FREIE UNIVERSITÄT BERLIN

Eye Movements in Programming Education II:

Analyzing the Novice's Gaze

Teresa Busjahn, Carsten Schulte,

Sascha Tamm, Roman Bednarik (Eds.)

TR-B-15-01
March 2015

FACHBEREICH MATHEMATIK UND INFORMATIK
SERIE B • INFORMATIK

Eye Movements in Programming Education II:

Analyzing the Novice's Gaze

Proceedings of the Second International Workshop

at the 9th WiPSCE Conference on Computing Education

Freie Universität Berlin, Germany

November 7th - November 8th, 2014

Welcome to the proceedings of the second workshop on "Eye Movements in Programming Education: Analyzing the Novice's Gaze".

Eye movement data yields rich information about source code reading. Observations of the visual behavior with an eye tracker can uncover programmer's visual attention in great detail. Unobtrusively and without extra cognitive load, viewing activities that the programmers are not aware of or fail to report for different reasons are captured. One of the challenges, however, lies in how to analyze the data.

This is the second edition of workshops that aim to create a network of academic and industrial participants with interest in various aspects of gaze in programming. The topics range from theoretical through methodological to applied topics of gaze in programming. We started with a focus on experts' gaze strategies during source code reading and developed a coding scheme to support analysis of reading strategies. This time, the attention shifted to novices.

The workshop was conducted in conjunction with the 9th WiPSCE Conference on Computing Education. It took place November 7th - 8th, 2014 at Freie Universität Berlin, Germany. A total of 21 people from 10 countries participated, three of them remotely. The event was kindly supported by SensoMotoric Instruments.

Before the workshop, participants were asked to analyze gaze data of two novice programmers reading Java and present their findings in a short position paper. The eye movement records came from a weekly Java beginner's course offered at Freie Universität Berlin, in which participants individually worked through an online course provided by Udacity (www.udacity.com/course/cs046). The beginner's course consists of six lessons, which were covered over a period of circa three months. After each lesson was completed, a recording was taken. For the workshop, we selected recordings corresponding to begin of class, mid-term and end of class.
The two exemplary novices that were studied had scarcely any prior programming knowledge. Workshop participants could choose, whether they want to analyze one of the novices or both. The complete dataset can be downloaded from www.emipws.org/datasets-2014/.

This technical report contains the position papers, workshop call, illustrations of the gaze data used, and a list of participants.

We would like to thank all participants for their excellent work,

Teresa Busjahn, Carsten Schulte, Sascha Tamm and Roman Bednarik

Contents

Applying Cognitive Theories to Novice Programmers

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

ABSTRACT

Identifying the differences between novice and expert programmers has been a long-standing question in the epistemology and learning research area for decades. With the advent of cheap eye tracking technology, experiments can be undertaken to objectively explore these differences at a much more fine-grained time-scale and with much larger numbers of experimental subjects than ever before. In this paper, we report on analyses of two novice computer science students who were recorded comprehending source code at three points in their first Java programming course. While we were able to infer possible cognitive explanations for their eye movement data, ultimately we did not have enough data to assure ourselves of its correctness. We plan to develop a cognitive model for program comprehension that combines symbolic execution, cognitive models, and eye tracking, that will hopefully be able to further progress the field in our understanding of the cognitive changes that occur as a programmer gains expertise.

1. INTRODUCTION

There is a long research history of trying to ascertain qualities that distinguish a novice software developer from an expert. While no one would mistake a first-time student learning Java programming with a software company employee having 15 years of Java application development experience, it has proven surprisingly difficult to objectively identify attributes manifested in the programming task itself that can serve as a judge of expertise.

Since the 1970s, researchers have subjectively identified several ways in which experts differ from novices. When asked to explain what a program does, experts can quickly spot the most salient parts of the program without having to read through the rest of the code. The researchers theorize that experts can do this by recognizing schemas [11, 8], or templates representing common coding patterns, and understand the purpose of those schemas without having to carefully examine the code within [7]. Crosby and Stelovsky confirmed this conjecture with eye tracking data of programmers looking at Pascal code [4].

When explaining what a block of code does, experts speak about its function, rather than its low-level execution. Experts plan ahead, adapting and applying general problem-solving skills to a given task (e.g. top-down [2] or bottom-up [13]), and sometimes employ specialized skills intended for more uncommon situations [10]. Brooks' top-down, hypothesis-driven plan uses the concept of beacons to link immediate knowledge about the program with domain knowledge in long-term memory. Beacons were theorized to be significant words, concepts, lines, or schema in the program text. Wiedenbeck showed that experts recall the lines containing likely beacons more than those that do not contain beacons with greater frequency than novices do [14]. Gellenbeck and Cook looked at various aspects that distinguish beacons from other parts of the program text, and found that structural grouping, headers, and mnemonically-defined identifiers improved the ability of programmers to find important concepts in the code [5].

Letovsky elaborated further on Brooks' top-down method to explain the hypotheses generated by subjects as they read the program. He found that subjects prompt themselves with a question when they encounter a confusing part of the code [9]. Subjects them come up with a possible answer to the question and look around the code for confirmation of their hypothesis. Once they have confirmed (or refuted) their answer, the subject resumes his prior task. Letovsky identified five kinds of inquiries: Why is the code written this way?, How is the code's goal accomplished?, What is this variable or subroutine I am looking at?, Does the code behave in the way I expect?, and Why does this code look funny or not do what I expect?

Many early research protocols were fairly simple. Programmers were asked to read a block of code and explain what it did. To make it more challenge, the code was taken away from the subjects when they had to explain it, forcing them to rely on their memory [3]. While this let researchers explore simple comprehension questions, it did not help to explain how subjects were performing the comprehension. Later experiments added think aloud, in which the subject was asked to verbalize his or her stream of consciousness while working. This lets the experimenter understand what the subject is consciously thinking about and can be quite revealing about the process.

The introduction of eye tracking technology has led to the promise of millisecond by millisecond data revealing where the subject is looking during the comprehension task. This

Eye Tracking for Programming Education II 2014 Berlin, Germany

offers two advantages: first, eye gaze location can be objectively and non-intrusively measured without interfering with the subject's thought process; second, eye gaze can be used as a proxy for attention, enabling researchers who develop cognitive models for programming to understand how and when information from the program can enters the subject's short term memory and can then be incorporated into the subject's growing understanding of the program.

2. WORKSHOP EXPERIMENT

In this workshop on eye gaze tracking for novice programmers, we have been asked to study and analyze eye gaze data from two subjects (EU10 and DO21) at three points in time during their attendance of an introductory Java programming course. Both subjects have had minimal training in programming in any programming language, and neither had any training in Java. The subjects' datasets come from programs they had to understand in Lessons 1, 4, and 6 of their Java course.

2.1 Video Analysis

In this section is a subjective description of what each subject looked at while performing their comprehension tasks.

1. Subject EU10

   Lesson 1 The subject read the program in text order, twice in a row. There were very few regressions or pauses in the eye gaze playback.

   Lesson 4 The subject first read the program in text order, taking about 25 seconds to do so. Then, the subject starts to read the program to see what it does when it runs. She starts at the body of the main() method, and reads the first line that defines the 'text' local variable. Then she jumps down to the last line containing the print statement that references the 'text' variable. Then, she switches to reading the program in dependency order, starting at the fourth line, which assigns a value to the 'word' variable. 'word' is assigned by executing the substring method on the value of the 'text' variable. The subject refreshes her memory of the value of the 'text' variable by going up to the first line, where 'text' was defined. There, she reads the string 'Hello World!'. Then she goes down one line to read the 'positionW' assignment statement, which is used by the assignment to 'word'. She "executes" the 'indexOf' method used to assign the value of the position of the 'W' character in the 'text' string by looking at each character of 'Hello World!' one letter at a time from left to right. She appears to be counting how many characters are there until the letter 'W', finding out the value of the assignment to 'positionW'. She then jumps to the third line, which assigns 'textLength' a value of the length of the 'text' variable. She uses the same tactic, again, counting the letters in 'Hello World!' one at a time to figure out its length.

   Next, she goes to the fourth line of the method with the assignment to 'word' and executes that statement in her head, which means she has to remember the values of 'positionW' and 'textLength' to calculate the answer. This is my own inference, however. Alternatively, she could have been expert enough to recognize that the 'substring' method simply gets the final word of 'Hello World!' by beginning at the position of the 'W' letter and going until the end.

   Finally, she reads the last line of the method again to evaluate the text.replace() method and come up with the answer 'Hello Sun'. She reads this line several times. Then she appears to refresh her memory of the antecedents, looking at the 'positionW' assignment, then the 'text' assignment (to remember the string), then the 'textLength' assignment, and then the 'word' assignment on the fourth line. She goes back to the 'text' assignment on the first line and appears to evaluate the 'substring' method again. She drops back down to the last line, and reads it again. She bounces back and forth between the assignment to 'text' and the 'text.replace()' function, and then looks at each line of the main() method one more time, as if validating her hypothesis about how this code should be executed.

   Again, she reads the assignment to 'word', looking at the arguments to the method over and over again. She goes back to the last line of the program, and bounces between the first line and last line. Then she rereads the 'textLength' assignment, the 'word' assignment, and the final line one last time before answering the question.

   Her answer says that the code replaces 'Word' with the string 'Sun'. We posit that she is referring to the variable 'word,' rather than a misspelled 'World'. Given this answer's equivalence to the final statement of the program, she could have given this answer without understanding the program at all. She did exhibit some understanding of the 'java.lang.String' methods through her execution of the 'indexOf' and 'length' methods, but it is a mystery whether she used this understanding in her answer.

   Lesson 6 The subject begins by reading the signature of the 'printMethod' method. She continues reading in text order to read the first 'for' loop. However, when she gets to the first use of the 'numberOfRows' variable, which happens to be the method's parameter, she jumps down to the printMethod() call site. She identifies that the value passed in is the number 3, and returns to the 'numberOfRows' method, where she had left off. She continues to read slowly, and with regressions, through the two nested 'for' loops.

   She now begins to solve the problem. She jumps down to re-read the 'main' method, and jumps up to re-read the 'for' loops, paying attention to the print statements that write a $*$ on the screen as each loop is executed. It appears as though she is gazing only at a single point on each line, as if she's mentally executing the code, her eyes signaling which line is attendant in mer mind at one time.

   Her answer to the problem is that there are 'two for loops for row and col,' which indicates she did not evaluate the code at all, but in fact, just described its control structure.

2. Subject DO21

   Lesson 1 DO21's first assignment is in pseudo-code, not Java. She starts reading the program in text or-

der, but stops on the first conditional statement. She re-reads the first line which has the data the conditional is testing. Then she goes back to where she left off, reading the program in text order until the end. Then she re-reads the first line with the data, goes back to the conditional its consequent. Finally, she goes to the second line of the program, reads the 'for' loop that surrounds the conditional. Then she reads the conditional and finishes. She answers the question correctly, using her own words, rather than explaining the surface features of the pseudo-code.

Lesson 4 This assignment is in Java. DO21 reads the code in text order until the second to last line of the method body where two print statements are. She reads the preceding two lines to see that the variable 'num2' is fetched from the user and printed out. Then she rereads the print statements again to discover that two numbers are added together and divided by two and assigned to the variable 'average'. She goes back up again above the variable 'num2' to see similar code that fetches an integer from the user, assign it to 'num1', and print it out.

She appears confused at how the 'num1' and 'num2' variables are assigned. She looks at the call to instantiate a new Scanner object. Then, she looks at the import statement that pulls the 'java.util.Scanner' namespace into the program. She starts to read the program in execution order, but only at a program slice where the Scanner object is used. She repeats herself again quickly.

Then she goes to the last line where the variable 'average' is printed out. She then goes back up to the lines where 'num1' and 'num2' are assigned.

She may have memorized the calculation to compute the average, or fetched it from her domain memory ('average' is a common math term) because she did not look at that line again before answering the question. Her answer is correct, but indicates a surface-level explanation of the code.

Lesson 6 DO21 reads the same code as subject EU10. First, she reads the lines of the 'printMethod' method in text order. Then she goes through the 'for' loops slowly, reading them over and over again, with intermittent jumps to read the value of the argument to the 'printMethod' because it is used in the outer 'for' loop. She appears to be executing the 'for' loops completely.

Her answer indicates that the subject understands the code and the purpose of its nested loops. In addition, she writes the output of the program correctly.

2.2 Analysis Informed by Prior Research

Subject EU10 exhibited signs that she understood how to execute Java code, but as Jeffries' research suggested of novices, her answers indicate she continues to have a low-level literal understanding of the code [7]. Jeffries also suggested that novices read code in text order. Both EU10 and DO21 did at the beginning, but as they progressed in the course, they interrupted this book reading with just-in-time inquiries about the code, as predicted by Letovsky [9]. This happens in Lesson 6, for Subject EU10. She asked a 'What' question about the 'numberOfRows' parameter to the method she was looking at. DO21 appears to ask a 'What' inquiry in Lesson 1. I think DO21 is more advanced than EU10 has been since the beginning of the course.

As proposed by Brooks [2] and elaborated by Wiedenbeck [14], beacons helped Subject DO21 decode the intent of the computation to produce the average of the two numbers. I believe this because she barely looked at the computation to produce the average, but she was able to answer the question correctly.

Aschwanden and Crosby showed in an eye tracking study that people look longer (mean fixation time) at lines of code that are important for program comprehension [1]. I calculated the average fixation duration for the most important lines of each subject's lesson in this study, and report them in Table 1. These two study subjects never looked at the important AOIs (words or groups of words) for longer than the unimportant AOIs. That could indicate that they remain novice enough even by the end of the Java class to pass for novices in Aschwanden and Crosby's study.

2.3 Improving the Experiment

There is a large amount of inter-subject variation among software programmers. This is especially true at the beginning of one's first programming course. It would help to have many more samples of each lesson done by different students in the course. In addition, I believe these two students did not seem to change very much in the course. I would like to see more repeated measures, i.e. where students do the same problem type again later in the course to see how they might have changed over time. This would enable this author to be more certain of his results.

3. FUTURE ANALYSES

Inspired by the work of Shneiderman and Mayer, Green et. al [12, 6, 13], and Rist developing cognitive models for writing code, I plan to create a cognitive model to explain program comprehension. A key aspect of this new model is the combination of eye tracking, symbolic execution, and a modern neurological understanding of cognition.

Symbolic execution is used in program analysis to understand what state a program is in at any stage of execution. As the analysis moves through the program, it builds up a model of what the program has done. The models often include a heap (for the values of variables), a stack (to keep track of function calls), a program counter (to keep track of the current line being executed). Contained implicitly in these models is a static heap to keep track of the definitions of the code itself and the layout of code in memory.

Human cognitive understanding works differently than computer understanding. Unlike a computer's FILO stack and random-access effectively infinite heap, a human's memory is multi-level and limited. It consists of a short-term memory store that can hold between 3 and 7 concepts for up to 1 minute, and a long-term memory store that can hold everything that is perceived to be important such as domain-specific concepts, strategies, schemas, templates. Long term memory also provides access to learned skills for planning, debugging, testing, and reformulation. There are other kinds of memory (e.g. spatial, prospective, associative, and episodic) which we will put into future versions of our model.

We model short-term memory as a cache using a least-recently-used replacement strategy. For example, as a person reads through source code, the program's variables and

Subject	Lesson	Most Important AOIs	Top Important (ms)	Top Actual (ms)
EU10	1a	L3P3, L4P3-5	L4P3-5 (420)	L2P8 (426)
	4a	L3-6P2, L3P4-5, L4P4,6, L5P4, L6P4,6,8-10, L7P3,5,7	L3P4,5 (423)	L5P1 (468)
	6	L2P7, L3P4-6,8-10,12,13, L4P4-6,8-10,12,13, L5P3	L5P3 (537)	L10P1 (592)
DO21	1b	L1P2,4,6, L2P3, L3P3-5, L4P2, L6P2	L1P4 (270)	L2P2 (501)
	4b	L2P3, L5,7P2,4, L6,8P2, L6,8P4, L9P2,5-7,9,10, L10P3, L11P3	L11P3 (233)	L6P6, L9P3 (283)
	6	L2P7, L3P4-6,8-10,12,13, L4P4-6,8-10,12,13, L5P3	L3P12 (359)	L10P8 (634)

Table 1: Highest average fixation duration for the Areas of Interest (AOIs) that were most important to the program and the one the subject focused their eyes on. The author selected the most important AOIs based on beacon theory, however, the highest average fixation duration always belonged to unimportant AOIs.

associated values enter his short-term memory. As more source code is read and processed, older variables and values may get evicted from short-term memory prior to being stored in long-term memory. When information is deemed important enough, and is practiced repeatedly, it can get stored in the effectively infinite long-term memory by connecting it to already-known domain knowledge.

In order to put information into short-term memory, a person must pay attention to it. We use the eye gaze information as a proxy for attention. When the person looks at a line of code, our model places that line and its meaning into short-term memory. As the person reads more lines of code and fills up the short-term memory, previous lines are evicted. If the person recognizes the signs (beacons) for a coding schema already stored in his long-term memory, he can replace several independent stored items with a single reference to the relevant schema and reduce the load on his short-term memory.

But, people read code in many different orders, including text order (like a book), control flow order (as the program executes), and data flow order (following some data values forwards or backwards through the program, while ignoring others). We use a symbolic execution engine to model control and data dependencies between lines of code. In order to understand the value of 'z = a + b', the engine must evaluate previously executed lines containing assignments to 'a' and 'b'. A correct program is written in an order that ensures that dependencies are met before a line of code is executed. A human, however, does not necessarily read code in linear order, or control- or data-dependent orders. Thus they might get to a line of code that they cannot understand until they read the right line of code that precedes it in control order. Humans do this easily and just-in-time when comprehending code [9].

In our cognitive model, we modify our symbol execution engine to model how a human understands code, given the limitations he has on short-term memory and his predilection for reading code out of control flow order. When a person reads too much code, information will fall out of short-term memory. Thus, if our model predicts he will lack knowledge of the value of a variable he must have to understand the code, he will have to direct his eyes (attention) back to the line of code that defines the variable's value. If the variable was found in the short-term memory, it would be accessible without re-reading it.

This model expects a person to comprehend the source code as well as a computer would. Thus, violations of this model could indicate the degree of expertise of the human. A novice may need to reread the same line of code several times, even though it is in the short-term memory. An expert may be able to skip reading the definition of a piece of code if he recognizes that the name and shape of the code resembles a template he already has accessible in his long-term memory.

I plan to build a prototype of this model by the time of the workshop, and hope to discuss its design and validity with other workshop participants.

4. REFERENCES

[1] C. Aschwanden and M. Crosby. Code scanning patterns in program comprehension. In *Symposium on Skilled Human-Intelligent Agent Performance. Measurement, Application and Symbiosis. Hawaii International Conference on Systems Science*, 2006.

[2] R. Brooks. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, 18(6):543–554, June 1983.

[3] R. E. Brooks. Studying programmer behavior experimentally: The problems of proper methodology. *Commun. ACM*, 23(4):207–213, Apr. 1980.

[4] M. E. Crosby and J. Stelovsky. How do we read algorithms? a case study. *Computer*, 23(1):24–35, Jan. 1990.

[5] E. M. Gellenbeck and C. R. Cook. Does signaling help professional programmers read and understan d computer programs? In J. Koenemann-Belliveau, T. G. Moher, and S. Robertson, editors, *Empirical Studies of Programmers: Fourth Workshop*. Ablex Publishing Corp., Norwood, NJ, USA, 1994.

[6] T. R. G. Green, R. K. E. Bellamy, and M. Parker. Parsing and gnisrap: A model of device use. In G. M. Olson, S. Sheppard, and E. Soloway, editors, *Empirical Studies of Programmers: Second Workshop*, pages 132–146. Ablex Publishing Corp., Norwood, NJ, USA, 1987.

[7] R. Jeffries. A comparison of the debugging behavior of expert and novice programmers, 1982. Paper presented at the meetings of the American Education Research Association.

[8] W. L. Johnson and E. Soloway. Proust: Knowledge-based program understanding. In *Proceedings of the 7th International Conference on Software Engineering*, ICSE '84, pages 369–380, Piscataway, NJ, USA, 1984. IEEE Press.

[9] S. Letovsky, J. Pinto, R. Lampert, and E. Soloway. A cognitive analysis of a code inspection. In G. M. Olson, S. Sheppard, and E. Soloway, editors, *Empirical Studies of Programmers: Second Workshop*, pages 231–247. Ablex Publishing Corp., Norwood, NJ, USA, 1987.

[10] M. Linn and J. Dalbey. Cognitive consequences of programming instruction. In E. Soloway and J. C. Spohrer, editors, *Empirical Studies of Programmers*, pages 57–81. Lawrence Erlbaum, Hillsdale, NJ, USA, 1984.

[11] C. Rich. A formal representation for plans in the programmer's apprentice. In *Proceedings of the 7th International Joint Conference on Artificial Intelligence - Volume 2*, IJCAI'81, pages 1044–1052, San Francisco, CA, USA, 1981. Morgan Kaufmann Publishers Inc.

[12] R. S. Rist. Program structure and design. *Cognitive Science*, 19(4):507–562, 1995.

[13] B. Shneiderman and R. Mayer. Syntactic/semantic interactions in programmer behavior: A model and experimental results. *International Journal of Computer and Information Sciences*, 8(3), 1979.

[14] S. Wiedenbeck and J. Scholtz. Beacons: A knowledge structure in program comprehension. In G. Salvendy and M. J. Smith, editors, *Designing and Using Human-Computer Interfaces and Knowledge-Based Systems*, pages 82–87. Elsevier, Amsterdam, The Netherlands, 1989.

An Exploratory Analysis of the Novice's Gaze

A Position Paper for the International Workshop on Eye Movements in Programming Education at WIPSCE14

Martin Löhnertz
Faculty IV/Computer Science
University of Trier
D-54269 Trier, Germany
loehnert@uni-trier.de

ABSTRACT

We analyse three sets of eye tracking data obtained from a novice's attempt to understand program code examples. We apply interpretation guided pattern segmentation and statistical methods to relate the given data to similar investigations from literature and previous workshops.

Categories and Subject Descriptors

K3.2 [**Computers and education**]: Computer and Information Science Education—*computer science education*; D.2.5 [**Testing and Debugging**]: Code inspection and walk-throughs—*reading code*

Keywords

code inspection, eye-tracking

1. INTRODUCTION

For an introduction to the general setting and the datasets (numbered 1,4 and 6) we refer to the preface of the volume.

When analysing human behaviour two approaches rival each other. On the one side is the inductive scientific approach that tries to eliminate all subjective factors and thus concentrates on - ideally numerically - measurable observations. While yielding indisputable statements these techniques often fail to produce any results in educational contexts as the number of independent parameters frequently is overwhelming and the sample sizes from sets of homogeneous observable objects mostly are small.

On the other side is the interpretative approach which assumes and accepts testees and observers to be uniformly humans and to share a common system of intentions, methods and adapted social or mental constructs, that allows the observer to relate his observations to his own behavioral patterns. While this allows to create conclusions based on very small sets of data it is prone to individual abberations, cultural barriers (e.g. in this context: reading directions) and misattributions. These can be averaged by increasing the number of human observers like it is attempted by the current workshop.

We will exemplarily apply methods from both paradigms. On the one hand we present our own (speculative) interpretation of the data and try to objectify this by applying schemes developed in a previous workshop. On the other hand we attempt some basic statistical analysis relating to established hypotheses of code reading behaviour.

2. SPECULATIVE DESCRIPTION OF THE TRACKING DATA

According to the questionnaire provided testee "DO21" is a native french[1] speaker with two years of programming experience in C++ but still low programming expertise due to very infrequent exposition to programming tasks (less than one hour per month). Her medium level competence in english (self-rated) probably was supportive when reading english variable names and program constructs on all three difficulty levels.

The first (pseudo-)code example tested is basically an english sentence formatted into a program-like structure. It comprises a loop and a simple fork. The testee reads it sequentially until an application of "cake price" is reached, then carefully re-reads the specification of "cake prices" in line 1 followed by a linear read of the whole program and two re-reads of the beginning with some regressions.

The second example presents JAVA-code with a completely sequential program structure, so from an algorithmic point of view it is simpler than the first probe. The data presented contains several eye movements leaving the code to coordinates (15,0). We considered these measurement errors - possibly blinking, although the used SMI RED-M device appears to support "blink recovery" what would imply some "blink detection". Random disturbances of the reading process should normally not result in exactly identical values, thus we ignored them. As the class name describes its function - creating a bias towards "expectation based comprehension"[5] - we suppose the program was understood after the first linear read. The testee then tries to understand the JAVA-scanner class and how it is used for input, followed by a short re-verification of the general function at the end.

The third example is strictly more difficult than the second one, but again contains no execution forks. It comprises a function call and two nested loops, where the number of iterations of the inner loop depends on the state of the outer. Again several offscreen events have to be eliminated. The testee starts reading top down and needs significant time before identifying the "main" routine and even more before trying to match actual and formal parameters. The following large number of recessions between the two loop constructs on the other hand is enforced by their nesting and not necessarily an indicator of difficulties. For this example we found it astonishing that the testee was able to actually

[1]Which has significantly higher redundancy than english [3].

Table 1: Analysis of example 1

Event #	Pattern	Strategy
1-8	LinearVertical	DesignAtOnce
9-20	RetraceDeclaration	?
20-33	LinearVertical	DesignAtOnce
33	Recession	FlowCycle
33-49	Flickering	FlowCycle

Table 2: Analysis of example 4

Event #	Pattern	Strategy
1-11	Flicking	none
11-86	Scan	DesignAtOnce
87-127	Flicking	FlowCycle
127-235	WordMatching	AttentionToDetail
236-266	Scan	TestHypothesis

”solve“ the problem by predicting the output. We would expect more prominent fixations on the critical ”col <= row“ part in the case of real understanding.

3. PATTERNS AND STRATEGIES

Preceding this workshop focusing on the ”novice“ programmer a workshop on the ”expert's view“ was held in 2013 [1]. There a set of movement patterns and interpretation strategies was collected (pp. 36-41), which we will try to apply to the new data. Most of these patterns and strategies have self explanatory names with the following exceptions: ”JumpControl“ means the following of the execution order; ”WordMatching“ is described as *visual pattern matching* which we assume to mean jumping between lexically identical objects, ”DesignAtOnce“ means a scan to understand the general idea, and ”FlowCycle“ denotes the repeated intensifying reread of some part.

When analysing **example 1** (Table 1) we noted that the usage term ”retrace declaration” is somewhat JAVA specific as there is no distinction between declaration and definition which is crucial to other programming languages.

As the code in **example 4** (Table 2) is completely sequential no differentiation between ”JumpControl“ and ”LinearVertical“ is possible. Most prominent appears the inspection of parts related to the class ”Scanner“ from event 127 to 235 which we classified ”WordMatching“ as the testee also revisited the ”import“ statement.

Example 6 (Table 3) starts getting interesting at event 224 when the testee identifies the main program and starts tracing the data-flow. Understanding of the code has probably happened between events 264 and 310.

We found it quite difficult to classify the examples available by the schemes developed for the ”experts-view”, as these expert strategies are seemingly not fully available to the testee, even at the last test. Many parts, that we classified ”LinearVertical“ would be better described as “linear vertical with some recessions”. To describe the novices attempts we would like a finer differentiation of regression patterns. Alternatively one could refine the time resolution and split it into ”linear vertical” and ”recession phases”, but these were very short - in time and length - and often appeared to be related more to general reading processes than to actual program evaluation.

Table 3: Analysis of example 6

Event #	Pattern	Strategy
1-30	Flickering	none
30-106	Flickering	DesignAtOnce
107-116	LinearVertical	DesignAtOnce
117-129	Flickering	Wandering
130-138	Flickering	DesignAtOnce
138-144	LinearHorizontal	AttentionToDetail
144-162	Scan	DesignAtOnce
163-165	LinearVerical	AttentionToDetail
165-171	LinearHorizontal	AttentionToDetail
171-177	Flickering	Wandering
177-183	LinearHorizontal	AttentionToDetail
183-195	Flickering	AttentionToDetail
196-201	LinearHorizontal	AttentionToDetail
201-223	Flickering	AttentionToDetail
224	RetraceDeclaration	DataFlow
225-226	JumpControl	AttentionToDetail
227-234	Flickering	AttentionToDetail
234-263	Flickering	TestHypothesis
234-263	Flickering	TestHypothesis
264-310	Flickering	Data Flow
340-342	LinearHorizontal	AttentionTodetail
343-399	Flickering	TestHypothesis

On the interpretation level discernable important events like ”testee has found main program“ should get more attention and should be used for a stateful model of the testees knowledge. After a fixed value ”3“ has become available in example 6, the testee proceeded significantly more efficiently. This is directly related to the level of abstraction needed in tight analogy to Piaget's levels of individual cognitive development [4].

4. STATISTICAL OBSERVATIONS

To gain a broader view of the field we tried to relate to the established concept of beacon orientation [2]. Beacons are code snippets prominently relevant to the overall structure of a program and are assumed to be used as ”jump targets” during reading. Assuming that actively using beacons will result in an increased amount of relatively wider jumps ”from beacon to beacon“ we counted the number of ”large jumps“ for quantitative different definitions of ”large“, depending on the length relative to the maximum jump length. As depicted by figure 1 no such tendency can be observed for any definition. The even relatively larger jumps in the first probe might by related to the increased size of the individual points of interest due to their representation by full words.

Another possible consequence of the beacon theory would be a relative reduction of short fixations, as the number of ineffective fixations should decrease. As depicted by Figure 2, this can be observed for the development from example 4 to example 6. But quite contrary to that the relative distribution for example 1 tends even more towards longer fixations. Again we conjecture that this is caused by the pseudocode style of the first example.

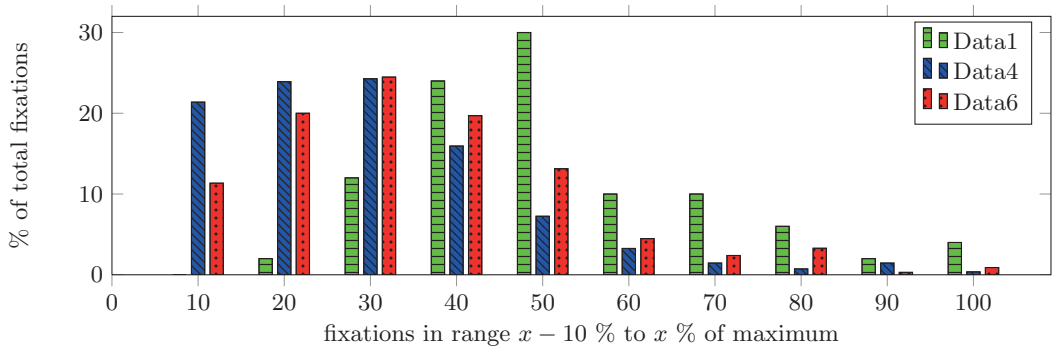


Figure 2: Percentage of fixations by duration relative to maximum

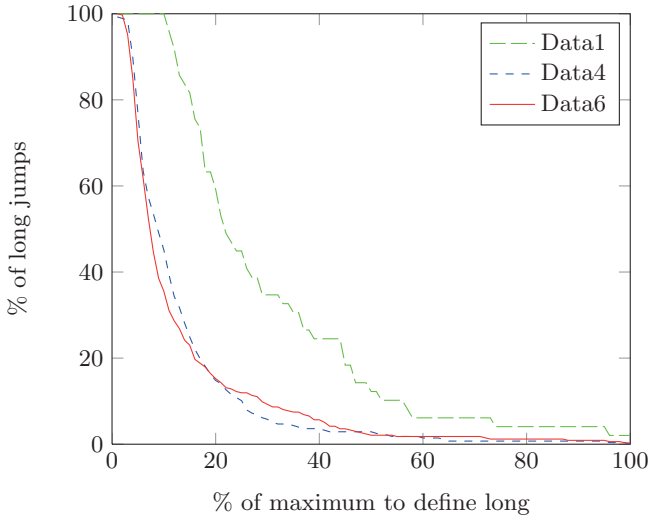


Figure 1: Percentage of long jumps depending on definition of long jump

5. DISCUSSION

A major problem of the data provided is that two parameters have been changed simultaneously: The skill of the student and the difficulty of the problem. It would be interesting to see the advanced student's performance on a simple program. The data from the first lesson probably is not suited to be compared to the other two due to the missing distinguation between structural and coding (i.e. JAVA) induced complexity. Furthermore it could be worthwhile to compare these traces with data from unsucessful reading attempts.

A more crucial question is how to create "operative" knowledge from these observations. In other eye-movement analysis contexts, e.g. marketing, there where obvious consequences to the results considering placement of images etc. While collecting more knowledge on student behaviour is a valid endeavour in itself some reflection what kind of knowledge will be useful in teaching practice should guide the procedure. E.g. a focus on comparing reactions to graphically or syntactically different representations of the same algorithm could provide applicable advices. Otherwise eye-tracking might just lead to tautologic results like "difficult algorithms are difficult to read".

6. REFERENCES

[1] R. Bednarik, T. Busjahn, and C. Schulte, editors. *Eye Movements in Programming Education: Analyzing the Expert's Gaze: Proceedings of the First International Workshop*, volume 18 of *Publications of the University of Eastern Finland. Reports and Studies in Forestry and Natural Sciences*, 2014.

[2] M. E.Crosby, J. Scholtz, and S. Wiedenbeck. The roles beacons play in comprehension for novice and expert programmers. In *Proc. PPIG 14*, pages 84–89. Brunel University, June 2002.

[3] F. Pellegrino, C. Coupe, and E. Marsico. A cross-language perspective on speech information rate. *Language*, 87(3):539–558, 2011.

[4] J. Piaget. *La naissance de intélligence chez l'enfant.* Delachaux et Niestlé, Neufchâtel, 1959.

[5] C. Schulte, T. Busjahn, and E. Kropp. Developing coding schemes for program comprehension using eye movements. In *Proceedings of 25th Annual Psychology of Programming Interest Group Annual Conference*, pages 84–89. University of Sussex, June 2014.

Understanding a Novice Programmer's Progression of Reading and Summarizing Source Code

Andrew Morgan
Software Engineering Research and Empirical Studies Lab
Department of Computer Science and Information Systems
Youngstown State University
Youngstown, Ohio 44555 USA
asmorgan@student.ysu.edu

Bonita Sharif
Software Engineering Research and Empirical Studies Lab
Department of Computer Science and Information Systems
Youngstown State University
Youngstown, Ohio 44555 USA
bsharif@ysu.edu

Martha E. Crosby
Department of Information and Computer Sciences
University of Hawaii at Manoa
Honolulu, Hawaii 96822 USA
crosby@hawaii.edu

ABSTRACT

The paper presents observations over the course of three months on the patterns and strategies a novice programmer (DO 21) uses while reading source code. The programmer was asked to read and summarize a program after completing three sets of lessons from an online course. Results indicate that the method of reading source code gets harder as the novice attempts to comprehend more difficult concepts. The analysis is presented in the form of a case study.

Keywords

eye tracking, source code reading, program comprehension strategies, computer science education

1. INTRODUCTION

Most universities teach students to start writing code early in introductory programming classes, without teaching them to read the code for understanding first. The task of comprehending code and the process used to teach students this core skill is at least as important as the task of writing code. In order to understand the process of reading and understanding code, a team of researchers from Freie Universitat Berlin and the University of Eastern Finland organized the first workshop on analyzing the expert's gaze held in Finland in November 2013. This year, the focus of the Koli workshop is on analyzing gazes of novice programmers.

2. METHOD OVERVIEW

A brief description about the method and study is now given. All participants of the workshop were granted access to three eye tracking sessions (data, visualizations, and videos) of one novice. Each of the eye tracking sessions were held after the novice completed certain lessons (namely lesson 1, lesson 4, and lesson 6) from an online Introduction to Java Programming Udacity course[1]. The novice was asked to study the program for as long as they wished and then provide a summary. The novice is referred to as DO21 in the paper. An optional data set for another novice named EU10 was later provided, but was not mandatory for analysis. The novices were both female and didn't have much experience programming. Workshop participants were urged to describe the data in terms of stages of development and asked for general thoughts on how this type of data could be analyzed. The eye tracking sessions were conducted on May 5th, June 16th, and July 14th of 2014 respectively.

[1]https://www.udacity.com/course/cs046

3. OUR PREDICTION

Before we had a chance to look at the data, we were asked about what we expect to find in the reading behavior and also what ideas we had on the progress that could be perceived. Our predictions follow. If the student has been progressing well through the course, we would expect to see the student getting better at comprehending the source code given and becoming more efficient within their analysis. The reading behavior should get more structured as the student progresses through the course. This more structured approach is further clarified as the individually unique technique the programmer uses to understand the presented source code. The approach, along the way, will become more structured as the programmer understands his or her own techniques to interpreting such code. This reading behavior should also then focus on the important parts of the code. What is important will vary based on what the task is. For example, a bug finding task would involve different reading behaviors compared to a task that just tells the subject to look over the code and give an overview. At the time of this prediction, we were not aware of the task (summarization) in this case. We also predicted that they might find the answer quicker after lesson 6 when compared to the one after lesson 1. Every programmer has a different workflow they follow, however given enough subjects, there should be some commonality that can be extracted. So how do we measure progress? We could record time to complete task as one example. Depending on how long these files are, we could possibility also segment them into intervals and compare them.

4. ANALYSIS

We now present our analysis of each of the three eye tracking sessions namely lesson 1, lesson 4, and lesson 6. These

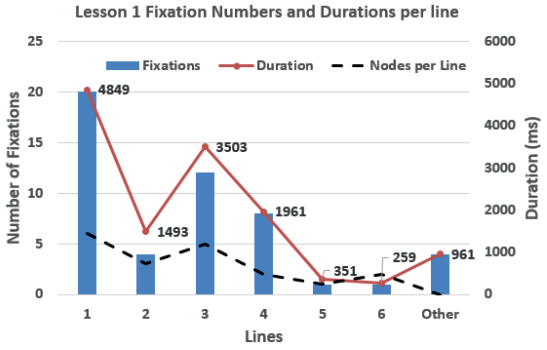


Figure 1: Fixations, Durations, and Nodes per line for Lesson 1 (pseudocode)

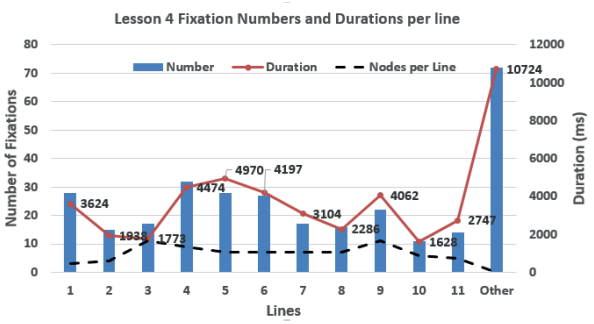


Figure 2: Fixations, Durations, and Nodes per line for Lesson 4 (CalculateAverage)

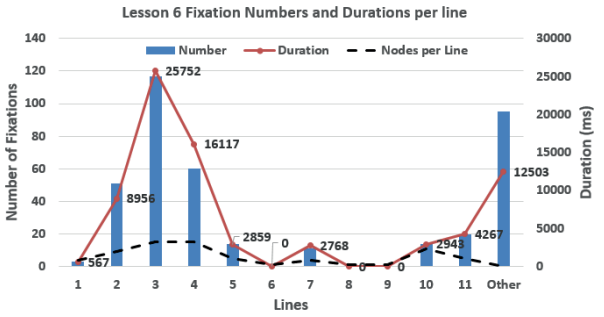


Figure 3: Fixations, Durations, and Nodes per line for Lesson 6 (PrintPattern)

lessons are named after the last lesson the novice did online. For each of the lessons, we present a graph outlining the number of fixations, total duration, and nodes per source line and provide some discussion about them. Note that even though a line graph is used to show duration and nodes per line, there is no implicit connection between duration or nodes. The novice DO21 also provided an accurate summary (the main task considered for this workshop) after she read through all the code in each lesson.

4.1 *Lesson 1* Analysis

The first lesson's recording was taken after the novice had six days of online lessons. The six line program was written in pseudo code and contained a for loop with an embedded "if - else" statement. The novice seemed to read the code as though it were text. Lines 1 and 3 received the most fixations and also contained the most nodes. We refer to a node as an area of interest in the data files. For example, a node could be individual words and phrases contained in a statement. See Figure 1. We did not see any continual regressions, however, we noticed that she read the entire program twice. In this particular case, the time spent reading the lines correlated with the number of fixations on those lines (which is not always the case). This type of behavior is very similar to what we would expect of reading text in a natural language. We also noticed that there were 4 fixations totaling 961 ms that did not fall on any given line in the source code.

4.2 *Lesson 4* Analysis

After lesson 1, the novice learned about objects and classes. The eye-tracking recording for lesson 4 was done 42 days after the recording for lesson 1. Refer to Figure 2. This source code snippet contained a Scanner object in which input was saved and later used for showing the average on the screen. The program was 11 lines long (we excluded the last two lines with braces since no fixations were detected in that area). The last two lines were mainly brackets so it could be that the subject perceived with peripheral vision that the brackets were there or could have also taken for granted that the program was bug free with no need to check for braces. It is also possible that the student might have already learned that the braces were of little importance. Most of the fixations focused on lines 4, 5, 6, and 9. Line 4 created the Scanner object. Lines 5 and 6 read in an integer and line 9 did the averaging of two numbers. If we compare the behavior of this subject with Lesson 1, it appears that this task was more difficult for her to solve. As DO21 tries to understand the code and build a mental model, she checks areas previously read. She reads through the program twice, i.e. we find two epics, from beginning to end. Between these two epics, we observed some sort of searching behavior. There were some regressions during the searching phase, where things were not looked at sequentially. The number of nodes, fixations and duration do not correlate in any particular way. In this lesson, there were 72 fixations (10,724 ms) that did not fall on any given line in the code.

4.3 *Lesson 6* Analysis

This lesson was recorded 29 days after Lesson 4. During this time, they covered decisions and loops. The source code contained a method called from within the main function. There was a nested for loop that printed stars in three rows with each row having one additional star than the previous. Refer to Figure 3. A lot of time is spent reading through the nested for loop in the method. Line 3 was the line most focused on, followed by line 4 and line 2 (method signature). DO21 spent nearly half a minute on line 3 throughout the session, which was the first for loop in the nested for construct. The total time spent in the method body was around 47 seconds (47,496 ms, 202 fixations) with about 9 seconds spent on the method signature (8,956 ms, 51 fixations), the third most looked at line in the program.

In this session, we counted about 7 epics (times DO21 went through the program from beginning to end). The

first time the novice read this program, the first focus was on lines 2 through 4 to understand what the method was doing. Later, the programmer proceeded to look at the main method. However, most of the gazes were focused on the method declaration's body. There was very little searching behavior and a lot more continual regressions between the lines in the nested for loop indicating a higher cognitive load because of higher task difficulty. In this lesson, there were 95 fixations totaling 12,503 ms that did not fall on any given line in the code.

4.4 Internal Testing for Further Analysis

A brief overview of the Java topics was introduced to a local novice programmer at Youngstown State University (Y10). In order to fully understand such data, the same tests were performed on this participant for each of the five source code snippets (combination of DO21's and EU10's). All regulations were similar to those for the workshop, and the programmer answered correctly to all summaries of the code. The only difference is that we conducted this small experiment all in one sitting.

When we compare Y10's eye gaze fixations to the two earlier subjects, we do see similar correlations. Y10's data had a tendency to experience Lesson 1 with a reading type behavior, while other lessons followed with a more problem solving type path with longer fixations and more focus on specific statements to understand meaning. Y10 answered all the summary questions correctly, however he took much longer time in terms of fixation duration for interpreting such code. In comparison, Y10 took up to two times the duration compared to DO21, and up to four times the duration as EU10.

5. STAGES OF DEVELOPMENT

Several studies describe the process of program comprehension but the evidence of *how* and *why* programmers perceive code is limited. Most studies explain *how* not *why* people read and comprehend programs. In the process of establishing a methodology for studying program comprehension, Weissman [25] found that initially students encountered problems with constructs of the programming language but eventually they were able to extract the programs meaning. By systematically investigating the effect on program comprehension of interactions between knowledge of the gist, features of the text and participant differences, it may be possible to determine when paradigm shifts (or stages as suggested by Flavell [7]) emerge.

Research suggests that stage shifts occur as novices become experts. Adelson [1] shows experts rely on abstract problem descriptions to understand code using semantics while novices are driven more by syntax and other categorization strategies. Davies [6], Gilmore and Green [8], Green and Navarro [10], Rist [18], Soloway and Ehrlich [21] and Bertholf and Scholtz, [3] argue that experienced programmers use programming plans during the comprehension process. Little is known about the progression of the processes involved as novices become experts. Evidence suggests that some people are more skilled than others, independent of the number of years programming [11]. However, the underlying reasons remain elusive.

Program comprehension has been described as 1) top-down by Brooks [4]; 2) bottom-up by Basili and Mills [2]; Shneiderman and Mayer) [20]; 3) knowledge based by Letovsky and Soloway [12], 4) as-needed by Littman et al. [13] and Soloway et al. [22]; 5) control-flow based by Green [9], Navarro-Prieto [14] and Pennington [15] and 6) integrated by von Mayrhauser [24]. Research by Clayton, et al. [5]. Shaft and Vessey [19] and von Mayrhauser and Vans [23] indicates the top-down approach is used to scan through source code. While bottom-up is used if people are unfamiliar with a particular application domain. While the integrated model of program comprehension is compelling, there is not clear evidence to support this model.

Application domain knowledge has been shown beneficial for program comprehension. People that are familiar with a domain tend to understand programs better than people that are not familiar with the domain [17]. Pennington [15], Petre et al. [16] and Navarro et al. [14] studied the mental representations used during program comprehension. Their studies present a model of how people build a mental image when trying to understand code. However, it is difficult to extract meaning from scan patterns alone. How do they relate to other studies that focus on models of program comprehension? Can scan patterns be classified in a meaningful way to clarify stages of comprehension? Comparing the scan patters of participants who understand the programs gist versus participants who do not may give insight into when paradigm shifts or stages occur.

6. SUGGESTIONS

One method of determining a novice's progression would be to show them source code that was similar to lesson 1 at lesson 4 and lesson 6. Similarly, it is necessary to show them source code similar to lesson 4 at lesson 6 in time. We can then see the learning that has occurred of the concepts learned at earlier sessions. Since this was not done in this study, we are not able to say for sure, but only guess as to what learning occurred. Another point is to design tasks that take advantage of the kind of mental structure they use to solve the problem. This can bring out the problem solving nature of the task that these programs analyzed did not have, even though they were semantically rich in syntax.

Regarding what points in time need to be examined more closely, the answer will really depend on what we are trying to determine. If our goal is to find stages in development then it is going to take a longer time to follow the person and have the person be their own control. This will help us determine when the novice actually exhibits expert like behavior (if ever) and this is the point at which we can say that the novice has started using "chunks" for example and behaves more like an expert. For example, we could determine if the novice has now started building hierarchical tree like representations to solve the task or if they still focus on a flattened out tree with no clue as to which path to take. This change of representation needs to come through with a good selection of tasks.

To fully understand the kind of data presented here requires multiple levels of analysis. The videos and fixation graphs by time do help. One thing that is also important is time spent at each fixation. The big circles are indicative of the task getting harder. Sometimes there could be a few fixations but a lot of time spent on them. Points of dis-connectivity in the fixation time line graphs also need to be examined (could imply cognitive load or thinking and reasoning).

We did not numerically analyze the additional EU10 dataset

since the programs used are syntactically and semantically a lot different at lesson 1 and lesson 4 compared to DO21. It would not be appropriate to compare them side by side. Even though the same program was used for lesson 6, EU10 did not summarize the code correctly, so we only provided a brief visual comparison.

7. DISCUSSION AND CONCLUSIONS

We predicted initially even before we saw the eye tracking sessions that the reading would become more structured. Reading pseudocode was more like a reading task, which in turn required two linear epics within, to understand. Whereas, the other two were also classified as reading but it was getting harder for DO21 to read the more complex constructs. DO21 did provide all correct summaries to the programs. The fact that the reading got more difficult can be seen in the data and videos of the sessions. It could be that the keywords used and structure of the program caused this to occur. Unfamiliarity with the methods can also cause this to happen.

We also noticed a lot of fixations that fell either on blank space on the screen or outside the screen (where the novice looked at something other than the computer screen). They may also have closed their eyes briefly to think about the task at hand. In the timeline graphs provided, these out-of-screen or out-of-line fixations can be seen as breaks in the line graphs. These breaks mainly occur during the searching that happens between the epics.

In visual comparison between Lesson 6 tests' between DO21 and EU10, we can again see the apparent time difference between the two. While DO21 appears to interpret presented code in a problem solving form, EU10 performs fewer epics on the code itself, along with spending less duration on the nested for-loop. The fixations for EU10 scattered the length of the code, compared to focusing such cognitive load on the main construct of the program. In return, the summarization of the code was answered incorrectly due to insufficient understanding within the for loop itself.

We were not able to identify clearly any stages in the videos or data. One might argue that each of the videos could be split into three phases where they first read the program and recognized it, followed by analyzing in detail, followed by a conclusive stage. However, this is not apparent while viewing the videos closely or looking at the numbers. It is too early in the learning process of a difficult skill to find such stages. It is unclear if they are chunking, for example. Stages are a profound shift in understanding and we didn't see this in the sessions presented.

Another possibility is to distinguish problem solving from reading. It might also appear that the novice is doing some form of problem solving in lesson 4 and lesson 6. Lesson 1 appears to be a reading task. However, in order for the problem solving theory to hold, they had to be working on a different type of task. Problem solving is a major research area in cognitive science. In programming, problem solving is a key skill. However, we do not believe the programs triggered any problem solving type of behavior. The task represented does not lend itself well to solving a problem where the participant requires to build and change their mental representation. In addition, eye movements alone cannot be used to show this. So we conclude that the novice DO21 is really in the process of reading and understanding the code in all lessons but due to the nature of the constructs used, things are getting harder to read. Do things get easier as time goes on? We didn't see it yet but it might in the longer future. This is when stages in behavior might become more apparent.

8. REFERENCES

[1] B. Adelson. *Structure and Strategy in the Semantically-Rich Domains*. PhD thesis, 1983.

[2] V. R. Basili and H. D. Mills. Understanding and documenting programs. *IEEE Trans. Software Eng*, 18:270–283, 1982.

[3] S. J. Bertholf, C. F. Program comprehension of literate programs by novice programmers. *Empirical Studies of Programmers: Fifth Workshop*, page 222, 1993.

[4] R. Brooks. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, 18:543–554, 1983.

[5] R. Clayton, S. Rugaber, and L. Wills. On the knowledge required to understand a program. *Working Conference on Reverse Engineering*, 1998.

[6] S. P. Davies. The nature and development of programming plans. *International Journal of Man-Machine Studies*, 32:461–481, 1990.

[7] J. H. Flavell. Stage-related properties of cognitive development. *Cognitive Psychology*, 2:421–453.

[8] D. J. Gilmore and T. R. G. Green. Programming plans and programming expertise, the quarterly. *Journal of Experimental Psychology*, 40A(3):423–442, 1988.

[9] T. R. G. Green. Cognitive approaches to software comprehension: results, gaps and limitations. *Extended abstract of talk at workshop on Experimental Psychology in Software Comprehension Studies 97*, 1997.

[10] T. R. G. Green and R. Navarro. Programming plans, imagery, and visual programming. *In Nordby, K., Helmersen, P. H., Gilmore, D. J., Arnesen, S. (Eds.) INTERACT-95*, pages 139–144, 1995.

[11] A. J. Ko and B. Uttl. Individual differences in program comprehension strategies in unfamiliar programming systems. *11th IEEE International Workshop on Program Comprehension (IWPC'03)*, 2003.

[12] S. Letovsky and E. Soloway. Delocalized plans and program comprehension. *IEEE Software*, pages 41–49, 1986.

[13] D. C. Littman, J. Pinto, S. Letovsky, and E. Soloway. Mental models and software maintenance. *J. Syst. Softw.*, 7(4):341–355, Dec. 1987.

[14] R. Navarro-Prieto. Mental representation and imagery in program comprehension. *Psychology of Programming Interest Group, 11th Annual Workshop.*, 1999.

[15] N. Pennington. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology*, pages 295–341, 1987.

[16] B. A. F. Petre, Marian. Mental imagery in program design and visual programming. *International Journal of Human-Computer Studies*, pages 7– 30, 1999.

[17] V. Ramalingam and S. Wiedenbeck. An empirical study of novice program comprehension in the imperative and object-oriented styles. *7th Workshop on Empirical Studies of Programmers*, 1997.

[18] R. S. Rist. Plans in programming: Definition, demonstration and development. *In E. Soloway and S. Iyengar (Eds.), Empirical Studies of Programmers*, 1986.

[19] T. M. Shaft and I. Vessey. The relevance of application domain knowledge: The case of computer program comprehension. *Information Systems Research*, 6:286–299, 1995.

[20] B. Shneiderman and R. Mayer. Syntactic semantic interactions in programmer behavior: A model and experimental results. *Intl. J. Comp. and Info. Sciences*, 18:219–238, 1979.

[21] E. Soloway and K. Ehrlich. Plans in programming: Definition, demonstration and development. *Empirical Studies of Programming Knowledge. IEEE Transactions on Software Engineering*, pages 595–609, 1984.

[22] E. Soloway, J. Pinto, S. Letovsky, D. Littman, and R. Lampert. Designing documentation to compensate for delocalized plans. *Communications of the ACM*, 31:1259–1267, 1988.

[23] A. von Mayrhauser and A. Vans. Comprehension processes during large scale maintenance. *16th International Conference on Software Engineering*, 1994.

[24] A. von Mayrhauser and A. Vans. Program understanding: Models and experiments. *Advances in Computers, M. C. Yovits and M. V. Zelkowitz, Eds. Academic Press Limited*, 40, 1995.

[25] L. Weissman. *A methodology for studying the psychological complexity of computer programs*. PhD thesis, 1974.

Primary investigation of applying Hidden Markov Models for eye movements in source code reading.

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

ABSTRACT

Source code comprehension is closely connected with reading process. To evaluate different factors that influence the reading process, we propose to build a mathematical model. This paper describes basic steps for building the eye movement analysis model during source code reading. We present the test model and the method of developing this kind of models.

The developing method of transforming the source code in Java programming language to abstract semantic elements and coded alphabet is presented here. The basic Hidden Markov Model (HMM) is created as fitting to the given source code. The results arouses more questions than provides answers or insights, but "Once you do know what the question actually is, you'll know what the answer means..." (Douglas Adams)

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous

Keywords

Source code, reading models, eye-tracker, HMM

1. INTRODUCTION

Source code reading is a highly complex process, in comparison to natural text reading, and involves many cognitive process. Eye movements that follow letters, math symbols and operants form curious graphs: from line to line, form top to bottom and back. Backwards – eye jumps are normal for source code reading (Crosby and Stelovsky, 1990), but they are very rare for normal or natural text reading (Rayner et al., 2009; Reichle et al., 2003; Nilsson and Nivre, 2009; Kotani et al., 2010). Reading process seems to correlate with information processing to reach the aim of understanding the semantic meaning of text. That is why the reading process in general is an interesting field of scientific investigations.

International Workshop at the 9th WiPSCE Conference on Computing Education Freie UniversitÃd't Berlin, Germany, November 7th - November 8th, 2014

Human vision process and visual perception is based on eye movements. Human eye moves even when a person is asleep. Eight different types of eye-movements are identified, but for analysis of information processing two basic types are used. The first type is fixations – very slow eye movements, that drift around one physical point. They are not an absolutely freezing state, but are still called fixations. Second type of eye movements are saccades. The saccade is a ballistic type of eye moments. The information processing is usually associated with fixations (Bridgeman et al., 1994; Otero-Millan et al., 2013). To simplify it is possible to say that the vision process proceeds in the following way: ... saccade -> fixation -> saccade -> fixation ... This rough approximation gives us an explanation of visual process as a potential system for modelling. To find the connection between vision process and the way people understand natural language several mathematics models were duilt(Reichle et al., 2003). These models can be used as an abstract concept for checking scientific hypotheses, for example, the question about next eye-movements fixation and about the lengths of saccade (like E-Z reader model (Reichle et al., 2003)).

From applied point of view, eye-movement reading models could be useful for differentiation between expert reader and novice, or for evaluation of influence of extra factors on reading process (and perception in general). That is why we decided to investigate eye-movement model for reading of the source code. Our motivation is to define main elements of source code and to find ways of model the process of viewing these elements.

2. EYE MOVEMENTS IN SOURCE CODE READING

The basic study by Crosby and Stelovsky shows differences in reading of source code(Crosby and Stelovsky, 1990). Figure 1 shows the fixations and lines – saccades from one fixation to another. We can suggests that information from source code is processed from fixation mostly. As fixations focus on distinct elements of source code, we should decompose the source code into basic elements. Identification of text elements is referred as annotation and is quite popular in this area of studies (Busjahn et al., 2014; Turner et al., 2014), therefore in order to understand how the information is processed we should annotate each fixation. We could not find any basic list of semantic annotation elements in previous studies. It seems that this kind of list should be different for different programming languages. For the given source code sample we defined the list of abstractions as follows:

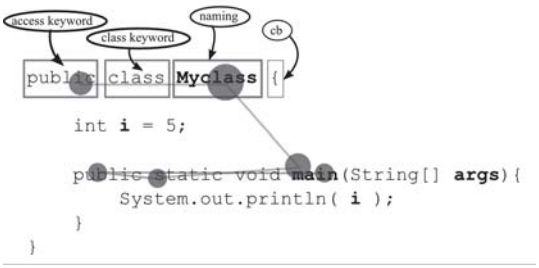


Figure 1: Eye movements during source code reading. First line is annotated with semantic abstraction elements.

- Definition. name(method, class, variable). -> N
- Method calling. -> Mc
- Variable using in math. operator. -> Uv
- Operator -> O
- Constant using in math. operator. -> Const
- Type in definition. -> T
- Semicolon and curved brackets. -> Cb
- Static. Definition. -> S
- Access specifier. Definition. -> A

The list is not full, but it was built from the original source code. The eye movements create patterns and strategies of how the source code is read(Busjahn et al., 2014). Analysis of these ■perception chunks■ is not a trivial process (Gobet and Simon, 1998; Cant and Jeffery, 1995). It is possible to count the number of fixations for each semantic block, for example we can count all fixations in ■access keyword■ block and, we can calculate total fixation counts and total processing time. If we code all abstract semantic blocks by letters we will have a picture like on figure 2.

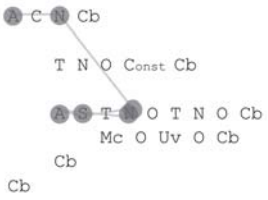


Figure 2: The corresponding of eye movements to annotation to abstract semantic block. Block coded to letters. Gaze fixations shown as ellipses.

One of the problem of comprehension the reading process are the two dimensions of stimuli. There are lines with text, even if the line consists of only one symbol like "{". Lets us modify this 2D stimuli into one dimension vector:

$$A, C, N, Cb, T, N, O, Const, Cb, A, S, T, N, O, T... \quad (1)$$

After this transformation we can analyse reading process in a why the alphabet is process. The example this transmission shows the list of abstractions after the "->" symbol (Access specifier. Definition. -> A).

3. HIDDEN MARKOV MODELS

The main thesis of Hidden Markov Models(HMM) was formed and published between 1960 and 1970 by Baum and colleagues. Today HMM are very useful in various information technology fields, like speech and image processing(Baum and Petrie, 1966; Baum et al., 1967, 1970).

Let us consider the concept of HMM in the ralation to our alphabet and observable letter sequence of abstract semantic elements. In the figure 3, we see 3 levels. The last, third level is the hidden level of HMM it self. This is the sequence of the states. The probability for the HMM moves to the next stage defined by transition probability matrix - *a*. In each time HMM could be in the one single state. The number of states is limited.

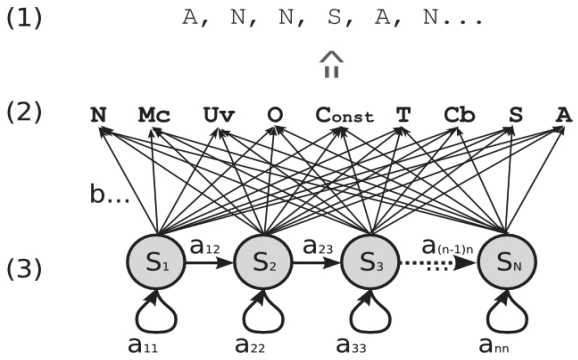


Figure 3: The concept of HMM with source code abstractions as an observed sequence. 1) Level is an observable sequence of gaze fixations on source code elements that corresponds to semantic abstraction code alphabet. 2) The alphabet of abstractions it self. 3) Hidden level of stats of HMM

The second level is our alphabet of semantic elements. Each HMM state corresponds to some probability *b* with letters from it. It means that if HMM is, for example, in a state one, there is a probability of b1(N), for choosing letter N, probability b1(Mc) for choosing letter Mc and so on. Finally we have the first level - level one). This level is a real observable sequence of letters from our semantic abstraction alphabet. The observation corresponds to gaze fixations on text elements of source code. In a speech recognition field the transition matrix *a* corresponds to the time of being in current state for HMM. There are some classical assumptions for the applications of HMM (see (Baum and Petrie, 1966)).

4. BASIC EYE MOVEMENT MODEL FOR SOURCE CODE READING

To build the basic eye movement model for source code reading we used our alphabet of abstraction. We annotated each fixation and found suitable letter for them. As a result we got the observable sequence of a person's source code reading. To build a HMM from that sequence we used K-means learner algorithm and Jahmm Java library. We create three states for our model. Finally, the model on figure 4 was created.

This model was tested on the second subject (second source

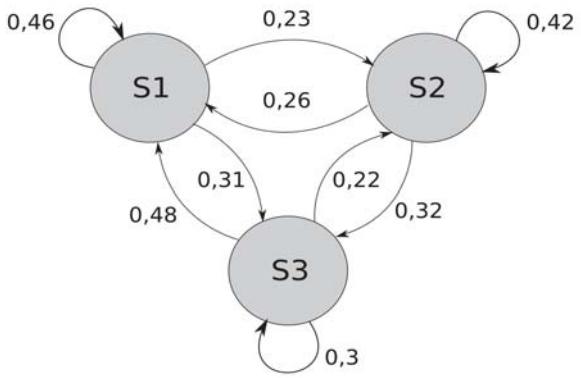


Figure 4: The 3 stats basic HMM for eye movement during source code reading.

code reader), but obviously we didn't receive brilliant results. In conclusion we could propose, that the current eye movements model of source code reading is just a concept for the future investigation and studying.

5. DISCUSSION

There are some limitations in our approach. First of all, our model will work only in one way: it is impossible to correlate abstraction semantic blocks with the real text elements (words) in the source code uniquely. This back-way process is based on probability.

The next aspect is that we understood is that each text element of source code has one abstraction. We can compare this with hieroglyph reading (Osaka, 1992), where each fixation corresponds with one hieroglyph. There are several fixations for the one six letter word in european languages. Variable naming occured in long words or even phrases, for example like this: *bufferedReader.readLine*. Taking into account this fact, we should integrate the natural reading model approach into the model of source code reading.

That list of abstract semantic elements is not full. Perhaps it should be done more accurate to include each computer language term as a separate semantic bloc. It means, that terms *public* and *private* should be in separate abstractions (in current annotation scheme they are in the one abstraction block - *Access specifier. Definition.*) Experimenting with different levels of abstraction is probably a good way to test the model's possibilities and limits.

Next question is the number of states for the model. In this case we decided to include three states, like in speech recognition algorithms. The question about an appropriate number of states should be investigated more.

6. REFERENCES

References

Baum, L. E., Eagon, J., et al. (1967). An inequality with applications to statistical estimation for probabilistic functions of markov processes and to a model for ecology. *Bull. Amer. Math. Soc*, 73(3):360–363.

Baum, L. E. and Petrie, T. (1966). Statistical inference for probabilistic functions of finite state markov chains. *The annals of mathematical statistics*, pages 1554–1563.

Baum, L. E., Petrie, T., Soules, G., and Weiss, N. (1970). A maximization technique occurring in the statistical analysis of probabilistic functions of markov chains. *The annals of mathematical statistics*, pages 164–171.

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

Busjahn, T., Begel, A., Orlov, P., Sharif, B., Hansen, M., Bednarik, R., and Shchekotova, G. (2014). Eye Tracking in Computing Education Categories and Subject Descriptors. In *ICER '14 Proceedings of the tenth annual conference on International computing education research*, pages 3–10, Glasgow, Scotland, United Kingdom. ACM New York, NY, USA Âl'2014.

Cant, S. N. and Jeffery, D. R. (1995). A conceptual model of cognitive complexity of elements of the programming process. *Information and Software Technology*, 37(7):351–362.

Crosby, M. E. and Stelovsky, J. (1990). How Do We Read Algorithms ? A Case Study. *Computer*, 23(1):24–35.

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

Kotani, K., Yamaguchi, Y., Asao, T., and Horii, K. (2010). Design of Eye-Typing Interface Using Saccadic Latency of Eye Movement. *International Journal of Human-Computer Interaction*, 26(4):361–376.

Nilsson, M. and Nivre, J. (2009). Learning Where to Look : Modeling Eye Movements in Reading. In *Proceedings of the Thirteenth Conference on Computational Natural Language Learning (CoNLL)*,, number June, pages 93–101, Boulder, Colorado. Association for Computational Linguistics.

Osaka, N. (1992). Size of saccade and fixation duration of eye movements during reading: Psychophysics of japanese text processing. *JOSA A*, 9(1):5–13.

Otero-Millan, J., Macknik, S. L., Langston, R. E., and Martinez-Conde, S. (2013). An oculomotor continuum from exploration to fi xation. In *Proceedings of the National Academy of Sciences 110.15*, pages 6175–6180.

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

Reichle, E. D., Rayner, K., and Pollatsek, A. (2003). The E-Z reader model of eye-movement control in reading: comparisons to other models. *The Behavioral and brain sciences*, 26(4):445–76; discussion 477–526.

Turner, R., Falcone, M., Sharif, B., and Lazar, A. (2014). An eye-tracking study assessing the comprehension of c++ and Python source code. In *Proceedings of the Symposium on Eye Tracking Research and Applications*, pages 231–234, Safety Harbor, Florida. ACM.

Notes on Eye-Tracking Data from a Novice Programmer

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

ABSTRACT
This paper presents notes on issues related to analysing eye-tracking data from a novice programmer progressing through an introductory course in Java programming. The data are used to make observations on evidence of development of code reading strategy, and these are related to models of program comprehension. Some suggestions for the timing of studies within the learning process and for useful visualisations and tools are included.

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

General Terms
Experimentation, Human Factors.

Keywords
Eye tracking, code reading, computing education

1. INTRODUCTION
Gaze data collected using an eye-tracking device can provide insights into problem solving, reasoning and search strategies, and eye trackers have been used in a wide range of fields such as HCI and human factors (Poole and Ball, 2006) There is considerable interest now in the use of eye-trackers to study program comprehension in both experienced and novice programmers, and there is potential to gain an understanding of thought processes to inform approaches to teaching and learning programming.

A workshop was held in 2013 to analyse eye-tracking data gathered from expert users reading code examples, to develop a scheme for coding features in the data with relation to program elements and reading strategies, and to consider the implications and potential for future research (Busjahn et al., 2014).

This work follows up on that workshop by transferring the focus to data gathered from a novice programmer progressing through examples from an introductory Java programming course. The aims are to find out whether developmental stages in the novice's thought process can be identified and to suggest approaches to eye-tracking research specifically related to the study of novices.

2. DATA
Data were gathered for a single learner, with no previous experience of Java, at three stages in an introductory course in Java programming. At each stage the participant was given a short sample program and asked to summarise the function of the program. The first sample, following on from a basic introductory lesson, was simple pseudocode, while the others were Java code. In the second example, which reads in two numbers and calculates their average, all the significant code is inside a main method and there are no control structures, while the third example consists of a class with a static method and a main method which calls that method and passes a parameter to it. The code in the main method consists of two nested for loops which draw a pyramid of astersisk symbols with extent determined by the method parameter.

These examples correspond to the types of programs which the participant would be expected to understand at the relevant stage of her study. The responses to the post-task questions indicate that the participant has been successful in understanding the overall purpose or result of each program.

Data was presented for analysis in a number of forms, including videos showing detected fixations in real time, scan path diagrams each showing all fixations and saccades from one experiment, superimposed on the sample code and timelines showing fixations by line of code graphed against time. Each sample program was divided into lines and also further divided into areas of interest (AOIs) within lines, and each fixation was where possible associated with the closest line and AOI.

3. STAGES OF DEVELOPMENT
One of the outcomes of the previous workshop was the refinement of a coding scheme which allows observed fixations to be expressed in terms which reflect the participant's thought process, in terms of program elements (singly or compounded into larger elements such as parameter lists or blocks), patterns and strategies. The latter two categories reflect higher-level views of the participants' actions or aims and require interpretation of the data.

It is interesting to consider the way a novice programmer's development might be reflected in the fixation codes that are present in their gaze data. The capabilities of a novice programmer (hopefully) expand rapidly during a course, and hence the range of codes likely to be identified should expand. The novice develops in terms of program knowledge, building upon previous lessons and learning how to use new constructs. Stages of program knowledge development may be strongly influenced by the course structure, for example "objects-first" or "objects-later". In parallel with this there should be development of skills generally related to writing and reading code, for example in terms of applying new strategies to solving problems and assimilation of programs while reading code.

The expected development of program knowledge was reflected in the code for the three samples. For example the first contains no Java. Only the final sample contains a method and associated method call within the code presented, so codes within related categories, such as Signature and Method Call will not be present before then. An interesting observation on the second sample is the large number of fixations on lines of code which import the Scanner class and instantiate a Scanner to be used to get user input. These lines, while necessary, are essentially irrelevant to the overall purpose of the program. The amount of attention paid

to these might indicate that the use of Scanner is unfamiliar to the participant who therefore needs to deduce their purpose. This might suggest a new code, for 'utility' or 'boilerplate' code which is required to make a program execute but whose detail is not fundamental to the purpose of a specific program. Arguably the details of the signature of the main method fall into this category also in cases where the program arguments are not used.

Development of general programming skills may be reflected in the emergence of patterns and strategies within the gaze data, and in characteristics of the data for each sample viewed as a whole. Some observations have been made of the novice programmer's gaze data to find whether there is evidence of this. The total time spent on each sample increases from 14 seconds in the first case, then 53 seconds and 86 seconds. This does not reflect the length of the code (the third sample has approximately the same number of lines as the second one) but does indicate a different thought process for the more complex structures in the third sample. In both the second and third cases there were many fixations off screen throughout which may indicate moments of thought about the information which has just been obtained by reading.

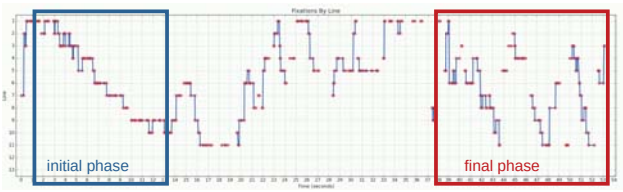


Figure 1. Timeline of participant's gaze for second program

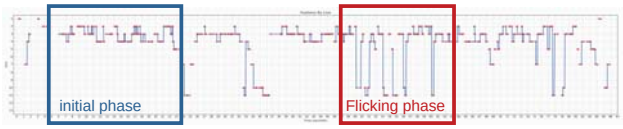


Figure 2. Timeline of participant's gaze for third program

The timeline for the second sample, shown in figure 1, shows evidence of the scan pattern (Uwano, 2006) or DesignAtOnce strategy]in which the participant reads the whole code first before focusing on specific parts. In fact. the second timeline shows quite distinct phases beginning with a scan and ending with a sequence of repeated movements which might be coded as TestHypothesis. This indicates some development of strategies which are not clearly present in the first.

However, these strategies are also not present in the third timeline, shown in figure 2. Here there is a phase early on which focuses on a small region of code that contains the definition of two nested for loops inside a method. There is a phase much later on where there is possibly evidence of Flicking between method definition and call. This may appear to be a retrograde step on the part of the participant in applying strategy. However, understanding the nested for loop statements in this example is essential to the task the participant was asked to do, and these statements carry more information than any other part of the program.. By determining how the loops interact it is possible to conclude what the program will do. The method call simply supplies an actual parameter which determines the number of rows in the pyramid. It could be argued that this approach shows development of the ability to apply a strategy which is best suited to a particular program.

4. ASSIMILATION PROCESS

Program comprehension by both expert and novice programmers has been the subject of an extensive body of research over many years (Schulte et al., 2010). A variety of assimilation strategies have been identified and described as, for example, top-down, bottom-up and opportunistic. Top-down models describe the assimilation process as one in which knowledge of the program domain is applied and mapped to the microstructure of the code, while bottom-up models involve starting with individual code elements and chunking these into higher-level abstractions. Opportunistic models reject the dominance of either of top-down or bottom-up processes.

Such processes describe the overall assimilation of the program and could be considered as a tier above the "strategy" and "pattern" tiers in the coding scheme, where the assimilation may consist of a number of phases each of which could include one or more strategy or pattern. It may be useful to consider the ways in which strategies coded in gaze data combine in phases to reflect specific types of assimilation process.

To take one example, in the bottom-up process described by Pennington (1987) two distinct models of the program evolve. The program model is essentially a control-flow model, while the situation model relates more closely to data flow. Programmers with a high degree of comprehension were observed to cross-reference frequently between these models, and an indicator of this might be successive phases in which DataFlow and ControlFlow strategies are coded.

Gaze data alone can only tell us directly about the participant's attention to features of the text surface, rather than the mental model of the function of the program, and any conclusions on the latter may be difficult to infer. Does the presence of DesignAtOnce in the initial phase indicate a specific assimilation process, for example top-down? This may apply in the case of the second sample as the participant discovers the domain or overall purpose of the program from the class name (CalculateAverage) and reads through the code to build an initial mapping. Other complementary data would be required to verify this, however, such as think-aloud protocols or textual analysis of responses to post-task questions using a coding scheme such as that described by Good & Brna (2004).

5. LEARNING PROCESS

The data presented here was taken from three points in time spaced throughout an introductory Java course. There is a question as to what points of time in a course will provide the most useful evidence of learners' development. Concentrating experiments in the first part of a course may capture rapid early development. This might be particularly the case for the "complete novice" who is experiencing programming for the first time. However, students may individually reach a stage of rapid development at different times as they cross some conceptual threshold, and it may be more useful to gather data fairly constantly throughout the course to give the possibility of capturing such transitions if they occur and determining different development processes among participants.

It may be useful to try to decouple the development of comprehension strategy from program knowledge by choosing samples at some stages which do not rely on the most recently acquired knowledge, but rather are similar in knowledge content to ones used at an earlier stage. Would the participant adopt a

different strategy for a similar problem as a result of having has developed more programming skills?

6. VISUALISATIONS AND TOOLS

Some visualisations were provided along with the data from the novice gaze experiments to aid in analysis. It is interesting to consider the usefulness of these, and what variations of these or new visualizations or tools would be of value.

Videos of each experiment presented the entire set of fixations and saccades in real-time, overlaid on the code text surface. Viewing the same data in eyeCode Fixation Viewer allowed starting and stopping and step-by-step viewing. Scan path diagrams showed all fixations at once overlaid on the code. These did not seem very useful here as the large number of fixations in each experiment made it difficult to discern particular areas of focus. It is possible that heat maps would be somewhat more helpful in highlighting clearly which AOIs received the most attention. However, while heat maps are widely used in eye-tracking research, the process of program assimilation seems more related to the movements between elements rather than the degree of focus on specific elements (although this may depend on the nature of the task given if something more specific than simply explaining the overall function of the program.

Timelines, as shown in figures 1 and 2, proved the most immediately useful in visualizing patterns and phases in this gaze data. Often, however, there are multiple successive fixations on a single line, and it would be useful for exploratory analysis to be able to expand each line in the timeline to see the sequence of fixations on the AOIs in that line. The overall clarity of the timeline may be diminished, however, and this may work best as an interactive visualization in which lines can be expanded as required and collapsed back to showing the line only.

The timeline shows temporal relationships clearly. Alternative visualisations showing spatial relationships may also be useful in identifying which other elements of the code were considered to be related to any given element. Again an interactive display would be useful for exploring the data. This display might be similar to that implemented by Ristovski et al. (2013) in their tool eCode, in which selecting an AOI interactively highlights other AOIs which the participant's gaze moved to immediately following from the selected one.

Visualisations based on raw gaze data with the text surface divided into lines or AOIs can be supplemented by visualisation of the data following coding (whether accomplished by manual or automated techniques), and the previous workshop demonstrated some useful approaches, for example based on the flow between specific identifiers or blocks.

Control flow in realistic programs passes between program components which are often in separate source code files. Even in introductory Java courses students may work with code distributed across several files. In most eye-tracking studies the sample program is presented to the participant as text within a single viewport. There may be value in integrating eye-tracking software with the IDE that is used when developing code. This may open up the possibility of analysing gaze for programs which are distributed across multiple source files. The IDE could potentially report to the eye-tracking software when a new code window is brought into view, although visualisation of this might be challenging. Furthermore, code editors in modern IDEs encapsulate understanding of the elements and structure of the code in order to offer features such as syntax and scope highlighting and context sensitive menus and information. This could be exploited to reduce the need for, or improve the reliability of, coding at tiers below the pattern tier.

7. REFERENCES

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

[2] Good, J., & Brna, P. 2004. Program comprehension and authentic measurement:: a scheme for analysing descriptions of programs. *International Journal of Human-Computer Studies*, 61(2), 169-185.

[3] Pennington, N. 1987. Comprehension strategies in programming. In *Empirical studies of programmers: second workshop* (pp. 100-113). Ablex Publishing Corp.

[4] Poole, A., & Ball, L. J. 2006. Eye tracking in HCI and usability research. *Encyclopedia of human computer interaction*, 1, 211-219.

[5] Ristovski, G., Hunter, M., Olk, B., & Linsen, L. 2013. EyeC: Coordinated views for interactive visual exploration of eye-tracking data. In I*nformation Visualisation (IV), 2013 17th International Conference* (pp. 239-248). IEEE.

[6] Schulte, C., Clear, T., Taherkhani, A., Busjahn, T., & Paterson, J. H. 2010. An introduction to program comprehension for computer science educators. *Proceedings of the 2010 ITiCSE working group reports*, 65-86.

[7] Uwano, H., Nakamura, M., Monden, A., & Matsumoto, K. I. 2006. Analyzing individual performance of source code review using reviewers' eye movement. In *Proceedings of the 2006 symposium on Eye tracking research & applications* (pp. 133-140). ACM.

Programming Code Reading Skills: Stages of Development Encountered in Eye-Tracking Data

Mareen Przybylla
University of Potsdam
August-Bebel-Str. 89
14482 Potsdam
+49 331 977 3104
przybyll@cs.uni-potsdam.de

ABSTRACT
In this position paper the methods used for analyzing eye-tracking data from code reading and comprehension experiments are described. Analyzing such data can contribute to our understanding about how program code is read and understood. The data to be analyzed were assigned to the author prior to a workshop on "Analyzing the novice's gaze" – a follow up for a similar workshop held the year before, where expert's data were analyzed. The aim was to find different ways of analyzing eye movement data. Experiences with and results of the analyses of the data are presented and ideas for further research and possible applications are discussed.

Categories and Subject Descriptors
K.3.2 [**Computers and Education**]: Computer and Information Science Education – *Computer science education, literacy.*

D.2.5 [**Software Engineering**]: Testing and Debugging – *Code inspections and walk-throughs, Diagnostics, Tracing.*

General Terms
Experimentation, Human Factors

Keywords
eye-tracking, eye movement, code reading, program comprehension

1. INTRODUCTION
When learning to program, code reading and comprehension is essential. Students need to be able to not only read and understand single lexical units, but also the overall functionality of programs. They have to be able to follow the program flow, the order in which the program is executed, in which methods are called, variables initialized, parameters handed over, etc. Otherwise they won't be able to find errors, debug their programs or modify existing programs in the manner intended.

The analysis of eye-tracking data from experts and novices can contribute to our understanding about how program code is read and how understanding is demonstrated. Prior to the analysis of the data, I would expect that when advancing their skills, students would show improvements concerning the understanding of program, concerning the time needed to get a grasp of the functionality of programs and concerning the identification of the relevant parts within program code to answer a particular question. In this position paper eye-tracking data of a novice programmer was analyzed prior to the workshop "Analyzing the novice's gaze" which is to be held at the WiPSCE conference in Berlin. The tasks given to students whose eye movements were tracked will be explained first, then the analysis method and results are presented and finally conclusions drawn and open questions discussed.

2. TASKS
The experiment took place after three particular lessons of a three month-long weekly java course at the Freie Universität Berlin (at the beginning of the term, mid-term and end of term). The participants were given short tasks after each of those lessons. The one that is going to be analyzed for this workshop was to read and understand the given code and then give a summary. The instruction given before the programs were presented was always the same: *Please read and comprehend the following source code. When you are done, press the left mouse button. Then you will be asked to give a SUMMARY.*

The programs given to the student were the following.

After lesson 1:

```
public class PrinterClass {
    public static void main ( String [ ] args ) {
        System.out.print ( "answer=" ) ;
        System.out.println ( 40 + 2 ) ;
    }
}
```

After lesson 4:

```
public class TextClass {
    public static void main ( String [ ] args ) {
        String text = "Hello World!" ;
        int positionW = text.indexOf ( "W" ) ;
        int textLength = text.length ( ) ;
        String word = text.substring ( positionW , textLength - 1 ) ;
        System.out.print ( text.replace ( word , "Sun" ) ) ;
    }
}
```

After lesson 6:

```
public class PrintPattern {
    public static void printMethod ( int numberOfRows ) {
        for ( int row = 1 ; row <= numberOfRows ; row ++ ) {
            for ( int col = 1 ; col <= row ; col ++ ) {
                System.out.print ( '*' ) ;
            }
            System.out.println ( ) ;
        }
    }
    public static void main ( String [ ] args ) {
        PrintPattern.printMethod ( 3 ) ;
    }
}
```

The task given to the student after each program was presented was always the same: *Please give a summary of the program.*

3. DATA ANALYSIS

Before analyzing the data, I went through the different tasks to set my expectations. The provided data was then analyzed through three stages. First, the videos containing visualizations of what the participant looked at were examined. Second, the timelines of each lesson were analyzed. Those two stages helped to identify the overall strategy of the participant on her way to understanding the code. Afterwards I studied the tables containing more detailed information about the fixation times in detail. This helped in identifying the accurate length of fixation times. I was particularly interested in those areas, where longer fixation times occurred. Only at the very end I looked into the existing coding scheme [1] and found several terms for what I found in the analysis. I wanted to stay away from the existing scheme for the beginning in order to analyse the data open-minded.

3.1 Getting an Impression

3.1.1 Lesson 1

A correct summary of the program would have been: this program displays the text “answer=42” (followed by a linebreak).

The summary given by the student (EU10) was:

> *answer =*
>
> *42*

This is both, too short for a summary and incorrect, as the actual output would not include a line break.

From the eye movement data I got the impression that the student tried to gain an overall understanding of the code in taking the approach of reading the whole program line by line. She then started jumping between the relevant bits for output construction. This occurred between seconds eight and twelve (fixations 26 - 38), which is between the two output lines three and four. Afterwards (with fixation 40, so exactly after half of the time used) she started over from the beginning. This second time she did almost exactly the same thing – either to understand what remained unclear or to check if her gained understanding was correct. This time, jumping back and forth happened between seconds 20 and 25 (fixations 60 - 80), again between the two elements that together form the output of the program (Figure 1).

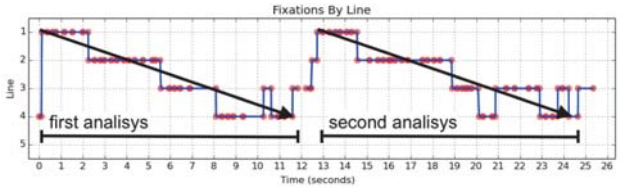


Figure 1. Method of analysis by EU10 after the first lesson.

This time, the jumps occurred more precisely between the expressions *print*, *println* and *answer=*, and were probably done to compare the two print methods. Presumably this is where the student decided that there should be a line break between the equal sign and the number 42 in the output.

In both cycles the jumps are probably done to find out about the relationship between the two parts of the output. This seems to be the hard part of understanding for the participant, as this is also the area in the code that was gazed at most of the time. In sum, 10.8 seconds, which is about 42% of the total time, she gazed at those three items.

The behavior of the student in this early phase of learning to program could be described as *double linear analysis* – an analysis that goes through the whole program line by line twice.

3.1.2 Lesson 4

A correct summary would have been: this program replaces the word “World” in the string “Hello World!” with the text “Sun” and displays “Hello Sun!”.

The summary given by the student was:

> *Replace Word with the string "Sun"*

This answer is not complete as it omits the output, but somehow correct. However, it is unclear if the participant’s typing error actually meant “World” instead of Word, or if she meant the variable word. Either way, this is what is replaced with “Sun” by the program.

From the eye movement data I got the impression that the student initially started with a strategy similar to the task after lesson 1. She read the code line by line, probably to get an overall understanding of what happens in the code. This is interrupted by short jumps amongst two lines between seconds five and ten (fixations 21 - 34), where she seems to think about the value to be assigned to the variable *positionW*. After completing this first cycle, in contrast to the previous time she never goes back to the class declaration and main method signature, which might indicate that she by now knows what those parts do and that no further analysis is needed there in order to find out about the functionality of the program (Figure 2). Instead, she constantly jumps between those lines within the code that actually create meaning (mostly lines 3, 6 and 7).

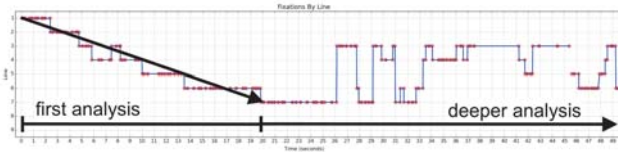


Figure 2. Method of analysis by EU10 after fourth lesson.

It seems that in order to understand the details, the participant moves her gaze between pairs of lines that reference each other. Obviously lines three and six where the original text and the word to be replaced are declared, are regarded as important for code comprehension by the student, similar to lines four and five, which define the beginning and end of the word to be replaced. There are several such observations, e.g. further meaning seems to be created between seconds 37 and 41 (fixations 121 - 122) and between seconds 42 and 45 (fixations 126 - 130), where slow linear movements indicate that the values for *positionW* and *textLength* could be determined. In the next 25 seconds the data shows rather long fixations on lines six and seven, possibly indicating that in thought the value for *word* and the output are generated. Another 20 seconds show eye movements between lines four, five, six and then three that might indicate a double check of the gained result in again looking at how the word to be replaced is identified from the text given.

While to this point the eye movements were explicable for the most part, what follows seems to be unsystematic and incomprehensible. Rapid movements between seemingly unconnected parts of the program code are followed by a long stare at *positionW* and *textLength* as if to find meaning in those two variables. Then, the very last part of the data again shows rapid movements across the code lines.

I would summarize the behavior of the student in this phase of learning to program as *linear-logical analysis* because in contrast to the earlier data, these data show clear examples of logically following the program flow.

3.1.3 Lesson 6

A correct summary of the program would have been: this program displays a triangle of asterisks, where the number of rows is handed over as a parameter – in this particular example 3. Each n-th row contains n asterisks (first row one, second row two, and so on). This concrete example therefore produces the following output:

*
**
***

The summary given by the student was:

two for loops for raw (sic!) and col

This is not a correct answer, as it neither describes the program as a whole nor gives the output produced by the program.

In the data it seems that the participant analyzed the program following the program flow rather than a dominantly linear approach (Figure 3).

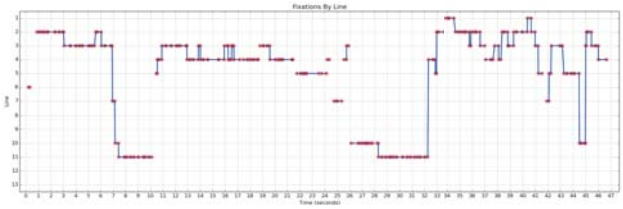


Figure 3. Method of analysis by EU10 after sixth lesson.

This becomes evident within the outer loop, where a parameter value is needed that was handed over to the method from the method call. The gaze then jumps down to the method call, without any larger stops on lines in between, which might indicate that the participant knows what she's looking for and where to find it. She then continues reading the code in her own logic, first comparing the variable initializations in the two loops, then looking at the loops' exit conditions. Afterwards very quick jumps between different parts of the program occur, which with good will might be interpreted as following the order in which the loops are exited. However, those movements are really fast and it is thus questionable, if the gaze is truly following the flow or just randomly jumping on the screen. After 26 seconds (fixation 102) the reading of the nested loops is finished, now the gazes moves to the main method and its body. Until here, the movements were quite comprehensible. What comes then is similar to what happened in the prior example: seemingly random movements across the code lines can be observed that allow for no further interpretation.

Summarizing, the behavior of the student in this final phase of the java course shows very clear evidence of program comprehension, even though she does not come up with the right answer. She analyzes the code in reading it by following its logical structure. I would therefore describe her approach as *logical analysis*.

3.2 Interpretation of the Results

Even though the analysis was conducted on a rather superficial level, it can be seen that some development took place over time. In the beginning, the participant read the code line by line and repeated the whole procedure from the beginning. This indicated that the structure of java programs was very new to her, she did not yet know which parts were really relevant. In the second example the student was still close to this linear analysis for the overall understanding of the program, but then started to follow the program flow in several shorter examples when trying to understand the details. In the last example, the linear analysis took no longer place. Instead, the code was read as if the program was executed. With the progress also came a lack of thoroughness. In the first task the student went through the whole procedure twice and used half of her time for checking the result. In the second task she seemed to check her findings quicker, using about a third of the time for validating her answer. In the last task such a behavior was no longer to be recognized. However, the seemingly random eye movements could not be interpreted and might be very rapid reassurance of the calculated answers.

3.3 Relation to the Coding Scheme

When looking at the existing coding scheme from the experts' data, I recognize several patterns that I also found in the data.

3.3.1 Scan

What I referred to as linear analysis is called Scan according to the coding scheme. This occurred more visibly in the novice stage and faded with growing expertise.

3.3.2 LinearHorizontal

LinearHorizontal reading was observed at all stages, usually at the beginning of the analysis. For the novice programmer this was observable throughout the whole time.

3.3.3 LinearVertical

LinearVertical reading could be observed at the earlier two stages, for the second task only at the beginning of the analysis, probably to get an overall impression of the program's functionality. Again, for the novice programmer (first task) this was observable throughout the whole time.

3.3.4 Flicking

Flicking occurred frequently at every stage in different forms (RetraceDeclaration, RetraceVariable), mostly when it came to understanding the details of a program.

3.3.5 JumpControl

JumpControl is what I referred to as following the program flow. It occurs more often for the more experienced programmer (here: task 3).

3.3.6 Thrashing

Thrashing did not appear in the first example, but was visible with the later two, both times after – in my impression – the program was understood. This is possibly a method of validation.

3.3.7 JustPassingThrough

JustPassingThrough lines did appear at all stages, but only rarely for the first task and quite often with growing expertise. I see a relation to Flicking here: the more flicking, the more JustPassingThrough.

4. DISCUSSION

As this analysis gave only very limited view into how novices learn to read code, none of what was found here should be generalized. However, the results in general meet my expectations. The participant has clearly shown that in the end of the course she was able to read program code in a way that follows the execution order – something that she did not do in the first task. For future research it is now important to analyze larger data sets and see if those patterns are found elsewhere, too. The approach towards the analysis of data (as described in section 3) was helpful and can be recommended to anyone not familiar with the existing coding scheme. If the scheme is known and well understood, it might be better to make use of it and possibly add additional codes. Concerning analysis tools, the material given was very helpful. Often I had the desire to see longer traces (approx. the last 15 gazes) and to slow down the video. A good analysis tool should be able to highlight those chunks of code that are related to each other based on the programmer's gaze and based on the program logic (control flow, data flow). What I feel would be very interesting is the analysis of eye-tracking data of students who are given faulty code and whose task is to fix the problem.

5. REFERENCES

[1] Bednarik, R.; Busjahn, T.; Schulte, C. (Eds.) 2014. *Eye Movements in Programming Education: Analyzing the Expert's Gaze*. Technical report. University of Eastern Finland, Joensuu, Finland.

How Novices Understand a Program?

Kshitij Sharma, Patrick Jermann, Pierre Dillenbourg
Computer Human Interaction for Learning and Instruction
École Polytechnique Fédérale Lausanne, Switzerland
{kshitij.sharma,patrick.jermann,pierre.dillenbourg}@epfl.ch

To analyze the gaze patterns of a novice, while (s)he is trying to understand a given program, we propose following measures:

1. Gaze transitions among different parts of the program. We consider the "sub-line" level areas of interest.
2. Gaze distribution over different semantic elements of the program ([1]). We define three categories of semantic elements: structural (the keywords and punctuation in the program), identifiers (names of methods and variables) and expressions (the statements modifying the identifiers).
3. Gaze transitions among different semantic elements of the program ([2]).
4. Defining entropy and stability episodes and then looking at the transition among different episodes ([3]). The entropy capture the number of elements looked at in a given time window and the stability captures the similarity of gaze over two consecutive time windows.

In the dataset, there were two novice programmers (we call them "eu10" and "do21"). All the answers given by do21 were very accurate while the answers given by eu10 were either very shallow or incorrect. Hence, we decide to compare the different gaze measures for these two participants. We observe the following:

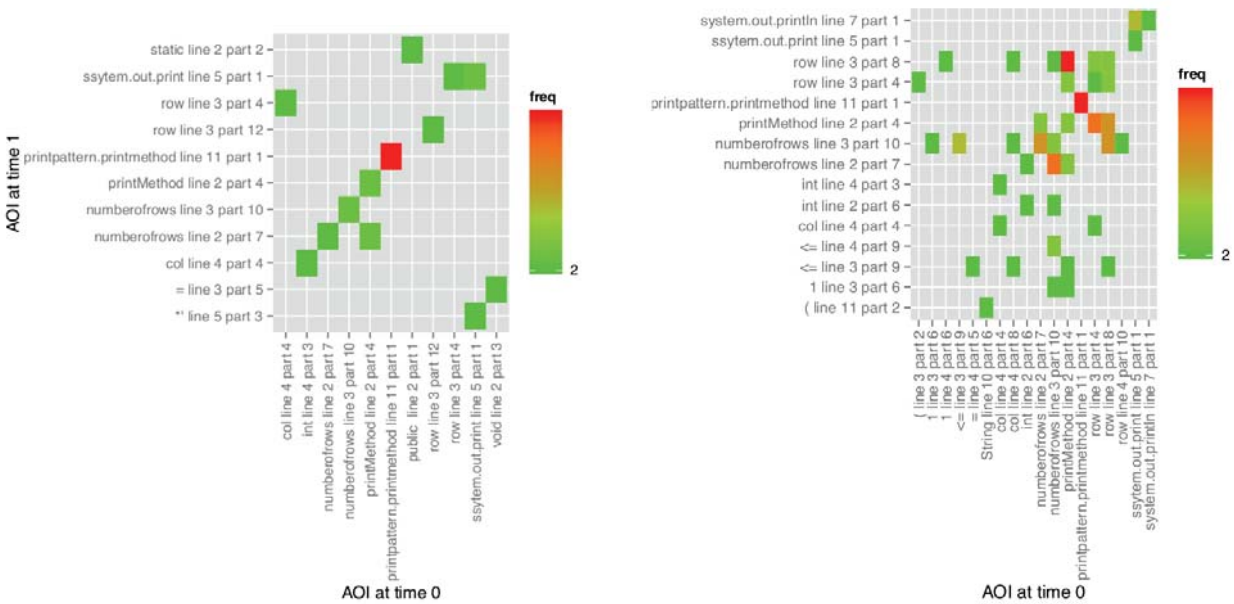
1. The only common program was "lesson 6" for them and we show the transition among different parts of the program (Figures 1(a) and 1(b)). We see that eu10 is focusing more on the "printPattern" method. Participant do21 has more transitions among the loop structure and the main method that enables do21 to provide concrete answer.
2. We compare the gaze distribution over structural, identifiers and the expressions for both the participants (Figure 2(a) and 2(b)). We observe that do21 has looks more on the expressions than eu10, which makes the comprehension easier for do21 as expression contain most data flow of the program.
3. We compare the gaze transitions among structural, identifiers and the expressions for both the participants (Figure 3(a) and 3(b)). We observe that both the participants have the most transitions among the expressions. However, eu10 also puts efforts to move among identifiers as well. This indicated that the participant is trying to guess the functionality by the names of the methods and variables, which hinders the complete and correct comprehension of the program.
4. We compare the gaze transitions among different entropy-stability episodes for both the participants (Figure 4(a) and 4(b)). We observe that do21 has most transitions among "dispersed-unstable" episodes. This reflects the fact that do21 spends most of the time in connecting different parts of the program because the gaze is often covering a bigger chunk of the program in a given time window and most of the time this chunk is different from the other. This behavior helps do21 understand the program in a better manner. On the other hand, eu10 has most transitions among "focused-unstable" episodes. This reflects the fact that in a given time window eu10 looks at a very small chunk of the program and the two consecutive chunks are different from each other. In terms of behavior this translates into an effort of reading the program line-by-line which hinders the understanding process ([2]).

References

[1] K. Sharma, P. Jermann, and P. Dillenbourg. Dual eye-tracking. In *Handbook of learning analytics and knowledge*. 2014, submitted.

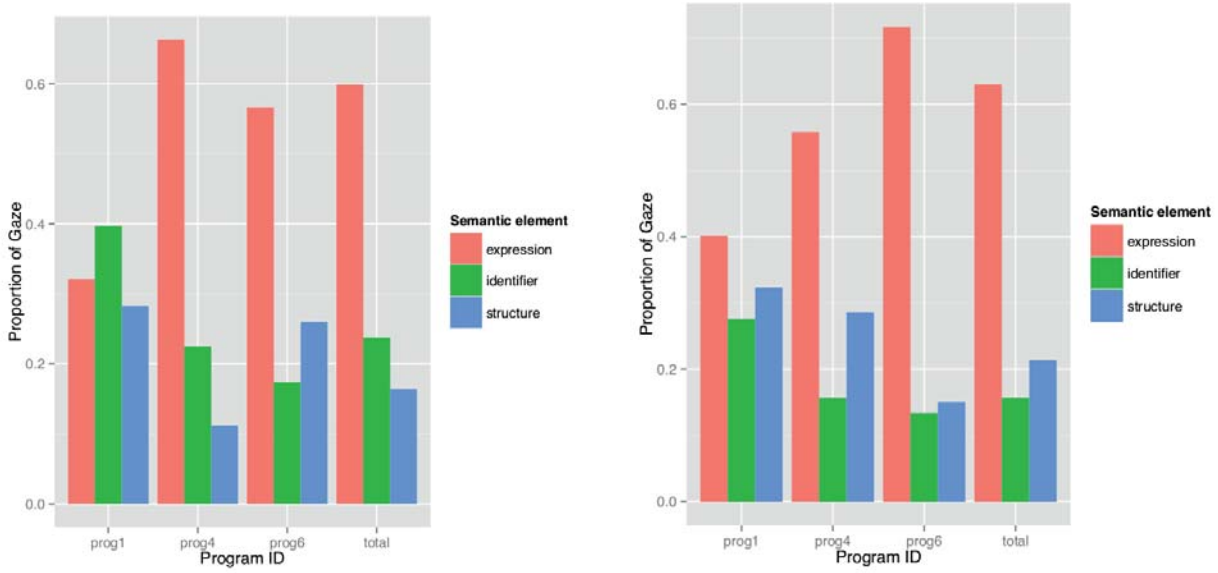
[2] K. Sharma, P. Jermann, M.-A. Nüssli, and P. Dillenbourg. Gaze evidence for different activities in program understanding. *24th Annual conference of Psychology of Programming Interest Group*, 2012.

[3] K. Sharma, P. Jermann, M.-A. Nüssli, and P. Dillenbourg. Understanding collaborative program comprehension: Interlacing gaze and dialogues. *Computer Supported Collaborative Learning (CSCL 2013)*, 2013.



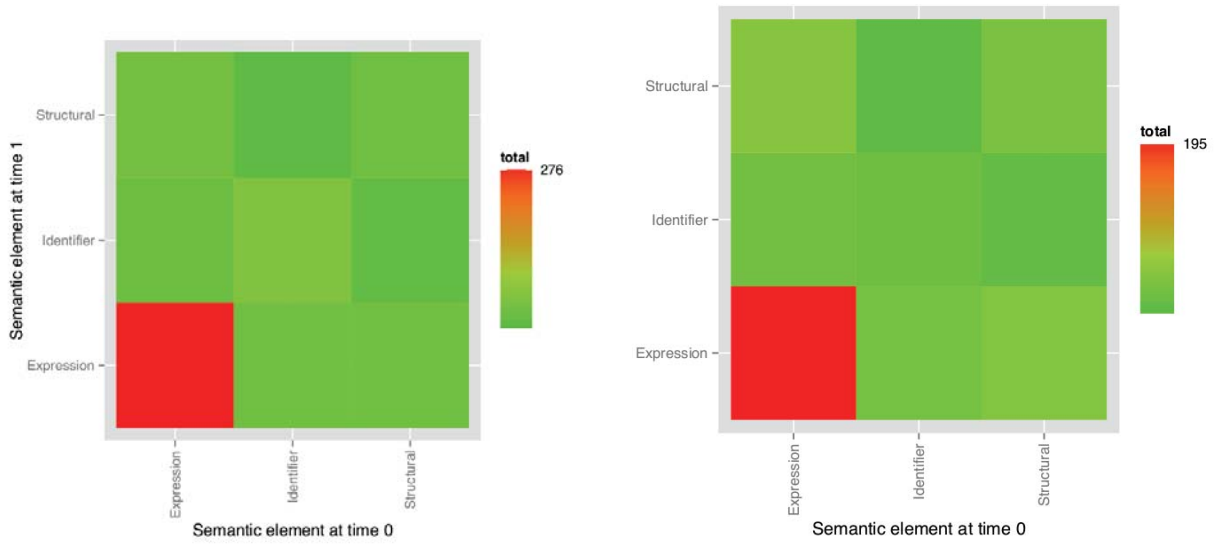
(a) eu10 (b) do21

Figure 1: Transitions



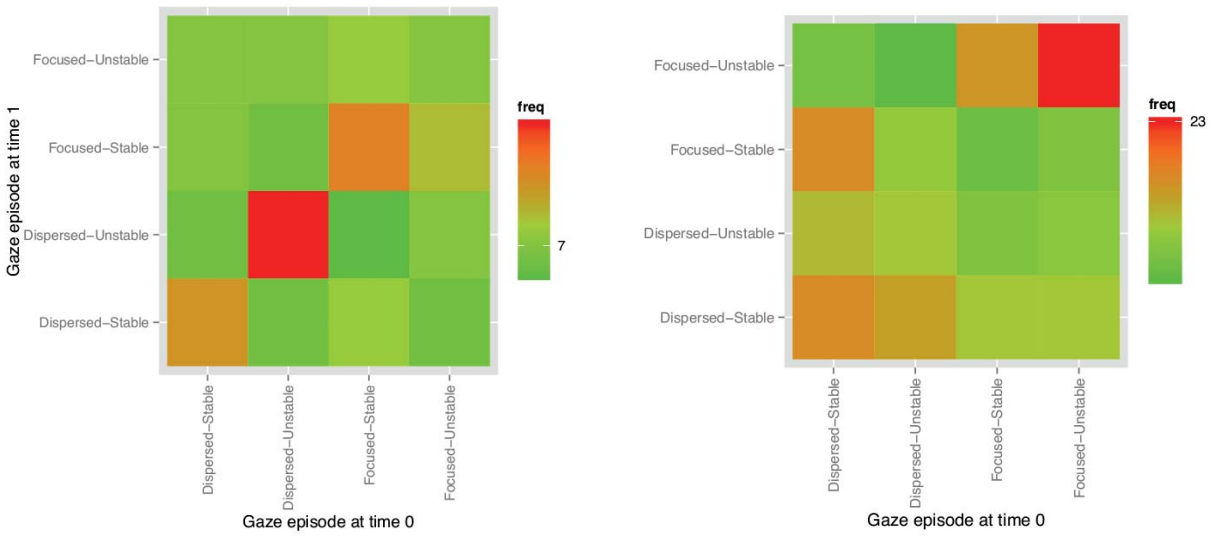
(a) eu10 (b) do21

Figure 2: Gaze distribution over semantic elements



(a) eu10 (b) do21

Figure 3: Transitions among semantic elements



(a) do21 (b) eu10

Figure 4: Transitions among different episodes

Eye movements in programming education 2: analysing the novice's gaze

Simon
University of Newcastle, Australia
simon@newcastle.edu.au

ABSTRACT
The gaze of a novice programmer was recorded as she examined small Java programs at various stages of a short programming course. The recordings taken at the start, in the middle, and at the end of the course were analysed, in the hope of finding a progression in code-reading competence as the course proceeded. At the start of the course the novice read the code just as one might read a story in a natural language, starting at the beginning and reading through to the end. She then repeated this process. In both subsequent recordings her gaze was more directed, moving between what might be seen as the salient features of the code. On both of these latter occasions she apparently failed to grasp what the code was doing. Nevertheless, there is clear evidence of the novice's progression from linear reading to a form of directed reading that might be more helpful to a programmer.

Categories and Subject Descriptors
K3.2 [**Computers and education**]: Computer and Information Science Education – *computer science education*

Keywords
Gaze analysis, computing education, programing education, eye tracking

1. INTRODUCTION
A novice programmer taking part in a short introductory programming course had her gaze recorded each week as she read small segments of program code in preparation for answering questions about the code. The course used Java as the vehicle of instruction, and the programmer had just a little prior experience writing Python. In an attempt to discern some form of progression in her code-reading skills during the course, we have examined the gaze recordings from the beginning, the middle, and the end of the course.

The programmer's answers to the questions about the code were substantially correct in the first recording and incorrect or incomplete in the next two recordings. Nevertheless, there is clear evidence of a changed way of reading as the course proceeds, a way that shows more understanding of the structure of the code.

2. READING PATTERNS AND STRATEGIES
In previous research [1] we identified a number of tiers in which a gaze recording could be classified. In this analysis we will focus just on the two higher-level tiers, pattern and strategy.

Pattern attempts to describe the gaze sequence by associating it with similar sequences that have been previously identified. The sequences identified to date are *JumpControl*, in which gaze follows the order of code execution; *Linear*, in which the gaze follows at least three lines sequentially, regardless of order of execution; *LineScan*, in which gaze concentrates on a single line in its entirety; *Scan*, in which gaze reads a sequence of lines briefly, then returns to concentrate on points of interest; *Flicking*, in which the gaze moves back and forth between two related items, such as the formal and actual parameter lists of a method call; and *Thrashing*, in which the gaze moves rapidly and wildly in a sequence that appears to make no particular sense. One further identified pattern is *Signatures*, in which gaze covers a number of method signatures before moving to the bodies of the methods. However, this pattern refers explicitly to the content of the code, while the others refer to the code only in more general terms; consideration should therefore be given to removing this one from the list, unless it is joined by others that also refer to code content. There is almost certainly scope for further patterns.

Strategy is the crux of the analysis. It is in this tier that the analyst tries to determine what the reader was thinking while reading the code. *DesignAtOnce*, typically associated with Linear and Scan patterns, suggests reading through part or all of the code in a linear manner, intending to acquire an overall understanding of it. *Debugging* is similar, but with gaze time more evenly distributed over the elements, and suggests a search for syntactic or semantic errors. *ProgramFlow* follows the expected sequence of program control, with the apparent intention of simulating program execution. *TestHypothesis* involves repetition of a pattern of gaze, and suggests further concentration in order to better understand a particular detail. *Trial&Error*, somewhat like *DesignAtOnce* but with faster reading, irregular jumps, and repetition, suggests a search for some part of the code that will lead to an initial understanding. *FlowCycle* involves following the same program flow sequence several times, perhaps to gain a first understanding of the flow, then to strengthen and reinforce it with repeated examinations of the same code.

3. ONE NOVICE'S CHANGE IN READING PATTERNS

3.1 First recording: start of course
In the first recording, made in the first week of the course, the novice read through the code twice in a purely linear manner, applying the LineScan pattern within the Linear pattern. This would seem to indicate the DesignAtOnce strategy. However, we must remember that strategies are not simply patterns: they are the analyst's attempt to determine what the programmer was thinking. In this context, with a novice reader at the beginning of a course, I believe that we see a strategy that might be called *StoryReading*. This would be applied by readers who have as yet learnt very little about program structure, and who therefore read code just as they would read a story in a natural language. It is abundantly clear that in this recording, this particular novice

programmer 'reads the story' of the code, then reads it a second time, perhaps in the hope of better understanding it. In the second reading there is a possible brief regression from the *println* command to the preceding *print* command; however, the programmer's answer to the question suggests that either she failed to register that the two were different, or she did not fully understand the difference between the effects of the two commands.

3.2 Second recording: middle of course

The second recording was made in the middle of the course, just after students had learnt about data types. The code, shown below, involves string processing that would probably appear quite involved to a novice who just encountered it, using the *indexOf*, *length*, *substring*, and *replace* functions.

```
public class textClass {
  public static void main (String [] args) {
    String text = "Hello World!";
    int positionW = text.indexOf("W");
    int textLength = text.length();
    String word = text.substring(positionW,
                    textLength – 1);
    System.out.print(text.replace(word, "Sun"));
  }
}
```

The programmer begins by reading the code in a somewhat Linear/LineScan manner, but already we see some variations from the earlier reading pattern associated with the StoryReading strategy. One is that the LineScan reading of each line involves more regression: reading the line is no longer a matter of starting at the left and proceeding to the right, but involves a number of jumps back to previous parts of the line.

The second change is particularly interesting, and appears to involve highly directed regressions to particular parts of the code. Specifically, when a variable is encountered, gaze returns to previous instances of the variable, as if the reader is checking the variable's history, to form a picture of what it now represents. This would suggest a new strategy of *Confirmation*: the reader has encountered a familiar variable, and is checking back to confirm what it represents and perhaps what its current value is.

Both of these changes appear to provide evidence for the reader's development from a simple story reader to a novice code reader.

The reader then works through the code more seriously, with many more regressions. At one point she appears to be calculating the values of *positionW* and *textLength*, as gaze moves deliberately along the literal string, first to the W, and then beyond it. This is interesting because a more experienced code reader is likely to appreciate that the code can be understood without knowing these values of these variables, but rather by understanding what roles they play.

The reader now proceeds to an even more detailed reading of the last line of the code, particularly of the call to *replace*, with many confirmation regressions to the initialisation of *text* and of *word*, the latter involving further confirmations of *position*, *textLength*, and *text*.

The reader then appears to have grasped the replace function call, but spends more time on the function's arguments, *word* and "Sun", again with confirmations to the declaration of *text*. This is followed by another long read of the computation of *word*, initially linear but then with multiple confirmations to *position* and *textLength*. Finally there is a more or less linear read of the last line, but again with brief confirmation glances.

After all this, the programmer's answer to the question suggests that she understood the notion of *replace*, but not the role or value of *word*, whose computation was arguably the most involved aspect of the code. However, it is possible that she actually meant to write something different. With just one more letter she might have written "Replace World with the string "Sun"", which would be far closer to a correct summary. Notwithstanding the correctness of her answer, her reading pattern appears to indicate that she knew exactly what to read and how to read it in order to try to understand this particular variable.

3.3 Third recording: end of course

The third recording was made at the end of the course, shortly after students had learnt about iteration, and the code involves a nested pair of loops with a simple *print* statement inside the inner loop and a *println* inside the outer loop. Also of interest, the principal operations are carried out inside a separate void method that is called from the *main* method.

The programmer again begins in a linear manner, which brings her to the void *printMethod* immediately after the class header. More or less as soon as she starts reading that, she appears to appreciate that *numberOfRows* is a parameter, not a local variable, and jumps forward to the method call within the *main* method. This would appear to be another form of the Confirmation strategy, one that seeks confirmation of something not yet encountered, and definitely shows further development as a code reader, in the form of an understanding of the code structure.

Gaze is fairly haphazard in the remainder of this reading, principally in the reading of the nested *for* loops, which is essentially a form of Flicking between the two. It would be easy to conclude that the reader has some sort of understanding of *for* loops, but no real grasp of the nesting of two loops. This is borne out by her answer to the question.

4. THE PROGRAMMER'S PROGRESS

Notwithstanding that this particular novice programmer answered the second and third questions incorrectly or incompletely, there would appear to be clear evidence of her development, particularly between the recordings made at the start of the course and in the middle of the course. In that time her reading appears to have progressed from StoryReading to a more code-focused strategy, characterised by multiple regressions and Confirmations to relevant pasts of the code.

5. REFLECTIONS

It is generally accepted that good readers are characterised by few regressions and short fixations. After examining these three recordings from the same novice programmer at three points of a programming course, I believe that this is a generalisation from natural-language reading, which does not apply particularly well to code reading.

I believe that regressions, as classified in these recordings as part of the Confirmation strategy, are an essential aspect of code reading. Unlike a story in natural language, a piece of program code is not a linear construct, and cannot be usefully read in a linear manner.

It might be true that code readers will have less need for the Confirmation strategy as their expertise increases. Nevertheless, it seems reasonable to conclude that the carefully directed use of regressions and forward jumps will increase as program code reading matures, as a necessary consequence of the non-linear structure of program code. The recordings of this novice help to support that conclusion.

Further data might help to confirm the observed progression from StoryReading to patterns more associated with code reading and understanding. Can we confirm that many or most readers appear to be StoryReading at the beginning of their courses? Does this apply to longer pieces of code? (The piece used in the first recording is so short that StoryReading might be an indication of the simplicity of the code rather than the early developmental stage of the reader.) In what week did this beginner advance from story reading to code reading? Do other readers advance in about the same week? If not, what is the range of weeks in which the change can be observed?

Having noted that different experts read code in different ways [1], we must expect different novices to read code in different ways. Nevertheless, for any novice who does start with story reading, the advance to code reading might be considered as a substantial step on the way to becoming an expert code reader.

6. REFERENCES

[1] T. Busjahn, C. Schulte, B. Sharif, Simon, A. Begel, M. Hansen, R. Bednarik, P. Orlov, P. Ihantola, G. Shchekotova, and M. Antropova (2014). Eye tracking in computing education. *Tenth International Computing Education Research Conference (ICER 2014)*, Glasgow, Scotland, 3-10.

Analyzing the Novice's Gaze in Program Comprehension

Jozef Tvarozek
Faculty of Informatics and Information Technologies
Slovak University of Technology in Bratislava
jozef.tvarozek@stuba.sk

ABSTRACT

In this position paper, I give my opinion on the eye movement records of two participants from a study on program comprehension. First I analyze each task for each participant individually, and then I try to analyze differences between the participants. Finally, I reflect on what I was expecting to find in the data and hope to give comments on improving the study.

Keywords

eye tracking, source code reading, program comprehension, computer science education

1. INTRODUCTION

The data examined in this paper consists of eye movement recordings for two participants, each having examined three program comprehension tasks:

1. after Lesson 1 (Introduction)
2. after Lesson 4 (Fundamental Data Types, after lessons on objects and classes), and
3. after Lesson 6 (Loops, after lesson on decisions)

Participants were asked to summarize the program seen after each task. The gaze data provided were collected using SMI RED-m eye tracker at 120 Hz with Ogama software.

2. PARTICIPANT 1

Participant 1 reported having 2 years of prior programming experience. In the first recording, pseudo-code about cakes having a price, and printing whether the price is odd or even was presented. The code can be found on the workshop website. The participant did read the code in linear fashion with regressions when details about the actual cake prices were relevant. The participant's program summary was correct and in sufficient detail, having demonstrated full understanding of the code. The pseudo-code was well-written, easy to read and verbose, almost plain English, and it did not pose any problems for the participant having prior programming experience.

The second task, presented a program for calculating average. In the first half of the recording, the participant read the program in linear fashion, with short regressions. In the second half, appearing to double check the mental representation. Again, the program was an easy read, almost plain English. Also, the class was named CalculateAverage and considering that the participant was informed before the study that the program is all correct and bug-free, there is not much left to extract during the eye tracking session. That is, when the program appears to be calculating average (as the name of the class, and variable's name average suggests), and it is said that the program is bug-free, then it *should* calculate average. Calculating average is a trivial mental exercise for a university student with two years of prior programming experience. The eye movement recording supports this hypothesis. The gaze data shows that the student did not even inspect the actual calculation of average (num1+num2)/2 in any detail. The program provided is linear with no tricky mental processing required; hence the participant's eye movement cannot exhibit any of such processing. The participant's summary was, same as previously, correct and in sufficient detail.

The third task, presented a program for printing a kind of simple ASCII art – an asterisk pyramid. The core of the program is a method containing two loops with an asterisk being printed within the inner loop, and a newline in the outer loop. The most difficult part of the comprehension task is to understand the loop ranges and how it can produce a pattern, and what pattern. At first, the participant appears to be reading the program linearly with small regressions. Afterwards, the participant begins to inspect the ranges in the loops more carefully, presumably iterating the cycles to arrive at the final pattern. As before, participant's program summary was correct in sufficient detail, demonstrating full comprehension.

Overall, the participant studied programs that were comparatively easy considering her previous programming experience. Basic English comprehension was sufficient for understanding the first two programs. The third program required a very basic understanding of loops.

3. PARTICIPANT 2

Participant 2 reported little prior programming experience. The first recording studied comprehension of a simple

Analyzing the novice's gaze workshop Freie Universitat Berlin, Germany, November 7th - November 8th, 2014

program consisting of two print statements. The participant read in two linear passes with very few regressions. Participant's program summary did provide the program's output that was almost correct, not accounting for newlines correctly, presumably due to misinterpreting when System.out.println method writes the new line character to output. Also, the program's summary provided by the participant was terse, suggesting the participant did not understand the question or was not motivated enough.

The second task involved a program replacing a word (World) within a string with another string (Sun). At first, the participant familiarized with the code in one linear pass with only small regressions, after that the participant struggled to build the mental model of the program (its variables). The participant's summary suggests only little understanding of the program's meaning, since the summary can be derived syntactically from the last line of source code. Deeper understanding would enable the student to capture the correct output, provided the student understood the question.

The third task involved the same program as was the case for Participant 2; that is two for loops printing a simple ASCII art – an asterisk pyramid. In the first pass, the participant read the code linearly and jumped forward to familiarize with the function call and possible input parameters, after the first pass the participant appears to double check class and method names, and to memorize the individual expressions, it does not appear that any analysis of control flow is performed by the participant. Missing mental model does not allow providing a summary in sufficient algorithmic detail. Only a simple surface-level summary was provided.

Overall the participant appeared to read mostly the surface form of the provided programs with no program structure-induced gaze jumps, building no mental model that would allow to answer the post-task question in more detail.

4. COMPARING PARTICIPANTS

The first task was easy for both participants. The second task was easy for Participant 1, while Participant 2 got comparatively more involved task that required building a mental model of variables and their run-time values. The third task was the same and provides a better vehicle for comparison.

The first participant did engage in detailed analysis of the loops while at the same time read the code mostly linearly. The second participant did exhibit more structure in reading – i.e. did not read only linearly but jumped – but ultimately failed to understand the program as demonstrated by the inability to provide correct program summary. One explanation is that the prior programming experience of Participant 1 enabled her to read the program linearly while building the mental model. On the other hand, the second participant had to jump during reading because she was not able to build a mental model effectively in a linear pass through the code.

5. WHAT DID I EXPECT TO FIND, AND WHAT CAN WE IMPROVE?

Before I examined the data, I expected students to gain better understanding of programming language structural elements (such as for loops and functions) over time. Programming is about building mental models about the program: its variables, control flow, memory state during execution, etc. I expected the observed reading pattern converges to a pattern in which student reads the code jumping between related elements: i.e. scans related items (e.g. if-else-break, for-break-continue, class-methods-arguments-returnValue) in succession. The data suggests that participant with longer prior programming experience (and also having a better programming skill) did in fact jump less between the related elements that the participant with no prior programming experience (and with less programming skill). The participant with higher skill might be able to extract the mental model from the code more effectively, and thus needs less jumping.

Overall the selection of tasks, did not favor this view on programming. Basic English comprehension was mostly sufficient to understand most of the tasks studied, while the final difficult one was answered sufficiently by only the student with previous programming experience. I expect more *programming* learning progress can be observed when participants are required to build mental models of the program beyond basic English comprehension. Modern coding standards recommend self-explanatory class names and variable names, but well named entities can give away too much to observe any significant code reading during eye tracking study. Considering tasks that require debugging and/or control flow analysis would presumably enable observing better code reading patterns.

6. ACKNOWLEDGMENTS

This article was created with the support of the Ministry of Education, Science, Research and Sport of the Slovak Republic within the Research and Development Operational Programme for the project "University Science Park of STU Bratislava", ITMS 26240220084, co-funded by the European Regional Development Fund, and Scientific Grant Agency of the Slovak Republic, grant No. VG 1/0752/14. I would like to thank the workshop organizers for providing the gaze data and all their efforts in organizing the event.

Eye Movements in Programming Education II

Analyzing the Novice's Gaze

International Workshop at the 9th WiPSCE Conference on Computing Education
Freie Universität Berlin, Germany, November 7th - November 8th, 2014

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

The second international workshop on Eye Movements in Programming Education focuses on the development of novice programmers.

Reading occurs in debugging, maintenance and the learning of programming languages. It provides the essential basis for comprehension. By analyzing behavioral data such as gaze during code reading processes, we explore this essential part of programming.

To participate send a mail to teresa.busjahn@fu-berlin.de. Note that it is possible to participate independent of attending WiPSCE.
Prior to the workshop, participants will get three data sets showing a novice's gaze during code reading tasks at the beginning, middle and end of an introductory Java course.
Participants will delve into the data in order to identify and reflect on stages of novice development. A short individual position paper of the results is required (max. 2-3 pages). A technical workshop report including the position papers will be published at FU Berlin.
During the workshop session in Berlin, we will work on how novice code reading skills develop and how this progress can be linked to observable gaze data. Afterwards, participants jointly prepare a publication describing the results.

IMPORTANT DATES

Making data and tools available for participants: end of August 2014

Deadline for position papers: Friday, October 17th, 2014

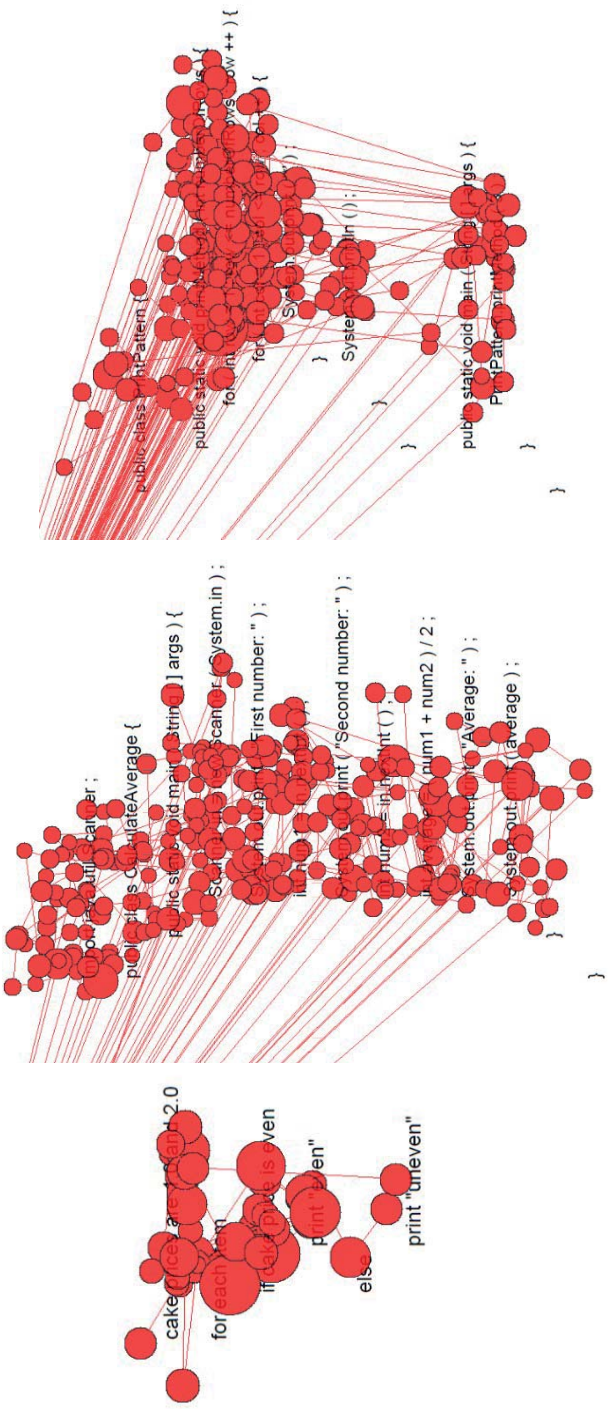
Pre-workshop get-together: Friday, November 7th, 2014 (evening)

Workshop: Saturday, November 8th, 2014

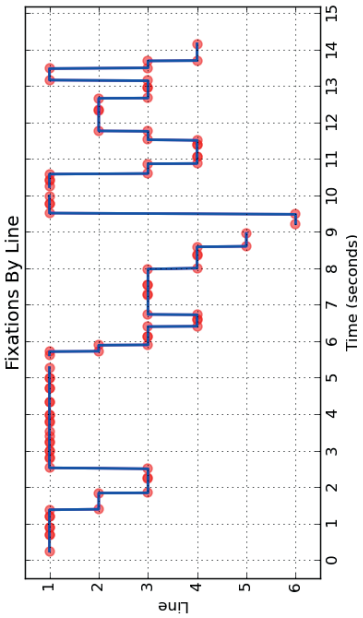
Illustrations of gaze data

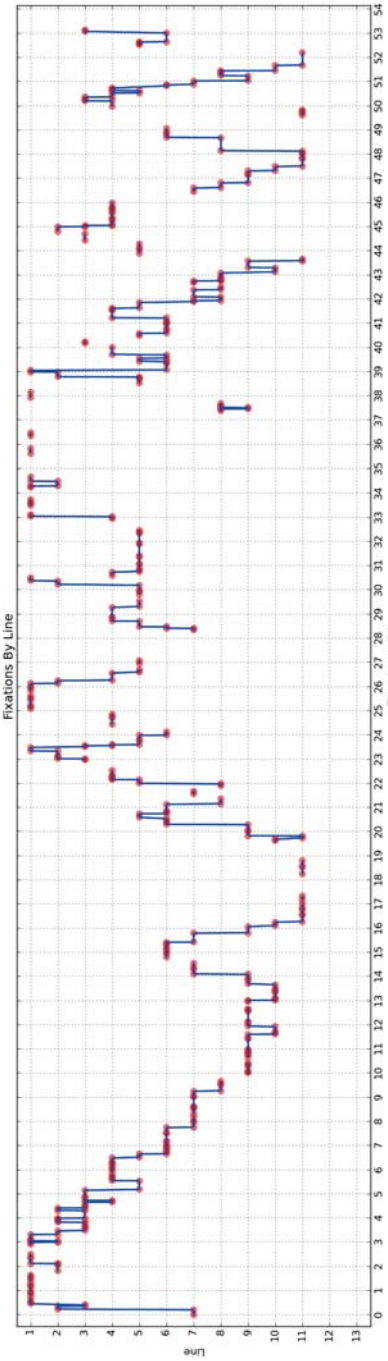
The eye movement data of the two participants DO21 and EU10 was recorded using Ogama (www.ogama.net) and an SMI RED-m Tracker (120 Hz).
Visualizations were done with eyeCode (https://github.com/synesthesiam/eyecode).

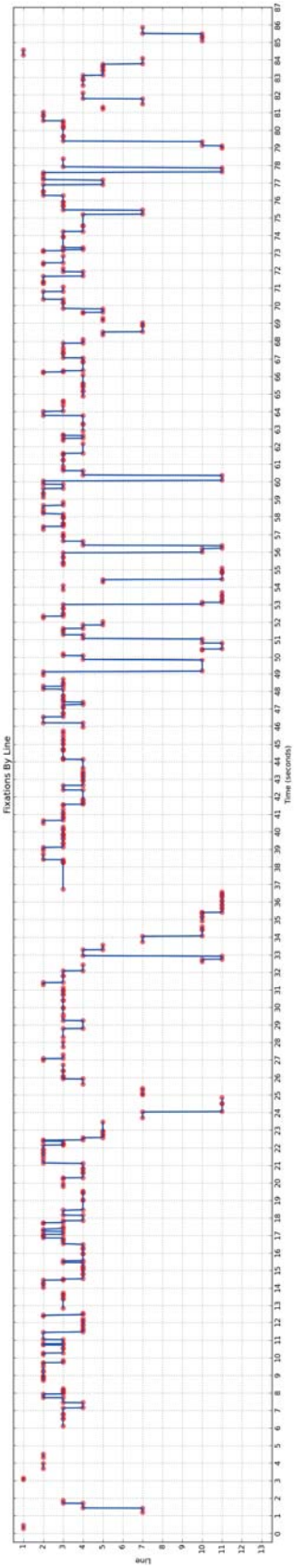
DO21



Scanpaths DO21

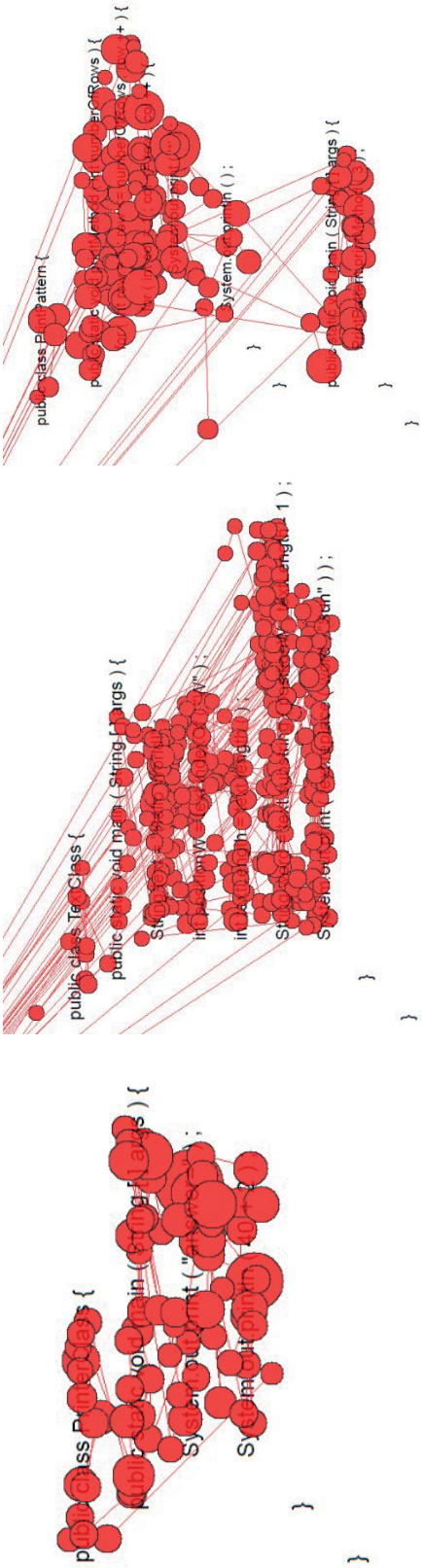




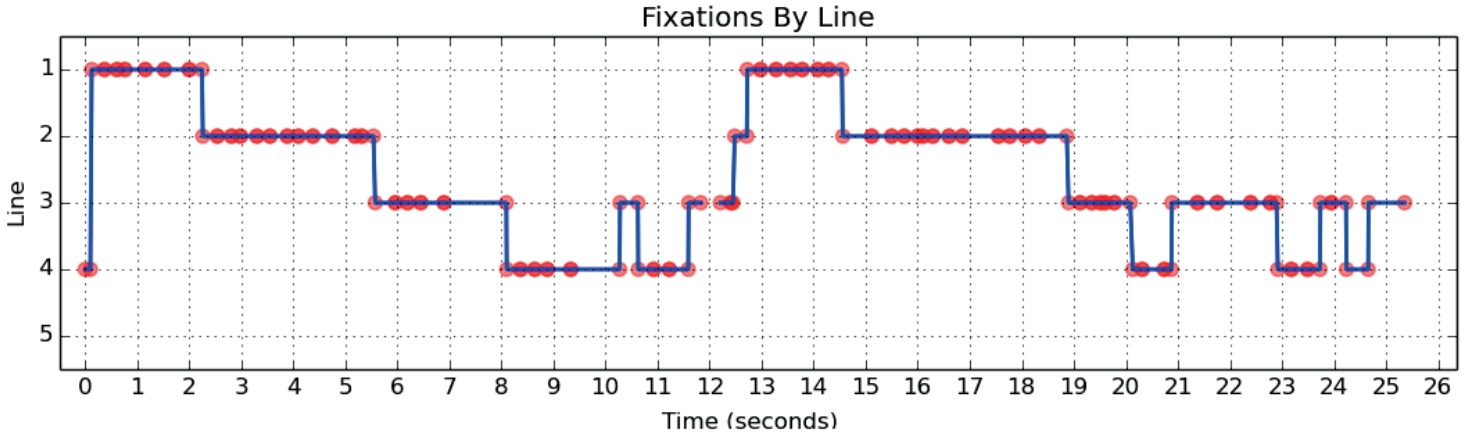


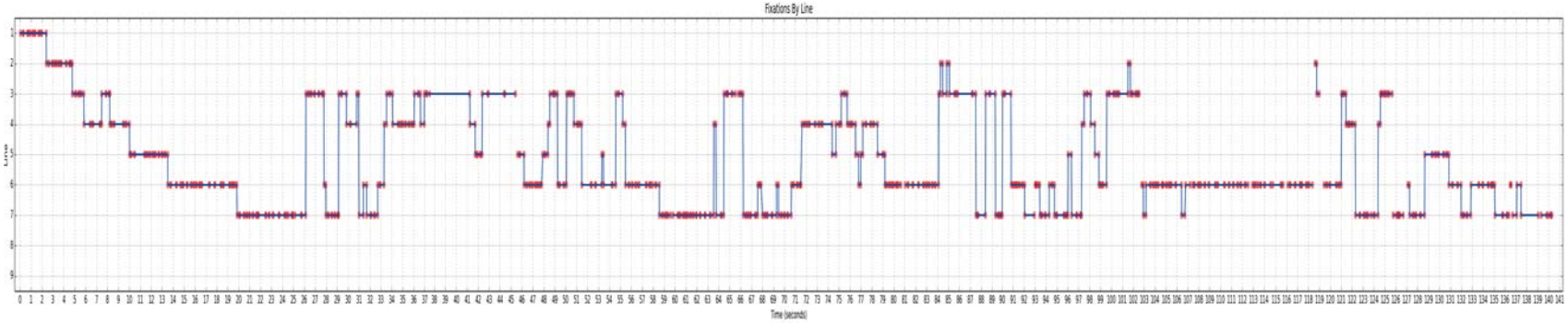
Timelines DO21

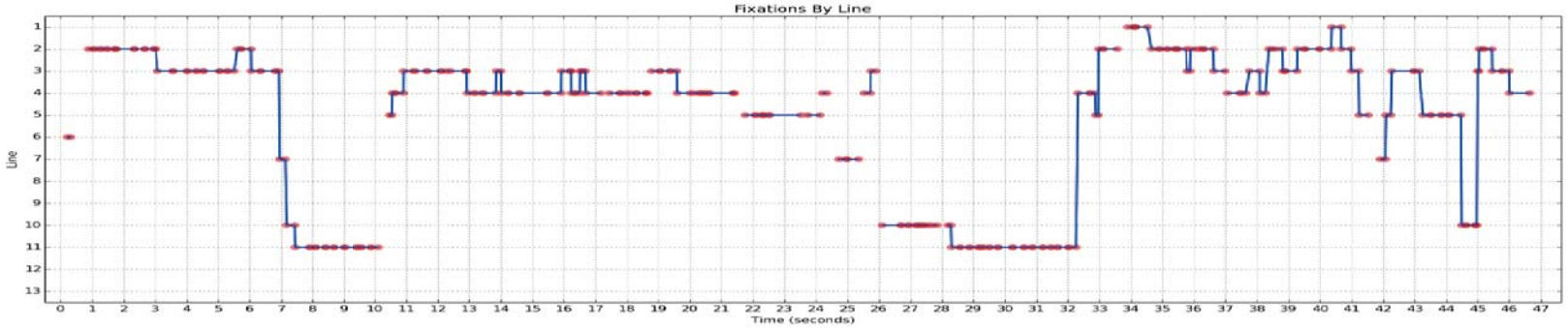
EU10



Scanpaths EU10


Fixations By Line
Line
Time (seconds)


Fixations By Line


Fixations By Line
Line
Time (seconds)

List of participants

Name	Institution
Bednarik, Roman	University of Eastern Finland, Finland
Begel, Andrew	Microsoft Research, USA
Buchholz, Sven	FH Brandenburg, Germany
Busjahn, Teresa	Freie Universität Berlin, Germany
Crosby, Martha	University of Hawai'i at Mānoa, USA
Heteren-Frese, Christoph van	Freie Universität Berlin, Germany
Lohmeier, Sebastian	Technische Universität Berlin, Germany
Löhnertz, Martin	Universität Trier, Germany
Orlov, Pavel	St. Petersburg State Polytechnic University, Russia
Paterson, James H.	Glasgow Caledonian University, UK
Pyykkönen-Klauck, Pirita	SensoMotoric Instruments, Germany
Przybylla, Mareen	Universität Potsdam, Germany
Rönnecke, Stefan	SensoMotoric Instruments, Germany
Schulte, Carsten	Freie Universität Berlin, Germany
Sharif, Bonita	Youngstown State University, USA
Sharma, Kshitij	École Polytechnique Fédérale de Lausanne, Switzerland
Simon	University of Newcastle, Australia
Sudol-DeLyser, Leigh Ann	CSNYC, USA
Tamm, Sascha	Freie Universität Berlin, Germany
Tvarozek, Jozef	Slovak University of Technology in Bratislava, Slovakia
Villasco, Clelia	SensoMotoric Instruments, Germany