavily grounded in research conducted in cognitive workload in Human-Computer Interaction. This model however builds on that research and attempts to build a relationship between cognitive load and programming comprehensions. Novice programmers are of particular interest in this study. If the level of understanding of core concepts in first year computer programming can be increased then the level of continuing students may increase.

4. ACKNOWLEDGEMENTS

This work was funded by the Irish Research Council Postgraduate Scholarship 2015.

5. REFERENCES

[1] Dale Shaffer, Doube Wendy, and Juhani Tuovinen. Applying Cognitive Load Theory to Computer Science Education. *Joint Conf. EASE & PPIG*, (April):333–346, 2003.

[2] Eija Haapalainen, SeungJun Kim, Jodi F. Forlizzi, and Anind K. Dey. Psycho-Physiological Measures for Assessing Cognitive Load. *Proceedings of the 12th ACM international conference on Ubiquitous computing*, pages 301–310, 2010.

[3] Jeff Klingner. Measuring cognitive load during visual tasks by combining pupillometry and eye tracking. *Perspective*, (May):130, 2010.

[4] Muhammed Yousoof and Mohd Sapiyan. Cognitive Load Measurement in Learning Programming Using NASA TLX rating scale. pages 235–245.

[5] Susan Bergin and Ronan Reilly. Programming: factors that influence success. *ACM SIGCSE Bulletin*, 37:411–415, 2005.

[6] Muhammed Yousoof, Mohd Sapiyan, and Khaja Kamaluddin. Reducing cognitive load in learning computer programming. *World Academy of Sciene, Engineering and Technology*, 12(March):469–472, 2005.

[7] Jackie O'Kelly, Susan Bergin, S. Dunne, Peter Gaughran, John Ghent, and Aidan Mooney. Initial findings on the impact of an alternative approach to Problem Based Learning in Conputer Science. *Pleasure by Learning*, 2004.

[8] Shamsi T Iqbal, Xianjun Sam Zheng, and Brian P Bailey. Task-evoked pupillary response to mental workload in human-computer interaction. *Extended abstracts of the 2004 conference on Human factors and computing systems CHI 04*, page 1477, 2004.

[9] Shamsi T Iqbal and Pd Adamczyk. Towards an index of opportunity: understanding changes in mental workload during task execution. *Conference on Human Factors in Computing Systems*, pages 311–320, 2005.

[10] Jeff Klingner, Rakshit Kumar, and Pat Hanrahan. Measuring the Task-Evoked Pupillary Response with a Remote Eye Tracker. *Proceedings of the 2008 symposium on Eye tracking research & applications.*, 1(212):69–72, 2008.

[11] Eckhard H. Hess. Attitude and pupil size. *Scientific american*, 1965.

[12] Marcel A. Just and Patricia A. Carpenter. The intensity dimension of thought: Pupillometric indices of sentence processing. *Canadian Journal of Experimental Psychology*, pages 310–339.

[13] Jackson Beatty and Brennis L Wagoner. Pupillometric signs of brain activation vary with level of cognitive processing. *Science (New York, N.Y.)*, 199(4334):1216–1218, 1978.

Experts vs Novices in programming: "Who knows where to look?"

Position paper

Pavel A. Orlov
1) University of Eastern Finland
School of Computing
P.O. Box 111, FI-80101,
Joensuu, FINLAND
2) Peter the Great St. Petersburg Polytechnic University
Department of Engineering Graphics and Design
195251, Polytechnicheskaya,
St.Petersburg, Russia
paul.a.orlov@gmail.com

ABSTRACT

In this position paper I open the discussion about attention and covert attention during source code comprehension. Covert attention plays a role in a selective process during visual perception, that involves central vision and extra-foveal (parafoveal and periphery) areas. Microsaccades were taken into account for the potential analysis as new elements. Finally I suggest some steps for future studies of covert attention during source code comprehension.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; J.4 [**Social and behavioral sciences**]: Psychology; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

General Terms

Theory

Keywords

Eye movements, eye-tracking, covert attention, source code comprehension

1. INTRODUCTION

Differences between experts' and novices' behavior in a professional domain are highly investigated [10]. Expert's task solving concepts are latent even to the expert himself. Understanding the expert's behavior can improve the educational and evaluative processes in a human resources field. Typically, experts solve tasks faster and with less errors. It is interesting to note that experts often could not tell us exactly how they solve problems (for example in chess playing [11]).

2. EYE-MOVEMENTS OF EXPERTS AND NOVICES

There are several theories that explain the differences between experts and novices. Gegenfurtner with colleagues deal with the theory of 1) long-term working memory, 2) the information-reduction hypothesis and 3) the holistic model of image perception [10]. The long-term working memory is based on the idea that there is the limitation of the working memory. And in this limited place humans store the mental representations of the visual world [2, 22]. The information-reduction hypothesis focuses on the learnt selectivity of information processing. Haider and Frensch claim that: ”people learn, with practice, to distinguish between task-relevant and task-redundant information and to limit their processing to task-relevant information” [12]. Finally, the holistic model of image perception focuses on the extension of the visual span [13]. Experts, following this theory, draw the initial holistic perception of the stimuli. They encode the scene into hierarchical structural components. This process should explore the information from the parafoveal areas [5].

The main points that characterize the expert's behavior are:

1. Experts' fixations durations are shorter than novices'.
2. Experts make more fixations on task relevant areas and solve the tasks with fewer fixations in general.
3. Experts have a wider perception span, longer saccades and shorter time to first fixation to the task relevant area.

These features of the expert's behavior need more careful explanations. For example, experts have a wider perception span, but what objects from this span do they direct their attention to and how do they select *the main* objects (relevant for task)? There is still a gap in the understanding of the expert's behavior in the field of programming.

After the virtual laparoscopic study Law with colleagues claim that eye movements are a possible factor for experts performing better than novices [14]. There are several possible explanations for that claim, thus I was faced with the following question: if a novice skips the learning process phase, but at the same time uses the same eye movement strategy as an expert, will he become an expert? In other words, do eye movements play a vital role for experts performing better than novices, or do they reflect expert's prior knowledge?

3. WHAT HAPPENS DURING FIXATIONS

In eye-tracking studies of source code reading fixations are used for the attention identification (following eye-mind metaphor [21]) [3]. To be more concrete, visual information is taken in during the fixational pauses between saccades [4]. In these fixational pauses, gaze could scan several lines of source code. The foveal area on the retina collects data from 2° of visual angle. If a subject sits 60 cm before the computer screen, this angle corresponds to a 90 px circle in diameter. There are a lot of source code objects obtained through this foveal area and it is represented in the neurone system of the human. A subject should select which object should be involved in the encoding for source code comprehension, in other words a subject should switch his attention from one object to another during one fixation. We should take into account the size of the foveal span and from this we can conclude, that the area for the information processing will be greater than if we use just the center of gaze fixation.

Following predictive coding theory that suggests generative models the brain might use for perception [9], humans build a full picture of the stimuli on the basis of their experience. The next fixation point is used to verify this picture with the reality, for that the next saccade is planned. The saccade planning process involves information from the out-of-foveal region (especially if the next fixation will be more then 2° from the current).

Reading studies of natural languages showed that readers are able to process parafoveal words [17]. During reading, humans can obtain useful visual information from the area (perceptional span) up to 14–15 letters to right of the fixation [20, 1]. The perceptual span is characterized by size and symmetry [18, 19]. Numerous studies show that the parafoveal information processing ability is one of the characteristics of the expert's behavior in natural language text reading [19, 15].

For the domain of programming, the perceptual span could be both horizontal and vertical (given the vertical layout of program code). That should increase the number of the objects for processing in working memory. Following the definition of the working memory, like "attention to manage short-term memory" [6], experts' ability of effective management of short-term objects prevents overloading of working memory. Being able to read a larger portion of the screen can reduce the load on working memory; perhaps this contributes to the experts' efficiency.

To summarize this paragraph, I suggest that the main focus of the latent process analysis of the source code comprehension should be based on the attention evaluation, not just on the places of the gaze fixations. For some research questions it is enough to assume that these things are the same, but not for all. New evidences that expert programmers use the extra-foveal information effectively (that was not found for programming yet), provide benefits for eye-movement modeling and for educational process.

4. ATTENTION EVALUATION

The visual perception studies show that visual attention plays a central role in the control of saccades. Engbert and Kliegl claim that "a key finding in research about visual attention is that the orientation of attention can differ from the orientation of gaze position. In this case, the term of covert attention is frequently used to indicate this separation, which is typically implemented in experimental conditions of attentional cueing" [8].

The human eye does not fade during the fixation. Small involuntary saccades, called microsaccades, take place on par with drifts and tremor. The role of microsaccades in the perception process is still in the center of research interest. But from the latest trend in the studies the dominant point of view can be deduced: microsaccades are not just noise, they play an important role in visual information processing [8, 7, 16]. And Engbert and Kliegl claim: "... microsaccades can be used to map the orientation of visual attention in psychophysical experiments" [8].

From the perspective of microsaccades analysis, the attention maps of novices and experts should be different. I suggest the potential benefits of such kind of analysis lie in a deeper understanding of source code comprehension of experts and novices. Further research in this field would be very useful in the evaluation of: 1) differences in the direction of microsaccades and comparing it with the direction of "normal" saccades, 2) attention switching between the semantic elements of source code, where microsaccades are directed, 3) the influence of source code elements on the probability and rate.

5. REFERENCES

[1] B. Angele, E. R. Schotter, T. J. Slattery, T. L. Tenenbaum, K. Bicknell, and K. Rayner. Do successor effects in reading reflect lexical parafoveal processing? Evidence from corpus-based and experimental eye movement data. *Journal of Memory and Language*, 79-80:76–96, 2015.

[2] A. D. Baddeley. Short-Term and Working Memory. *The Oxford handbook of memory*, pages 77–92, 2000.

[3] R. Bednarik. *Methods to Analyze Visual Attention Strategies: Applications in the Studies of Programming.* PhD thesis, University of Joensuu, Joensuu, 2007.

[4] B. Bridgeman, A. H. C. Van der Heijden, and B. M. Velichkovsky. A theory of visual stability across saccadic eye movements. *Behavioral and Brain Sciences*, 17(02):247–258, feb 1994.

[5] N. Charness, E. M. Reingold, M. Pomplun, and D. M. Stampe. The perceptual aspect of skilled performance in chess: evidence from eye movements. *Memory & cognition*, 29(8):1146–1152, 2001.

[6] N. Cowan. What are the differences between long-term, short-term, and working memory? *Progress in Brain Research*, 169(07):323–338, 2008.

[7] R. Engbert. Microsaccades: A microcosm for research on oculomotor control, attention, and visual perception. *Progress in brain research*, 154(June 2005):177–92, jan 2006.

[8] R. Engbert and R. Kliegl. Microsaccades uncover the orientation of covert attention. *Vision Research*, 43(9):1035–1045, 2003.

[9] K. Friston. Prediction, perception and agency. *International Journal of Psychophysiology*, 83(2):248–252, 2012.

[10] A. Gegenfurtner, E. Lehtinen, and R. Säljö. Expertise Differences in the Comprehension of Visualizations: A Meta-Analysis of Eye-Tracking Research in

Professional Domains. *Educational Psychology Review*, 23(4):523–552, 2011.

[11] F. Gobet and H. A. Simon. Expert Chess Memory : Revisiting the Chunking Hypothesis. *Memory*, 6:225–255, 1998.

[12] H. Haider and P. a. Frensch. Eye movement during skill acquisition: More evidence for the information-reduction hypothesis. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 25(1):172–190, 1999.

[13] H. L. Kundel, C. F. Nodine, E. F. Conant, and S. P. Weinstein. Holistic component of image perception in mammogram interpretation: gaze-tracking study. *Radiology*, 242(2):396–402, 2007.

[14] B. Law, M. S. Atkins, A. J. Lomax, and C. L. Mackenzie. Eye Gaze Patterns Differentiate Novice and Experts in a Virtual Laparoscopic Surgery Training Environment. In A. T. Duchowski and R. Vertegaal, editors, *Proceedings of the 2004 Symposium on Eye Tracking Research & Applications*, volume 1, pages 41–48, 2004.

[15] C. Y. Leung, M. Sugiura, D. Abe, and L. Yoshikawa. The Perceptual Span in Second Language Reading: An Eye-Tracking Study Using a Gaze-Contingent Moving Window Paradigm. *Open Journal of Modern Linguistics*, 4(05):585, 2014.

[16] S. Martinez-Conde, S. L. Macknik, X. G. Troncoso, and D. H. Hubel. Microsaccades: a neurophysiological analysis. *Trends in Neurosciences*, 32(9):463–475, 2009.

[17] G. W. McConkie and K. Rayner. The span of the effective stimulus during a fixation in reading. *Perception & Psychophysics*, 17(6):578–586, 1975.

[18] K. Rayner. Eye Movements in Reading and Information Processing : 20 Years of Research. *Psychological Bulletin*, 124(3):372–422, 1998.

[19] K. Rayner, M. S. Castelhano, and J. Yang. Eye movements and the perceptual span in older and younger readers. *Psychology and aging*, 24(3):755–60, sep 2009.

[20] K. Rayner, S. J. White, G. Kambe, B. Miller, and S. P. Liversedge. On the processing of meaning from parafoveal vision during eye fixations in reading. *The mind's eye: Cognitive and applied aspects of eye movement research*, pages 213–234, 2003.

[21] E. D. Reichle. Editorial: Computational models of eye-movement control during reading: Theories of the "eye-mind" link. *Cognitive Systems Research*, 7(1):2–3, 2006.

[22] J. J. Todd and R. Marois. Capacity limit of visual short-term memory in human posterior parietal cortex. *Nature*, 428(6984):751–754, 2004.

Readability Metrics for Program Code: How is Reading Ease Reflected in Gaze?

James H. Paterson
Glasgow Caledonian University
Cowcaddens Road
Glasgow G4 0BA, UK
James.Paterson@gcu.ac.uk

Katrin Hartmann
Glasgow Caledonian University
Cowcaddens Road
Glasgow G4 0BA, UK
k.hartmann@gcu.ac.uk

ABSTRACT

This paper describes a range of models that have been proposed for the readability of program code. We consider ways in which eye movements can be used to provide evidence for the validity of these models. It is proposed that gaze metrics based on fixation count and duration may correlate with readability scores. We also consider the prospects for developing a model based on gaze to measure readability. The usefulness to educators of measures of readability is discussed.

Categories and Subject Descriptors

K.3.4 [**Computer and Information Science Education**]: Computer science education, information systems education

General Terms

Experimentation, Human Factors.

Keywords

Eye tracking, code reading, readability, computing education

1. INTRODUCTION

Natural language text can be used to communicate information and ideas, with the intention that the reader should understand the content of the text as a result of reading it. Readability is a useful precursor to understanding, and depends on factors that include the vocabulary, syntax and presentation of the text. Text that is difficult to read is a barrier to understanding of the content. Measurement of text readability is useful in many situations, for example in specifying standards for the way official documents are written. The Flesch Reading Ease Score (FRES) is a widely used metric for readability. It is a simple measure, based on average word and sentence lengths, but despite of (or perhaps because of) its simplicity it has proven to be useful over many years.

Program code is primarily written to provide instructions to the computer. However, it is also a means of communicating the nature of the computation involved to a human reader, for example to another programmer who has to maintain the code. The readability of the code can have an effect on how easy or difficult it is for a programmer to understand the function of the code. Just as it may be possible to write natural text to communicate the same idea in different ways, with different degrees of readability, it is often possible to write code that causes the same function to be performed by the computer in different ways, some more readable than others. A number of measures analogous to FRES have been proposed for software readability [3,5,13]. These will be described in the following section.

Code examples used to teach programming are a special case where code readability is of particular importance [3]. Unlike code written to be part of a functioning computer system, the primary purpose of code examples is to promote understanding of programming concepts, constructs and techniques. Measures of readability may be of value in evaluating the quality of code examples so that educators can be confident that the concept being exemplified is not obscured by code that is difficult to read.

Models of code readability are underpinned by empirical data on human perceptions of readability. We propose that eye-tracking potentially has a valuable role to play in validating these models and more generally in evaluating code readability.

2. SOFTWARE READABILITY MEASURES

Buse and Weimer [5] implemented a tool that assesses readability of programs with a model constructed using machine learning techniques based on ratings of readability of code snippets made by 120 human participants (we refer to this model as BW).

Posnett et al. [13] propose a simpler model (we refer to this model as PHD) that calculates readability from code metrics (Halstead's volume, lines and Entropy). They assert that this model actually outperforms the BW model.

Börstler et al. [3] propose an even simpler model, very similar in concept to FRES, which they call the Software Readability Ease Score (SRES). Like FRES, this score is based on average word length (AWL – defined for software as the average length of lexemes) and average sentence length (ASL – words per statement or block) and is expressed as:

$$SRES = AWL - 0.1ASL$$

It should be noted that SRES does not take into account any factors related to the presentation of the code. Hargis [10] describes levels of readability in which there is a surface level of legibility ("things that affect the reader's eyes") upon which further levels build, up to learnability and doability ("things that affect the reader's mind"). The authors of [3] argue that their measure considers the inherent factors of software readability, corresponding to the higher levels of readability. However, while most natural language text is intended to be read in sequential order, code carries important structural information, for example control flow and class/method definitions, and assimilation of that structure while reading is important for understanding. It is widely considered to be good practice for programmers to pay attention to presentation of code, for example indentation and white space, to make code readable. PHD is slightly more sensitive to presentation in that it includes the total number of lines in the measure, including white space, but does not make any meaningful use of the way structure is highlighted in the code. Similarly, color coding is widely used in code editors but none of the measures described here consider its effect on readability.

CLOZE tests, in which humans are asked to fill in missing elements from text with every *n*th word obscured, have also been long used to measure text readability. Such tests have been proposed in relation to program code as a technique for assessing comprehension [8, 9], although not specifically for measuring code readability.

3. VALIDATING READABILTY MEASURES

How do we know that the models described in section 2 actually measure readability of code in a meaningful way? The BW model is based on human perceptions of readability, rated on a five-point scale for each code snippet. Some correlation is demonstrated between the model and software quality metrics. The PHD model is based on code metrics but is validated using the same empirical data as BW (the authors of [5] make this data generally available). Börstler et al. [3] compare SRES with the other readability models, with a range of measures of software quality and with a metric for the quality of object-oriented example programs for teaching as perceived by human experts. They note in doing so that there is empirical evidence for the success of certain software metrics in measuring maintainability and reliability.

Eye-tracking offers the potential to provide, alongside the perceptions reported by humans, another source of evidence for readability with which to validate readability models and measures. If we can identify characteristics in gaze that are indicative of ease or difficulty of reading code then these could be correlated with the predictions of the models. It is often possible to write the same program, function or method in many different ways, so it should be possible to construct pairs of code examples that are functionally equivalent but which are clearly differentiated by their measured readability using one or more of the models, perhaps similar to the "beauty-and-the-beast" example in [3]. We would then try to identify differences in the eye movements of participants while reading the code. What would we look for in that data? Gaze data contains a record of eye movements during the whole process of code reading and comprehension. The comprehension process involves factors such as strategy, focus and reading order and these are all reflected in the data. We need to consider metrics that we believe to be influenced specifically by readability. Fixation duration may be a useful indicator. Rello et al. [15] note that shorter fixations are associated with better readability while longer fixations can indicate that processing loads are greater, and hence used fixation duration across the whole text as measure to quantify readability in a study on number presentation in text. The number of regressions can also be an indicator of reading difficulty [14].

The BW model captures a range of token features, such as identifier length, number of identifiers, number of key words, and so on, and evaluates the predicting power of these for readability. For example, there is a strong negative correlation with number of identifiers, and, perhaps surprisingly, a very weak correlation with identifier length. Isolating the effect on eye movements of varying individual token features may give further insight into the effect of these and validate, or not, their reported predictive powers. Sharif et al. [16] have studied the effect of identifier style (camel-case and underscore) on accuracy, time, and visual effort and have used a range of measures based on fixation counts and durations. These techniques could potentially be adapted to measure the effects of other token features in code.

As noted in section 2 we are not convinced of the validity of the exclusion of presentation factors from the models. Eye movements may help to determine the effect of, for example, indentation, space and color. Techniques for the presentation of code are generally intended to highlight the structure of code or the purpose of tokens, and may be expected to make it easier, for example, to focus on important features or read in purposeful order. Eye movements which are believed to be indicative of particular comprehension strategies may be more clearly apparent when the code is presented in a way that supports those strategies.

4. MODELING READABILITY WITH GAZE

A further use of eye-tracking would be to adopt a similar approach to the BW model, but to use gaze metrics rather than code features as classifiers. This could be useful particularly if it proves difficult to demonstrate correlation between gaze and code-metrics based models of readability.

The question is: can we predict the readability of code from gaze data obtained from participants reading that code? In the past eye gazing data has been used to predict cognitive states [1]. We want to produce a classifier which given eye tracking data of a person looking at code will predict the readability of the code. To this end we envisage applying statistical and/or machine learning classification techniques. A range of statistical learning techniques, decision trees and forests and possibly deep learning can be applied.

A classifier is a statistical or machine learning technique that allocates a sample to a category on the basis of an observed set of attributes or features [12]. In this case we aim to produce a classifier that outputs a numerical value for readability of the code based on the gaze data. We generate the classifier from a large sample of training data. Each element of training data is gaze data from a participant looking at code together with a measure of the readability of that code. The latter could, for example, be the BW data on perceived readability or an equivalent data set. Given that this process would involve a substantial number of participants for gaze data gathering then these participants could be asked also to rate readability.

It is difficult for classification algorithms to work directly with raw eye gaze data. Therefore in a preliminary step we propose to extract from the sample data a set of attributes that describe gaze behaviour. Examples of such features are fixation, fixation duration, distance between fixations or saccades [1]. We expect there to be 20 – 30 features of gaze behaviour that may be relevant. Any individual feature may or may not be directly related to readability, e.g. maximum gaze duration may indicate difficulties reading the code or it may simply be due to general interest in the implementation or because the participant has lost interest in the code and is still looking at the code while thinking about another matter. We then use standard statistical or machine learning algorithms to produce a classifier.

Once we have produced a set of training data consisting of attributes with classification a classification algorithm will generate a predictor, i.e. an algorithm that automatically classifies gaze data to predict readability. If the output of the classifier is a sufficient match to the true readability we have shown that it is possible to predict readability from gaze data.

Another possibility for generating a readability score is deep learning [11]. The advantage is that deep learning works in a semi-supervised way and can extract new features. The disadvantage is that deep learning requires a relatively large set of training data [7] and at this point it is not clear whether deep learning can be applied given the likely sample size of the data or whether a sufficiently large sample can be generated. Decision trees and decision forest techniques can also be applied as classifiers that work with smaller data sets [4].

One issue to consider when using gaze data to as classifiers, rather than code features, is that this data is not intrinsic to the code. Gaze data is a product of human interaction with the code, and may differ according to the expertise of the participants. Differences have been observed between the eye movements of novices and experts, reflecting the development of new reading strategies as expertise is gained [6]. Another approach would be to use the above techniques to develop a gaze "reading-ease" metric. This could be used to distinguish experts form novices, as experts should probably show more features indicating easy reading.

5. READABILITY IN CODE EXAMPLES

Our primary interest as educators is in ensuring that the code examples we give to students do not hinder understanding the concepts they are intended to illustrate. Further, we may wish to present code examples in specific ways to highlight concepts that we wish learners to pay attention to, for example by using color coding that is different from the usual coding that the editor in an IDE applies to all code, or by annotating code. We are also generally freer than teams developing products are to consider readability as a factor in the choice of programming language we use to teach concepts and techniques.

If we are able to relate gaze to readability of code this could be useful for educator in various ways. We can potentially gain new insights into what makes code readable, particularly for novices, and use these, or guidelines based on them, in the creation of code examples. Measures analogous to FRES could be used in a similar way, with recommendations being made for the readability level of code presented to learners at different stages of their development. Code that exemplifies complex constructs or algorithms is likely to be less readable than simple examples, just as it may be difficult to express complex ideas in very simple natural language. However, measures of readability, perhaps taking into account the formatting of code, such as colour coding, spacing and naming, may help to ensure that we present examples in as readable a form as possible given the complexity of what is to be exemplified. It would also be interesting to measure the readability of code that is commonly presented to learners through examples in textbooks, in a similar way to a previous study of the overall quality of these examples [2].

There may be scope to embed the measurement of readability into learning and development tools. If we can relate difficulty that a learner is having understanding a section of code to their gaze behaviour this would be a very useful tool for IDEs or assisted learning management systems because it offers the opportunity to automatically detect when a student is experiencing difficulties with a particular section of code. This would be individualized measurement of readability, making use of the characteristics that are generally indicative of difficulty of reading to determine whether the individual is experiencing that section of code as difficult to read. The system may then have options to change some aspect of the code or its presentation to guide the reader towards understanding.

6. CONCLUSION

Readability is an important part of code comprehension and eye-tracking has a role to play in understanding what makes code readable, through validation of proposed code-based models or as a tool in its own right for evaluating readability. There are, of course, significant challenges in determining cognitive processes in code reading from gaze data, and separating out indicators of readability as a component of these processes will not be easy to do.

7. REFERENCES

[1] Bednarik R, Eivazi S, Vrzakova H. 2013. A Computational Approach for Prediction of Problem-Solving Behavior Using Support Vector Machines and Eye-Tracking Data. *Eye Gaze in Intelligent User Interfaces Gaze-based Analyses, Models and Applications*. 111-134.

[2] Börstler, J.,Nordström, M. and Paterson, J.H. 2011. On the Quality of Examples in Introductory Java Textbooks. *Trans. Comput. Educ*. 11, 1, Article 3.

[3] Börstler, J., Caspersen, M. E., & Nordström, M. 2015. Beauty and the Beast: on the readability of object-oriented example programs. *Software Quality Journal*, 1-16.

[4] Breiman, L. Random forests. 2001. *Machine Learning*, 45:5–32.

[5] Buse, R. P., & Weimer, W. R. 2010. Learning a metric for code readability.*Software Engineering, IEEE Transactions on*, 36(4), 546-558.

[6] Busjahn, T., Bednarik, R., Begel, A., Crosby, M., Paterson, J. H., Schulte, C., Sharif, B. & Tamm, S. 2015. Eye Movements in Code Reading: Relaxing the Linear Order, In *Proceedings of the 23rd IEEE International Conference on Program Comprehension.* IEEE.

[7] Deep Learning. 2015. Datasets [Online]. Available at: http://deeplearning.net/datasets/ [Accessed: 13th October 2015].

[8] Garner, S. 2005. The CLOZE procedure and the learning of programming. *Proceedings of International Conference on Learning, Granada, Spain,* 5-13.

[9] Hall III, W. E., & Zweben, S. H. 1986. The cloze procedure and software comprehensibility measurement. *Software Engineering, IEEE Transactions on*, (5), 608-623.

[10] Hargis, G. 2000. Readability and computer documentation. *ACM Journal of Computer Documentation (JCD)*, 24(3), 122-131.

[11] Li Deng and Dong Yu. 2014. Deep Learning: Methods and Applications. *Found. Trends Signal Process.* 7, 3–4 (June 2014), 197-387.

[12] Machine Learning, Neural and Statistical Classification Edited by D. Michie, D.J. Spiegelhalter and C.C. Taylor. Ellis Horwood Limited, 1994.

[13] Posnett, D., Hindle, A., & Devanbu, P. 2011. A simpler model of software readability. In *Proceedings of the 8th working conference on mining software repositories* (pp. 73-82). ACM.

[14] Rayner, K., Chace, K. H., Slattery, T. J., & Ashby, J. 2006. Eye movements as reflections of comprehension processes in reading. *Scientific Studies of Reading*, 10(3), 241-255

[15] Rello, L., Bautista, S., Baeza-Yates, R., Gervás, P., Hervás, R., & Saggion, H. 2013. One half or 50%? An eye-tracking study of number representation readability. *Human-Computer Interaction–INTERACT 2013*, 229-245.

[16] Sharif, B, & Maletic, J. 2010. An eye tracking study on camelcase and under_score identifier styles. In *Proceedings of the 2010 IEEE 18th International Conference on Program Comprehension* (ICPC '10). IEEE Computer Society, Washington, DC, USA, 196-205.

What are programmers looking for?

Vera Solomonova
St. Petersburg State Polytechnic University
Department of Engineering Graphics and Design
195251, Polytechnicheskaya,
St.Petersburg, Russia
vera_solomonova@mail.ru

Pavel A. Orlov
1) University of Eastern Finland
School of Computing
P.O. Box 111, FI-80101,
Joensuu, FINLAND
2) St. Petersburg State Polytechnic University
Department of Engineering Graphics and Design
195251, Polytechnicheskaya,
St.Petersburg, Russia
paul.a.orlov@gmail.com

ABSTRACT

Source code comprehension usually classifies as a reading process. This classification is based on the fact, that source code includes words from the English language. But to evaluate and understand the comprehension process it is not enough to use methods and frameworks from reading studies in the natural-language field. Here we discuss the potential of using a different metaphor for source code reading – visual searching. Finally, we suggest the explication of basic terms from the visual searching domain for source code reading, like target, target's path and distractors.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

General Terms

Reading, searching

Keywords

Visual strategies, searching, task solving, source code comprehension

1. SEARCHING STRATEGY VS. READING PATTERNS.

Programmers during their usual work, spend a lot of time to read source code and to search troubleshooting and solutions. As it may seem at first glance, the work with source code has a lot of similarity with reading texts in a natural language. Source code as a plain text uses the same characters, the same words and letters. Both of these texts are read in general from left to right top down. It is not usual for programming experts, but the source code can be really read line by line like a work of art [1]. But is the reading process really going on like this? Do programmers actually read the source code?

Busjahn with colleagues show that the source code can be studied following natural-language text frameworks, but not without their review. For example, they claim: "...[source code] element frequency is not a relevant factor in the variability of first fixation duration and first dwell time during code reading..." [2].

What does actually happen when the programmer *reads* the source code? We want to offer the correspondence with the visual research process, consider the problem of source code comprehension from this point of view. In most cases, a programmer, before he starts to read, already has a question (problem) associated with the code. For example, a programmer does not read the entire array line by line, but searches pieces of the relevant visual information, like a scanner, not like a reader. This idea corresponds to the information-reduction hypothesis, that suggests that experts (in a various domains) select *the main* pieces of information to process, they do not process all information from the visual stimuli [4, 3]. This selective behavior is difficult to fit with natural-language reading.

If we assume, that a programmer's goal is to find the answer by source code comprehension, then a source code should contain some targets (that are significant for the decision making) and distractors (that noise for searching). It is a challenging task to find direct and strong correspondence targets and distractors from the visual searching field. In other words, a programmer does not look for the concrete character or symbol in a source code.

Let us bring the possible structure of the source code comprehension from the visual searching perspective:

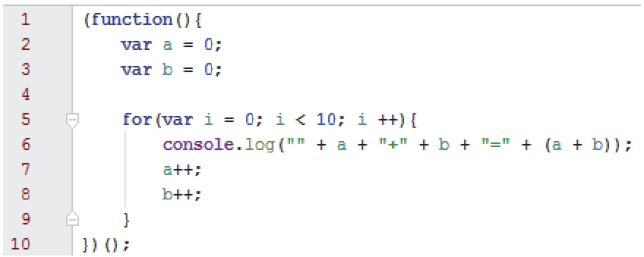
1. Target(s) – one or several places in a source code, that play a key role for task solving.
2. Target's path – the visual path to the target.
3. Distractors (noise) – source code elements, all build-in syntactic constructions (like: *if, else, for, while, ()*, {}, *etc.*).

Following this structure, a programmer should find the target along a target's path, skipping distractors. This metaphor corresponds more to the passage of a maze, than natural-language reading.

2. ILLUSTRATIVE EXAMPLES.

Let us consider examples for the hypothesis described above. For example, we have a code fragment (Figure 1).

What are the target, target's path and distractors (noise) here? According to our hypothesis, if there is a "target", there must be a question. Suppose that the question is this: what will be the output of this source code? The answer is obvious, expressions like: "0 + 0 = 0". For this example, the concept of target, target's path and distractors (noise), in our opinion, can be interpreted as follows:



```
1   (function(){
2       var a = 0;
3       var b = 0;
4
5       for(var i = 0; i < 10; i ++){
6           console.log("" + a + "+" + b + "=" + (a + b));
7           a++;
8           b++;
9       }
10  })();
```

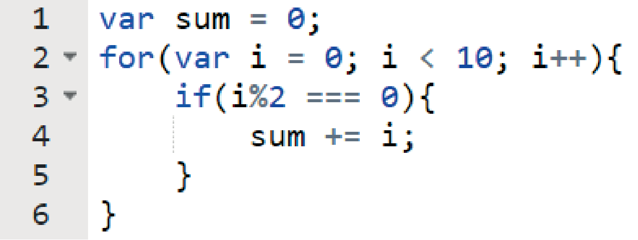
Figure 1: Code fragment (JavaScript).

Target: We assume that this is the sixth line. The line writes the output to the console.

Target's path: the path is likely to pass through the variable declaration, the terms of the loop, the loop body.

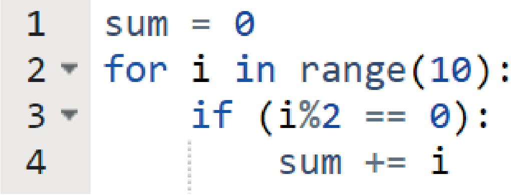
Distractors (noise): in this case - the function words *function*, *for*, *console.log*, and {}, *()*.

Now let's consider this: we have two pieces of code that are functionally identical (see Figures 2 and 3). The first piece of code is written in JavaScript, the second in Python. These examples illustrate our understanding of noise. It is easy to notice that the second code fragment has much fewer lines, but it does not make it less readable. From this observation we can indeed conclude that most of the arrays of source code can contain so-called noise.



```
var sum = 0;
for(var i = 0; i < 10; i++){
    if(i%2 === 0){
        sum += i;
    }
}
```

Figure 2: Code fragment (JavaScript).



```
sum = 0
for i in range(10):
    if (i%2 == 0):
        sum += i
```

Figure 3: Code fragment (Python).

The above examples are not yet a proof for our point of view, but allow to consider analogies between reading source code and visual search. Only a series of experiments can bring clarity to the issue.

3. REFERENCES

[1] T. Busjahn, R. Bednarik, A. Begel, M. Crosby, J. H. Paterson, B. Sharif, S. Tamm, Busjahn Teresa, Bednarik Roman, Begel A, M. Crosby, Peterson Jim, Schulte Carsten, Sharif Bonita, and S. Tamm. Eye Movements in Code Reading : Relaxing the Linear Order. In *International Conference on Program Comprehension (ICPC'15)*, Italy, Florence, 2015.

[2] T. Busjahn, R. Bednarik, and C. Schulte. What influences dwell time during source code reading?: analysis of element type and frequency as factors. In *Proceedings of the Symposium on Eye Tracking Research and Applications*, pages 335–338, Safety Harbor, Florida, 2014. ACM.

[3] A. Gegenfurtner, E. Lehtinen, and R. Säljö. Expertise Differences in the Comprehension of Visualizations: A Meta-Analysis of Eye-Tracking Research in Professional Domains. *Educational Psychology Review*, 23(4):523–552, 2011.

[4] H. Haider and P. a. Frensch. Eye movement during skill acquisition: More evidence for the information-reduction hypothesis. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 25(1):172–190, 1999.

Studying Various Source Code Comprehension Strategies in Programming Education

Jozef Tvarozek, Martin Konopka, Pavol Navrat, Maria Bielikova
Slovak University of Technology in Bratislava, Faculty of Informatics and Information Technologies
Ilkovičova 2, 842 16 Bratislava, Slovakia
{jozef.tvarozek, martin_konopka, pavol.navrat, maria.bielikova}@stuba.sk

ABSTRACT

Source code comprehension is a fundamental task of software development. However, source code is not just a free-flowing text, but a structured document which may be difficult to comprehend. We are interested in understanding how novice programmers look at source code in relation to its structure, how they comprehend it, and how to use this information in order to improve the learning process. We study how programmers (university students) read source code in different types of programming tasks, and use eye tracking to reveal comprehension strategies in a classroom equipped with such devices during regular classes.

Categories and Subject Descriptors

K.3.2 [**Computers and Education**]: Computer and Information Science Education—*Computer science education*

General Terms

Experimentation, Human Factors, Measurement

Keywords

Program comprehension, Programming education, Code reading, Eye tracking, Debugging

1. INTRODUCTION

Eye tracking technology provides interesting insights into human cognitive processes in certain tasks. Text-based and reading tasks are an ideal subject for study, and program comprehension is a special type of reading task [1, 2, 3]. Program comprehension in education in particular offers opportunities to not only study how humans think when performing a task of program comprehension, but also to improve the overall learning process through improving learning material based on eye tracking data.

In this paper we discuss a method to study the effect of different types of tasks on source code comprehension strategies. For that we use the eye tracking devices Tobii X2-60 combined with our online learning system Peoplia [5] used by undergraduate students in programming courses. Aim of our study is to evaluate how students gaze at source code for different programming tasks and their variations.

2. RELATED WORK

The importance of source code comprehension, as a fundamental task of software development [3], has motivated researchers to study developers interacting with code, whether with interviews [3], transcribing activity, collecting interaction data, or mostly eye tracking [1, 4, 7]. Source code reading strategies affect developers' success rate of understanding tasks [1] or debugging [7]. Eye tracking technology is used for recording eye movements [2], or for, e.g., pupil size [4], and then for describing actual comprehension strategies. Different models that conceptualize program comprehension combine the source code's external representation, developer's cognitive structure and assimiliation process. These models were also studied from a computer science education perspective [6]. We aim to study students with different levels of programming experience.

3. TYPES OF PROGRAMMING TASKS

In our work we focus on studying how novice programmers, e.g., students, comprehend source code during programming tasks of different types. We attempt to identify relations between types of those tasks, developers' experience, and their results. We distinguish between these types of programming tasks and their respective subtypes:

- Static code reading – students comprehend a source code fragment (no editing) to answer questions about:
  - Output – given the program's code and input, identify the respective output.
  - Input – given the program's code and expected output, identify its correct input.
  - Bug – identify a bug in the given program code.
- Programming tasks – students implement a program in a single source code file, during which we focus on:
  - Dynamic code reading – students read code, scroll within the file, and change its contents.
  - Task description reviewing – students repeatedly check the task description; they focus back and forth between code and task description.
- Combination of development with static code reading – unfinished source code is displayed and disabled for editing until their first edit in order to:
  - Fix a bug – the code contains one or more bugs.
  - Complete solution – the code is an incomplete or buggy solution to a non-trivial problem.
- Code reading for debugging – students encounter different errors to fix during programming tasks:

- Compile-time error – given the compiler output, students locate the cause of the problem in the source code.
- Output error – program's output is not as expected, students search for the problem.
- Run-time error – stepping through the code to locate a bug, e.g., null reference exception.

In our setting, students complete these tasks as exercises in university programming courses. While *programming tasks* are often used in programming education, *static code reading* and question answering tasks are more suitable for novice programmers who are not yet able to write their programs from scratch. As the course progresses, we introduce more complicated programming task types.

4. COMPREHENSION STRATEGIES

Different types of *static code reading* tasks involve different comprehension strategies. When identifying *output*, students may follow control flow from the beginning of the code. Identifying *input* is, however, different, as it requires the students to understand control flow, then retrace back [2], and follow data flow. We aim to give students different inputs, and let them identify the correct one for the output to see how they comprehend source code.

In programming classes, students new to programming often fail to interpret compiler errors (some because of language barrier of English as a second language). Rather than using suggestions given by the compiler, they switch back to code and look for problems. We expect to find different comprehension strategies after reading compiler output.

Additionally, we are interested how students read task descriptions. Some read the description only at the beginning of the task, others (who may be struggling with the problem) check back often during their work. In order to determine student's experience level, we also track students' success rate in previous exercises during the course.

To sum up, we are interested in answering these research questions:

- How do students' source code comprehension strategies vary for different instances (e.g. outputs) of the same exercise, e.g., of the input identification task type?
- Does students' success rate correlate with the way how they review the problem description?
- How do students read and understand compiler errors?

In order to study the variation in comprehension strategies using eye tracking, we aim to analyze the particular type of how a programmer follows the flow in code: data flow vs. control flow vs. only following program's surface features. We also expect the quantitative aspects of reading (e.g. tokens per minute, regressions) to contribute less to detection of the type of flow the programmer uses and more to the ability level of the programmer. When studying the first research question, we expect systematic approaches of assimilation process [6], i.e., the bottom-up and top-down models, to appear the most.

5. EXPERIMENTS SETUP

We aim to evaluate the comprehension strategies within undergraduate programming courses during regular classes in our classroom laboratory equipped with eye tracking technology. In programming courses at our faculty, students use our online learning system Peoplia [5] to work on exercises of *programming task* type in C language, as well as of *static code reading* type in pseudo-code or C language. The system encompasses programming exercises viewing, source code editing, compiling, testing, and submitting answers for evaluation. Currently, we work on extending the system with eye tracking to record student's gaze on source code, problem's description, compiler output, etc.

As we aim to conduct experiments during regular classes, we expect students (experiment subjects) to be of different levels of expertise. We are not able to randomize assignment of students into classes, but they mostly work independently of each other.

6. CONCLUSIONS

Opening possibilities of using eye tracking devices in recent years attract our attention to study source code reading and comprehension strategies in programming education. As opposed to other works in this field, we set up experiments to take place during regular programming classes. Based on the hierarchy of task types, we proposed three research questions to study code comprehension strategies for different tasks, or for minor variations of the same task.

7. ACKNOWLEDGMENTS

This work was partially supported by the Scientific Grant Agency of the Slovak Republic, grants No. VG 1/0752/14, VG 1/0646/15, and it is the partial result of the project "University Science Park of STU Bratislava", ITMS 26240220084, co-funded by the ERDF.

8. REFERENCES

[1] R. Bednarik and M. Tukiainen. An eye-tracking methodology for characterizing program comprehension processes. In *Proc. of the Symp. on Eye Tracking Research & Appl.*, ETRA '06, 125–132, 2006. ACM.

[2] T. Busjahn, R. Bednarik, A. Begel, et al. Eye movements in code reading: Relaxing the linear order. In *Proc. of the 2015 IEEE 23rd Int'l. Conf. on Program Comprehension*, ICPC '15, 255–265, 2015. IEEE.

[3] C. L. Corritore and S. Wiedenbeck. An exploratory study of program comprehension strategies of procedural and object-oriented programmers. *Int'l. J. Hum.-Comput. Stud*, 54(1):1–23, Jan. 2001.

[4] T. Fritz, A. Begel, S. C. Müller, et al. Using psycho-physiological measures to assess task difficulty in software development. In *Proc. of the 36th Int. Conf. on Software Eng.*, ICSE 2014, 402–413, 2014. ACM.

[5] P. Navrat, J. Tvarozek. Online programming exercises for summative assessment in university courses. In *Proc. of 15th Int'l. Conf. on Systems and Technologies.*, CompSysTech '14, 341–348, 2014. ACM.

[6] C. Schulte, T. Clear, A. Taherkhani, et al. An introduction to program comprehension for computer science educators. In *Proc. of the 2010 ITiCSE Working Group Reports*, ITiCSE-WGR '10, 65–86, 2010. ACM.

[7] B. Sharif, M. Falcone, and J. I. Maletic. An eye-tracking study on the role of scan time in finding source code defects. In *Proc. of the Symp. on Eye Tracking Research and Appl.*, ETRA '12, 381–384, 2012. ACM.

Call for papers 2015 – Models to data

Eye Movements in Programming 2015, emipws.org

Models to Data

3rd International Workshop at the **15th ACM Koli Calling Conference on Computing Education Research**

Location: University of Eastern Finland, Joensuu, Finland, November 22-24, 2015

The third international workshop on Eye Movements in Programming will focus on advancing the methodological, theoretical and applied aspects of eye-movements in programming. Previously we focused on understanding expert (2013) and novice (2014) source code reading behaviors, and we launched the first distributed data collection in this discipline. By analyzing the pre-shared datasets and discussion of bottom-up data-to-models approaches, the workshop produced new methods, tools and understanding of source code reading gaze patterns.

The third edition aims at investigating the opposite direction. We invite contributions departing from theoretical perspectives and grounds to present new hypotheses about gaze behaviour in comprehension, debugging, and other tasks of programming. These may include, but are not limited to, affective computing in programming, vision based models, readability, and new theories of program comprehension. Contributions are expected to present implications to industrial programming practice or programming education.

In preparation for the workshop session, participants submit a short paper (about two pages) discussing 'how various aspects of comprehension leave their trace in gaze'. All submissions will undergo a light review by the program committee. A technical workshop report including the position papers will be published. At the workshop we will discuss the proposed operationalizations of comprehension and how to validate them against the data gathered during the Distributed Collection of Eye Movement Data in Programming (http://bit.ly/emip_data). Furthermore, we will prepare a joint publication with the results.

To participate send an email to Roman Bednarik (roman.bednarik@uef.fi). Participation is independent of attending ACM Koli Calling. Workshop web-site: emipws.org.

Keynote speakers:
Dario Salvucci
Tuomo Häikiö

Important dates:
– deadline for position papers: **October 14, 2015**
– feedback to authors: November 1, 2015
– get together: Sunday, November 22, 2015
– workshop: November 23-24, 2015

Workshop organizers:
Roman Bednarik (University of Eastern Finland)
Teresa Busjahn, Carsten Schulte, Sascha Tamm (Freie Universität Berlin)

Program committee:
Andrew Begel, Microsoft Research, USA
Sven Buchholz, FH Brandenburg, Germany
Martha Crosby, University of Hawaiʻi at Mānoa, USA
Bonita Sharif, Youngstown State University, USA
Jozef Tvarozek, Slovak University of Technology, Slovakia

Position paper (about 2 pages):

Invited Topics: Models to data / Theoretical foundations of attention in programming/ How comprehension leaves its trace in gaze

The aim of the position papers is to discuss aspects of a cognitive model or process, a strategy, a certain element of program comprehension, etc. The arguments should be derived from literature (psychology, computer science etc.) or experience (e.g. teaching novices, working with programmers in industry).

Authors are encouraged to describe the model and operationalize it, so it is possible to validate the occurrences of this particular visual behavior in actual gaze data. At the workshop session, we will discuss how to evaluate the presented models on the dataset from the Distributed Data Collection; the dataset includes over a hundred of programmers of different expertise from total novice to expert with many years of professional experience reading Java (and Python).

Central questions:

- Given your model, who looks where for how long in what order? What patterns do you expect?
- Who: Novices or experts
- Where: Region on any abstraction level, e.g. signature – body, code categories like keywords and identifiers
- How long: e.g. a brief scan to gain an overview vs. focused reading
- What order: e.g. reading starts at the first line or the main-method

Format of the submission: ACM 2 column format – www.acm.org/sigs/publications/proceedings-templates (Option 2 for LaTeX)