REPORTS AND STUDIES IN
FORESTRY AND
NATURAL SCIENCES

ROMAN BEDNARIK, TERESA BUSJAHN,
CARSTEN SCHULTE (EDS.)

# *Eye Movements in Programming Education:*

## *Analyzing the Expert's Gaze*

PUBLICATIONS OF THE UNIVERSITY OF EASTERN FINLAND
*Reports and Studies in Forestry and Natural Sciences*

UNIVERSITY OF
EASTERN FINLAND

# *Eye Movements in Programming Education: Analyzing the Expert's Gaze*

**ROMAN BEDNARIK, TERESA BUSJAHN, CARSTEN SCHULTE (EDS.)**

# *Eye Movements in Programming Education: Analyzing the Expert's Gaze*

*Proceedings of the First International Workshop*

Publications of the University of Eastern Finland
Reports and Studies in Forestry and Natural Sciences
No 18

University of Eastern Finland
Faculty of Science and Forestry
School of Computing
Joensuu, Finland
2014

Grano Oy
Joensuu, 2014
Editor Prof. Pertti Pasanen, Prof. Pekka Kilpeläinen,
Prof. Kai Peiponen, Prof. Matti Vornanen
Distribution:
Eastern Finland University Library / Sales of publications
P.O.Box107, FI-80101 Joensuu, Finland
tel. +358-50-3058396
http://www.uef.fi/kirjasto
ISSN (nid): 1798-5684
ISBN (nid): 978-952-61-1538-2
ISSN-L: 1798-5684
ISSN (PDF): 1798-5692
ISBN (PDF): 978-952-61-1539-9

Bednarik, Roman; Busjahn, Teresa; Schulte, Carsten (Eds.)
Eye Movements in Programming Education: Analyzing the Expert's Gaze.
Itä-Suomen yliopisto, School of Computing, 2014
Publications of the University of Eastern Finland. Reports and Studies in Forestry and Natural Sciences, no 18
ISSN (nid.): 1798-5684
ISSN (PDF): 1798-5692
ISSN-L: 1798-5684
ISBN (nid): 978-952-61-1538-2
ISBN (PDF): 978-952-61-1539-9

# Eye Movements in Programming Education:

## Analyzing the Expert's Gaze

Proceedings of the First International Workshop

at the 13th KOLI CALLING INTERNATIONAL CONFERENCE ON COMPUTING EDUCATION RESEARCH, 2013

**School of Computing, UEF, Joensuu, Finland**

**November 13th - November 14th, 2013**

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

Code reading is an essential part of program comprehension and a common activity in debugging, maintenance and learning a programming language. Nevertheless, Computer Science Education Research and Teaching mostly focus on code writing. Better insights in code reading are valuable to support programmers from novice to expert. The first international workshop "*Eye Movements in Programming Education: Analyzing the Expert's Gaze*" is an approach to gain deeper understanding of the comprehension processes behind observable eye movements during code reading.

The workshop was organized in association with the 13th KOLI CALLING Conference in Computing Education and took place November 13th - November 14th, 2013 at the School of Computing, UEF, Joensuu, Finland. A total of 15 people participated in the workshop, four of them remotely. The event was supported by the Joensuu University Foundation.

Before the workshop, participants were given two sets of eye movement records of expert programmers reading Java. The data can be downloaded from www.mi.fu-berlin.de/en/inf/groups/ag-ddi/Gaze_Workshop/koli_ws_material. We asked the participants to analyze and code these records with a provided scheme. Based on this analysis position papers have been written describing the eye movement data and commenting on the coding scheme, as well as on the application of eye movement research in computer science education. The coding scheme concerned code areas in different level of detail, observable eye movement patterns and presumed comprehension strategies. The scheme was revised following suggestions given in the position papers and during the workshop. Additionally, several group members developed visualization tools both for eye movements and the results of the coding process and provided them in their position papers.

This technical report contains the position papers. Furthermore it includes the workshop call, the eye movement materials used, the revised coding scheme, and a list of participants.

We would like to thank all participants for the great work,

Roman Bednarik, Teresa Busjahn and Carsten Schulte

# *Contents*

# Eye Movements in Programming Education. Analyzing the Expert's Gaze

Maria Antropova
Research Team Lead at JetBrains
Russia, Saint Petersburg
Universitetskaya nab.7-9-11, k.5, lit.A
+7-921-311-4431
maria.antropova@gmail.com

Galina Shchekotova
Analyst at JetBrains
Russia, Saint Petersburg
Universitetskaya nab.7-9-11, k.5, lit.A
+7-921-763-7648
gshchekotova@gmail.com

## ABSTRACT

There are two main strategies of subject behavior during eye-movement experimental research. The first pattern is based on the inductive approach and the second one is based on the deductive.

## Keywords

Eye-movement analysis, code-reading patterns.

## 1. SUBJECT DESCRIPTION

### 1.1 The first subject

The first subject spent much more time on learning the program, he was unsparing in his efforts for the methods analysis and matching parameters in methods and constructor.

In the top of patterns behavior for this subject there are many pairs like:

'Main'→ 'Height'

'Width' → 'Constructor'

'Constructor' → ' Main '

'Constructor' → 'Height'

'Width' → 'Constructor' → 'Width'

'Height' → 'Constructor' → 'Height'

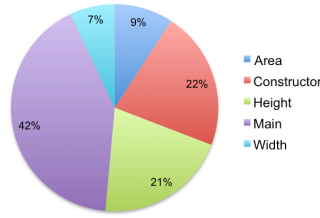
'Constructor' → 'Width' → 'Height' → 'Constructor'

This means that the subject was trying to actually understand how the method works and which parameters were used and how they were used in each method. For the 'Height' and 'Width' block he spent 25% of the time. This behavior explains his task description (to answer on certain question about program output).

It seems like the subject was following combined strategy: Scan strategy is more valuable than JumpControl and LineScan.

The subject was learning the correspondence between input parameters and variables in constructor. That is why in the top of patterns we see a lot of pairs like 'Constructor' → 'Main' and 'Main' → 'Constructor'. Time which the first subject spent for Constructor block is less than the second subject, because the first one looked at the Constructor all the time very briefly, only for understanding of the parameters order.

**Time distribution. Subject 1**




### 1.2 The second subject

The second subject spent less time than the first subject for the task completion (40% less than the first one). The reason is that second subject had a different task and had to answer multiple-choice questions. Also he used other technique, which is more convenient and fast for the short program and this technique is based on scan strategy. In the top of patterns behavior for this subject are:

'Constructor' → 'Width'

'Main' → 'Constructor'

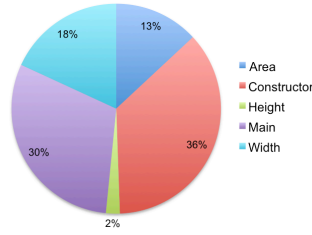
'Area' → 'Main'

'Area' → 'Main' → 'Constructor'

'Constructor' → 'Area' → 'Main'

The second subject probably uses Linear or LineScan (or combined) strategy because of the task description.

**Time distribution. Subject 2**



## 1.3 Comparison table

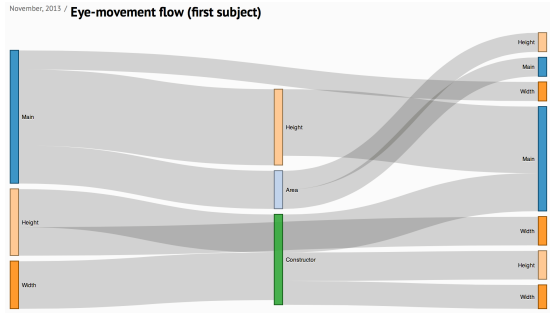
Take a look at the table comparison of the two subjects by several metrics related to the code scheme and the subject's behavior: Attributes, Main, Constructor, Height and Width blocks, Actual Parameter List in Main, Return, Pattern, Duration.

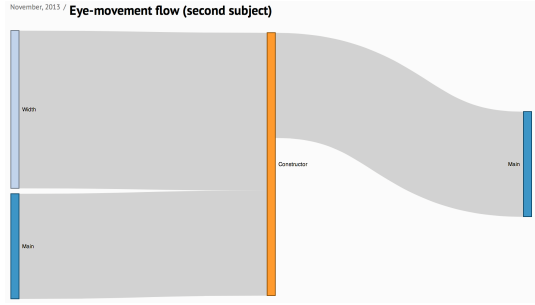
**Table 1. Comparison table for considered subjects**

| Code | First Subject | Second Subject |
|---|---|---|
| Main | Often but with short duration | Rarely but with long duration |
| Constructor | Often and with long duration in the first part of the session. It needs for the area calculation | Rarely and not very intensively, only for understanding how the object is created. The second subject spent on 10% more time for Constructor block than the first one. |
| Height and Width blocks | The first subject spent on these blocks 25% of the time. | The second subject spent on these blocks 13% of the time; it is 12% less than the first one. |
| Actual Parameter List in Main | Learning the parameter list close to the end of the session for the area calculation, then moving to Constructor and object methods for calculation | Looked at the parameter list in the end of the session to keep in mind two created objects (rectangles). |
| Return | Looked at Return block very often to calculate the value of the area | Looked at the Return block only for understanding how does method work |
| Pattern | Combined: more valuable is Scan strategy, than JumpControl and less LineScan. | The main strategy is LineScan - the best one for the general understanding of the short code. |
| Duration | The first one spent more time than the second one (the reason is strategy or experience difference). | |
| | | |

## 1.4 Eye movement flow

This is an example of eye movement flow, which presents big difference between subjects behavior. First subject moves between different code blocks a lot, second one has different behavior and has not too many jumps.




## 2. THE CODING SCHEME

In the presented coding scheme there is no Area block.

The presented coding scheme is too detailed for the research of small code length. For example, researching current two subjects we didn't need such tier of the scheme as Strategy (Debugging, etc).

Based on Block analysis we can conclude that there are two main strategies of code reading: from the special to general and the other way around.

The first strategy is more effective in case of big program with many modules of a code (or in case of specific task), the second strategy is more effective in case of short code (in this case more experienced user can guess what is going on in the program). This is a very important difference that should be taken into consideration in the experiment design. Other parameters should be looked in other experiments with code of different length and with subjects with different strategies.

## 3. MAIN QUESTIONS

**What yields the tagging of "primitive" events? What ideas/thoughts/associations did arise?**

It helps to identify general patterns more clearly. Also it gives some clue about actual cognitive task if we don't know about it. Also, probably it helps to follow the order of task stages.

**We know about global understanding strategies from program comprehension research (data flow, control flow, top-down, bottom-up, as-needed...). Do we find those in the gaze?**

In case of having only gaze data it is possible to find it only if we have a very simple program on one screen, because it becomes to be impossible to recognize the strategy if we don't know anything about scrolling.

Otherwise it is possible but still difficult because we have to mark on the gaze file points with coordinates changing (if the program is more than one screen).

**What patterns did you find and what are suitable names for them?**

We have found two main patterns: for the first subject we can call it inductive approach (from the special to general), for the second subject it is more deductive way (from the general to special). It very depends on the task, which the subject has to solve (and probably depends on subject experience, type of program paradigm, etc).

**How are patterns connected to cognitive strategies? Which patterns are indicative of which strategies?**

Patterns are practical realization of cognitive strategies in process of task making. Different cognitive strategies probably have different patterns. For the inductive approach combined strategy is more typical (mix of Scan strategy, JumpControl and LineScan). For the deductive strategy LineScan is more typical.

**Are there further strategies? And what would be suitable names for them?**

As we mentioned above, the strategy depends on the task. There are many types of tasks: debugging, code review, refactoring, etc.

## 3.1 Application of Eye Movement Research in Computer Science Education

The best way of implication is conscious use of code reading strategies depending on code size, program structure and other parameters.

The explication of code reading strategies and deliberate usage of these strategies depending of situations helps students to make their strategies more effective.

With high probability, there are also differences in code reading process due to the approach to programming (OOP, Functional, Procedural), that should be also considered in the education process.

# 4. ACKNOWLEDGMENTS

Our thanks to Eye-movement workshop organizators for allowing us to participate in this event.

# Analyzing Programming Tasks

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

## 1. INTRODUCTION

In this position paper, I first describe the eyetracking patterns of the two participant videos I watched and coded. Next, I reflect on the methods and validity of manual coding and interpretation, and finally, I add my own thoughts on the utility of eyetracking data for understanding and helping programmers create and maintain software.

### 1.1 Task Segment 1

Participants were asked to understand what the area() method would do. The first participant spent 22 seconds linearly reading the code from top to bottom. He then went backwards and read through the class methods and constructor for 6 seconds. Then he explored a constructor call from the main(), and spotted similarly named instance variables throughout the rest of the code. Then it seemed that he switched to tracing the code in each of the instance methods, flipping back and forth from the constructor call to the instance method in order to figure out which values were being used in the computations. Finally, he traced through the execution of the rect2.area() method call, jumping from the this.width() call to the definition of width() and from the this.height() call to the definition of height().

Diving a bit deeper from 00:33 – 00:36, the subject explored the meaning of the width() method. Triggered by the call to width() in the area() method, he read the width() method body from start to finish, then traced the definition of this.x1 to the constructor call where this.x1 was assigned. He then traced that back to the parameter list which contained an x1 parameter. Then he jumped back down to the Rectangle constructor call in main() to see which value was passed in as the first argument to the Rectangle constructor call.

While this could be characterized as a strategy of execution tracing in reverse (a.k.a. debugging), I think the user was really executing a pattern by tracing similar words backwards through the code file. So, he saw x1 in width(), saw it again in the Rectangle body, and then again in the parameter list. Afterwards, he used the notion of parameter-argument positions to find the appropriate value passed into the Rectangle constructor from the main() method.

If we wanted to identify when the subject was tracing in a debugging strategy vs. pattern matching words, we could modify the study instrument and change the Rectangle constructor parameter names to be different than the instance variable names. Similarly, we could also create a second constructor in which the positions of the parameters (and grouping) are permuted from the first, to see if actual knowledge of method calls was being used to spot the correspondence between the caller and the callee arguments.

### 1.2 Task Segment 2

The second participant read linearly through the class and constructor until he reached the width() method. Then he traced the definition of each used instance variable to the constructor. He did the same when reading the height() method, but switched back to linearly reading the code when he reached the area() method. This took about 25 seconds. Then he started tracing the first constructor call to see how each argument was assigned to a particular parameter and assigned into a similarly named instance variable. Finally, answering the question, he looked at the rect2.area() method call, read the definition, and then presuming he understood the code correctly, computed the math in his head to figure out the rectangle's area, and finished the task.

The second participant worked more quickly than the first to minimize code scanning and concentrate more directly on the rect2.area() method call. Between 00:40 — 00:55, he traced the Rectangle constructor call that created rect2. He first connected the first argument to the first parameter, then to the first instance variable assignment. Then he read the next line of the constructor call and worked backwards to the parameter and to the argument to the constructor call to validate some internal hypothesis about which argument values were assigned into each instance variable.

## 2. REFLECTIONS

The process of coding eyetracking data can be divided into two parts: segmentation/identification and interpretation. For programming tasks, the first part is automatable, provided the subjects' IDEs can be queried to turn (x, y) pixel positions offered by the eyetracking device into program constructs at various levels of textual, lexical, syntactic, and semantic abstraction. The second part is subjective, requiring the observer to interpret the rationale behind the user's eye movements. This is easiest to do when the user thinks aloud (and the narration is recorded in sync with the user data). But, interpretation can easily be biased by the observer's prior knowledge of programming and pedagogy. We can mitigate this by having many independent observers interpret the same data, allowing unsupported inferences to be detected, negotiated, and eliminated [1]. Tailoring the

research questions requiring interpretation towards purely observable phenomena can also help.

After segmenting and coding the data using the observable measures, I have several thought about the process:

1. ELAN has some awkward user interface constructs that make it difficult to process multiple related tiers of codes. One specific example is that some clearly hierarchical code tiers should have corresponding start and end time stamps in each tier, but the system does not automatically align the annotation boundaries for you.
2. I do not trust my annotation timestamps to be accurate within one second. I would trust an automated labeler much more. The implication here is that I would not feel comfortable trusting many quantitative analyses based on annotation times or lengths. I would trust an analysis based solely on the order of annotations within a single tier.
3. Without think aloud, I can only offer speculations on the programming strategies employed by the participants. Even on a smaller time scale, there are so many things that could be going through the participants' heads while they code that influence where their eyes are pointed. Before I would believe anyone else's speculations, I would conduct an experiment to confirm the theories found in the Empirical Studies of Programmers workshop series through some carefully designed code comprehension experiments [3, 2, 4].

With respect to what I found the participants to be doing, it was possible to see what I thought were eye movements (saccades) influenced by various semantic and operational properties of the code (all timestamps for first video): data flow (following a single object in memory as its value changes through the program, e.g. 00:46–00:50), intraprocedural control flow (scanning lines of code in program execution order (real or simulated), interprocedural control flow (following call-chains in real or simulated execution, e.g. 00:52–00:59), word (pattern) matching (simple visual pattern matching, e.g. 00:26, 00:27.8–00:28.2), linear scanning at the block level and the line level (reading through the textual lines of code, e.g. 00:02–00:20, 00:26.3–00:27.6), and reverse data flow data (tracing assignments backwards through control flow in service of debugging and/or program execution comprehension, e.g. 00:23–00:26).

I do not think the two examples we saw were intricate enough to help us understand much about program comprehension strategies, and certainly nothing about programming or debugging strategies. I offered suggestions in the previous sections describing the two segments as to how to alter these examples to validate any theoretical concepts related to difficulty, confusion, or fatigue.

One major complaint about the 1980s Empirical Studies of Programming work is that most of the program comprehension theories were derived from experiments on students reading tiny programs away from a computer where they could code or run them. Thus, the lowest level strategies used by experts in real work may appear similar, but there will be evidence of higher and higher-level strategies and plans in *in situ* empirical data (should we have some) that will confound the simpler theories.

## 3. THE BIGGER PICTURE

Software developers continue to make mistakes when writing code, despite improvements in programming languages, high-level abstractions, better development tools, better communication tools, more responsive development methodologies, and even the availability of Internet search. Mining software repositories (MSR) research correlates empirical data about the software and the process by which it was developed to discover attributes that indicate poor code quality and/or poor productivity. However, this research does not explain why mistakes are made, but only where they occur most often.

Developers do take steps to mitigate the risk pointed out by MSR analyses. For example, they might more rigorously test code that has been implicated in prior bugs. However, I feel that to improve the basic situation, we need to go to the root of the problem, when developers are actively reading, writing, and modifying code. In a pilot study I conducted last year with my colleague, Thomas Fritz, from the University of Zurich, we recorded 6 Microsoft software engineers working for five minutes to modify some code we gave them and for five minutes on their own task they had that day. We found each developer expressed (via think aloud protocol) temporary confusion, and got lost (re: navigation) in their code several times in that short time span, even when working on their own code with which they were very familiar. Perhap, developers make more mistakes when they are confused or lost (and do not make mistakes when they are not). Thus, if we could detect and/or stop them from programming in these emotional states, we could improve code quality and productivity.

In the last year, I have been using eyetracking, electrodermal activity sensors, and EEG sensors with professional programmers doing comprehension tasks (very similar to the ones in this workshop) to identify correlations between the biometric sensor readings and programmer confusion, task difficulty, and surprise. My goals are to discover which sensors correspond most precisely to these emotional attributes, which combination of sensors are easiest to deploy and offer the best online prediction accuracy, and correlate the sensor readings to areas of the code where developers cause bugs or experience lowered productivity.

Ultimately, I would like to use instantaneous measurement of biometric data and design an appropriate analysis to enable the design of IDE-based programmer *interventions* that could stop developers from making bugs before they make it into the source code. For instance, EDA readings can help determine when someone is not paying attention to their work (e.g. they just had lunch) and warn them if they try to edit a region of the code known to be at high risk for bugs.

During my work, I have had to learn a lot about experimental design of small comprehension tasks, biometric sensor measurements, analysis of noisy human-sourced data, and

still find ways to discover significant results with non-trivial effect sizes. I hope to find others at this workshop to trade tips and tricks for this experimental data, and develop a set of practical methods for design and implementing experiments and analyses. I would also like to find out how best to adapt experimental methods and analyses from the medical and cognitive psychological fields for tasks that involve many fewer, yet much more complex (related to more areas of the brain) activities that are representative of computer science tasks and skills.

## 4. REFERENCES


[1] B. Kitchenham, D. I. K. Sjøberg, O. P. Brereton, D. Budgen, T. Dybå, M. Höst, D. Pfahl, and P. Runeson. Can we evaluate the quality of software engineering experiments? In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '10, pages 2:1–2:8, New York, NY, USA, 2010. ACM.

[2] G. M. Olson, S. Sheppard, and E. Soloway, editors. *Empirical studies of programmers: second workshop*. Ablex Publishing Corp., Norwood, NJ, USA, 1987.

[3] E. Soloway, B. Shneiderman, and S. Iyengar, editors. *Empirical Studies of Programmers: First Workshop*. Greenwood Publishing Group Inc., Westport, CT, USA, 1986.

[4] S. Wiedenbeck and J. Scholtz, editors. *ESP '97: Papers presented at the seventh workshop on Empirical studies of programmers*, New York, NY, USA, 1997. ACM. 608977.

# Analysis of two eyetracking renders of source code reading

Katerina Gavrilo
Saint-Petersburg, Russia
katrinaalex@gmail.com

## ABSTRACT
In this paper, the specific and subjective description of two short segments of data is given. Author proposes some thoughts on the usage of the eye movement data in computer science education research.

## Keywords
Eye movement, source code review, eye-tracking metrics, cognitive strategies, program comprehension, pattern

## 1. INTRODUCTION
Connection between comprehension processes and eye movement data has been analyzed for many years now. Although this field of computer science is rather young, we should not underestimate the results we have already got. A lot of different researches have been made in this field. Some studies have more physical specification [1], another ones — cognitive [2]. Recently a number of programming orientation works has been written [3,4].

Having initial data (two subjects per program) and various annotations to program, a number of particular observations is given as a result of assignment.

## 2. GENERAL INTERPRETATION
According to our goal, which is to find existing connection between eye movement data and cognitive processes during programming, we analyzed data we have. For a review at our disposal we have one Java program and two subjects. To each of them two different comprehension questions about the same program were given. As a result we recorded two video fragments with different subjects.

## 3. DATA, PATTERNS, STRATEGIES
As raw material we have two videos from where we get information about subjects' eyes behavior, several characteristics (fixations, location and duration of those fixations) and saccades amplitudes.

For making the analysis we apply a number of given rules which describe some of the eye-movements. These rules represented as digest of attributable patterns for eye gazes and also strategies, which are based on existing patterns.

In our research we address to some patterns as scan pattern, liner, retrace declaration pattern [4], retrace reference pattern [4]. As for strategies we will try to derive them from patterns we found.

Even though the source code for two participants is the same, patterns and strategies are noticeably different. That occurs because comprehension task, which were given to the subjects before the source code was shown, are not the same. Let us discuss the gaze data step by step.

### 3.1 Figure 1
The task for this figure was to say the return value of 'rect2.area()' after the program was executed. The whole fragment takes 1,5 minutes.

First 19 seconds we consider as a scanning process through the whole source code. During this part time participant becomes acquainted with it. At the end of the program probationer facing finds the line with where he find 'rect2.area()', the — data he has to know the value of. So then he is going back to place where variables for x's and y's were declared. From that place he descends reads the code again. And again the moment participant he reaches the 'area()' method description participant he looks step by step "go" through the previous places where the variables has have been recently referred. For example he faced the 'area()' method and sees there 'width()' and 'height()' methods, so he goes to them returns to them. After he gets what is in there the values of these variables, next gaze he stops his sight is at the constructor, where x and y values are defined. This route is repeated a number of times with some insignificant deviations.

At some point we have three blocks of code between which gazes are travelling. These are area with given parameters (5, 5, 10, 10), which have to be counted for having output value, area between where width and height methods are described and constructor with parameter definitions. After a sequence of brief fixations longer fixations appear. That is caused by some cognitive processes. Presumably there the participant counts the result, because he is looking at the entering parameters (5, 5, 10, 10).

### 3.2 Interpretation of Figure 1
We found scan pattern in the beginning of the video. For the most part of the whole test there were a lot of oft-recurring saccadic eyesight jumps between places where variables had been recently referred to or declared. Those patterns we call *retrace declaration[4]* pattern and *retrace reference pattern[4]*. As for strategies, we would define here DesignAtOnce and Trial&Error.

We would propose to describe the mix of patterns and strategies as a process when you first check the risk of any deal. Like before transport some goods through the unknown route, first one go there without merchandise and see where to turn and where the traffic lights are. And when one is sure about everything he takes the goods with him to finish the deal. Comparing this example with our task, the route here is the algorithm while the goods are the parameters. This strategy is called *touchstone*.

### 3.3 Figure 2

The task for this figure was to find a way to give an answer to a multiple-choice question about the algorithmic idea. The whole fragment takes 56 seconds.

First 25 seconds we can define as detailed and thoughtful scanning. During scanning the participant equally pays attention to signatures, lists of parameters, body of functions. We can notice that when the similar description of a method or a variable appears, fixation time is much shorter. For example after the participant has examined the width method long enough he did not spend much time examining the height method, because they are similar. When probationer reaches output commands he spends there quite a long period of time (the sum of fixation intervals is bigger compared to other blocks), thinking and analyzing the type of information he will have as an output. Also we can see a moment when the participant was comparing the 'rect1.area()' to 'rect2.area()'. Next "block" of his action is juxtaposition of entered parameters (5, 5, 10, 10) to how they are described in the constructor. Afterwards eyesight is coming back to 'public static void main' with predominant attention on line 20. The fixation duration is getting noticeably longer. Then participant has his eyes directed to the 'area()' method. This part seems like he is attentively investigating what the method is doing. After all sights are going back to the 'public static void main' zone and last 6 second we observe that the fixations are longer and they gathered just near the end of the code.

### 3.4 Interpretation of Figure 2

It is uncertain if the scan pattern is appropriate in this case, because usually it means that the participant is briefly looking over the source code and then coming back to parts, which he thinks deserve more attention. It could be perceived as slow *motion scan pattern*. Beside that we can distinguish the *linear pattern*. So we come to strategies DesignAtOnce and ProgrammFlow, when the subject's intention is to understand the general idea of algorithm and figure out the outcome of the program.

### 3.5 Comparison between figures

Compared to the Figure 1, Figure 2 was more consistent, regular and calm, so to say. In the Figure 1 the whole picture was assembled by the participant from the pieces, which where all around the code in random places. The participant of Figure 2 made his picture very accurate. It seems that the second participant was memorizing information during the reading the code, from the first step. That is logically explained by the task he was given.

## 4. POTENTIAL USE

Each person has their own model of cognitive comprehension and by studying them we can individualize the material we have. That could be used in computer science education to improve quality of studying materials.

There are several areas of application for the eye movement data analysis, if it were researched more thoroughly. For example, that could be used for finding "bugs" in the code. This can be observed and as the results we could have some rules of differences between novices and professionals (which are actually already observed) with which it is possible to check the candidates for some job for example.

Also this method could be used as a great base in education. For instances, as some special aspects for IDE interface design, that could be even auto-tuned with live eye-tracking data. If some parameters are getting too low or too high that means that the person has some problems in this particular block of code, therefore some tooltips, hints or buttons could appear. So that certainly could be used for creating IDE for learning programming. No doubt that this field has a great potential for educational field.

## 5. REFERENCES


[1] Gippenreiter Y.B. (1978) Movement of human eye. Moscow: Moscow University publisher (in Russian).

[2] Velichkovsky B.M. (2006) Cognitive Science: The Foundations of Epistemic Psychology. Moscow: Smysl/ Academia (in two volumes, in Russian).

[3] CROSBY, M. E., AND STELOVSKY, J. 1990. How Do We Read Algorithms? A Case Study. IEEE Computer 23, 1, 24–35.

[4] Uwano, H., Nakamura, M., Monden, A., Matsumoto, K. (2006) Analyzing individual performance of source code review using reviewers' eye movement. In Proceedings of the 2006 Symposium on Eye Tracking Research &Amp; Applications (San Diego, California, March 27 - 29, 2006). ETRA '06. ACM, New York, NY, pp. 133–140.

# Towards Automated Coding of Program Comprehension Gaze Data

Michael Hansen
Indiana University
School of Informatics and Computing
2719 E. 10th Street
Bloomington, IN 47408 USA
mihansen@indiana.edu

Robert L. Goldstone
Indiana University
Dept. of Psychological and Brain Sciences
1101 E. 10th Street
Bloomington, IN 47405 USA
rgoldsto@indiana.edu

Andrew Lumsdaine
Indiana University
School of Informatics and Computing
2719 E. 10th Street
Bloomington, IN 47408 USA
lums@indiana.edu

## ABSTRACT

Gaze data collected during program comprehension provides insight into programmers' thought processes. Manual coding of this data, however, can be tedious and subjective. We define and demonstrate an automated coding scheme for most categories in this workshop's coding scheme. We discuss potential sources of error when abstracting from fixations to areas of interest and patterns, and consider alternative definitions for some codes. For the high-level `Strategy` category, we inform coding decisions with metrics computed over a rolling time window.

## Categories and Subject Descriptors

H.1.2 [**Information Systems**]: User/Machine Systems—*software psychology*

## 1. INTRODUCTION

Gaze data collected during program comprehension provides an insight into programmers' thought processes that is difficult to gain using common performance measures [1]. The process of interpreting and coding this gaze data, however, is tedious and highly subjective. To aid in the discovery of strategies for use in programming education, automated coding can be done with fixation data obtained directly from the eye-tracker. By building on the abstraction gained from lower-level automated coding – e.g., from fixations to blocks, lines, parameter lists, etc. – we demonstrate that codes from most categories in this workshop's coding scheme can be automatically and reasonably assigned.

Automated coding requires precise definitions of each category and code. At a low level, this means defining *areas of interest* (AOIs) based on syntax or semantics, and then deciding to which AOI (if any) each fixation belongs. Section 2 discusses the details of AOI creation and fixation assignment. These details must be explicit because the process of quantizing fixations introduces new potential sources of error. Section 3 defines all automatically-assigned codes in terms of AOI rectangles or lower-level codes. These definitions fit the authors' intuitions, but should not be taken as absolute or final. To aid in the manual assignment of `Strategy` codes, we make use of several fixation metrics computed over rolling time windows in each trial (Section 4).

## 2. QUANTIZING FIXATIONS

Fixations are quantized gaze positions over time. To abstract further, we draw rectangles around areas of interest (AOIs) and assign each fixation to zero or more AOIs. For simplicity, we assume the AOI rectangles in the `Block`, `SubBlock`, `Signature`, and `MethodCall` categories do not overlap. Codes in these categories, therefore, are mutually exclusive (not the case for `Pattern`).

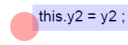



**Figure 1: Example assignment of a fixation to an AOI. A circle is drawn around the fixation point, and the AOI with the largest overlap is assigned.**

To determine whether or not a fixation belongs to an AOI, we do the following: (1) draw a circle around the fixation point with radius $R$, and (2) choose the AOI rectangle with the largest area of overlap (Figure 1). The choice of $R$ depends on the size of the experiment screen and how far away the participant was sitting. Using $R = 20$ pixels, Figure 2 shows a timeline for subject 1's trial where each fixation has been quantized by line. Particular high-level patterns, such as `Scan` (highlighted), become readily apparent with such plots. Caution must be exercised, however, because noise at the lowest levels (raw gaze data) may result in a wrong AOI or code assignment.

## 3. CODING SCHEME DEFINITIONS

To facilitate automation of the coding process, we must precisely define each portion of the coding scheme. Even for very basic codes, such as `Body` from `SubBlock`, different reasonable definitions are possible. For example, should a fixation be coded as `Body` if it hits an opening curly brace (`{`)? For functions defined with K&R style braces, the opening brace is part of the signature line, and would likely not be considered part of the body:

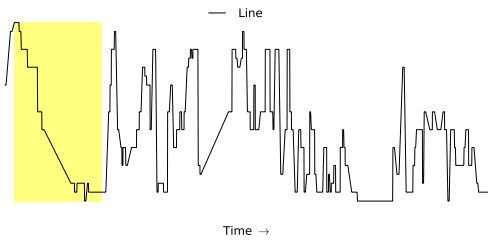



**Figure 2: Timeline of line fixations for subject 1 (entire trial). The automatically identified `Pattern:Scan` portion is highlighted (2.034-18.642s).**

```
public Rectangle(int x1, int y1, int x2, int y2) {
    // constructor body
}
```

With more compactly defined functions, such as `width()`, the separation between body and signature is not as clear:

```
public int width () { return this.x2 - this.x1 ; }
```

We suggest the following definitions for `SubBlock`. The opening brace is counted as part of the signature, whether or not the function is defined on a single line. To be consistent, the closing brace (`}`) is never considered part of the body. Figure 3 shows areas of interest overlaid on the `rectangle` program according to these definitions.

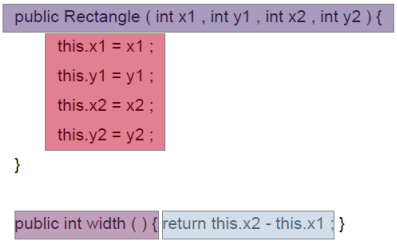



**Figure 3: `SubBlock` areas of interest for constructor and `width` method. Signature and body are consistently separated.**

## 3.1 Signature and MethodCall

Both `Signature` and `MethodCall` have `Name`, `Type`, and parameter list codes. For a signature like `main`'s:

```
public static void main (String[] args) {
    // ...
}
```

we consider `public static void` to be the type, `main` to be the name, and the arguments **plus** surrounding parentheses to be the formal parameter list. When coding method calls, however, we only consider `Name` and `ActualParameterList`. While the type and name of a method call are distinct linguistically (e.g., `System.out` and `println`), they are physically combined as a single "word" (`System.out.println`). Unlike signatures as well, the types and names of method calls are both in the same grammatical category (identifiers), as opposed to being in separate categories (keywords and identifiers). For these reasons, we do not separate type from name for `MethodCall` (Figure 4). Lastly, we do not code nested calls hierarchically (e.g., `foo(bar())`) because it would cause within-category overlap of the AOIs.

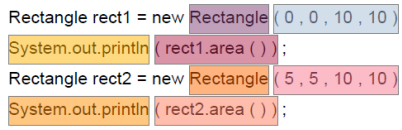



**Figure 4: `MethodCall` areas of interest for `main` method. We do not distinguish between `Name` and `Type`.**

## 3.2 Pattern

The most basic pattern, `Linear` is defined as the subject following at least 3 lines in text order. We follow this definition with one caveat: blank lines are not taken into account. For example, fixations on lines 1, 2, then 4 for the `rectangle` program are coded as `Linear` because line 3 is blank.

The `JumpControl` pattern, while seemingly simple, hides a great deal of complexity. Whether or not a transition between two lines follows execution order depends on where the subject is in evaluating the program! For example, a transition between line 11 (`width()` definition) and line 15 (`area()` definition) follows execution order only if the subject is currently evaluating the call to `this.width()` in the body of `area()`. For now, we code any line transition that *could* follow execution order as `JumpControl`. Future definitions of this code should take previous fixations into account in order to guess where the subject is in the call stack.

`LineScan` is defined in English as the subject reading the whole line in "rather equally distributed time." For simplicity, we operationalize this definition by splitting each line into a set of equally-sized rectangles (Figure 5). A `LineScan` is coded for any set of consecutive fixations that hit at least 3 distinct rectangles on a single line. While this does not explicitly address the "equally distributed time" portion of the English definition, it assigns codes that match the authors' intuitions for the sample data. Another option would be to use the rolling metrics discussed in Section 4 – e.g., fixation spatial density and duration.

Building on `LineScan`, we can simply define `Signatures` as a line scan of a signature line (`SubBlock:Signature`) immediately followed by a fixation inside the corresponding function/constructor body (`SubBlock:Body`). With this definition, we identify two instances of the pattern in subject 2's trial (`width` starting at 7 seconds and the constructor starting around 26 seconds).

The `Scan` pattern, inspired by results from Uwano et al. [4], can be operationalized using two sets of constraints. A `Scan`

starts the first time a fixation moves down the screen relative to the previous fixation, and stops when one of two conditions is met: either (1) more than 3 fixations move up the screen, or (2) more than 1.5 seconds are spent on the same line. The highlighted portion of Figure 2 has been identified using this definition, and matches well with the authors' intuitions.


**Figure 5: A single line split into equally-sized rectangles. We code a `LineScan` if 3 or more distinct rectangles are fixated consecutively.**

## 4. STRATEGIES & ROLLING METRICS

Codes from the categories described above can be assigned based (mostly) on observation. The `Strategy` category of codes, however, requires more interpretation. To aid in the identification and interpretation of strategies, we compute three fixation metrics over the course of each trial using a rolling window. Windows are 4 seconds in size and are shifted by 1 second during each step. On average, a single time window will contain about a dozen fixations.

Our first two metrics are simply fixation count and mean fixation duration [3]. Respectively, they are the total number of fixations in a time window and the mean duration of those fixations. Our third metric, fixation spatial density [2], is computed as follows: (1) divide the screen into a grid, and (2) calculate the proportion of cells in the grid which contain at least one fixation. We divide the portion of the screen containing code vertically into 10 equally-sized rectangles. A spatial density of 1, therefore, means that all 10 rectangles were fixated at least once in a time window.

Figure 6 shows our three rolling metrics computed for subject 1's trial (time windows with no fixations were dropped). Troughs in spatial density (solid blue line) correspond to windows in which subject 1 was concentrating on one or two lines. In some cases, this was correlated with an increase in fixation count (dashed green line), which may be useful for distinguishing between the `Debugging` and `TestHypothesis` strategies. The sharp increase in mean fixation duration just after the 70 second mark (dashed-dotted red line) corresponds with the subject focusing on the final line of the program:

```
System.out.println(rect2.area ());
```

The subject's task in this trial is to obtain the value of `rect2.area()`. Given the increased fixation duration and drop in both fixation count and spatial density at this point (at approximately 65-75 seconds), we hypothesize that the subject is performing the necessary mental calculation to compute the area of `rect2`. There are several off-screen fixations at 70-75 seconds in the video, supporting this hypothesis. While we may not be able to pinpoint shifts in strategy using this kind of visualization, we can quickly identify interesting time windows to investigate further.

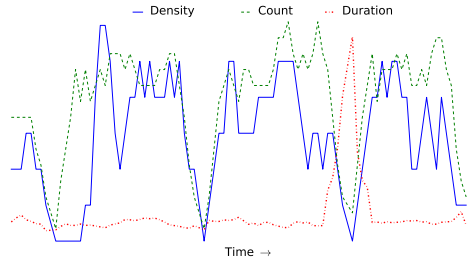



**Figure 6: Rolling fixation metrics for subject 1 (entire trial) with a window size of 4 seconds and a step size of 1 second.**

## 5. CONCLUSION & FUTURE WORK

We have defined and demonstrated an automated process for coding non-`Strategy` categories from the workshop's coding scheme. In most cases, this process assigns codes that match well with the authors' intuitions. In the context of programming education, automated coding helps researchers quantify differences between experienced and novice programmers. Such differences could inform the design of an automated tutor capable of providing highly-contextualized feedback to a student. For example, alternative strategies could be presented to students who fail to locate a bug in an exercise.

Automated coding also forces the coder to think precisely about areas of interest and how to define high-level codes, increasing confidence in subsequent analyses. Because the process is automated, it can be run with different, competing code definitions. Multiple quantitative cognitive models could also be used to inform coding (e.g., `JumpControl`), with deviations from expectations helping to refine the models.

For future work, we would like to achieve automated coding of the `Strategy` category in a way that agrees with human coders. This may not be possible without more precise definitions of `Debugging`, `DesignAtOnce`, etc. Previous psychology of programming research, combined with focused eye-tracking studies where only one strategy is likely to be used, will be crucial to achieving this goal.

## 6. ACKNOWLEDGMENTS

We would like to thank the workshop organizers for their efforts in constructing the coding scheme and providing the gaze data. All software will be made available online after the workshop. Grant R305A1100060 from the Institute of Education Sciences Department of Education and grant 0910218 from the National Science Foundation REESE supported this research.

## 7. REFERENCES


[1] R. Bednarik, N. Myller, E. Sutinen, and M. Tukiainen. Program visualization: Comparing eye-tracking patterns with comprehension summaries and performance. In *Proceedings of the 18th Annual Psychology of Programming Workshop*, pages 66–82, 2006.

[2] L. Cowen, L. J. Ball, and J. Delin. An eye movement analysis of web page usability. In *People and Computers XVI-Memorable Yet Invisible*, pages 317–335. Springer, 2002.

[3] A. Poole and L. J. Ball. Eye tracking in human-computer interaction and usability research: Current status and future. In *Prospects, Chapter in C. Ghaoui (Ed.): Encyclopedia of Human-Computer Interaction. Pennsylvania: Idea Group, Inc*, 2005.

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

# Notes on Eye Tracking in Programming Education

Petri Ihantola
Aalto University
Department of Computer Science and Engineering
Finland
petri.ihantola@aalto.fi

## ABSTRACT

Eye tracking is an interesting approach to trace how programmers read source code. Although it is relatively straightforward to find out where a programmer focus his or her eyes and how focus travels, interpreting this is much more difficult. Why a programmer looks at something and why his eyes move to something else? In this report, I describe my interpretations of two short eye traces where experienced programmers have read a short Java program to find out what it does. I briefly discuss potential pitfalls of interpreting eye tracking data and possible avenues of future research.

## 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

Eye tracking is measurement of eye activity combined with information about the surrounding reality. This includes measuring where a person looks at, how his or her gaze travels as a function of time, and even how the diameters of pupils reacts to different stimuli. Eye tracking data is gathered with eye tracking devices. These can be divided between head mounted (e.g. special glasses) and remote ones (e.g. a monitor with with an accurate camera measuring users eye focus).

In programming education, eye tracking has been used to analyze both novice and expert programmers since early 90's. Since that, as illustrated in Figure 1, an increasing number of studies has been carried out.



Eye traces are rarely sufficient by themselves. Thus, to better support reasoning about the cognitive processes related to reading source code, eye tracking data is often accompanied with, for example, think aloud and retrospective think aloud information. The latter is created by replaying the eye tracking videos to the subjects after they have been recorded and asking participants to explain what they did, why they looked certain parts of the code, why they navigated the source code with their gazes as they did, etc.

In this short essay, I have analyzed two eye tracking recordings where experienced programmers read code in order to understand what it does. Recordings were created by using a mobile eye tracking device attached to a monitor. This results to a video where the screen view is on the background and eye traces are drawn on top, as illustrated in Figure 2. Because of the setup, there is no information what participants look at when they do not look at the screen. The original data did not include any think aloud information or other interpretations about what the participants eye gazes.

## 2. DESCRIPTIONS OF THE TRACES

In this section the behavior of both participants is briefly described. Before diving into the stories, I advice my readers to read the program in Figure 1 by themselves, and find out what it does.

### 2.1 Participant A

Participant A started by skimming through the definitions of instance variables and the constructor. After that, he or she went straight into the main method and skimmed through it. Perhaps the participant found out from the main method that reading the whole would be beneficial, as he or she next linearly skimmed through all the methods (declared before the main). After the last method, the participant started to refer back to the code what he just went through. First, perhaps because the area method, that was the last method, uses width and height methods, the participant went to look at them. After that, perhaps because width and height methods used the instance variables, the participant went back to the constructor.

Towards the end of the session, the participant does more and more jumping and looking back and forth in the code. It may be that he starts tracing the creation of the rectangle from main method, but after tracing what the constructor does, he or she continues to other method definitions instead of returning to the main method, as the execution does. This could be to find out what the methods will return with this particular rectangle object. This is possible to find out al-

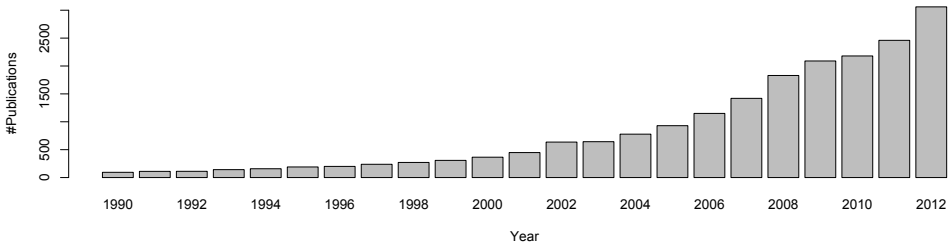



**Figure 1: Number of publications per year matching to "eye tracking" and "programming" query in Google Scholar. Numbers are not accurate because the search engine may misclassify publication years, not all publications (especially older ones) are digitally available, etc.**

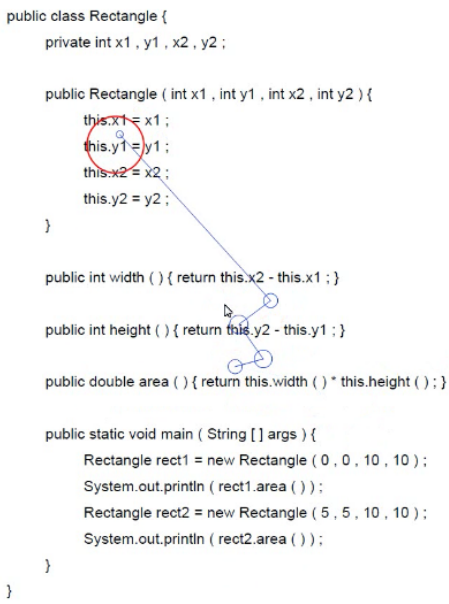



**Figure 2: A screenshot from the eye gaze data analyzed in this study. The red circle shows where the participant looks at the moment. Blue lines and circles provide information about the history where the participant looked at before.**

ready at this point because there are no methods that would change the state of an object. Indeed, when the participant returns to the main method, he or she does not need to start tracing when the area method is called.

## 2.2 Participant B

Participant B starts reading the code linearly from the first line, that is the class definition. During the first 8 seconds he or she goes briefly and linearly through definitions of the instance variables and the constructor. After that, during the next 6 seconds, he or she goes through the width method. While reading this one line method, participant scans the line back and forth and also quickly checks how the variables used in this method were initialized earlier in the constructor. The next method (i.e. height()) is almost the same as the previous width method and the participant just skims it though very briefly. The participant actually starts reading the method backwards from the end of the line – perhaps because the the two consecutive lines are so similar that it is sufficient to check how the variables used in this height method differ from the previous width method. The next method (i.e. area()) is different than the previous two methods methods and the participant spends a couple of seconds on scanning this line back and forth.

Finally there is the main method from where the execution starts. This main method has two very similar segments where a rectangle object is first created and then the area of that rectangle is printed on the screen. The participant first goes back and forth the lines inside the main method – perhaps to ensure that there is nothing wrong locally in that method. Finally, perhaps to ensure that the Rectangle really works as the participant expects, he or she seems to trace the execution related to the creation of one of the rectangles and calling the area method of that object.

# 3. DISCUSSION

## 3.1 Different Strategies in Reading the Code

As described in the previous section, participants A and B used slightly different methods in reading the code. In addition to differences in what participants looked at, the time they needed to find out what the program does differed. Participant A spend about one and a half minutes reading the code, whereas B read did that in about a minute. A

significant difference between the approaches of participants A and B is that A did a lot more long jumps and backwards referencing in the code. At some point, participant A seems to look almost everything at the same time. Participant B's approach, on the other hand, was very linear. He started from the beginning and he or she referred back to previously read sections only a few times – typically not more than once to same blocks.

At the end of the sessions, both participants started tracing what happens when an object is created. After that, participant B continued the tracing by returning to *main* and after that to the *area* method as it was called. Participant A did not return to *main* method but continued directly to other methods from the constructor.

## 3.2 What is the Task

There are different use cases when programmers read source code. For example, programmers read code of their own and code written by others. In the latter case, programmer may or may not know who has written code. In addition, programmers may or may not have some trust on that person. I argue that when reading code of others, it makes a difference if an experienced programmer is reviewing a patch from an unknown source, if he or she is reviewing code from someone trusted. This is why all eye tracking studies should report the context in details. It is an interesting avenue for future research to study how much and how the contex affects code reading strategies of experts.

I also assume that size of the code base affects to how (experienced) programmers start reading it. However, it looks like that so far most eye tracking research has focused on small programs only.

# 4. CONCLUDING REMARKS

I analyzed two short recordings of eye gaze data where experienced programmers were asked to find out what a small Java program does. I did not have previous experience from this kind of manual annotation of eye traces and I found the task quite laborious. Some of my tasks were something that should be automated. However, despite my lack of experience in analyzing eye tracking data, I found it possible to observe differences, but also similarities, in how participants read the code. As there were only two samples, I did not find annotating the data as useful as viewing them side by side.

# Eye Movements in Programming Education: Analyzing the expert's gaze

A position paper for a workshop at
Koli Calling 2013: International Conference on Computing Education Research

Suzanne Menzel
School of Informatics and Computing
Indiana University
150 S. Woodlawn Ave.
Bloomington, IN 47405
menzel@indiana.edu

## ABSTRACT

This position paper describes the author's experience with the ELAN tool for annotating the recorded eye movements of two expert programs during a code-reading exercise. From observable patterns in the gaze, strategies that the subjects may have been employing are inferred. Ideas for future research directions and the possible applications to improving Computer Science education by explicitly teaching reading skills to novices is discussed.

## 1. INTRODUCTION

This project attempts to infer the high-level cognitive processes at work during the reading of a simple Java program by an expert programmer, where the reading behavior is encoded as eye movement data. For this phase, the data for two subjects was provided as an animation.

Both subjects read the same simple 18-line Java program, but were given different instructions regarding the question they would be asked following the reading. The first subject read for 1 minute and 32 seconds, with the knowledge that the follow-up question would involve the return value of a specific method call. The second subject read for only 56 seconds, and expected to be asked a multiple choice question regarding the algorithmic idea. Both subjects were told that the code was free of errors, thereby eliminating the need to verify "compiler level" details.

## 2. ANNOTATIONS

Time segments in each animation were coded, using multiple tiers, in the ELAN Linguistic Annotator tool [1]. A controlled vocabulary was used to limit the set of possible annotations appearing in a given tier. From the observable positions and patterns, the author attempted to infer the problem-solving strategy being employed by the programmer, i.e., to see what was going on "behind the eyes".

## 3. EXPERIENCE WITH ELAN

The tiers and vocabulary were created by the workshop organizers and provided to the participants, although we were encouraged to adapt the template to our needs. Thus, my primary interaction with ELAN was to "mark up" time segments in the given animations with given annotations. Although there is ample documentation of the system available online, the acclimation to the system could have been faster and easier had a brief tutorial of the annotation procedure been provided.

Initially, I was unclear as to how detailed the annotations should be, how much coverage was reasonable, and how exacting should be the start and end points. Also, I wanted to complete the annotations for one subject in a single sitting, so I desired a ballpark estimate of how much time it could be expected to take. I sought guidance from one of the organizers, Teresa Busjahn, who shared with me her personal approach to doing the annotations and told me that it took her about two hours per video. I gratefully adopted her procedure. This was to proceed in two passes. During the first pass, only Blocks are annotated. This identifies the basic code segment the reader is concerned with during each time period. The remaining levels were covered in the second pass.

The tiers for SubBlock, Signature, and MethodCall allow for fine-tuning the description of the observable events. Generally, I didn't find these helpful, especially those that distinguished between Name and Type. This was largely due to a lack of confidence that developed in knowing the precise word corresponding to the gaze point. In the instructions to participants, we had been warned by the organizers that "the gaze point might be somewhat askew (due to head movements etc.) and that an area of several characters around the middle of the fixation can be perceived. The perceived information may span about a thumbnail around the center of the fixation." There were times when I debated my decision about the line of text that was being scanned, and making a contingent decision regarding the word on the line

seemed like a stretch.

Each video was annotated in a single session. The first took about four hours. The second video was shorter, had fewer high-level transitions, and I was more practiced with the ELAN system, so it took me under three hours.

The most interesting and important tiers are Pattern and Strategy, as this is where I relied on my intuition (garnered over three decades of teaching programming) to speculate on how the subject had decided to go about the task of comprehending the program. I am sure that I relied, at times, on my own expectation of how I would have read the program myself and where I would have proceeded next from a given point. Because there were times when it seemed that there were overlapping strategies in play, I added two additional tiers, SecondaryPattern and SecondaryStrategy. I had no trouble selecting one strategy as the dominant force guiding the subject, which is why I labeled the recessive strategy as Secondary.

# 4. INTERPRETATIONS

It is likely that the prompt influenced the subjects' approach to the reading, with the first person focused entirely on program execution and output, whereas the second needed to recognize the program's algorithm. In some real sense, the cognitive load on the first subject was less than that on the second. It is a mechanical process to trace a given program (to "be the computer"), whereas the second subject had the additional burden of formulating an abstract understanding of the code.

The two subjects exhibited vastly different behaviors, most notably in the duration of time spent in one area before moving on. An interesting statistics might be to calculate the total distance traveled by each subject.

## 4.1 Impressions of Subject1

This subject was "all over the place", with many sporadic jumps and short visits to code blocks. This is evidenced by the comparatively large number of Block annotations (92) and the frequent use of the Trial&Error strategy.

Given the concrete "what does this Area method return" prompt, I was surprised at the small amount of time spent tracing the code and viewing the Area method. This subject seemed to be overly concerned with syntax. A good deal of time was spent reading the Height method, and wandering from place to place. The effort exerted on a Debugging strategy is surprising given that the subject was informed, in advance, that the program contained no syntactic or runtime errors.

## 4.2 Impressions of Subject2

This subject's gaze was characterized by a careful, methodical, top-down scan of the code, followed by a DesignAtOnce and ProgramFlow strategies. Compared to the first subject, the gaze is more controlled and less fragmented. The total number of Block annotations is just 21. The systematic top-down reading is broken with the occasional brief TestHypothesis, which appear to be used to reinforce or confirm prior assumptions.

After the initial line by line reading, the transitions generally seem to follow the program execution. The gaze seems to pick up where it left off in the reading when returning to a code block for further review. Some annotations are clearly just stops on the way to someplace else, which would be better coded as JustPassingThrough.

This subject exhibited concentrated and localized effort. Not only were the Block annotations longer, the gaze would linger on a single line for a sustained period.

Sometimes the gaze would indicate close reading of whitespace. For example, from about 0:52 to the end shows the subject studying a blank area in the lower right. This makes me wonder if the calibration is too error prone to allow reliable coding of tokens within a line. Perhaps this could be mediated by using a larger font and smaller code segments.

# 5. VISUALIZATIONS

Mike Hansen, one of the workshop participants, created some wonderful visualizations of the eye movement data, showing which program lines the subjects fixated on.

It might be interesting to overlay a "heat map" on top of the code that shows the fixations. In cases where the subject is given a prompt to evaluate an expression, one might expect a more uniform coating than if the subject was trying to extract algorithmic meaning from the code.

# 6. FUTURE EXPERIMENTS

Java has a lot of "noise". It might be more interesting for run experiments using a language such as Scheme, which packs an algorithmic punch in a small amount of code. I would rather identify successful readership skills to discern the "algorithmic gist" of a program, as opposed to the syntactic structure.

Consider, for example, the following simple recursive procedure. The reader would be asked to evaluate, say, `(mystery '(4 7 3 8 5 2))`, and also told that the evaluation does not result in an error (so as to lighten the cognitive load). It would be interesting to note whether subjects notice the `cddr` in the `else` clause.

```
(define (mystery ls)
  (cond
    [(null? ls) '()]
    [(even? (car ls)) (mystery (cdr ls))]
    [else (cons (car ls) (mystery (cddr ls)))]))
```

Another interesting possibility is to ask the subject to employ a *Think Aloud* strategy, as much as possible, and then collect audio during the reading, as well as the gaze data. This could be used in a control group to help refine the categories in the Strategy tier.

# 7. CODING SCHEME

Some observations about the coding scheme:

1. The coding scheme provided by the organizers, and the corresponding ELAN template, omitted a code inside

the Block tier for Area. I was certain this was an oversight, so I just added that tag to the vocabulary. Also, the organizers described a TestHypothesis code for the Strategy tier in their provided materials. This was inadvertently omitted from the ELAN template.

2. The `Type` code in the `MethodCall` tier is confusing because method calls do not include type information. If the intent is to annotate the time when the gaze is over a declaration, then `Decl` is a better identifier. However, it seems that the assignment is the more interesting artifact, as in `Rectangle rect1 = new ...`, and in that case I'd suggest the code `Assignment`.

3. ProgramFlow was perhaps the easiest strategy to identify with confidence.

4. When I performed the annotations, I was unaware of the fact that participants had been assured of the error-free nature of the code they were reading. Thus, I made an assumption about them being in Debugging mode when they appeared to be carefully checking a line character by character or when they flickered from one place to another, quickly, as if verifying a small detail. In retrospect, some of these later cases may have been better categorized as TestHypothesis.

5. It is interesting to speculate how the subjects may have altered their usual reading strategies to accommodate for the fact that they knew the code was error-free. Professional programmers hardly ever have this luxury and it is probably second nature for them to verify syntax during reading. I suspect that they would not have been able to entirely suspend this behavior.

   It seems a bit of a misnomer to classify this activity as Debugging. After all, there are no bugs! I would call this AttentionToDetail. In most cases, there is a slowness to AttentionToDetail, but the subject could also be verifying a global property, such as that argument/parameter types agree or that the semi-colons are present in the right places.

6. The Debugging strategy seems to be characterized by very small jumps, where the subject is presumably validating the syntax. In contrast, DesignAtOnce is capturing high-level algorithmic thinking, thus, features rather large steps as the gaze sweeps over the text.

7. I associated the TestHypothesis code with Worry. I imagined that subject might have found the need to corroborate some assumption, as in "Wait, did I understand that correctly...". This is different from Debugging (or the proposed AttentionToDetail) in that there is a connection between what was being read previously and what is being checked, and that the gaze will return to the original point.

8. I found the Trial&Error identifier a bit difficult to grasp. At some point, I translated this in my mind to Wandering, and that seemed to help, although it might be better to have this be a separate strategy. I used this code for times when it appeared that the subject was backtracking, seemingly searching for a point to resume the reading after a particular path of reasoning had been exhausted—essentially a transition period or a brief rest between bursts of effort.

# 8. REFLECTION

I am reminded of the work done by Matt Jadud to try to extract students' cognitive processes from their compilation behaviors [2].

If we can gain insights into how experts read code, perhaps those concrete code-reading skills could be explicitly taught to learners in CS1. Using observable low-level behavior avoids the pitfalls of relying on human testimonials. In many cases, the strategies being employed by the expert may be so ingrained and practiced that the person is not even aware of them on a conscious level.

The idea that expert knowledge sometimes needs to be teased out and made concrete is something that has been studied, in the context of undergraduate education, for some time at Indiana University. A technique known as "Decoding the Discipline" was developed, initially for History [3][5], but later applied to other disciplines including Computer Science. In [4], the authors state that "Since faculty did not learn to think like historians through explicit instruction, they find it difficult to articulate what it means to think like historians." and "We present history as a model for other disciplines. They too need to uncover their ways of knowing and to teach them explicitly to students".

The cornerstone of the technique involves an intelligent non-expert interviewing the expert to discern the "tacit knowledge" that is inherent in the field, thereby bringing it to the surface. Once the hidden knowledge is made concrete, appropriate ways of developing similar skills in the new learner can be addressed. The interesting aspect of this project with the eye movements is the prospect of taking the human out of the loop because, many times, the human is unable or unwilling to honestly self-reflect. I suspect that expert programmers may have a difficult time articulating exactly how they go about reading a program, even while they are doing it, because they are so skilled at the task that they make many rapid, unconscious decisions and may fail to discern the discrete steps that form their overall strategy. They may also fail to report the "dead ends" or "false starts" in their lines of reasoning, something that would be preserved in the gaze data.

I find this to be a very exciting and rich research direction. I am eager to hear what others at the workshop think about the potential application to Computer Science education. I can imagine that this work might lead to the creation of a tool for teaching reading skills that shows the student where to look.

# 9. REFERENCES


[1] ELAN. http://tla.mpi.nl/tools/tla-tools/elan/. A professional tool for the creation of complex annotations on multimedia resources.

[2] M. C. Jadud. Methods and tools for exploring novice compilation behaviour. *ICER*, September 2006.

[3] J. K. Middendorf and D. Pace. Decoding the disciplines: A model for helping students learn disciplinary ways of thinking. *New Directions for Teaching and Learning*, (98), Summer 2004.

[4] L. Shopkow, A. Diaz, J. K. Middendorf, and D. Pace. From bottlenecks to epistemology in history. *Changing*

*the Conversation about Higher Education*, pages 17–37, 2012.

[5] L. Shopkow, A. Diaz, J. K. Middendorf, and D. Pace. The history learning project "decodes" a discipline: The union of research and teaching. *Scholarship of Teaching and Learning In and Across the Disciplines*, 2012.

# Visual evaluation of two eye-tracking renders of source code reading.

Paul A. Orlov
University of Eastern Finland
Yliopistokatu 2. P.O. Box 111
FI-80101 Joensuu, Finland
paul.a.orlov@gmail.com

## ABSTRACT
In this paper, I describe the reading process of source code. By analyzing gaze data during code reading processes, were defined eye-movement patterns which are essential part of programming comprehension. Software Visual Evaluation Tool (VETool) was developed for visual evaluation of eye-tracking data and renders. Two general patterns of eye-movements were found. The Jump Control pattern was at the beginning for both subjects. And for second subject was normal to use the Line Scan pattern.

## Categories and Subject Descriptors
D.3.3 [**Programming Languages**]: Language Constructs and Features – *abstract data types, classes and objects, control structures. H.5.2 [**User Interfaces**] : Interaction styles, Theory and methods*.

## General Terms
Human Factors, Measurement, Languages.

## Keywords
Eye-tracking, source code reading, data visualization, pattern, strategy.

## 1. INTRODUCTION
The analysis of eye - movements is used for understanding of human behavior and different psychological aspects. In his basic work Yarbus describes, that eye-movement patterns and vision strategy depend on the task (Yarbus, 1965). His thesis actual not only for strong visual tasks like visual searching or reading. More abstract tasks also determined visual strategy. For example, when experimenter gives first task about age evaluation of the person and second task about evaluation of emotional conditions of the person on picture. In that two tasks both visual strategy and eye-movements patterns are different.

The measurements of eye movements are very common for understanding of humans activity, visual information processing and comprehension of objective reality. And the problem is how to interpreted eye movements. In psychology of programming (PoP) eye-movements patterns mostly corresponds with the meaning of stimuli. It means that we should assume, that if subject look at the source code element, like variable definition, he should think about this variable. This hypothesis plays vital role if we try to understand mental process through eye-movements, for example, source code comprehension.

Workshop at the 13th KOLI CALLING INTERNATIONAL CONFERENCE ON COMPUTING EDUCATION RESEARCH.
Joensuu, Finland, November 13th - November 14th, 2013

## 2. OBSERVATIONS AND INTERPRETATIONS
In two video renders subjects have to understand a Java program and to answer questions about them. Eye movements (gaze fixation, saccades and gaze path ) were visualized for evaluation. Both participants are experts in programming. First part of task context was the instruction. How often programmers should answer questions about source code in real live? The physical experiment context rebuild usual Integrated Development Environments (IDE) view, size, colors and formatting of source code. In definition of "task" we have to agree that current conditions relevant only for laboratory study.

For visual evaluations were used ELAN software and were build new software tool for dynamic visualization of ELAN and eye-tracking data Visual Evaluation Tool (VETool). VETool is an open source software. Source code of VETool can be found here: https://bitbucket.org/orlovpa/visual-evaluation-tool-vetool .

### 2.1 Comparing by "primitive" events

#### *2.1.1 Attributes (Block), Type (Signature annotations), Formal Parameter List (Signature annotations), Name (Signature annotations)*
Both subjects looks at these elements at the beginning of reading and rarely after middle of total time. First points of fixation are in the physical center of stimuli, but then, both subjects moves their gaze to class attributes. Also duration of looking at class attributes and at the name of signature are quite same. Times for each block are different.

Attributes and Type elements are shown on Figure 1.

#### *2.1.2 Constructor*
Similar situation could be found in duration of seeing on Constructor block. For this block of source code there are two activity intervals: at the 5 sec (at the beginning) and at 25 - 40 sec. First subject looks more often back to this block than second one. Constructor element shown on Figure 2.

#### *2.1.3 Area*
Participants look at this block first time at about 20 sec and then, they back to this block once or twice.

#### *2.1.4 Width*
This block shows different looking times and in different time intervals. First subject backs here several times for about 1 sec each. Second subject looks here twice and first time quite long. But total time of looking on this block could be similar.

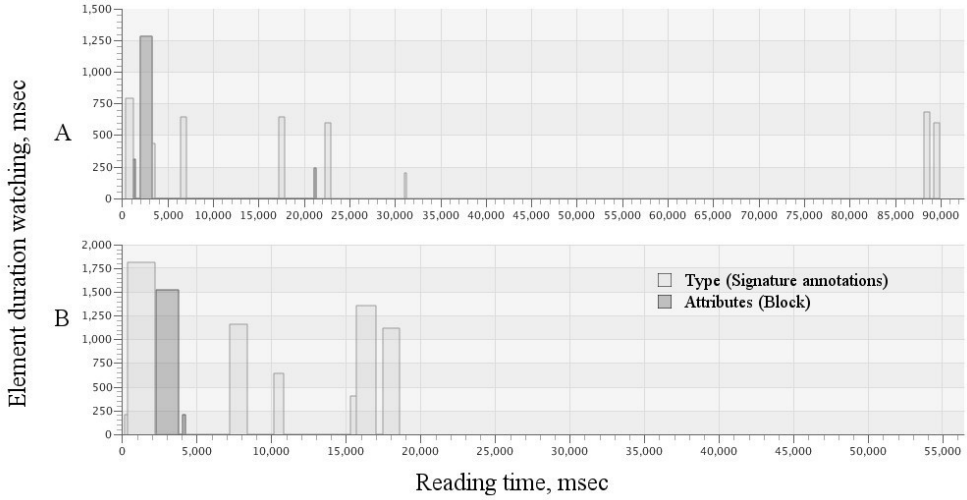



**Figure 1: Time cyclogram for Attributes and Type elements. Letter A shown first subject. Letter B shown second subject.**

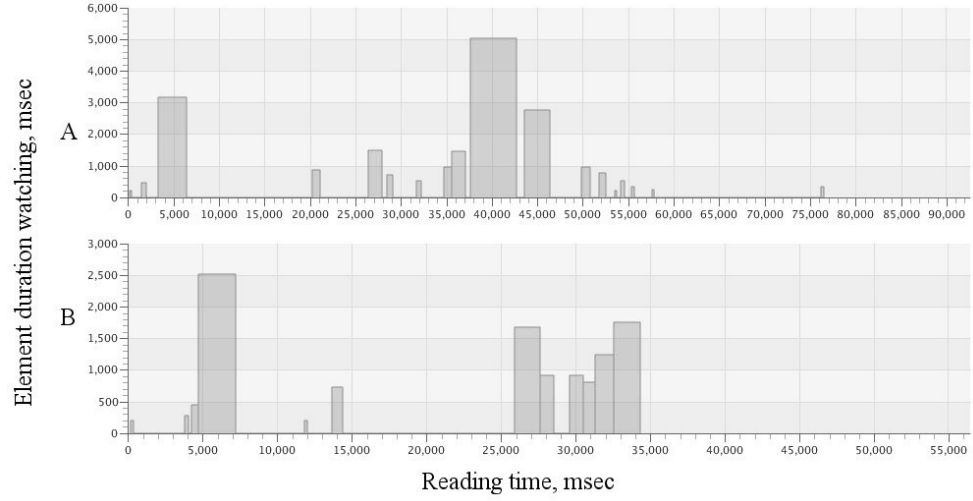



**Figure 2: Time cyclogram for Constructor element. Letter A shown first subject. Letter B shown second subject.**

### *2.1.5 Main (Block), Body (Sub-Block), Actual Parameter List and Names (Method calls) and Method returns element.*

The Main block is the popular for looking for both subjects. And the similarity is in the interval of seeing, both subject looking here from the middle time. They spend at this block much time (more than 20 sec). In the Main block they look at Body sub-block and at method calls inside sub-block. That is why these three elements are very similar for total time and interval. All these blocks are interesting for subjects gaze after the middle of total spending time and at the end. Main element shown on Figure 3.

In other “primitive” events I could not find any interesting moments. All of them are quite individual for participants.

## 2.2 Comparing by patterns

I found two general patterns that were used by subjects. The Jump Control pattern was at the beginning. This situation is the same for both subjects. And for second subject was normal to use the Line Scan pattern. One interesting moment could be the way of Line Scan. There are reading from right to left also. This “back” reading should not be the same with backwards saccades (regressions) in normal reading. Line scan patterns shown on Figure 4.

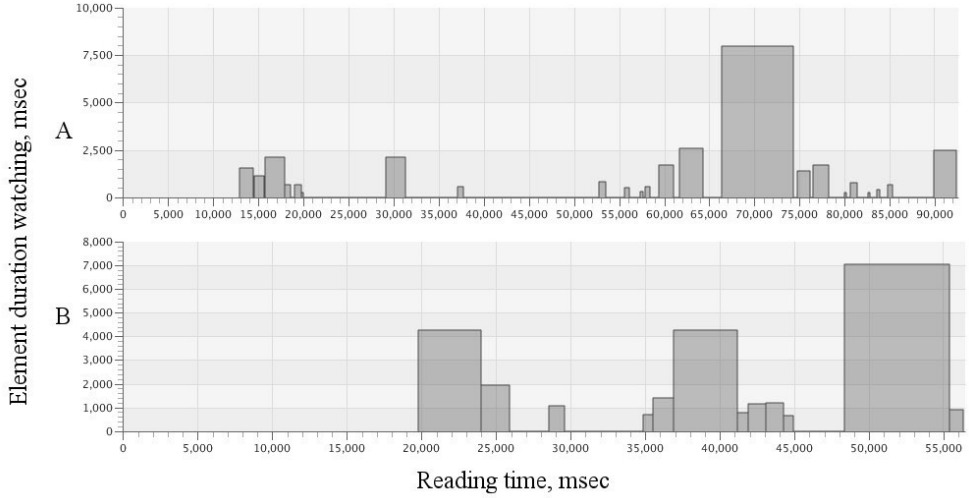



**Figure 3: Time cyclogram for Main element.**
**Letter A shown first subject. Letter B shown second subject.**

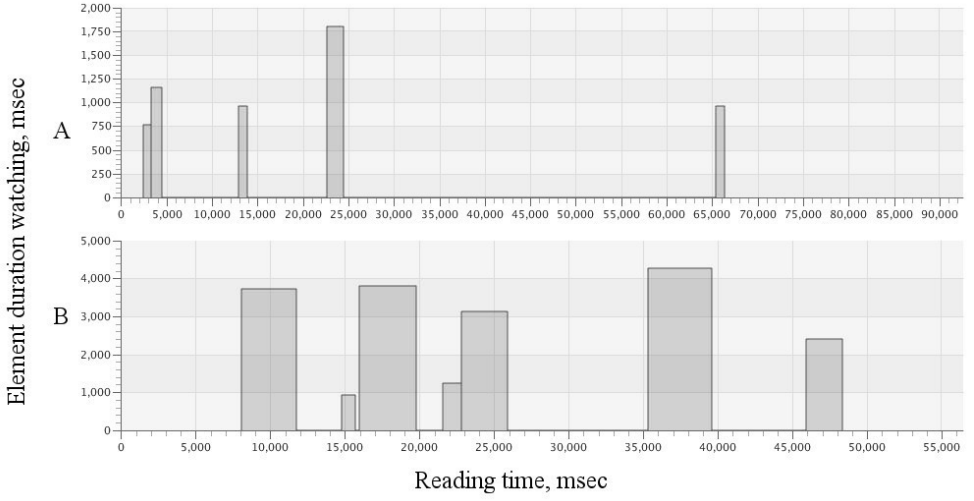



**Figure 4: Time cyclogram for Line Scan pattern.**
**Letter A shown first subject. Letter B shown second subject.**

## 3. INTERPRETATIONS AND DISCUSSION

In PoP eye-movements patterns should also depend on task. The term “task” should be given a definition. The task is a mental construction in humans mind that formed by instruction and context (Gippenreiter, 1978; Rayner, 1998). Humans can be instructed in different forms, like verbal or visual. Instructions comes from objective reality, but human does their interpretation through the individual context. Context can be influenced by subjective emotional factors, previous experience, social and physical factors (Muller et al., 2012). In PoP there are different aspects of context also, but numerous papers skips context (except studies with gaze controlled systems and gaze contingent systems).

Visual evaluation of these two eye-movements shows that there is not one significant picture. Even though subjects are both experts in programming, they have mostly different eye-movements patterns. Only at the beginning of reading (working) they have, may be, Jump Control, but then they go in individual ways. It seems, that it is necessary to take into consideration more factors to determine the context of the task.

Finally, if we would like to identify the Strategy, we have to go back to the Task definition. In current study subjects are expert in programming, and they decide (may be at first gaze fixation), what kind of strategy is necessary to use. If they got instruction like this: “There is a bug, find it!”, they will use different strategy. Subjects were informed about the task, before they read the code. Subject1 was told, that there will be a question about the return value of “rect2.area()” after the program was executed. And for subject2 the information was, that there will be a multiple-choice question about the algorithmic idea. So, they both use Program Flow and both try to not only understand, but remember this program to answer questions. Of course, that kind of task and problems in real professional life are not exactly the same.

This situation shows that there are so much interesting finding in this field of science in future!

## 4. ACKNOWLEDGMENTS

I thank all organizers ( Roman Bednarik, Teresa Busjahn & Carsten Schulte) of Workshop at the 13th KOLI CALLING INTERNATIONAL CONFERENCE ON COMPUTING EDUCATION RESEARCH. I like to express profound gratitude to Teresa Busjahn for the idea to build such interesting materials for the workshop.

## 5. REFERENCES

[1] Gippenreiter, Y. B. (1978). Движения человеческого глаза [Movements of the human eye] (p. 256). Moscow: Изд-во Московского государственного университета.

[2] Muller, M. G., Kappas, A., & Olk, B. (2012). Perceiving press photography: a new integrative model, combining iconology with psychophysiological and eye-tracking methods. Visual Communication, 11(3), 307–328. doi:10.1177/1470357212446410

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

[4] Yarbus, A. L. (1965). Роль движений глаз в процессе зрения [Eye movements and vision]. (p. 173). Moscow: Изд-во “Наука.”

# Finding Patterns and Strategies in Developers' Eye Gazes on Source Code

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

***Abstract*—This paper presents observations on patterns and strategies expert developers use while reading source code. An interpretation of two code segments of two expert developers is given in the context of a coding scheme. Results indicate that the method of reading source code varies based on the task however some similarities are noted. Implications of these results to Computer Science education are presented.**

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

## 1. Introduction

Computer Science is currently being taught at most major Universities with a focus on code writing without really introducing methods on how to first read the code. Reading code is important because it is the first thing developers do as part of most software tasks such as bug fixing and impact analysis. Glass [1] states that we should approach learning a programming language the same way we learn any other language. First, a child learns how to read a language and later develops writing skills.

How do we read and comprehend code? In order to answer this question, a group of researchers at Freie Universitat Berlin and the University of Eastern Finland conducted a workshop at the Koli Calling 2013 conference dedicated to provide some insight into this question. They provided workshop participants with a two short videos of expert developer's eye gaze and a coding scheme. The workshop participants were required to annotate the eye gaze using ELAN and find patterns and strategies based on their annotations. Table 1 gives the task given to each subject. The subjects were expert developers. These tasks were given to the subjects before the source code was even shown to them. Both subjects were given the same 23 lines of source code.

In order to answer the above question on how programmers read and comprehend code, we introduce specific research questions based on the types of activities software developers are engaged in while they are reading the code. The research questions we attempt to address are:

- RQ1: What specific parts of the program do programmers look at most/least?
- RQ2: What comprehension strategies are used together?
- RQ3: Does the eye movement depend on the task being solved?
- RQ4: What are the similarities and differences in eye gaze between different tasks?

We do not generate any hypotheses for the above research questions since this is a purely observational study and reflection of our interpretation of the results. The videos are qualitatively assessed in a somewhat structured manner based on the coding scheme given.

The next section gives our interpretation of the eye gaze in the two code segments. In Section III, we present some discussion about the coding scheme used and modifications we made. Finally we conclude with how eye movements can be used in computer science education.

## 2. Source Code Interpretations and Observations

The coding was done by the second author of the paper using the coding scheme provided to the workshop participants. The ELAN files that represent our coding can be downloaded from http://www.csis.ysu.edu/~bsharif/koliworkshop13/.

The main coding events fall into four main categories: Block, SubBlock, Signature, and MethodCall. Each of these categories are further decomposed into codes based on the identifiers, methods, and functions in the program given to the two programmers.

### *A. Subject 1 – Specific Task*

The first subject was told that they would be asked about the return value of rect2.area() as shown in Table 1. We categorize this as a specific task because they knew about a specific method that they would be asked about.

The subject spent the first 20 seconds scanning [2] the program from top to bottom. The latter part of the time was spent reading the constructor and the three methods and mapping actual parameters to the formal parameters for the rect2 object. The highest number of fixations were on the rect2 object and the height() method. We could come to the conclusion that the subject was trying to calculate the area of rect2 based on his/her eye gaze. This is indicative of the task given. In other words, the subject was trying to trace the program and the eye movements were scattered between the main function and the methods called from main showing that the subject was trying to mimic the behavior of a compiler.

***Table 1. Tasks, Patterns, and Strategies***

| Subject | Task | Time | Scan Time | Patterns | Strategies |
|---|---|---|---|---|---|
| Subject 1 | Asked about the return value of rect2.area() | 92 secs | 20 secs | First 20 seconds: LineScan, Linear<br>Later: JumpControl, LineScan | First 20 seconds: DesignAtOnce, Trial&Error<br>Later: DesignAtOnce, ProgramFlow, Debugging |
| Subject 2 | Multiple choice question about the algorithm | 56 secs | 26 secs | First 26 seconds: LineScan, Linear<br>Later: JumpControl, LineScan | First 26 seconds: DesignAtOnce, TestHypothesis<br>Later: ProgramFlow |

The pattern evident in the first 20 seconds was mainly LineScan and Linear. In the latter part of the eye gaze, the patterns that emerged were many alternating JumpControl and LineScans. The strategy evident in the first 20 seconds was DesignAtOnce and Trial&Error. In the latter part of the video the strategy that emerged were DesignAtOnce, ProgramFlow, and Debugging in that order.

Subject 1 was asked afterwards: "What is the return value of rect2.area()?" and gave the correct answer – 25.

*B. Subject 2 – General Task*

The second subject was told that they would be asked about the algorithmic idea in multiple choice form. We categorize this as a general task because they were not told about anything specific to look for a priori.

The subject spent the first 26 seconds scanning the program from top to bottom. We do not observe any mapping of actual to formal parameters. The number of fixations on the rect2 object was the highest. There was equal emphasis with respect to number of fixations on the width() and area() method definitions. From the latter part of the eye gaze after the first scan, the subject focused again on the Rectangle constructor, looked at how rect2 was being instantiated, and checked the area method again before looking at the rect2 object. We believe that this subject focused more on rect2 because it was the last object in the program.

The pattern evident in the first 26 seconds was also mainly LineScan and Linear. In the latter part of the eye gaze, the patterns that emerged were also JumpControl and LineScan with very little context switching between the two patterns.

The strategy evident in the first 26 seconds was DesignAtOnce and TestHypothesis in that order. In the latter part of the video the strategy that emerged was ProgramFlow. Each of these patterns and strategies are mentioned in Table 1.

Subject 2 was asked the following multiple choice question and chose a, which was incorrect. The correct answer was b.

This program

a) computes the area of rectangles by multiplying their width (x1-x2) and height (y1-y2)

b) computes the area of rectangles by multiplying their width (x2-x1) and height (y2-y1)

c) computes the area of rectangles by multiplying their width (x1-y1) and height (x2-y2)

d) I'm not sure.

*C. Other Observations and Caveats*

This small experiment on two subjects shows that if the subject does not know what to look for it is difficult for them to remember details like parameter ordering as seen by the answer given by Subject 2.

The differences in strategies begin to appear after the initial scan. It could be possible that small differences in the initial scan could cause the second phase to follow different strategies. However, since we only got one video for each task, we were not able to consider this possibility.

We noticed that the eye movements of Subject 2 were much more focused compared to Subject 1. In other words, we did not detect many stray glances in Subject 2's data. We define a stray glance as something they look at that does not necessarily involve comprehension or something that we cannot explain. Subject 2 quickly read the code and tried to trace it. There were not many places where Subject 2 re-reads the lines (regressions) – at 7 seconds through 10 seconds the subject re-reads the width method. We also have to point out that Subject 1 did have three sections of the video where the eye data was not available so the above observation should be considered with caution.

With respect to the two videos in question, it is not possible to generalize or come to any conclusions without more data about how the number of fixations might relate to how difficult the task is. We could conjecture that the more fixations a line has, the more difficult it is to comprehend but more studies are needed to validate this claim. It is also possible that fixation duration or pupil diameter might be a better alternative. Another possibility could be to look at smaller time windows of 10 or 15 seconds instead of looking at fixations in the entire dataset.

*D. Possible Threats to Validity*

First, we cannot come to any general conclusions based on only one data point for each task. Both subjects had a high number of fixations on the object rect2 (line 20). It would be interesting to see how the fixations would change if subject 1 was asked about the return value of rect1 instead of rect2. A possible explanation here would be that because rect2 is the last object in the program, subject 2 also focused on it more than rect1. The specific durations of the fixations were not provided. It is possible for a method to have few fixations but have longer durations of them indicating higher cognitive load.

### E. Preliminary Insights into Research Questions

We give some preliminary insights and start to answer the research questions posed in the Introduction based on the two videos. These will be further refined in future work.

- *RQ1: What specific parts of the program do programmers look at most/least?* – The programmers looked at private member variables sparingly and only in the beginning. They focused mainly on the constructor and the main function body. All the methods in this program were a single line and the programmers looked at each of those as well although with less frequency.
- *RQ2: What comprehension strategies are used together?* – We answer this question based on the coding scheme given. The DesignAtOnce was used during the scanning phase along with (Trial&Error and TestHypothesis). In the latter phase DesignAtOnce, ProgramFlow, and Debugging were observed.
- *RQ3: Does the eye movement depend on the task being solved?* – Yes it is very evident from the two videos that the low-level eye gaze behavior follows different trends for the two tasks involved.
- *RQ4: What are the similarities and differences in eye gaze between different tasks?* – Both the programmers first scanned the entire code from top to bottom indicated by the DesignAtOnce strategy. Both also tried to understand how the program executes (ProgramFlow). The differences were evident in the period after the initial scan. The subject with the specific task focused on trying to find the answer to the method call area() whereas the subject with the general task focused on understanding the Rectangle constructor and the area() method without a need to find the specific result of the method calls.

## 3. Modifications to Coding Scheme

It is highly possible that two different coders will code the same video in entirely different ways. There are some subtle differences between the strategies presented. With respect to Trial&Error, it is hard but not impossible to gauge the reading speed from the videos.

We found the coding scheme quite comprehensive overall covering all scenarios found in the two videos. We made some modifications to the coding scheme and list them below.

- Added class, main, area codes to the Block Tier.
- Added visibility code to the Signature Tier. This will determine if they looked at the visibility of the methods such as the keyword public or private.
- Added PrintLine code to the MethodCall Tier.
- We added a child tier FormalParameterList for Signature to be more specific when a person looks at the formal parameter list. This child tier has four codes x1, x2, y1, and y2.
- In order to annotate two patterns at the same time we added two child tiers to Pattern (Pattern1 and Pattern2)

We did not find the Pattern Signatures in the two videos provided. Note that we did not specifically comment on the use of the PrintLine or visibility code that we added in the strategies above, leaving that as future work. This does not mean that they will not be useful in another type of task such as a bug fixing task, an impact analysis task or new feature task.

In future work, we plan to have the videos coded by another coder, compare the findings and calculate the inter coder reliability rating.

## 4. Future Work on Eye Movements in Computer Science Education

The use of eye movements has tremendous potential in Computer Science education. First, beginning programmers can be shown eye tracking videos of expert programmers performing tasks such as the ones presented here. The novices get to see firsthand how code is supposed to be read. This increases their awareness while they read code by themselves.

Second, beginning programmers can track themselves while they are solving a task and later analyze in retrospect what they were thinking while solving the task. All this builds self-awareness that eventually teaches a beginner how to learn to read code efficiently. Of course an eye tracker would be required for this purpose.

There are several research questions we pose with respect to using eye tracking for Computer Science education:

- What task is the most difficult for beginning programmers?
- How do beginning programmers write code after they read and comprehend it?
- What strategies are used in program debugging?
- What tools could help beginning programmers increase their productivity?

In order to answer the above questions, a systematic family of empirical studies need to be carried out in a way that they can be replicated in the future thereby adding to the body of knowledge and evidence of computer science education.

## References


[1] R. L. Glass, *Facts and Fallacies of Software Engineering*: Addison-Wesley Professional, 2002.

[2] H. Uwano, M. Nakamura, A. Monden, and K. Matsumoto, "Analyzing individual performance of source code review using reviewers' eye movement," in *2006 symposium on Eye tracking research & applications (ETRA)*, San Diego, California, 2006, pp. 133-140.

# Eye movements in programming education: analysing the expert's gaze

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

## ABSTRACT
This is Simon's contribution leading up to the workshop on eye movements in programming education that is to be held in conjunction with Koli Calling 2013. It encompasses brief descriptions of two short segments of gaze-tracking data, thoughts about the coding scheme, and general thoughts about the use of gaze tracking in computing education research. The contribution has been revised in response to comments by the workshop leaders.

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

## General Terms
Measurement

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

## 1. INTRODUCTION
Two expert programmers were invited to read the same short piece of code in the expectation of being asked a question about it. The code is a class representing a rectangle, with two pairs of x-y coordinates as its attributes, and methods to return its length, width, and area. The main method declares two rectangles and prints the area of each.

There are clear differences between the approaches of the two participants. They were in fact told to expect different questions, one involving tracing the code and one involving the algorithm. However, the differences in gaze appear deeper than this difference in what they were expecting to be asked, and suggest that different readers read code in markedly different manners.


*Koli Calling '13*, November 14-17 2013, Koli, Finland.

http://dx.doi.org/...

## 2. SAMPLE DESCRIPTIONS
Reader 1's gaze might politely be described as erratic. Considered in real time, it flashes wildly about the code, generally spending very little time on any one point. Viewed over time, there is a clear pattern of returning to certain focal points, points that are pertinent to the question that the reader was told to expect; but the gaze fixations are so brief as to leave the analyst wondering whether it is possible to gain any comprehension of the code. For example, in the ten seconds between about 52s and 1m02s, gaze shifts more than a dozen times between the main method, the constructor, and the width, height, and area methods, typically spending less than a second on each point of interest.

Reader 2, by contrast, appears to read the code slowly and methodically. There are elements of linear scanning, and gaze remains far longer on areas of interest. By contrast with the ten-second span described above for reader 1, between about 34s and 44s reader 2 focuses on just one line of code, the declaration of rectangle2. After one-second glances at height and width, there is a steady four seconds on area followed by another eight seconds on the declaration of rectangle2. The impression is of a slow and deliberate analysis of the code, suggesting that most of it is understood the first time it is considered.

It is tempting to suggest that reader 1 is unlikely to have understood the code in the time during which the gaze was recorded. However, both readers are professional programmers, so this seems unlikely – unless it turns out that this reader was unable to correctly answer the subsequent question.

Reader 1 was expecting a question about the output of rect2.area(). It is clear that the scanning was indeed addressing this particular question: for example, the gaze returns frequently to the declaration of rect2, and very seldom to the declaration of rect1. The gaze also frequently returns to the area() method, and to the height() and width() methods that are called by area. The reader has clearly identified the relevant parts of the code and is working on absorbing them; yet there is no hint of the methodical linear (or rather, flow of control) reading that one might expect to be associated with code tracing.

On the other hand, reader 2 was told to expect a question about the overall algorithm. This is a broader question that would entail comprehension rather than tracing; yet reader 2 displays more of the flow-of-control reading style that one might think would be associated with tracing.

It is clear that these two expert programmers have markedly different code-reading styles. As a code reader myself, I have no difficulty seeing how reader 2's approach could lead to program comprehension; for me the challenge is to hypothesise a way in which reader 1's erratic reading can lead to the same outcome.

After making these observations I learnt that reader 1 was asked 'What is the return value of rect2.area()?', and gave the correct answer of 25. Reader 2 was given the multiple-choice question:

> This program
>
> a) computes the area of rectangles by multiplying their width (x1-x2) and height (y1-y2)
>
> b) computes the area of rectangles by multiplying their width (x2-x1) and height (y2-y1)
>
> c) computes the area of rectangles by multiplying their width (x1-y1) and height (x2-y2)
>
> d) I'm not sure.

In response to this question, reader 2 chose option *a*, whereas the correct answer is option *b*. Option *a* expresses a correct outcome, but not the exact implementation that leads to that outcome. A generous interpretation would be that reader 2 understood the nature and purpose of the algorithm, but did not remember its detail.

By contrast, reader 1's wild flashing about the code does seem to have taken it in, as reader 1 correctly answered the question.

## 3. THE CODING SCHEME

The coding scheme consists of a number of 'tiers', each of which can be coded with a choice of values. The tiers are summarised below.

**Block** indicates in which block of code the participant's gaze is working. The simple code used for this example has six basic code blocks: the (rectangle) class attributes, the constructor, the main method, and three further methods, height, width, and area. Therefore Block has six possible coding values, one for each of these.

**SubBlock**: some of the blocks have identifiable sub-blocks in which a reader's gaze might rest. The sub-blocks that have been coded are Signature, Body, and Return. While the return statement is part of the body of a method, its particular importance means that it is likely to be the focus of some concentration by the reader. Another part of the body that is similarly likely to receive attention is method calls, which are coded in a separate tier. It is interesting that a method signature is coded as a sub-block and additionally in its own tier, whereas method calls are coded just in their own tier. I can see the benefit of the separate tier, which permits the coding of which part of the signature is occupying the reader's gaze. But at the same time I fell that *body* is potentially a huge sub-block, if we analyse gaze on more substantial code, with many other statement types that would each merit their own tier, types such as assignment, iteration, selection, and more.

Perhaps my confusion here is that this is a coding scheme specific to this single piece of code, whereas I have been trying to envisage it applied to bigger and more varied code passages. When I do see it applied to different passages, I expect that my confusion will dissipate.

**Signature**: when gaze rests on the signature sub-block, this tier further indicates whether it dwells on the method name, its type, or its formal parameter list.

**MethodCall**: when gaze rests on a method call, this tier is used to indicate whether it is focusing on the method name or its actual parameter list. The method's type is also included in this tier, but we note that this information is not included in a method call, so this value will never be used, and should be removed from the scheme.

**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 (of any type) sequentially, regardless of order of execution; LineScan, in which gaze concentrates on a single line in it entirety; Scan, in which gaze reads a sequence of lines briefly, then returns to concentrate on points of interest; and Signatures, in which gaze covers a number of method signatures before moving to the bodies of the methods. There would seem to be scope for many further patterns. Two possible patterns that I have identified are 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.

**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. As with patterns, there would seem to be scope for further strategies. For example, I could envisage a use for a FlowCycle strategy, in which the same program flow sequence might be followed several times; the intent might be to gain a first understanding of the flow, strengthening an reinforcing it with repeated examinations of the same code. The Flicking pattern might then suggest the simplest level of the FlowCycle strategy. In addition, the Debugging strategy might in fact be broader than its name suggests, as we might see similar gaze patterns (but infer different intentions) in readers who are trying to comprehend a piece of code that is not believed to contain bugs.

## 4. REFLECTIONS

The two gaze-tracking examples considered here lead to the observation that different experts have markedly different code-reading styles.

How would one use gaze tracking in computing education – for example, in the teaching of introductory programming? One approach might be to examine the gaze of programming novices and determine how closely it resembles that of experts: the more expert-like the novice's gaze, the more expert-like the novice would appear to be. Unfortunately, the very small sample of expert programmers examined in this work suggests a major flaw: that experts do not examine code in the same way; that there are at least two, and possibly many more, different ways of examining code in order to successfully comprehend it.

While there is much literature linking the ability to read and comprehend code with the ability to write it, there is also

substantial evidence that many programming novices have not yet acquired the ability to read and comprehend program code. This suggests another weakness in the idea of comparing the gaze of novices with that of experts: the experts are presumed to be able to read code, whereas novices are not.

There is still clearly value in analysing students' patterns of gaze. This technique could be used, for example, to determine when students are looking at entirely the wrong section of code, or to determine that they are looking wildly all over the code without ever settling on any particular piece. But this is not necessarily the same as comparing their gaze with that of experts.

## 5. ACKNOWLEDGEMENTS

This work would never have been done without the impetus and inspiration of Teresa Busjahn, Carsten Schulte, and Roman Bednarik. For this I am deeply indebted to them.

## 6. REFERENCES

[1] None at this point . . .


# Eye Movements in Programming Education

## Analyzing the Expert's Gaze

Workshop at the 13th KOLI CALLING INTERNATIONAL CONFERENCE ON COMPUTING EDUCATION RESEARCH

Joensuu, Finland, November 13th - November 14th, 2013

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

Computer Science Education Research and Teaching mainly focus on writing code, while the reading skills are often taken for granted. 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.

This first workshop gives participants an opportunity to get insights into code reading with eye movement data. However, as this data only reflects the low level behavioral processes, the challenge to tackle is how to make use of this data to infer higher order comprehension processes. We will take on this challenge by working on a coding scheme to analyze eye movement data of code reading. The links between low and high level behaviors will help computing science educators to design, realize and reflect on the teaching of code reading skills.

Furthermore, we aim to open discussion about the ways of explicit teaching of readership skills in computing education. Therefore we will discuss the role of reading skills in teaching programming, facilitated by position papers of each participant.

To participate send a mail to teresa.busjahn@fu-berlin.de. It is possible to participate independent of attending Koli Calling. Participants will get eye movement data of reading and comprehension processes of expert programmers, and a coding scheme for annotating the process. You will annotate the video, and reflect on the (perceived) intentions behind the visible pattern. Applying and refining the coding scheme on the data gives insight into the higher order comprehension strategies of the reader.

A short individual reflection and position paper of the results and perspectives for teaching programming is required by the participants [max. 2-3 pages]. As a result, participants will jointly prepare a paper with the data and the refined coding scheme.

IMPORTANT DATES

- Making data and tools available for participants, as well as instructions for coding and position paper: beginning of September 2013
- Deadline for submissions: October 14, 2013
- Workshop: November 13th (evening) – November 14th

Visit www.mi.fu-berlin.de/en/inf/groups/ag-ddi/Gaze_Workshop/expert/ for details.

# *Sample visualizations of gaze data*

The eye movement data was recorded using Ogama (www.ogama.net) and an SMI RED-m Tracker (120 Hz). Visualizations were done with eyeCode (http://eyecode.synesthesiam.com/stories/koli-calling.html).

## SUBJECT 1

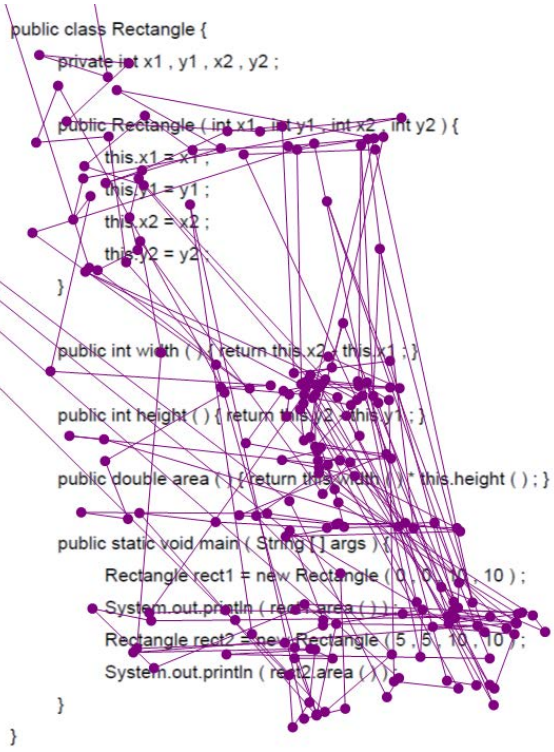
Instruction given to participant before the source code was shown:

You will be asked about the RETURN VALUE of 'rect2.area ()' after the program was executed.

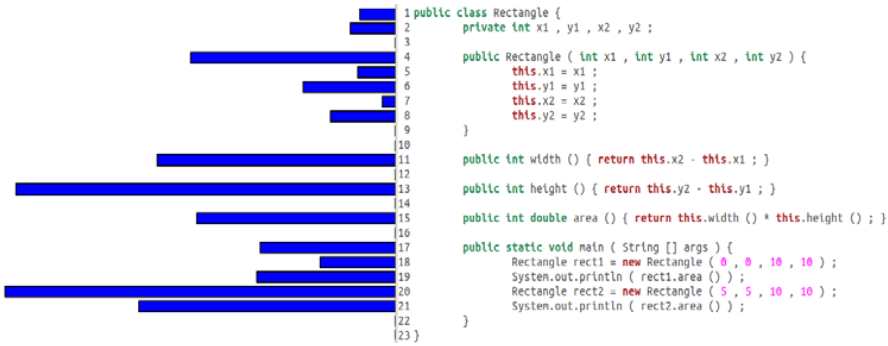
Comprehension task given after the source code was shown:

What is the return-value of 'rect2.area ( )'?

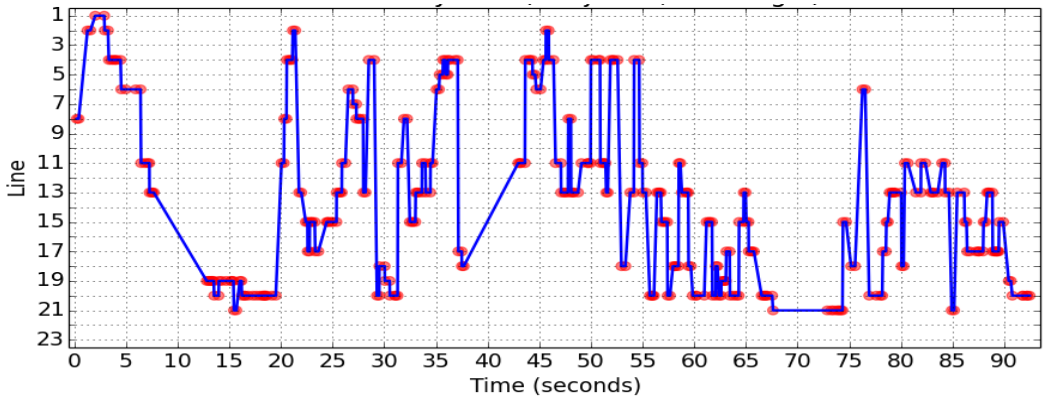
Subject's answer: 25




*Scanpath - Subject 1*




*Fixations per lines - Subject 1*




*Timeline - Subject 1*

## SUBJECT 2

Instruction given to participant before the source code was shown:

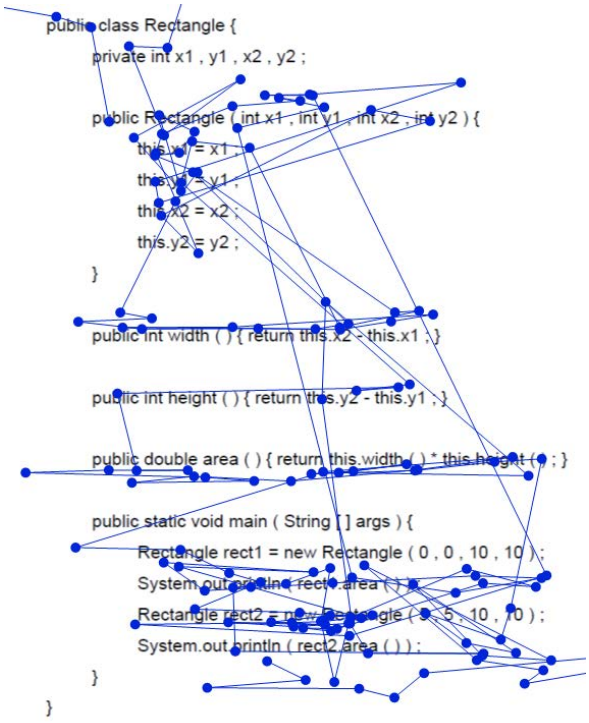
You will be given a MULTIPLE CHOICE question about the algorithmic idea.

Comprehension task given after the source code was shown:

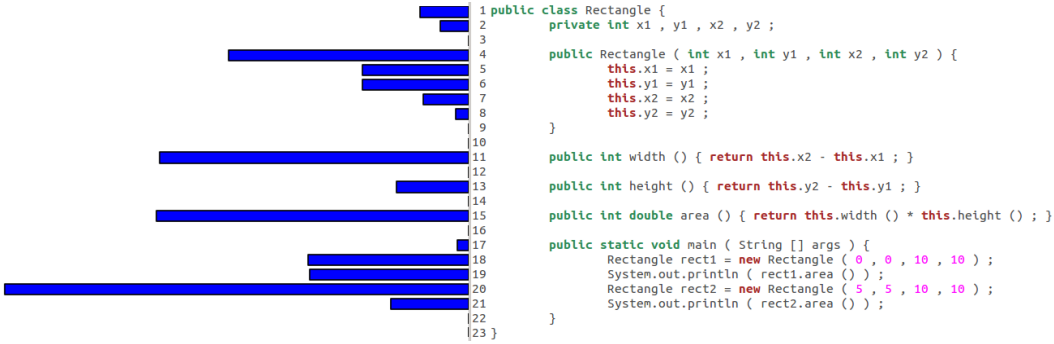
This program

a) computes the area of rectangles by multiplying their width (x1-x2) and height (y1-y2)
b) computes the area of rectangles by multiplying their width (x2-x1) and height (y2-y1)
c) computes the area of rectangles by multiplying their width (x1-y1) and height (x2-y2)
d) I'm not sure.

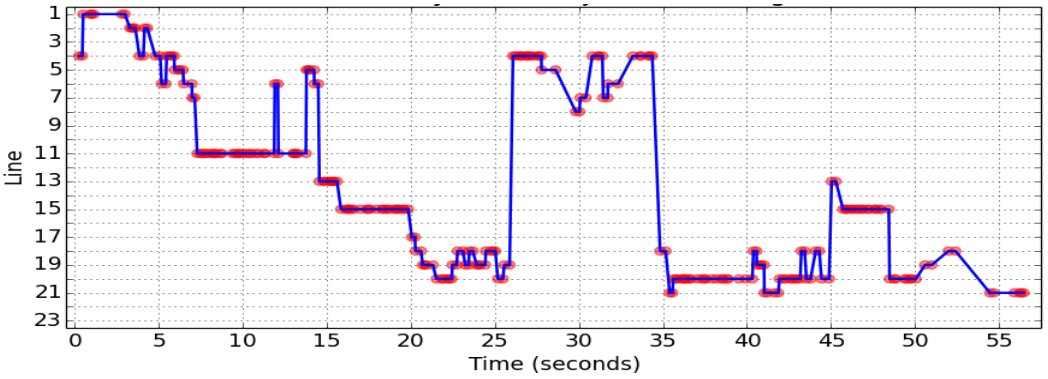
Subject's answer: a




*Scanpath – Subject 2*




*Fixations per lines - Subject 2*




*Timeline - Subject 2*

# *Revised coding scheme*[1]

| **Tier/ Category** | **Codes** | **Description** | **Classification** |
|---|---|---|---|
| (Lexical) Element | Public1, Double1 | (Lexical) element on which the fixation occurs, e.g. an operator or identifier | Observable |
| Line | Line1, Line2 ... | Line on which fixation occurs | Observable |
| Block | Attributes, Constructor, Height, Main, Width, Area | General area in which fixation occurs, e.g. the height-method, the main-method etc. | Observable |
| SubBlock | Body, Return, Signature | Specific region in which fixation occurs, e.g. a signature or a line containing a return-statement. Can be nested. Granularity depends on structures of interest. | Observable |
| Signature | FormalParameterList, Name, Type, Visibility | Precise code section | Observable |
| FormalParameterList | x1, x2, y1, y2 | Precise code section | Observable |
| MethodCall | ActualParameterList, Name, PrintLine | Precise code section | Observable |
| Pattern | Flicking, JumpControl, JustPassingThrough, LinearHorizontal, LinearVertical, RetraceDeclaration, RetraceReference, Scan, Signatures, Thrashing, Word(Pattern)Matching | **Flicking**: The gaze moves back and forth between two related items, such as the formal and actual parameter lists of a method call.<br>**JumpControl**: Subject jumps to the next line according to execution order.<br>**JustPassingThrough**: Fixations are on a blank spot and clearly just stop on the way to someplace else. | Observable |

1 The scheme was developed with the specific Rectangle program in mind.

| | | | |
|---|---|---|---|
| | | **LinearHorizontal**: Subject reads a whole line either from from left to right or right to left, all elements in rather equally distributed time.<br><br>**LinearVertical**: Subject follows text line by line, for at least three lines, no matter of program flow, no distinction between signature and body.<br><br>**RetraceDeclaration**: Often-recurring jumps between places where variable is used and where it had been declared (Uwano et al. 2006). Form of Flicking.<br><br>**RetraceReference**: Often-recurring jumps between places where variable is used and where it had been recently referred to (Uwano et al. 2006). Form of Flicking.<br><br>**Scan**: Subject first reads all lines of the code from top to bottom briefly. A preliminary reading of the whole program, which occurs during the first 30 % of the review time (Uwano et al. 2006).<br><br>**Signatures**: Subject looks at all signatures first, before looking into method/constructor body.<br><br>**Thrashing**: The gaze moves rapidly and wildly in a sequence that appears to make no particular sense.<br><br>**Word(Pattern)Matching**: Simple visual pattern matching. | |

| | | | |
|---|---|---|---|
| Strategy | AttentionTo-Detail, DataFlow, Debugging, Deductive, DesignAtOnce, FlowCycle, Inductive, Interprocedural-ControlFlow, Intraprocedural-ControlFlow, StrayGlance, TestHypothesis, Touchstone, Trial&Error, Wandering | **AttentionToDetail:** Readers are trying to comprehend a piece of code that is not believed to contain bugs. In most cases, there is a slowness to AttentionToDetail, but the subject could also be verifying a global property, such as that argument/ parameter types agree or that the semicolons are present in the right places.<br><br>**DataFlow**: Following a single object in memory as its value changes through the program. Can also occur backwards through control flow in service of debugging and/or program execution comprehension.<br><br>**Debugging**: Similar to DesignAtOnce, but more equally distributed fixation durations, and more equally distributed time of fixation for all text elements. Based on pattern LinearHorizontal and LinearVertical. The subject's intention is to find syntactical or semantic errors. Very small jumps, where the subject is presumably validating the syntax. (Note: Maybe debugging is more a goal, than a strategy.)<br><br>**Deductive**: From general to special, from definition to use, typically includes LinearHorizontal. | Interpretation |

| | | | |
|---|---|---|---|
| | | **DesignAtOnce**: LinearHorizontal or Scan, hardly any jumps back. The subject's intention is to understand the general or algorithmic idea, without having the need to go into details. Aiming at understanding by linear reading of the complete (needed) code. Can easily be confused with excessive demand/trial and error, might also include TestHypothesis on local levels. Captures high-level algorithmic thinking, thus features rather large steps as the gaze sweeps over the text 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.<br><br>**FlowCycle:** The same program flow sequence is followed several times, the intent might be to gain a first understanding of the flow, strengthening and reinforcing it with repeated examinations of the same code. The Flicking pattern might then suggest the simplest level of the FlowCycle strategy.<br><br>**Inductive**: From the special to general, from context to definition, typically combined strategy (mix of Scan, JumpControl and LinearHorizontal). | |

| | | | |
|---|---|---|---|
| | | **InterproceduralControl-Flow**: The subject follows call-chains in real or simulated sequence of control flow. Intention is to understand the execution or to get the outcome of a code section. Focus is on execution between blocks.<br><br>**IntraproceduralControl-Flow**: The subject scans lines of code in real or simulated program execution order. Intention is to understand the execution or to get the outcome of a code section. Focus is on execution on block level.<br><br>**StrayGlance:** A glance where something is looked at that does not necessarily involve comprehension or something that we cannot explain.<br><br>**TestHypothesis**: Repetition of a pattern or gaze path. Occurs in connection with DesignAtOnce or ControlFlow. The subject's intention is to check for some details in understanding. Hints at some issue where either the person was distracted, or which is more difficult to comprehend. Involves repetition of a pattern of gaze, and suggests further concentration in order to better understand a particular detail. | |

| | | | |
|---|---|---|---|
| | | **Touchstone**: Analogue to checking the risk of any deal. Before transporting some goods through an unknown route, first you go there without merchandise and see, where to turn and where the traffic lights are. And when you are sure about everything, you take the goods with you to finish the deal. Comparing this example with program comprehension, the route is the algorithm, while the goods are the parameters.<br><br>**Trial&Error**: Similar to Design-AtOnce, but with higher reading speed, and some irregular jumps and repetitions in reading. The subject's intention is to cope with cognitive overload and to try to find some place to start the understanding process. Connected to JustPassing-Through and Wandering.<br><br>**Wandering:** It appears that the subject was backtracking, seemingly searching for a point to resume the reading after a particular path of reasoning had been exhausted, essentially a transition period or a brief rest between bursts of effort. | |

# *List of participants*

| | **Name** | **Mail** |
|---|---|---|
| 1 | Antropova, Maria | maria.antropova@gmail.com |
| 2 | Bednarik, Roman | roman.bednarik@uef.fi |
| 3 | Begel, Andrew | andrew.begel@microsoft.com |
| 4 | Busjahn, Teresa | busjahn@inf.fu-berlin.de |
| 5 | Gavrilo, Katerina | katrinaalex@gmail.com |
| 6 | Hansen, Michael | mihansen@umail.iu.edu |
| 7 | Ihantola, Petri | petri@cs.hut.fi |
| 8 | Menzel, Suzanne | menzel@indiana.edu |
| 9 | Orlov, Paul | paul.a.orlov@gmail.com |
| 10 | Schulte, Carsten | arsten.schulte@fu-berlin.de |
| 11 | Sharif, Bonita | bsharif@ysu.edu |
| 12 | Shchekotova, Galina | intendia@gmail.com |
| 13 | Simon | simon@newcastle.edu.au |
| 14 | Vrzakova, Hana | vrzakova.hana@gmail.com |
| 15 | Wang, Peng | pwang@student.uef.fi |

Remote participants

**Roman Bednarik,**
**Teresa Busjahn,**
**Carsten Schulte (Eds.)**

# *Eye Movements in Programming Education: Analyzing the Expert's Gaze*


This is the proceedings of an international workshop on the emerging topic of eye movement data analysis in programming education. The first workshop edition focused on "Analyzing the expert's gaze". It was held in November 2013 at the School of Computing, University of Eastern Finland.

UNIVERSITY OF
EASTERN FINLAND

Publications of the University of Eastern Finland
*Reports and Books in Forestry and Natural Sciences*

ISBN: 978-952-61-1538-2 (nid.)
ISBN: 978-952-61-1539-9 (pdf)
ISSNL: 1798-5684
ISSN: 1798-5684
ISSN: 1798-5692 (pdf)