Eye Movements in Programming:
Spring Academy 2017

Sascha Tamm, Roman Bednarik, Teresa Busjahn,

Carsten Schulte, Hana Vrzakova, Lea Budde (Eds.)

TR-B-17-02
May 2017

Welcome to the proceedings of the Fourth International Workshop on Eye Movements in Programming, the EMIP 2017 Spring Academy

The EMIP 2017 workshop was the fourth in the series of meetings around the topics related to theories, applications, and methods in eye-movement tracking in programming. The previous workshops were held in Joensuu, Finland (2013 & 2015), and in Berlin, Germany (2014).

The 2017 workshop took place at Freie Universität Berlin, Germany, and started with a social gathering on March 15, 2017. A total of 25 participants attended the workshop, and one presentation was delivered remotely.

The program consisted of a keynote, talks, interactive sessions, and demonstrations. We had the pleasure to listen to the keynote talk of Jochen Laubrock, entitled “Eye Movements as Measures of Cognitive Processing of Text and Image when Reading Comics” and enjoyed the demonstration of eye-tracking analysis tools presented by Markus Plank from SMI. The submissions underwent a light review process judging their suitability for inclusion, or were directly invited by the program committee.

We would like to thank all participants and associated partners for their contribution towards the workshop and help with growing the EMIP academic and industrial networks.

Sascha Tamm, Roman Bednarik, Teresa Busjahn, Carsten Schulte, Hana Vrzakova, and Lea Budde
emipws.org

Contents

Investigating Eye Movements in Natural Language and C++ Source Code – A Replication Experiment

Patrick Peachock and Bonita Sharif
Department of Computer Science and Information Systems
Youngstown State University
Youngstown, Ohio 44555 USA
prpeachock@student.ysu.edu and bsharif@ysu.edu

ABSTRACT

Is there an inherent difference in the way programmers read natural language text compared to source code? Does expertise play a role in the reading behavior of programmers? We conduct a controlled experiment with novice and non-novice programmers while they read small short snippets of natural language text and C++ source code. This study is a replication of an earlier study by Busjahn [1] but uses C++ instead of Java source code. The study was conducted with 33 students, who were each given ten tasks: a set of seven programs, and three natural language texts. Using several linearity metrics presented in an earlier study [1], we analyze the eye movements on source code and natural language. The results indicate that novices and non-novices both read source code less linearly than natural language text. We did not find any differences between novices and non-novices between natural language text and source code. Our findings are discussed with respect to Busjahn's study along with future work.

CCS CONCEPTS

• **Human-centered computing ~ Empirical studies in HCI**

KEYWORDS

eye tracking study, C++, program comprehension, natural language

1 INTRODUCTION

In 2015, Busjahn et al. conducted an eye tracking study on how people read code vs. natural language text [1]. They found that novices read source code less linearly than the natural language texts. In addition, the experts were found to read code even less linearly than the novices. In this paper, we report on a replication of the Busjahn et al. study because we wanted to determine if their results still hold for C++ on a different sample.

2 EXPERIMENTAL DETAILS

The main purpose of the study was to compare the reading and comprehension of natural language text and C++ source code. The variables are mentioned below.

Independent Variable: Type of stimulus (source code and natural language text).
Dependent Variables: A set of linearity measures from Busjahn et al. [1]. The linearity measures include:

- Vertical Next Text: The percentage of forward saccades that either stay on the same line or move one line down.
- Vertical Later Text: The percentage of forward saccades that either stay on the same line or move down any number of lines.
- Horizontal Later Text: The percentage of forward saccades within a line.
- Regression Rate: The percentage of backward saccades of any length.
- Line Regression Rate: The percentage of backward saccades within a line.

Element Coverage (ratio of words looked at) and Saccade Length (distance between each successive pair of fixations) were used to measure the differences between non-novices and novices. The linearity measures were used to analyze reading styles between source code and natural language text.

Participants and Tasks: Thirty-three participants were recruited from the introduction to programming and data structures classes. The study was conducted in the sixth and seventh week of classes. Three natural language texts and seven C++source code snippets were used. Fifteen of the participants were female with about 18 of them male. The number of words ranged from 74 to 80 in the natural texts. The lines of code in the snippets ranged from 7 lines up to 21 lines of code. The constructs used in the source code tasks were if/else, average, division, while loop, division, and the mod operator. After each task, the participant was asked to answer a comprehension question such as writing a summary, filling in a blank, or pick a multiple choice answer. They were not able to view the code snippets while answering the question.

Apparatus: A Tobii X60 eye tracker was used to collect the data. Tobii Studio was used to interface with the eye tracker to gather gazes. The IV-T fixation was used as the fixation filter on the raw gaze data.

3 RESULTS AND OBSERVATIONS

In order to analyze the results, we first used *EyeCode*, an AOI generation tool, that takes an image with fixation data, to create line and word areas of interest mapped to the fixation (*https://github.com/synesthesiam/eyecode/*).

A significant difference ($p < 0.001$) was found among novices reading NT vs. SC in all the linearity measures except for regression rate. The saccade length and element coverage was also significantly different ($p < 0.001$ and p=0.03) between source code and natural language among novices. In the case of non-novices, all the linearity measures were significantly different between source and natural language (p < 0.01). No statistical difference was found between novices and experts in any of the linearity metrics between source code and natural language texts for all the tasks combined. See **Figure** 1 for the bar charts showing the average between the linearity measures for novices. **Table** 1 summarizes the dependent variables and their measurements alongside Busjahn et al's study.

Table 1: Summary and Comparison to Busjahn et al. study [1]

Busjahn et al. Study	This Study
Regression Rate (27% SC, 16% NT) Novices 31% Experts 39%	Regression Rate (16% SC, 17% NT) Novices 16% Non-novices 17%
Element Coverage (60% SC, 83% NT) Novices 52% Experts 41%	Element Coverage (34% SC, 30% NT) Novices 33% Non-novices 31%
Novices mostly follow story order vs. execution order of program	Novices mostly follow story order. (execution order was not analyzed)
Difference found in linearity metrics between SC and NT (except line regression rate)	No difference found between SC and NT in any linearity metrics.

We see that there is much disparity between the regression rate and element coverage in our study compared to the Busjahn et al. study. Regression rate is almost double for novices and similarity for experts. In the Busjahn et al. study we did fixation correction manually before running Eyecode on the stimuli. In this study, we did not do any manual fixation corrections and used the data as it was recorded. All data was recorded only after a good calibration was obtained however, it is possible for the above numbers to be different after the corrections are made.

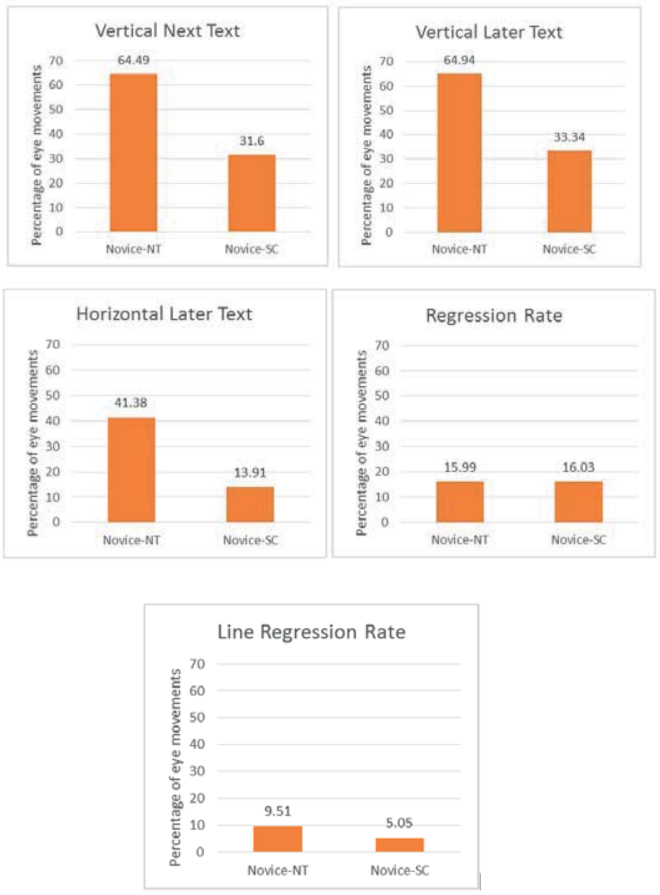


Figure 1: Average linearity measures for all novices.

4 CONCLUSIONS AND FUTURE WORK

The paper did find inherent differences in the way novices read natural language text compared to source code. The same applied to non-novices. However, we did not find a difference (with or without considering task type) for novices and non-novices however, non-novices also read code less linearly than natural language texts.

As part of future work, we are looking into a second phase of the study with the same group of students to determine if eye movements differ at the end of the semester indicating if any learning has occurred.

Acknowledgments
We would like to thank Teresa Busjahn for help with running *Eyecode* on the stimuli. We would also like to thank all the participants that took part in our study.

REFERENCES

[1] T. Busjahn, R. Bednarik, A. Begel, M. Crosby, J. H. Paterson, C. Schulte, *et al.*, "Eye Movements in Code Reading: Relaxing the Linear Order," in *ICPC*, Florence, Italy, 2015, pp. 255-265.

The effect of syntax highlighting on source code reading

Tanya Beelders
Department of Computer Science and Informatics
University of the Free State
Bloemfontein, South Africa
(+27)51 401 9320
beelderstr@ufs.ac.za

Jean-Pierre du Plessis
Department of Computer Science and Informatics
University of the Free State
Bloemfontein, South Africa
(+27)51 401 3701
DuPlessisJL@ufs.ac.za

The main aim of this study is to determine whether inexpensive black-and-white ereaders are a viable alternative for information technology (IT) textbooks. The IT students at the university where the study was conducted programme in Visual Studio®, which automatically provides them with syntax highlighting. However, currently it is not uncommon for textbooks to be provided in black-and-white format which is different to what they are used to seeing in the integrated development environment. The question then arises as to whether this is a disadvantage to students while studying. The prevalence of online resources which could replace hard-copy textbooks could be a better solution. Additionally, textbooks are traditionally expensive, more so than the electronic equivalent, hence it might be financially prudent to use electronic resources. Hence, tablets and ereaders could be used as a medium to deliver academic texts. The question now is whether a cheaper black-and-white electronic device will suffice or whether there is a need for a more expensive device capable of rendering colour.

Therefore the initial phase of the study investigated whether there was any difference in reading behaviour of students who read a piece of source code with or without syntax highlighting ([1]; [2]). Findings suggest that while the code with syntax highlighting is more aesthetically pleasing, it does not make the code easier to read ([1]). Participants reading code without syntax highlighting did however enter a more concentrated reading phase faster than their counterparts who received the code with syntax highlighting ([2]) but once again they did not experience significantly more difficulty reading the code ([2]).

Since syntax highlighting does not appear to affect the reading behaviour of students, it appears to be immaterial (from an objective standpoint) whether the code presented uses syntax highlighting or not.

The current phase of the study is investigating whether the device used has an effect on code reading. Factors such as page breaks, rotation, quality of display could influence the reading behaviour. Therefore, data is currently being collected on a number of devices in order to determine whether the device itself plays a role. This will answer the initial question as to whether resources could be provided on an electronic device and of what stature the device should be.

References

[1] Beelders, T.R. and du Plessis, J-P. (2016a). Syntax highlighting as an influencing factor when reading and comprehending source code. *Journal of Eye Movement Research*, 9(1), 1-11.

[2] Beelders, T.R. and du Plessis, J-P. (2016b). The influence of syntax highlighting on scanning and reading behaviour for source code. In *Proceedings of SAICSIT 2016*, Johannesburg, South Africa.

Scanpath comparison with the use of web applications SMI2OGAMA and ScanGraph

Extended Abstract

S. Popelka
Palacký University Olomouc
17. Listopadu 50, 77146, Olomouc
Czech Republic
stanislav.popelka@upol.cz

J. Dolezalova
Palacký University Olomouc
17. Listopadu 50, 77146, Olomouc
Czech Republic
jitka.dolezalova@upol.cz

ABSTRACT

The contribution will introduce two web tools for analysis of eye-tracking data. The first one – SMI2OGAMA is a simple tool for conversion of data recorded with SMI software. Outputs of this tool can be directly imported into OGAMA – open-source application for analysis of eye-tracking data. OGAMA allows creating a sequence of visited Areas of Interest. This file can be used directly in our second tool called ScanGraph. ScanGraph allows finding participants with a similar strategy of stimuli inspection. The application performs a scanpath comparison based on the String Edit Distance method, and its output is a simple graph. Groups of similar sequences/participants are displayed as cliques of this graph.

KEYWORDS

Eye-tracking, Conversion, Scanpath comparison, Web application

1 INTRODUCTION

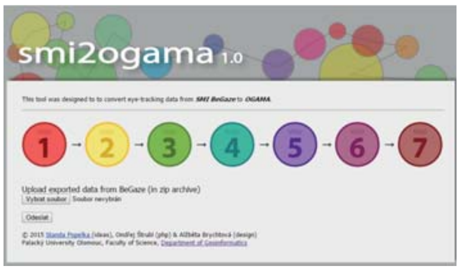
The beginnings of interest in distinctive scanning patterns can be found in the study of Noton and Stark [8], who reported a qualitative similarity in eye-movements when people viewed line drawings on multiple occasions. The scanpath comprises sequences of alternating saccades and fixations that repeat themselves when a respondent is viewing stimuli. One of the most frequently used methods is String Edit Distance, which is used to measure the dissimilarity of character strings. As Duchowski et al. [5] mentions, scanpath comparison based on the String Edit Distance introduced by Privitera and Stark [9] was one of the first methods to quantitatively compare not only the loci of fixations but also their order.

When using String Edit Distance, the grid or Areas of Interest (AOI) have to be marked in the stimulus. The gaze trajectory (scanpath) is then replaced by a character string representing the sequence of fixations with characters for AOIs they hit. A sequence of transformations (insertions, deletions, and substitutions) is used to transform one string to another. Their similarity is represented as the number of transformation steps between two analyzed strings [1].

Open Gaze and Mouse Analyzer (OGAMA) [10] allows to create and export this character string directly as a part of the scanpaths module. The user only needs to draw Areas of Interest or generate a grid and OGAMA displays the character strings and calculates Levenshtein distances between them. This calculation does not take into account the different lengths of strings, so only the strings are used in our tool called ScanGraph. So our tool disregards this calculation and uses just the string for its own calculations. Although OGAMA allows also a recording of the data, in the majority of cases we are using SMI device and software for creation of experiments and recording of the data. Recorded raw data can be converted into OGAMA import format and analyzed here. To do this conversion, we have developed a simple tool called SMI2OGAMA.

2 SMI2OGAMA

SMI2OGAMA is a web application developed in PHP and is available at http://eyetracking.upol.cz/smi2ogama/. Its use is very simple. User exports gaze positions as raw data from SMI BeGaze as single files for each participant. The zip file with these raw data is uploaded into SMI2OGAMA and the conversion to OGAMA format runs on the server. After a couple of seconds, the zip file with converted files is downloaded. In the next phase, user imports these files into new OGAMA project (one by one, because timestamps are not unique). Finally, stimuli images are copied into SlideResources folder of OGAMA. The interface of SMI2OGAMA contains more information about this process including illustration figures (Fig 1).



This tool was designed to to convert eye-tracking data from *SMI BeGaze* to *OGAMA*.

Upload exported data from BeGaze (in zip archive)

© 2015 Standa Popelka (ideas), Ondřej Štrubl (php) & Alžběta Brychtová (design)
Palacký University Olomouc, Faculty of Science, Department of Geoinformatics

Figure 1: Interface of SMI2OGAMA web application

3 SCANGRAPH

ScanGraph (http://eyetracking.upol.cz/scangraph/) uses visualization of cliques in simple graphs (Fig 2). It displays the simple graph with cliques. These cliques show similar sequences based on the input parameter of the degree of similarity. ScanGraph was developed in PHP and C# (Backend) and D3.js (Frontend).

The calculation of similar groups could be described in several steps. At first, the distance matrix is constructed using Levenshtein distance [6], Needleman-Wunsch algorithm [7] or Damerau-Levenshtein distance [3]. However, it is not appropriate to use the absolute value of the distance to comparing the similarities. Hence, the values are normalized. Each of the element a_{ij} of the matrix is divided by the higher value of the length of strings i and j. The next step is creation of the adjacency matrix, which depends on the user's choice of "advised graph", parameter p or percent of possible edges.

"Advised Graph" is a graph with 5 % of possible edges and the corresponding value of parameter p. The second option is a construction of user-defined graph according to parameter p or % of edges. The higher value of $p; p \in \langle 0; 1\rangle$, the higher similarity of the given sequences.

The output of the ScanGraph is a simple graph and listed cliques of this graph (right part of the fig. 2). After clicking on the list, selected clique (a group of similar participants) is highlighted and their character strings are displayed at the bottom. In the case of fig. 2, the parameter was set to 0.6, so groups of participants whose sequences of visited AOIs were similar to at least 60% are displayed. Biggest clique contains three participants. All three of them were Noncartographers (which is indicated by their color).

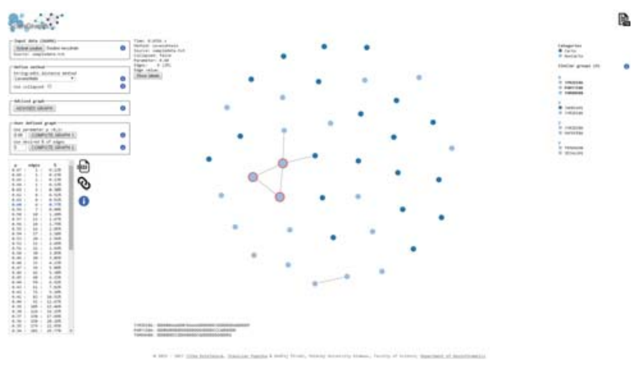


Figure 2: Interface of ScanGraph web application

Detailed information about its principles and possibilities was described in [4]. Since the publication of this article, several updates were performed. Damerau-Levenshtein distance that takes into account the transpositions of adjacent characters. A heuristic algorithm for clique finding is not needed anymore. Cliques are now found with the use of Bron-Kerbosch algorithm [2]. In the first version of ScanGraph, only analysis of one stimulus was possible. Now, the tool allows to read zip files with character strings for multiple stimuli and calculates the similarity of the participants across them.

4 CONCLUSIONS

Two web applications are briefly described in this contribution. The first one – SMI2OGAMA allows to convert data recorded by SMI software into the environment of OGAMA. In OGAMA, character strings representing the sequences of visited Areas of Interest can be generated. These sequences are used as an input into the second described application – ScanGraph. ScanGraph allows calculating similarities of these sequences (and hence of the participant's trajectories) using three algorithms. Results are displayed as cliques of a simple graph.

ACKNOWLEDGMENTS

The preparation of this publication has been supported by the Internal Grant Agency of the Palacký University in Olomouc (project IGA_PrF_2016_008, Advanced monitoring, spatial analysis and visualization of urban landscape).

REFERENCES

[1] Anderson, N. C., Anderson, F., Kingstone, A., Bischof, W. F. (2014) *A comparison of scanpath comparison methods*. Behavior research methods, pp. 1-16.

[2] Bron, C., Kerbosch, J. (1973) *Algorithm 457: finding all cliques of an undirected graph*. Communications of the ACM, 16(9), pp. 575-577.

[3] Damerau, F. J. (1964) *A technique for computer detection and correction of spelling errors*. Communications of the ACM, 7(3), pp. 171-176.

[4] Dolezalova, J., Popelka, S. (2016) *ScanGraph: A Novel Scanpath Comparison Method Using Visualisation of Graph Cliques*. Journal of Eye Movement Research, 9(4), pp. 1-13.

[5] Duchowski, A. T., Driver, J., Jolaoso, S., Tan, W., Ramey, B. N., Robbins, A.(2010) Scanpath comparison revisited. In *Proceedings of Symposium on Eye-Tracking Research & Applications*, ACM, 2010, pp. 219-226.

[6] Levenshtein, V. I. (1966) *Binary codes capable of correcting deletions, insertions, and reversals*. Soviet physics doklady, 10(8), pp. 707-710.

[7] Needleman, S. B., Wunsch, C. D. (1970) *A general method applicable to the search for similarities in the amino acid sequence of two proteins*. Journal of molecular biology, 48(3), pp. 443-453.

[8] Noton, D., Stark, L. (1971*) Scanpaths in saccadic eye movements while viewing and recognizing patterns*. Vision Research, 9//, 11(9), pp. 929-942.

[9] Privitera, C. M., Stark, L. W. (2000) *Algorithms for defining visual regions-of-interest: Comparison with eye fixations*. Pattern Analysis and Machine Intelligence, IEEE Transactions on, 22(9), pp. 970-982.

[10] Voßkühler, A., Nordmeier, V., Kuchinke, L., Jacobs, A. M. (2008) *OGAMA (Open Gaze and Mouse Analyzer): open-source software designed to analyze eye and mouse movements in slideshow study designs*. Behavior research methods, 40(4), pp. 1150-1162.

Evaluating the Readability of Example Programs for Novice Programmers

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

ABSTRACT

This paper describes a proposed study of readability of code examples designed to support learning programming. The study focuses on measuring readability for comprehension. It considers only code structure rather than presentation and relates perceived readability and comprehension to a simple reading ease score and to eye movements during code reading. Characteristics of the example programs to be used in the study are described. Insights are sought into the value of metrics and eye movements in evaluating readability.

Categories and Subject Descriptors

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

General Terms

Experimentation, Human Factors.

Keywords

Computing education, code reading, eye tracking

1. INTRODUCTION

Examples are valuable tools in teaching programming. Example programs work as role models and must therefore be consistent with the principles and rules we are teaching. Börstler et al [3] noted that it is often difficult to find or develop examples that are both well aligned with pedagogical principles and practices and that correctly illustrate the principles and guidelines of the object-oriented paradigm. Example programs should also be understandable by learners with the level of expertise for whom they are designed. An important aspect of understandability is readability. If code has a high level of readability then it is easy for the reader to recognize syntactical elements which leads to the ability establish relationships between those elements and build a mental model of the operation of the code.

Readability of text depends on its content and its presentation, and this is true for code as it is for natural language text. Presentation aspects, such as type size and font, influence readability at the level of legibility. Hargis [10] describes this as the surface level of readability, and described readability for comprehensibility as being at a higher level, influenced more by aspects such as sentence structure and length, and vocabulary.

Börstler et al [4] proposed a simple metric, Software Readability Ease Score (SRES), to measure code readability, analogous to the Flesch reading ease score [8] which is widely used for natural language text. This metric focuses on readability for comprehensibility. They present two versions of an example program ("beauty" and "the beast") which differ largely in overall structure while providing the same functionality – "beauty" is composed from simple functional components, "beast" essentially has a single monolithic component. The SRES score for each is computed and compared to a range of other metrics. SRES contrasts in its simplicity and focus on features analogous to those of natural text with other software readability metrics that have been proposed (see section 2). However, it correlates well with the quality of object-oriented example programs as perceived by human experts and is proposed as a useful tool for helping educators in the selection and development of suitable example programs.

This paper proposes a study of readability of code examples typical of those presented to novices in introductory programming courses, focusing on comprehensibility. Following the philosophy of SRES, surface level presentation will not be the focus of the study, as these are essentially independent of the code itself and can be easily changed if the code is read in a code editor. Arguably, indentation and spacing are also at surface level of readability, and can typically be amended automatically in code editors without changing the code itself (unless they are syntactically significant in the programming language). Comments will also not be considered here. Interestingly, Börstler and Paech [5] studied the influence of comments and found that these influence perceived readability but not comprehension. Finally, differences in readability between different programming languages is not within the scope of this proposal, which is limited to Java, a language that is widely used in introductory courses[3].

Readability studies typically record perceived readability as informed by the participants and comprehension, as measured by the performance on comprehension tasks. Eye movement data will add a further dimension to this study, allowing observation of, for example, total time spent on a reading task, time spent on specific features of the code, and patterns and strategies evidenced in the gaze data, which can be analysed according the coding scheme developed during a previous EMIP workshop [2].

Possible research questions are:

- Does a readability ease score analogous to a natural language reading ease score predict the readability of code examples used in teaching novice programmers?
- Can differences in reading strategy of functionally equivalent code examples be observed in eye movements provide evidence of reading ease?
- What features of such code examples influence readability?

2. OTHER READABILITY MEASURES

Buse and Weimer [6] 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.

Posnett et al. [12] propose a somewhat simpler model that calculates readability from code metrics (Halstead's volume, lines and Entropy). They assert that this model actually outperforms the BW model.

CLOZE tests, in which humans are asked to fill in missing elements from text with every nth 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 [9], although not specifically for measuring code readability.

Dorn [7] introduced a "generalizable" model, which includes visual, spatial, alignment, and linguistic features, and includes aspects such as syntax highlighting, variable naming and structural. Building on that work, Scalabrino et al. [13] present a set of textual features based on source code lexicon analysis and assert that textual features complement structural ones to improve the accuracy of code readability models.

3. CHARACTERISTICS OF EXAMPLES

Examples for novice programmers may be designed to illustrate a specific syntax feature, or a programming technique which makes use of a feature. Listing 1 shows an example method used to illustrate the implementation of an algorithm using an if-else statement in Java. The preamble to the example would be a discussion of the rules which apply to the problem domain (calculating an energy bill) and how these might be expressed as an algorithm. The learner would be expected to understand from this example the way in which the code has implemented the algorithm and the role of the if-else statement in this.

```
public double calculateBill(int current, int previous)
{
   if ((current-previous)<=40)
   {
      return (current-previous)*30.52;
   }
   else
   {
      return (40*30.52)+((current-previous-40)*14.76);
   }
}
```

Listing 1. Version 1 of a method that implements a simple algorithm using an if-else statement and returns the value calculated using the algorithm

This example could equally well be written as in Listing 2, which is functionally equivalent. In this version named constants are used to replace numbers, and there are variables holding intermediate values during the calculations. The latter change has the effect of dividing evaluation of the expressions into evaluating a larger number of simpler expressions and combining the results. There are, of course, other variations possible adopting specific aspects of the approach shown either of the two listings, and a range of versions will be studied.

```
public double calculateBill(int current,  int previous)
{
   double STANDARD_RATE=30.52;
   double DISCOUNT_RATE=14.76;
   int DISCOUNT_LEVEL=40;
   int unitsUsed;
   double bill;
   unitsUsed = current-previous;
   if (unitsUsed<=DISCOUNT_LEVEL)
   {
      bill=unitsUsed*STANDARD_RATE;
   }
   else
   {
      double standard_rate_cost=DISCOUNT_LEVEL*STANDARD_RATE;
      double discount_units=unitsUsed-DISCOUNT_LEVEL;
      double discount_rate_cost=discount_units*DISCOUNT_RATE;
      bill=standard_rate_cost+discount_rate_cost;
   }
   return bill;
}
```

Listing 2. Alternative version of the method shown in Listing 1.

So which, if either, is the "beauty" and which is the "beast"? SRES, like the Flesch score, is based on two metrics – Average Word Length (AWL) and Average Sentence Length (ASL). The interpretation of these has been adapted for code, so that AWL is influenced by lexeme length (for example identifiers, key words), while ASL corresponds to the number of words per statement. The weighting given to AWL is very low in SRES, in contrast to natural reading scores. This seems reasonable as long identifiers don't generally carry the same implication of complexity as long words in natural language, and in fact can carry information to aid readability compared to very short identifiers. Hofmeister et al. [11] focused specifically on identifier length and semantics and showed that identifier names using proper words lead to a faster comprehension, evidenced through defect detection, than identifier names using abbreviated words or single letters. For this reason, SRES might favour Listing 2. However, the short definition given by Boerstler et al. for ASL also refers to number of words per block, which would presumably favour Listing 1. Note that these are observations based on the general philosophy of the metric – the specific strategy used to calculate SRES for the example programs will be based on the detailed description given by Abbas [1].

A set of examples has been developed, drawn from course materials used in an introductory Java programming module. In each case, functionally equivalent versions have been prepared which differ in the structure of the code. The examples do not differ in variable names or method names, except where additional variables or methods are required to accommodate difference in structure. Other presentational aspects, including indentation, line spacing, spacing around operators, positioning of opening braces, and syntax highlighting will be consistent in all examples and the way they are displayed to the participants.

Equivalent examples do differ in aspects of structure. These relate closely to code features that are likely to affect comprehensibility, and which Sedano found to be the most significant, other than identifier names, in a study of perceived code readability [14]:

- Structure of and conditions in control flow statements
- Decomposition by extracting methods, similar to Börstler's "beauty and the beast"

4. CONCLUSION

Readability is an important aspect of code examples developed to support the learning novice programmers. Instructors could benefit significantly from having the means to evaluate, in a simple way, the readability of their example programs, or from having guidelines based on evidence of what aspects of simple code most strongly in similar examples. The study proposed here will evaluate the usefulness of a simple reading ease metric for predicting readability. Eye-tracking and gaze data may have a valuable role to play in providing insights into the process of reading code examples where the same functionality is expressed with different structures and syntax. There are, of course, significant challenges in determining cognitive processes in code reading from gaze data, and it is hoped that valuable insights will be gained through the focus on readability.

5. REFERENCES

[1] Abbas, N., 2010, January. Properties of “good” java examples. *Masters thesis, Umeå University, Umeå, Sweden*.

[2] Bednarik, R., Busjahn, T., and Schulte, C. 2014. Eye movements in programming education: Analyzing the expert’s gaze. *Technical report, University of Eastern Finland, Joensuu, Finland*.

[3] 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.

[4] 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.

[5] Börstler, J. and Paech, B., 2016. The Role of Method Chains and Comments in Software Readability and Comprehension—An Experiment. *IEEE Transactions on Software Engineering*, 42(9), pp.886-898.

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

[7] Dorn, J., 2012. A general software readability model. *MCS Thesis available from (http://www. cs. virginia. edu/~ weimer/students/dorn-mcs-paper. pdf).*

[8] Flesch, R., 1948. A new readability yardstick. *Journal of Applied Psychology*, 32(3), p.221.

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

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

[11] Hofmeister, J.C., Siegmund, J. and Holt, D.V., 2015. Influence of identifier length and semantics on the comprehensibility of source code. *Department of Psychology, University of Heidelberg, Germany*.

[12] 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.

[13] Scalabrino, S., Linares-Vásquez, M., Poshyvanyk, D. and Oliveto, R., 2016, May. Improving code readability models with textual features. In *Program Comprehension (ICPC), 2016 IEEE 24th International Conference on* (pp. 1-10). IEEE.

[14] Sedano, T., 2016, April. Code Readability Testing, an Empirical Study. In *Software Engineering Education and Training* (CSEET), 2016 IEEE 29th International Conference on (pp. 111-117). IEEE.
.

Ambient and Focal Attention During Source-code Comprehension

Position paper

Pavel A. Orlov
Peter the Great St. Petersburg Polytechnic University
Department of Engineering Graphics and Design
195251, Polytechnicheskaya,
St.Petersburg, Russia
paul.a.orlov@gmail.com

ABSTRACT

This preliminary analysis of ambient and focal attention during source-code comprehension uses the methodology of Krejtz et al. [2]. I applied the original formula to the source-code reading tests and found that subjects who program less than one hour per month use ambient-like vision. At the same time, focal vision usage dominates in subjects who do no programming at all. Level of English proficiency and gender play important roles: subjects with a high English level use ambient vision; subjects with a low English level in perform focal vision mode. Females use ambient-like vision more than males. Finally, I open the discuss about the role of ambient/focal 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*

Keywords

Eye movements, eye-tracking, ambient vision, focal vision, source code comprehension

1. INTRODUCTION

Analysis of eye movements during source-code comprehension allows researchers the opportunity to evaluate visual attention. Following the idea that gaze position goes through the attentional visual object, the place of gaze fixation, fixation durations, and scanning pattern are frequently used metrics for analysis. However, the question remains as to why the programmer turns their attention to the next source-code element – that is, why do they select it? Was this selection determined by the source-code structure and "bottom-up" attentional process, or was it a volitional act that corresponds to "top-down" approach?

There are two modes of acquiring information during perception of visual stimuli: exploration and inspection [9, 5]. The scene exploration process is employed when a subject looks for a target in the visual scene and makes short fixations and long saccades to find it. After the *ambient* phase, when a potential target has been found, the subject inspects it and determines whether it is a target. This second stage–*focal vision*–corresponds with long fixations and short saccades' amplitude [10, 8].

The target inspection is a volitional act of the subject; hence, the focal vision corresponds with the "top-down" visual attention process and may be interpreted by the direct control theories of visual attention [9, 4]. Ambient vision is less costly and dominates in parallel search, while focal attention is required for serial search in visual searching tasks [3, 2]. The domination in parallel search can be explained by the "bottom-up" attentional process, when a pop-out effect provides faster target localization [7].

The evaluation of the ambient and focal modes of vision may provide insight into source-code comprehension. However, I did not find such studies in the psychology of programming field.

2. AMBIENT AND FOCAL ATTENTION EVALUATION

I am following the methodology of Krejtz et al. to identify ambient and focal attention from eye movement data [2]. The authors presented the Coefficient K as the difference between zero-score values of the fixation durations and saccadic amplitudes (see Formula 1). The positive and negative values of the Coefficient K indicate focal or ambient viewing, respectively.

$$K_i = \frac{d_i - \mu_d}{\sigma_d} - \frac{a_{i+1} - \mu_a}{\sigma_a},\ such\ that\ K = \frac{1}{n}\sum_n K_i \quad (1)$$

K_i was calculated for each fixation with duration d_i and saccadic amplitude a_{i+1} that goes after the $i^t h$ fixation – where μ_d, μ_a are the mean fixation duration and saccade amplitude, respectively, and σ_d, σ_a are the fixation duration and saccade amplitude standard deviations, respectively, computed over all n fixations [2]. Mean fixation durations and saccade amplitudes were calculated for all subjects and for all stimuli.

3. RESULTS AND DISCUSSION

The analysis was conducted on the open eye-tracking data by the EMiPWS group (in publishing). Data from 209 subjects were reviewed. Fixations, saccades, and blinks were detected by the algorithm described by Engbert and Kliegl [1] in the *saccade* package (the package is available at https://github.com/tmalsburg/saccades) for R programming language [6]. I've included Java programmers only and excluded data from 5 participants because of an anomaly in fixation duration detection.

A two-way ANOVA of the mean Coefficient K revealed that the frequency of programming has a statistically significant effect (F(4, 386) = 5.838, $p < .001$). The effect of the frequency of Java programming was also significant (F(4, 386) = 2.644, p = .033). Not surprisingly, I also found a significant interaction between the frequency of programming and the frequency of Java programming (F(16, 386) = 1.768, p = .034).

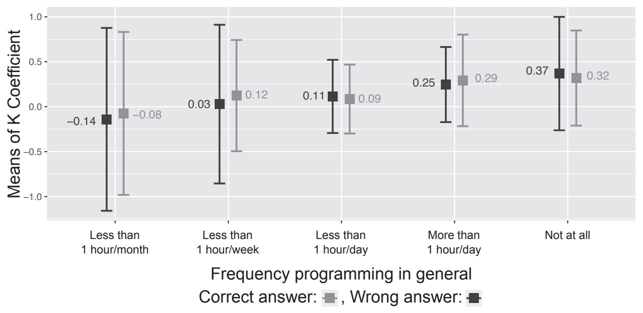


Figure 1: Means of K Coefficient by frequency of programming.

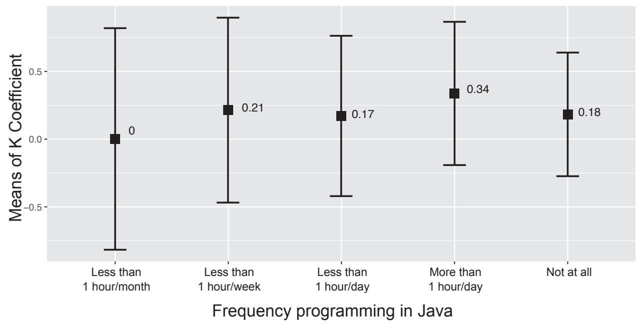


Figure 2: Means of K Coefficient by frequency of programming in Java.

Figure 1 and Figure 2 show that the difference in the means of the Coefficient K is small. However, the figures show a trend in which the K Coefficient increases as programming frequency increases. Unexpectedly, I found that when subjects do not program at all, their Coefficient K is quite high while subjects who program less than one hour per month have Coefficients K of less than zero. Subjects who do not program at all still solve tasks correctly and use mostly focal vision, which is similar for subjects who program more than one hour a day. These two groups probably use the direct control of visual attention for different purposes.

A more ambient-like mode of vision was used by subjects who are proficient in the English language (a one-way ANOVA: F(2, 408) = 14.62, $p < .001$). Figure 3 shows that subjects with a high English level have a K Coefficient of about zero, while subjects with a low English level have a higher K Coefficient. This result may be explained by that fact that the stimulus was formed using a mix of Java programming language and English. Subjects with have higher English levels obtain the semantic information easily.

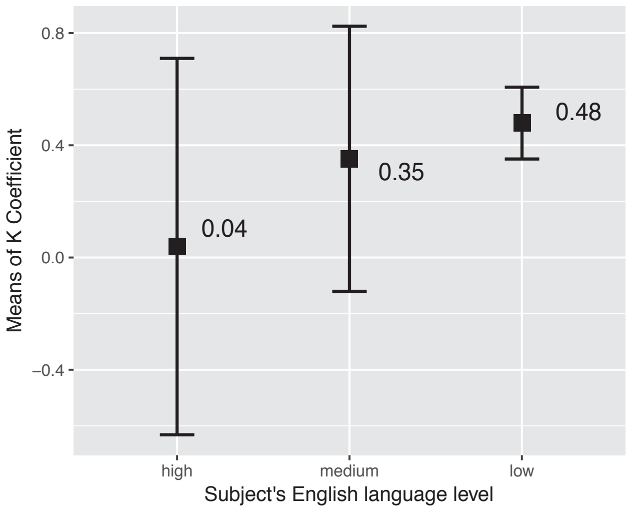


Figure 3: Means of K Coefficient by English level.

Finally, it was found that programmer's gender plays an interesting role against the ambient-focal vision modes. Gender had a significant effect (a one-way ANOVA: F(1, 409) = 15.8, $p < .001$) on the K Coefficient. Female subjects perform using ambient vision more than focal vision during source-code comprehension. Conversely, male programmers use focal vision rather than ambient vision (see Figure 4).

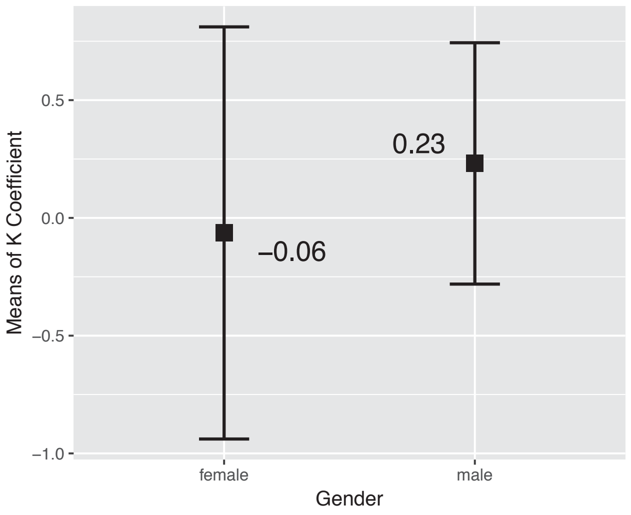


Figure 4: Means of K Coefficient by gender.

This preliminary analysis of the ambient-focal attention opens a discussion of their role in source-code comprehension. Specifically, should we apply Formula 1 for such tasks as source-code reading? How can the knowledge of ambient-focal usage be applied to software development issues for educational purposes?

4. REFERENCES

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

[2] K. Krejtz, A. T. Duchowski, I. Krejtz, A. Szarkowska, and A. Kopacz. Discerning Ambient / Focal Attention with Coefficient K. *ACM Transactions on Applied Perception*, 13(3):20, 2016.

[3] H.-C. Nothdurft. Focal attention in visual search. *Vision Research*, 39(14):2305–2310, 1999.

[4] S. Pannasch, J. R. Helmert, R. M{\"u}ller, and B. M. Velichkovsky. The analysis of eye movements in the context of cognitive technical systems: three critical issues. In *Cognitive Behavioural Systems*, pages 19–34. Springer, esposito a edition, 2011.

[5] S. Pannasch, J. R. Helmert, K. Roth, and H. Walter. Visual fixation durations and saccade amplitudes : Shifting relationship in a variety of conditions. *Journal of Eye Movement Research*, 2(2):1–19, 2008.

[6] R. C. Team. R: A language and environment for statistical computing. R Foundation for Statistical Computing, Vienna, Austria. 2013, 2014.

[7] A. M. Treisman and G. Gelade. A feature-integration theory of attention. *Cognitive Psychology*, 12(1):97–136, 1980.

[8] P. J. a. Unema, S. Pannasch, M. Joos, and B. M. Velichkovsky. Time course of information processing during scene perception: The relationship between saccade amplitude and fixation duration. *Visual Cognition*, 12(3):473–494, 2005.

[9] B. M. Velichkovsky. *Cognitive Science: Foundations of epistemic psychology. [Kognitivnaya nauka : Osnovy psihologii poznaniya]*, volume 2. Smusl: Publisher center "Akademia", Moscow, 2006.

[10] B. M. Velichkovsky, S. M. Domhoefer, S. Pannasch, and P. J. A. Unema. Visual Fixations and Level of Attentional Processing. In *ETRA '00 Proceedings of the 2000 symposium on Eye tracking research \& applications*, pages 79–85, 2000.

Towards robust data from program comprehension studies with fine-grained interaction monitoring and eye tracking

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

ABSTRACT

In this paper, we describe a subset of interaction events to monitor in a code editor together with raw gaze data and information about the eye tracker device used in the program comprehension study in order to completely reconstruct programmers' activity with source code for any time of the recording session afterwards. The motivation behind recording as much as possible, but still in the feasible amount, is to avoid any bias that online preprocessing methods can bring to the data, to allow reevaluation of methods used in the data analysis, as well as to share the data with other researchers for possible replication studies or replay programmers' work with source code without screencast recording, as we show with our implementation of such player.

Keywords

Program comprehension, Interaction monitoring, Eye tracking, Replication

1. INTRODUCTION

Various tools for recording data can be used in program comprehension studies with eye tracking, mostly based on the goals of the study and size of the source code fragments used in the study. Ranging from analytics software supplied by the eye tracker vendors with no specific support for source code as a stimulus [1], e.g., Tobii Pro Studio, to direct integration with a development enviroment, e.g., with the notable iTrace plugin for the Eclipse IDE [4]. Integration of eye tracking with a code editor allows us to conduct studies closer to the real-world workflow of programmers. Together with another Eclipse plugin Mylyn we may also record how programmers interacted with code and employ that in the study [3].

For our purposes of program comprehension studies [5], we opt to use lightweight code editor, e.g., Microsoft Visual Studio Code or Monaco Editor[1] integrated in a website, and record gaze data with an external tool using the Tobii Analytics SDK. Our approach is to record raw gaze data and all the necessary fine-grained interactions of a programmer with source code in the editor and with the editor itself during the experiment to allow us:

- completely reconstruct the state of the code editor at any time of the recording afterwards,
- reevaluate raw gaze processing methods, e.g., fixation filters, gaze alignments, smoothing filters,
- reevaluate code analysis methods at any time of the recording, e.g., the generatation of abstract syntax tree of the source code document,
- reevaluate gaze-to-code mapping algorithms, e.g., mapping of gaze positions to the nearest AST element,
- replay the programmer's gaze and interactions.

[1]https://microsoft.github.io/monaco-editor/

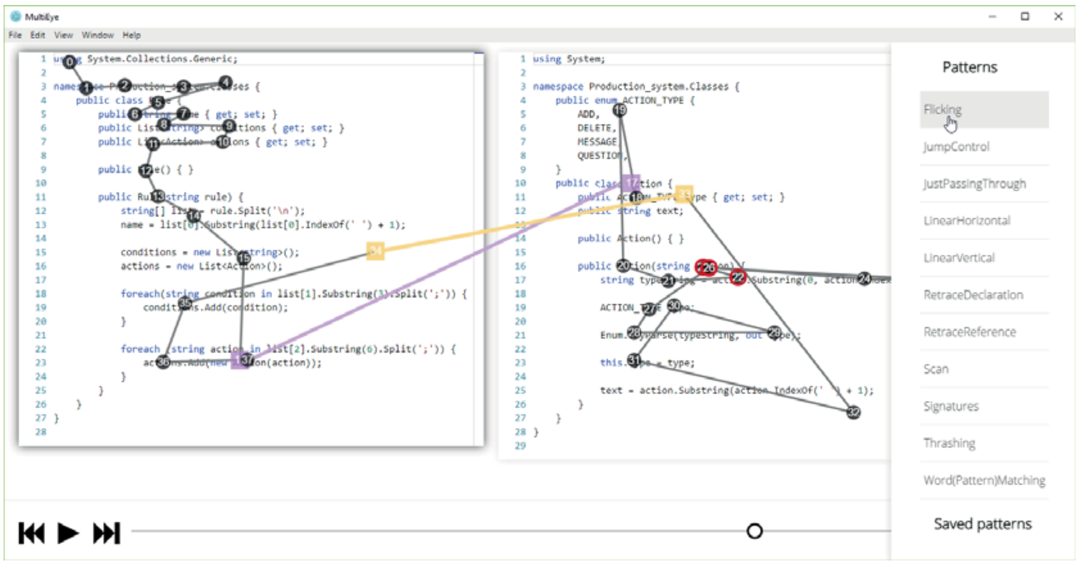
2. REQUIRED DATA FOR REPLICATION

To meet our requirements for data collection, we record and store solely raw data provided by the eye tracker together with all state changes and programmers' interaction events in the code editor. No online data pre-processing (such as online fixation filtering) is performed during the recording to avoid any introduction of bias into the data. This should be performed only afterwards as part of the data analysis task.

Each raw gaze data sample contains positions of programmer's gaze on the screen, eyes in space and validity of the recording. We also store information about the eye tracker device itself, firmware version, track box coordinates, calibration used for recording, and basic information about the participating programmer.

To preserve the state and contents of the code editor over time, we have identified the following subset of fine-grained interaction events:

- Screen resolution and DPI settings,
- Code editor window:
  - Size and position on the screen in pixels,
  - Window focus and state changes,
  - List of code editor viewports open in the window.
- Code editor viewport – includes source code document contents and line numbers, scrollbars, etc.:
  - ID of the viewport,
  - Path to currently open source code document,
  - Size and position in the window,
  - Scrollbar visibility and size,
  - Focus changes.
- Code editor contents – the area containing source code:
  - Size and position in the viewport,
  - Vertical and horizontal scroll position,



```
using System.Collections.Generic;

namespace Production_system.Classes {
    public class Rule {
        public string Name { get; set; }
        public List<string> conditions { get; set; }
        public List<Action> actions { get; set; }

        public Rule() { }

        public Rule(string rule) {
            string[] list = rule.Split('\n');
            name = list[0].Substring(list[0].IndexOf(' ') + 1);

            conditions = new List<string>();
            actions = new List<Action>();

            foreach(string condition in list[1].Substring(3).Split(';')) {
                conditions.Add(condition);
            }

            foreach (string action in list[2].Substring(6).Split(';')) {
                actions.Add(new Action(action));
            }
        }
    }
}
```

```
using System;

namespace Production_system.Classes {
    public enum ACTION_TYPE {
        ADD,
        DELETE,
        MESSAGE,
        QUESTION,
    }
    public class Action {
        public ACTION_TYPE type { get; set; }
        public string text;

        public Action() { }

        public Action(string action) {
            string typestring = action.Substring(0, action.Index

            ACTION_TYPE type;

            Enum.TryParse(typestring, out type);

            this.type = type;

            text = action.Substring(action.IndexOf(' ') + 1);
        }
    }
}
```

Figure 1: The MultiEye player displaying the recording of the program comprehension study with options to annotate identified pattern in the gazeplot.

  - Font settings – font family, size, line height, typical character width,
  - Text cursor position and selections.

- Code editor contents changes
  - Source code changes, if reported by the editor,
  - Keyboard events.

Recording the described data allows us to reconstruct position of the code editor viewport on the screen, verify whether the programmer's gaze was within its boundaries at any particular time and what code element he or she was focusing on. Even if the programmer may have edited the program source code, using these events we may replicate the changes, perform code analysis, and partially automate the task of coding the gaze with the coding scheme [1].

3. REPLAY AND DATA ANALYSIS

Recording all the relevant events in the code editor allows us to reconstruct and replay the programming session. Such player of program comprehension studies may be used for visualizing data analysis results, e.g., gaze plot or heatmap evolving over time with source code changes, switching attention between multiple source code documents.

Inspired by the Tobii Pro Studio software for analyzing experiment data, we work on the MultiEye player for replaying and annotating recordings of program comprehension studies (see fig. 1). The player displays gaze plot in source code documents over time, together with changes in source code and switching between documents, so the researcher may easily annotate groups of fixations with patterns based on the coding scheme from the EMIP workshops [2].

4. CONCLUSIONS

In this paper, we outlined minimal subset of data recorded in program comprehension studies to allow other researchers to replicate our study, reevaluate data analysis methods, or even replay the recording session. Currently, we work on tools for conducting and analyzing program comprehension studies in our UXI Group Lab (https://www.pewe.sk/uxi/) with 20 workstations with Tobii X2-60 eye trackers. Proposed subset of data can be recorded in all current popular code editors, e.g., Eclipse, Microsoft Visual Studio, Microsoft Visual Studio Code, and web editors like Code Mirror, Monaco, etc.; and required raw gaze data can be recorded using the SDKs of all major eye trackers manufacturers – Tobii and SMI.

5. ACKNOWLEDGMENTS

This work was partially supported by the STU Grant scheme for Support of Young Researchers and it is the partial result of the project Human Information Behavior in the Digital Space, APVV-15-0508, funded by the Slovak Research and Development Agency.

6. REFERENCES

[1] T. Busjahn et al. Eye tracking in computing education. In *Proc. of the 10th Annual Conf. on Int'l. Computing Education Research*, ICER '14, 3–10, 2014. ACM.

[2] Coding Scheme, Eye Movements in Programming Workshop, revision Feb. 15th., 2014, online: http://emipws.org/sample-page/2013-analyzing-experts-gaze/coding-scheme/#scheme

[3] K. Kevic et al. Eye gaze and interaction contexts for change tasks – Observations and potential. In *Journal of Systems and Software*, 2016. Elsevier.

[4] T. R. Shaffer et al. iTrace: enabling eye tracking on software artifacts within IDE to support software engineering tasks. In *Proc. of ESEC/FSE 2015*, 954–957, 2015. ACM.

[5] J. Tvarozek, M. Konopka, et al. Studying various source code comprehension strategies in programming education. In *Proc. of EMIP'15: Models to Data, Reports and Studies in Forestry and Natural Sciences*, no. 23, 25–26, 2016.

Comparing Novice and Expert Eye Movements during Program Comprehension

Johannes Hofmeister*, Jennifer Bauer†, Janet Siegmund‡, and Sven Apel§
University of Passau

Email: *johannes.hofmeister@uni-passau.de, †bauerjen@fim.uni-passau.de,
‡janet.siegmund@uni-passau.de, §apel@uni-passau.de

Norman Peitek
Leibniz Institute for Neurobiology,
Magdeburg
npeite@lin-magdeburg.de

***Abstract*—A recent eye-tracking study indicates that novices and experts use different strategies to comprehend source code. We are planning to replicate these results to understand the mechanisms behind the different strategies. To control for effects caused by the used stimulus materials, we are proposing a static code measure to differentiate source code. The suggested measure expresses the expected linearity of reading the code. We are planning to observe novices in a longitudinal study along their CS1 programming course, and compare changes in their comprehension strategies over time with the strategies of an expert sample. An inexpensive eye-tracker will be used to obtain data about different visual patterns.**

I. Introduction

A recent eye-tracking study found that experts and novices follow different paths while reading code [1]. The study reports several interesting differences between these groups. For example, the novices' saccades where on average shorter, and they focused on 52.4% of presented code elements, whereas experts' saccades were longer, and more focused, covering 41.3% of the presented elements. Further, the authors found that novices follow *story reading order* (i.e., a linear left-to-right-top-to-bottom reading progression similar to natural-language texts), whereas experts exhibit less linear reading patterns. This effect is likely based on experts' increased experience. Experts manage to identify and focus on relevant parts and are able to switch between more complex contexts; thus, their saccades appear more assertive.

To understand the described effects, we are planning a replication study. To this end, we analyzed the operationalization of linearity, and found a difference regarding the linearity of the snippets between experts and novices, as the authors also noted. For example, the group of snippets seen by experts had a median of 3 methods and a median length of 26 lines, whereas the novice snippets had only a median 1.5 methods and a median length of 15 lines of code. The snippets, which the novices saw, contained fewer methods. The increased amount of methods in the expert code snippets required experts to jump more between different code entities than novices, who saw mostly code with fewer methods, thus exhibiting fewer saccades. We hypothesize that the amount of methods and their structure (e.g., amount, location in the code, and spatial relation between call and definition, etc.) may affect the order of reading. This raises the question whether this effect results from novices' reliance on linear text reading skills, or from properties of the snippets presented. Thus, we would like to replicate the study, while controlling for differences caused by the stimulus material, to isolate and further understand the effect.

II. Operationalizing Linearity

To operationalize linearity of code snippets, we defined and evaluated a measure. Simply put, we describe a program's inherent structural *linearity* as a relation between the overall jump length and the number of methods.

We built this measure based on the following reasoning: Sequences of operations (i.e., statements) are read in the order of their occurrence. Calls to structural units (e.g., routines, functions, methods, etc.) cause deviations from this linear path, requiring the reader "to jump" to the definition of the symbol, if it is locally available. A *jump* can be understood as a logical saccade, that is, the reader's forward movement from a method call to the definition of the called method.

The idea is illustrated with an example in Listings 1 and 2, which show equivalent codes with different levels of linearity. When developers encounter a method call, they will have to jump the definition of called method (e.g., for Listing 1, from the call of inBetween to its declaration). The distances (in number of lines) between all calls in a program and the matching method declarations are summed and later divided by the program's average method length.

This implies that a program with very long methods is considered more linear than a program with fewer, shorter methods. Jumping between a call and a called method's signature makes the program less linear, thus a program with many methods is less linear than a program with fewer methods. Method length affects the jump length. Long methods by themselves are more linear than shorter methods, but also affect the length of each jump.

To provide an example, Listing 1 shows a single jump (i.e., a single call to the additional method besides main), whereas Listing 2 exhibits two jumps (i.e., call to both constructor and method). For Listing 1 our measure is calculated like this: The call and the method declaration are 6 lines apart, and the average method length is 3.5 $((3 + 4)/2)$. Thus, the snippet has a linearity score of 1.7. Listing 2 contains two jumps $(11 + 6 = 17)$, and has an average method length of

$((4+3+4)/3 = 3.6)$, and, thus, scores $17/3.6 = 4.9$, and is considered less linear than the code in Listing 1. These values help to compare code snippets against each other on an ordinal scale (i.e., Listing 2 is less linear than Listing 1).

Listing 1: Two methods

```
public class Interval {

    public static inBetween(int number, int lower,
        int upper) {
        return (lower < number) && (number < upper);
    }

    public static void main(String[] args) {
        Boolean result = inBetween(5, 1, 10);
        System.out.println(result);
    }
}
```

Listing 2: Complex structure

```
public class Interval {
    int lower = 0;
    int upper = 0;

    public Interval (int lower, int upper) {
        this.lower = lower;
        this.upper = upper;
    }

    public Boolean contains(int number) {
        return (this.lower < number) && (number <
            this.upper);
    }

    public static void main(String[] args) {
        Boolean result = new Interval(2, 10).
            contains(3);
        System.out.println(result);
    }
}
```

III. Experiment Design

This goal of this paper is to answer the following research questions:

1) RQ_1: Do novices read code differently than experts?
2) RQ_2: Are differences best explained by participants' strategies or properties of the stimulus material?

To answer these questions, we are planning the following experiment:

A. Participants

To replicate the original study as closely as possible, we investigate a group of novices and experts. The novices are part of a longitudinal study. To this end, we recruit them from a programming course at the University of Passau, and invite them to participate in the study before, during, and after the course. By tracking the learning efforts of the students in an educational setting, we control for the participant's lack of experience with reading less linear source code snippets. Experts are recruited at an industry venue and via the Internet.

B. Materials

In a first iteration, we are planning to replicate the results in the study using the original code snippets. In a later iteration, we investigate whether the measure suggested above is feasible to differentiate other code snippets. We add further snippets, ensuring that they are comparable based on their linearity, to investigate whether experience or the properties of the code account for inter-individual differences.

For our replication, we use a Tobii Eye-X eye tracker. The tracker is unobtrusive, inexpensive, and offers sufficient spatial and temporal resolution to trace participants' visual attention. The measures suggested by the original authors, such as element coverage and saccade lengths, should be reproducible with this apparatus.

IV. Expected Results

By replicating the original results, we aim to further understand differences between novices and experts. Other studies have shown that experts and novices apply different strategies (e.g., experts apply top-town strategies driven by beacons, whereas novices rely on bottom-up comprehension strategies) [2], [3]. The goal of our replication is to find further support for these processes and identify details of their mechanisms.

V. Discussion

The proposed measure represents a value that differentiates different code snippets based on their linearity, that is, a reader's expected positional progression when reading the program. We evaluated this measure for its face validity, in a pilot study. The results are pending.

Our measure accounts for static differences in the code. It is sensitive to subtle variations (e.g., insertions of empty lines) and can be augmented to represent further structures that affect linearity of code for example, ambiguous jumps (such as overloads) are multiplied with the number of possible targets for the jump, and loops, or recursive structures are penalized by squaring the jump distance. These kinds of structures render the program less linear.

Values for this measure form an ordinal scale or, in other words, create a partial order: Values with similar integer parts (e.g., 1.2 and 1.6) are considered similarly linear, as they reflect minor differences, whereas larger differences (e.g., 2.4 and 8.6) indicate different progressions while reading the program (e.g., a long method called from the top or the bottom of the program).

Acknowledgments

This work has been supported by the DFG grant SI 2045/2-1.

References

[1] 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 *2015 IEEE 23rd International Conference on Program Comprehension*, pages 255–265, 2015.

[2] N. Pennington. Stimulus Structure and Mental Representations in Expert Comprehension of Computer Programs. *Cognitive Psychology*, 19:295–341, 1987.

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

Do Computer Programmers With Dyslexia See Things Differently? A Computational Eye Tracking Study

Ian McChesney
School of Computing and Mathematics
Ulster University
Newtownabbey, County Antrim, BT37 0QB
Northern Ireland
+44 28 9036 6129
ir.mcchesney@ulster.ac.uk

Raymond Bond
School of Computing and Mathematics
Ulster University
Newtownabbey, County Antrim, BT37 0QB
Northern Ireland
+44 28 9036 6129
rb.bond@ulster.ac.uk

ABSTRACT

In this extended abstract, we describe the early stages of a study using computational eye gaze analysis to investigate the gaze behaviour of computer programmers with dyslexia.

CCS Concepts

• Software and its engineering~Maintaining software • Social and professional topics~Computing education programs

Keywords

Eye Tracking; Gaze Analysis; Program Comprehension; Dyslexia

1. INTRODUCTION

Dyslexia is defined as "a specific learning difficulty which affects the ability to recognize words fluently and/or accurately; causes problems with spelling, auditory short-term memory, phonic skills, multi-tasking, remembering instructions, and organizational skills" [4]. Approximately 10% of the population live with dyslexia and individuals with dyslexia experience the condition in different ways, and there is much debate surrounding its identification and support [1].

Computer programming is primarily a text-based activity, and as such it may present additional challenges to the dyslexic programmer over and above the typical cognitive challenges of software development. The challenges faced by programmers with dyslexia have been described both in relation to learning to program [8],[9] and working in industry [3],[7]. However, there is limited empirical work on the program comprehension strategies of programmers with dyslexia. Consequently there are many research questions to be addressed in this area; for example, how do models of reading such as the Dual Route Model [6] apply when reading program code? How does the visual aspect of program code (indentation, camel case, and IDE features) assist programmers with dyslexia? Furthermore, can cognitive assistants or better artificial intelligent agents be developed to assist neurodiverse programmers?

The exploratory study described here, currently in its early stages, is using eye tracking technology to gather data on the gaze behaviour of programmers with dyslexia.

2. STUDY ORGANIZATION

Subjects recruited from our undergraduate computing courses were presented with three programs based on the protocol used in the EMIP 2014 workshop [2] (referred to here as the Cake, CalcAvg and PrintRow programs). In this phase of the study we have 7 computer programmers with dyslexia and 5 in the control group. Recording sessions used the Tobii x60 eye tracking device and related Tobii Studio software (v2.3). For each program, subjects were presented with an instruction screen, the code screen and an evaluation screen, and were asked to verbalize their reading and understanding of the code (by thinking-aloud). After interpreting each program, subjects were also asked to rate how confident they were in their understanding of the program (1-low, 10-high). Subjects completed a short questionnaire which collected their personal characteristics such as their programming experience and preferred programming languages. The questionnaire also asked the subject if they had dyslexia and, if so, to self-rate this as mild, moderate or severe. The study was approved by our Faculty Ethics Filter Committee.

3. QUALITATIVE OBSERVATIONS

Each recording session was reviewed by the authors and using specified criteria the subject's understanding of each program was graded on a scale of 1-10 (1-low, high-10). Using the EMIP coding structure as a vocabulary toolbox, each program session was characterized in terms of the general pattern and sequence of gaze exhibited. While recognizing the subjective nature of this assessment, initial analysis using the coding structure did not show any distinct differences between the two groups.

4. QUANTITATIVE ANALYSIS

Using the Tobii Studio (v2.3) software, a range of visualizations and metrics have been explored.

4.1 Heat Maps

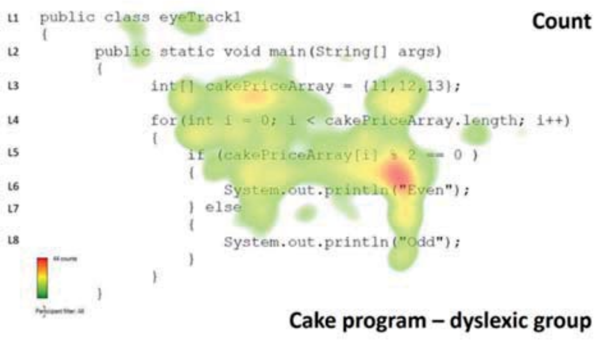
In Figure 1 (Cake program), we can see that the dyslexia group gaze (count) shows a distinct high volume of fixations on the modulus operator on line 5, with a focus just below the line of code. For the control group, gaze is focused on lines 4 and 5, namely the For loop header and following If statement condition respectively, with a focus again just below the lines of code.

For the CalcAvg program, the control group gaze is in the middle of the method, concentrated on line 7. For the dyslexic group, this is more vertically diffuse across lines 3-11.

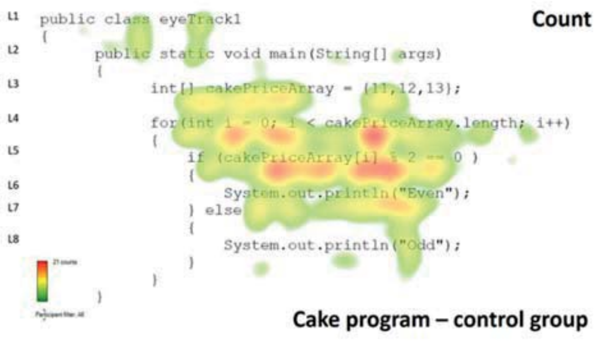
For the PrintRow program, the dyslexia group and the control group both exhibit gaze which is concentrated on the outer and inner For loops headers. In the control group, gaze is concentrated directly on the variables `row` (in the outer For header) and `col` (in the inner For header). Interestingly, in the dyslexia group, this gaze is more diffuse across these two lines, and is also concentrated "between" the lines (between line 3-4 and lines 4-5).

Computing heat maps within the first 5 seconds of viewing can depict a programmer's automatic intuition and Gestalt perception in understanding a program. For the Cake and CalcAvg programs, a qualitative pattern is that the dyslexia group tend to limit their scan to the initial lines in the programs (LinearHorizontal), whereas for the control group gaze is more vertically dispersed

(LinearVertical). This is not the case however for the PrintRow program where the dyslexia group gaze is, albeit briefly, drawn to the method call in the public static void main method at the bottom of the screen –this is not the case for the control group.



```
L1  public class eyeTrack1
    {
L2      public static void main(String[] args)
        {
L3          int[] cakePriceArray = {11,12,13};
L4          for(int i = 0; i < cakePriceArray.length; i++)
            {
L5              if (cakePriceArray[i] % 2 == 0 )
                {
L6                  System.out.println("Even");
L7              } else
                {
L8                  System.out.println("Odd");
                }
            }
        }
```



```
L1  public class eyeTrack1
    {
L2      public static void main(String[] args)
        {
L3          int[] cakePriceArray = {11,12,13};
L4          for(int i = 0; i < cakePriceArray.length; i++)
            {
L5              if (cakePriceArray[i] % 2 == 0 )
                {
L6                  System.out.println("Even");
L7              } else
                {
L8                  System.out.println("Odd");
                }
            }
        }
```

Figure 1. Heatmaps (count) for Cake program

4.2 Metrics

Considering Total Fixation Duration (TFD) and Fixation Frequency (FF) metrics at the program level, there appears to be no significant difference between the dyslexia and control groups for any of the three programs (Table 1, TFD, two-sample t-test).

Table 1. Total Fixation Duration, two-sample t-test (0.05 confidence level)

		Dyslexia group	Control group
Cake	$\bar{x}\ (\sigma^2)$	53.02s (305.52s)	46.69s (309.50s)
	P	0.2756	
CalcAvg	$\bar{x}\ (\sigma^2)$	77.60s (2175.76s)	60.69s (767.99s)
	P	0.2254	
PrintRow	$\bar{x}\ (\sigma^2)$	88.01s (1942.97s)	81.74s (1777.81s)
	P	0.4049	

Each line of the program (ignoring braces) was defined as an area of interest (AOI). For these, metrics First Fixation Duration (FFD) and TFD were evaluated. Significant differences in gaze duration were revealed in only a small number of AOI instances as shown in Table 2. Considering FFD, in each significant case, the dyslexia group spent less time on the AOI than the control group. For TFD, the only significant difference was in the Cake program, line 6, where the dyslexia TFD duration was greater than the control group.

Each program interaction consisted of reading an instruction screen followed by reading the program code screen. There was no emerging pattern between the two groups regarding overall reading time during the experiment.

Table 2. AOIs with distinct gaze patterns [(P)rogram, (L)ine]

	AOI	Dyslexia group ($\bar{x}$)	Control group ($\bar{x}$)	p	Line description
FFD	P1, L4	0.2086s	0.3000s	0.02	`For loop iterating over array`
	P2, L10	0.2300s	0.3700s	0.06	`system.out.println("Average")`
	P3, L2	0.1414s	0.2060s	0.02	`public static void printMethod`
TFD	P1, L6	5.1029s	2.0560s	0.08	`system.out.println("even")`

Some correlations (using Pearson's correlation coefficient) between experiment variables have been investigated. Line length (number of characters) is used as a proxy for complexity (other measures could be used, e.g. Halstead's complexity metrics). For the Cake program, both groups displayed similar behaviour, namely the longer the line length the greater TFD (dyslexia group, *r*=0.62, control group, *r*=0.71). For the CalcAvg program, this pattern held, albeit weakly, for the dyslexia group (*r*=0.20), whereas for the control group it was the opposite case (a weak negative correlation, *r*= - 0.01), i.e. for the control group, the longer the line length, the lower the TFD. For PrintRow, both groups exhibited a positive correlation on these variables, but weaker for the control group (*r*=0.15) compared with the dyslexia group (*r*=0.42).

On completion of the experiment, the authors assessed each subject's understanding of the code on a scale 1-10 (1-unable to explain program, 10-full explanation of program given). Correlation of this score to the TFD showed no overall trend. However, limiting cases to those who did not demonstrate a good understanding of the programs (scoring 6 or less), there appears to be a slight negative correlation with FF (*r*= - 0.27), i.e. the lower the understanding, the higher the FF values at the program level.

5. INITIAL OBSERVATIONS

There are many limitations in the study at this stage, most obviously the number of subjects. However, it does show that there are many interesting avenues to explore in comparing the gaze of programmers with and without dyslexia. Arguably, most notable of all at this stage, is that there appears to be no striking differences in how these two categories of programmer read code?

6. NEXT STEPS

Further eye tracking sessions have been scheduled to increase the number of subjects. It is hoped a broader data set, and drawing upon related research in other fields (e.g. [5]) will help identify fruitful research questions for further investigation. Does reading program code employ the same visual and cognitive models as reading text and how is this exhibited by a programmer with dyslexia? Do the visual, orthographic and phonetic differences in program code compared with prose make it easier for a programmer with dyslexia to comprehend a program? Do programmers with dyslexia see things differently?

7. REFERENCES

[1] Armstrong, D. and Squires, G., 2014. *Key Perspectives on Dyslexia: An essential text for educators*. Routledge.

[2] Busjahn, T., Schulte, C., Tamm, S. and Bednarik, R., 2015. Eye movements in programming education II: *Analyzing the novice's gaze. Technical Report TR-B-15-01*, Freie Universität Berlin, Department of Mathematics and Computer Science, Berlin, Germany.

[3] Coppin, P., 2008. Developing drawing and visual thinking strategies to enhance computer programming for people with dyslexia. In *Visual Languages and Human-Centric Computing, 2008. VL/HCC 2008. IEEE Symposium on* (pp. 266-267). IEEE.

[4] "dyslexia." In *A Dictionary of Education*, edited by Wallace, Susan. : Oxford University Press, 2015.

[5] Kim, S., Lombardino, L.J., Cowles, W. and Altmann, L.J., 2014. Investigating graph comprehension in students with dyslexia: An eye tracking study. *Research in developmental disabilities*, 35(7), pp.1609-1622.

[6] Law, C. and Cupples, L., 2017. Thinking outside the boxes: Using current reading models to assess and treat developmental surface dyslexia. *Neuropsychological rehabilitation*, 27(2), pp.149-195.

[7] Morris, M.R., Begel, A. and Wiedermann, B., 2015. Understanding the challenges faced by neurodiverse software engineering employees: Towards a more inclusive and productive technical workforce. In *Proc. 17th Intl ACM SIGACCESS Conf.on Computers & Accessibility (pp. 173-184). ACM.*

[8] Powell, N., Moore, D., Gray, J., Finlay, J. and Reaney, J., 2004. Dyslexia and learning computer programming. *Innovation in Teaching and Learning in Information and Computer Sciences*, 3(2).

[9] Stienen-Durand, S. and George, J., 2014. Supporting dyslexia in the programming classroom. *Procedia Computer Science*, 27, pp.419-430.

Enhancing fMRI Studies of Program Comprehension with Eye-Tracking

Norman Peitek
Leibniz Institute for Neurobiology
Magdeburg, Germany
Norman.Peitek@lin-magdeburg.de

Janet Siegmund
University of Passau
Passau, Germany
siegmunj@fim.uni-passau.de

André Brechmann
Leibniz Institute for Neurobiology
Magdeburg, Germany
Brechmann@lin-magdeburg.de

ABSTRACT

Understanding program comprehension is a fundamental research question, which requires multiple methods to fully understand it. In this paper, we propose to combine eye-tracking and functional magnetic resonance imaging (fMRI) study to gain additional data for interpreting programmers' cognitive processes during top-down comprehension. In this paper, we present a proposal for such a combined approach.

1 INTRODUCTION

Eye-tracking is a promising technique to understand the program-comprehension process. For example, the study of Busjahn and others [1] compared eye movements of novices and experts while reading code. Other novel approaches in combination with eye-tracking have been studied to gain insight into programmers' cognitive processes. Fritz and others [4] combined multiple psycho-physiological measures (i.e., electroencephalography (EEG), eye-tracking, electrodermal activity) to predict the task difficulty based on the participants' response. Similarly, Lee and others [5] used EEG and eye-tracking to predict the task difficulty and programmer expertise.

Currently, we are conducting a study on top-down comprehension with functional magnetic resonance imaging (fMRI). Neuro-imaging allows us to observe activated brain areas, and, thus, derive associated cognitive processes during programming tasks. Previous studies have successfully used fMRI [2, 3, 6], and we intent to explore a combination of fMRI and eye-tracking. In our current study, we found that fMRI alone is not sufficient to distinguish the fine-grained differences in cognitive processes without asking the participants to provide aloud protocols, which will interfere with the comprehension process. Eye-tracking may provide additional data to better understand how programmers comprehend code by observing their exact eye movements throughout an entire session in an fMRI scanner.

With our current study, we aim at understanding how code aspects, that is beacons and layout, affect top-down comprehension. To this end, we asked participants to understand source-code snippets in a top-down process. To enable participants to use top-down program comprehension, we trained them on code snippets before the fMRI scanner

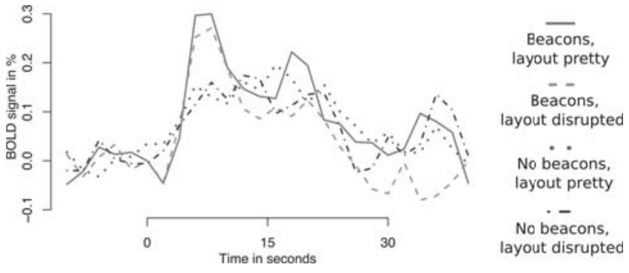


Figure 1: Time course of the BOLD response per condition for BA 21.

session. We created four different versions of the code snippets to evaluate the role of beacons (present or not) and layout (pretty-printed according to code conventions or disrupted, e.g., by adding obscuring line breaks). Additionally, participants should locate syntax errors. As common in fMRI sessions, each task (i.e., understanding a snippet or locating syntax errors) was followed by a rest condition that provides a baseline condition.

Preliminary analysis of the fMRI data showed that neither beacons nor layout had a differential effect on overall brain activation. For example, we could not find a significant difference in the activation pattern when beacons were present compared to when they were not present. Additionally, the fMRI data showed no effect between top-down comprehension and syntax-error finding, indicating that understanding source code and locating syntax errors does not require different cognitive processes. These are interesting results that seem to contradict common knowledge on program comprehension. However, fMRI data alone does not provide sufficient information to explain these results.

One shortcoming of fMRI is the low temporal resolution. Typically, the activation of a brain area happens in a matter of seconds, but actual activity is happening considerably faster, that is, within milliseconds. Thus, we cannot unravel rapid processes during program comprehension (e.g., the initial adjustment of a comprehension strategy when presenting a manipulated source-code snippet). With eye tracking, we plan to address this temporal gap. As example of what we hope to better understand, we present the activation of one brain area in Figure 1. The figure shows how the intensity of the activation over the duration of one comprehension task evolves, including 15 seconds before and after a task (i.e., the rest conditions). We can see that independent of the conditions, the change of activation increases to a peak

EMIP17 Spring Academy, Berlin, Germany
2017.

after a few seconds, and then slowly drops to the level before the task began. Two of the conditions (both with beacons in the source code) show an initial higher BOLD response peak at about 7 seconds. fMRI alone does not allow us to explain this phenomenon. Eye-tracking could give us insight how a participant forms a strategy during the comprehension process when beacons are available and, thus, adjust their approach to comprehend the code.

2 METHODOLOGY

Combining multiple psycho-physiological measurements can mitigate weaknesses of a single-approach study. For our study design, we plan to add eye tracking to the fMRI scanner to gain further insights into the programmers' comprehension, especially when the evolvement of activation of a brain area is not sufficient. The fMRI scanner has the option to add an EyeLink 1000 Plus[1] eye-tracker, which is able to track eye movements at 1000 Hz monocular and with 0.25°to 0.5°average accuracy. The high temporal resolution and spatial accuracy allows us to record smallest eye-movements while participants comprehend source code.

However, several adjustments to the experiment setup need to be made. First, the stimulus software we use to display the code snippets during the fMRI session (Presentation, available at http://www.neurobs.com/), has a plugin to communicate with the EyeLink system. Consequently, the stimulus software needs to incorporate receiving and storing the eye-movement data, additionally to managing the snippet display and response data. This needs to be tested with pilot studies. Second, to initialize the EyeLink, we need to run a calibration and validation process. This can either be done with Presentation or directly with the EyeLink. We need to evaluate, which method works more reliably. Third, we need to add the preceding eye-tracker initialization before starting the fMRI pre-measurements, which would make the fMRI session longer. Currently, the session starts directly with fMRI scans after a participant has been moved into the scanner. Once the fMRI measurement has started, the participant should be as motionless as possible to reduce motion artifacts in the fMRI signal. However, the eye-tracker calibration may need some head-position adjustments to succeed. Hence, we should perform the calibration only before the fMRI measurement and avoid it while the experiment is ongoing. Finally, we need to prevent drift of the eye-tracker. Drift is the growing signal offset over time. An ongoing drift throughout the 31-minute fMRI session deteriorates the eye-tracker's accuracy, such that we cannot reliably measure where participants are looking at.

3 OPEN QUESTIONS

The proposed change to our experiment setup raises two questions:

- **Online Drift Correction or Re-Calibration** The EyeLink system documentation describes a possible drift for longer eye-tracking sessions. This could lead to a significant offset of the eye-tracking signal. Without high accuracy of the eye-tracker, we lose critical information what part of the source code a participant is looking at. Consequently, we need to decide to either implement an online drift correction (e.g., looking at one fixed point during the rest periods) or to recalibrate and validate multiple times throughout the experiment.
- **Experiment Length** The current experiment lasts 31 minutes. With pre- and post-measurements, the time for a participant in the fMRI scanner adds up to around 45 minutes. We gathered feedback from previous participants that the experiment is already at the maximum of their concentration and that they feel exhausted afterwards. This is understandable, as participants have to lie as motionless as possible, but have to concentrate on a cognitively demanding task. If we add eye-tracking, the calibration and possible re-calibration would lengthen the experiment further and might lead to a declined performance during the later programming tasks. Shortening the experiment as alternative would decrease statistical power, making it more difficult for us to detect relevant effects.

Before implementing the proposed additions to the experiment setup, we need to solve these questions.

[1] http://www.sr-research.com/eyelink1000plus.html

4 ACKNOWLEDGMENTS

The authors' work is supported by DFG grant SI 2045/2-1.

REFERENCES

[1] T. Busjahn, R. Bednarik, A. Begel, M. Crosby, J. H. Paterson, C. Schulte, B. Sharif, and S. Tamm. 2015. Eye Movements in Code Reading: Relaxing the Linear Order. In *2015 IEEE 23rd International Conference on Program Comprehension*. 255–265. DOI:http://dx.doi.org/10.1109/ICPC.2015.36

[2] J. Duraes, H. Madeira, J. Castelhano, C. Duarte, and M. C. Branco. 2016. WAP: Understanding the Brain at Software Debugging. In *Proc. Int'l Symposium Software Reliability Engineering (ISSRE)*. IEEE, 87–92.

[3] Benjamin Floyd, Tyler Santander, and Westley Weimer. 2017. Decoding the Representation of Code in the Brain: An fMRI Study of Code Review and Expertise. In *Proc. Int'l Conf. Software Engineering (ICSE)*. IEEE. To appear.

[4] Thomas Fritz, Andrew Begel, Sebastian C. Müller, Serap Yigit-Elliott, and Manuela Züger. 2014. Using Psycho-physiological Measures to Assess Task Difficulty in Software Development. In *Proc. Int'l Conf. Software Engineering (ICSE)*. ACM, 402–413.

[5] Seolhwa Lee, Danial Hooshyar, Hyesung Ji, Kichun Nam, and Heuiseok Lim. 2017. Mining Biometric Data to Predict Programmer Expertise and Task Difficulty. *Cluster Computing* (2017), 1–11. DOI:http://dx.doi.org/10.1007/s10586-017-0746-2

[6] Janet Siegmund, Christian Kästner, Sven Apel, Chris Parnin, Anja Bethmann, Thomas Leich, Gunter Saake, and AndrÃČÂľ Brechmann. 2014. Understanding Understanding Source Code with Functional Magnetic Resonance Imaging. In *Proc. Int'l Conf. Software Engineering (ICSE)*. IEEE, 378–389.

Looking at Indentation

Analysing Gaze Movement during Program Comprehension with Indentations

Jennifer Bauer
University of Passau
bauerjen@fim.uni-passau.de

Johannes Hofmeister
University of Passau
johannes.hofmeister@uni-passau.de

Janet Siegmund
University of Passau
siegmunj@fim.uni-passau.de

Abstract

Indentations are commonly seen as important layout-aspects in code. To study the effects of indentations on gaze movement, we replicate a study about the relationship between indentations and comprehensibility during which the participants' gaze will be recorded via eye tracking. As a result, we expect that there is also a relationship between indentations and aspects of reading behaviour.

1. Motivation

Indentations in source code are an essential part of programming style. Developers use indentations to convey a program's structure to human readers. Some languages even embrace them in their syntax (e.g., Python). Indentations are believed to improve readability and consequently ease program comprehension. Shneiderman and others tested the difficulty of comprehending a Pascal code snippet depending on the level of indentation (0, 2, 4, 6 spaces) [1]. In their study, the participants had to read code and their comprehension was measured using a quiz with multiple-choice questions and them writing a short text about the program's function. They found that an indentation of 2 to 4 spaces leads to the best performance in terms of mean score in the comprehension quiz. This seems reasonable, because when there is too much indentation, the program is shifted to the right and scanning becomes more difficult. On the other hand, when there are no indentations, identifying associated parts of code gets more difficult due to the lack of visual structure in the code.
While the results of this study are interesting, they are outdated, and do not reflect modern representations of source code. In our study, we want to replicate the original study with two changes:
First, instead of Pascal, we use Java, because it is one of the most widely used programming languages today. Second, we will record the eye movements of participants with an eye tracker to get an understanding of where participants look, depending on the indentation.

We will address the following research questions:

1. **Which level of indentation is optimal for program comprehension?**
   With this question, we want to evaluate whether the results of the original study still hold today for modern programming languages. This gives us valuable insights about the effect of indentation, and allows us to evaluate common coding conventions (and possibly give recommendations for updating them).
   By using an eye tracker, we can determine the points of fixation on the code and saccades between them. Few saccades should indicate a good level of indentation.
2. **What challenges do different levels of indentation pose while reading code?**
   An optimal level of indentation can increase readability of code, such that programmers can extract its meaning faster. When having none or small indentations in the code, readers might take longer to understand the general idea and structure of the code. This could cause an increased number of saccades, as the readers have to jump more in the code to figure out the overall structure. On the other side, when there are very large indentations, jumping between lines with indentations becomes longer, which might fatigue the eyes too fast.

2. Experiment Design

The experiment is designed as follows:

2.1 Material and Task

We use several Java code snippets and with various levels of indentation, that is, 0, 2, 4, 6 and 8 spaces. We include 8 spaces, because the differences between the different indentations are quite small. For each code snippet, the participants' comprehension will be tested. To this end, we intend to ask question about the output of the program (e.g., 'What is the output of the program, if the input is 4?') and ask for a short textual description about the program's function (e.g., 'Describe what the program does'). We also intend to include a question about the appearance of variables in the snippet (e.g., 'Does the variable *length* play a role in the program?'). These measures combined will allow us to evaluate whether a participant understood one snippet, as they test both recognition and recall. There will be also a question about the participant's subjective rating of difficulty of the program on a scale of 1 to 7. This can later be compared to their score on the other questions, which could reveal, whether the participants experienced the code as difficult as their results on the question might suggest. To evaluate the participants' programming experience, we will use an questionnaire [2], as experience influences the handling of unexpected (in this case unexpectedly indented) code.

2.2 Participants

The participants in our study will be both novices and experts differentiated by the questionnaire about programming experience. We include both experience levels, because we expect that experts might not be as much affected by missing or excessive indentations as novices. Shneiderman and others found, for example, that novices are more uncomfortable with non-intended code than experts.

2.3 Apparatus

In our study, we use a Tobii-EyeX eye tracker. This tracker is unobtrusive and easy to use, as it is attached to the screen like a bar, and it samples with 60hz, which is sufficient for our measurements. Additionally, it has the advantage of a low price and an easy installation.

2.4 Execution

Due to the eye tracking, we conduct our experiment with one participant at a time. The participant will be informed that he/she will be eye tracked and given Java programs and questions about them. Next, the eye tracker has to be calibrated and the recording will start. The participant is then given the code snippets and answers the according questions one after another within the given 20 minutes. At the end, the participant completes the questionnaire about programming experience.

2.5 Variables

In our experiment, we alter the independent variable of indentation of the code snippet with 0, 2, 4, 6 or 8 spaces. Programming experience is an independent variable in our study as we want to measure the dependent variables also as a function of experience.
As dependent variables, we measure comprehension as described above. We determine the points of fixation on the snippet with their x- and y-coordinates. A fixation occurs when the participant spends more then 0.3 ms on one point.We also measure the number of saccades, which take place when there is no fixation, as well as their length, i.e. the distance between their start and end coordinates.

2.6 Planned Analysis

The following relations concerning indentation will be analysed:

- Relation between level of indentation and comprehension (correct answers of the questions/ correct description of the program's function in the short text) to find the optimal indentation
- Relation between level of indentation and points of fixation on the code to determine, whether different indentations change the parts of the code on which the participants concentrate on
- Relation between level of indentation and numbers of saccades to determine, which indentations make the participants jump more in the code
- Relation between level of indentation and length of saccades to determine, whether which indentations require the participants to make longer jumps in the code.

To this end, we examine statistic differences of the respective factors depending on the level of indentation.

3. Expected Results

In general, we expect to get similar results to the original study for the relation of code indentation and comprehension. The participant will probably also rate the difficulty of the code differently depending on the level of indentation, because novices are trained on correct style of code and might feel uncomfortable with from their point of view unusual indentation. Non-intended code should be rated equally difficult to comprehend by experts and novices, as it does not give hints about the structure of the code. Concerning the gaze movement, we expect that indentations have an effect on saccades and their length, as a lack of indentations probably makes it more difficult to grasp the structure of the code, whereas long indentations cause more jumping between lines of code.

References

[1] R. Miara, J. Musselman, J. Navarro and B. Shneiderman: 'Program Indentation and Comprehensibility' (1983)

[2] J. Siegmund, C. Kästner, J. Liebig, S. Apel and S. Hanenberg: 'Measuring and Modeling Programming Experience' (2014)

EMIP’17 Spring Academy

The fourth International Workshop on Eye Movements in Programming

Freie Universität Berlin, Germany, March 16th – March 17th, 2017

The Eye Movements In Programming community organizes the 2017 spring meeting in Berlin. The previous workshops were held in Joensuu, Finland (2013 & 2015), and in Berlin, Germany (2014). The 2017 EMIP Spring Academy in Berlin will feature speakers from the domain of gaze in programming, including social aspects, vision and educational perspectives. The program also includes focused workshops on topics related to the analysis of gaze data in programming and introduces new systems, tools, and techniques.

The event is suitable for both academic and industrial participants wishing to update and broaden their views on the role of attention in programming and education.

Participants are invited to submit a short abstract, in case they wish to make a presentation and/or discuss ideas during the workshop.

Invited speakers / contributors:

- **Tanya Beelders**, University of the Free State
- **Sven Buchholz**, University of Applied Sciences Brandenburg
- **Teresa Busjahn**, Freie Universität Berlin
- **Pavel Orlov**, Peter the Great St. Petersburg Polytechnic University
- **James Paterson,** Glasgow Caledonian University, UK
- **Markus Plank**, SensoMotoric Instruments (SMI)
- **Carsten Schulte & Lea Budde**, University of Paderborn
- **Bonita Sharif**, Youngstown State University

Program:

The Spring Academy will feature a mixture of invited talks, hands-on demos with SMI BeGaze and contributions by participants. We are aiming for a highly interactive workshop with overview presentations, the opportunity for networking activities and presentations in smaller groups.

Details on the program will be based on participants’ submissions.

Date:

March 15 evening: social gathering

March 16-17: workshop

Venue:

Freie Universität Berlin, Germany