2015 IEEE 23rd International Conference on Program Comprehension

Eye Movements in Code Reading: Relaxing the Linear Order

Teresa Busjahn
Institute of Computer Science
Freie Universität Berlin, Germany
Email: busjahn@inf.fu-berlin.de

Roman Bednarik
School of Computing
University of Eastern Finland, Finland
Email: roman.bednarik@uef.fi

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

Martha Crosby
Information & Computer Sciences
University of Hawai'i at Mānoa, HI, USA
Email: crosby@hawaii.edu

James H. Paterson
Dept. of Computer, Communications & Interactive Systems
Glasgow Caledonian University, UK
Email: james.paterson@gcu.ac.uk

Carsten Schulte
Institute of Computer Science
Freie Universität Berlin, Germany
Email: carsten.schulte@fu-berlin.de

Bonita Sharif
Computer Science & Information Systems
Youngstown State University, OH, USA
Email: bsharif@ysu.edu

Sascha Tamm
Center for Applied Neuroscience
Freie Universität Berlin, Germany
Email: tamm@zedat.fu-berlin.de

***Abstract*—Code reading is an important skill in programming. Inspired by the linearity that people exhibit while natural language text reading, we designed local and global gaze-based measures to characterize linearity (left-to-right and top-to-bottom) in reading source code. Unlike natural language text, source code is executable and requires a specific reading approach. To validate these measures, we compared the eye movements of novice and expert programmers who were asked to read and comprehend short snippets of natural language text and Java programs.**

Our results show that novices read source code less linearly than natural language text. Moreover, experts read code less linearly than novices. These findings indicate that there are specific differences between reading natural language and source code, and suggest that non-linear reading skills increase with expertise. We discuss the implications for practitioners and educators.

I. INTRODUCTION

For over five thousand years, human history has been written down in a variety of forms which represent our cognitively rich natural spoken languages. Even when manifested in written form, the lexical, syntactic, semantic, and cultural complexities inherent in spoken communication make reading and writing difficult skills to learn and master.

For over fifty years, computer programs have been written in a variety of source code languages. While having no commonly spoken form, the design of source code languages is inspired by natural languages, each with its own formal style. Though computer programs are primarily thought of as being executed by a computer, their source code form must be understandable in order to be read and written by humans. To become an expert computer programmer, one must master source code reading and writing skills.

Fortunately, programmers can leverage their mastery of natural language literacy to learn to read and write source code (SC). However, programs differ from natural language texts (NT) in two important ways. First, programs are lexically and syntactically different from natural language texts. Lexically, they are composed from a limited vocabulary, with some words used as programming language keywords, and others used more freely as identifiers to name program constructs like variables and methods. Programmers must consciously choose meaningful names in order to communicate to others what the program does and how it works [17]. Syntactically, programs are laid out and organized differently than natural language texts. They feature greater use of formally defined structures and multiple forms of indented layout (both horizontal and vertical).

The second difference is semantic. According to the Kintsch text comprehension model [15], natural language text is typically understood in two concurrent phases: text (how it is written down) and domain (what it means). Source code comprehension however needs a third dimension of comprehension: execution [22]. Thus, in order to understand a program's goals, programmers must not only master the ability to read its words and structures, but must also be able to trace source code execution to discover its operational semantics [23]. Sorva [28] posits that expert programmers rarely make an explicit distinction between text structure and text execution because they feel it is obvious. This might not be so obvious for novices.

For the past two years, we have organized workshops focusing on the interpretation of eye tracking data for program comprehension [6]. We and others have come to the realization that due to the similarities between source code and natural language, many of the same processes must underlie source code reading [25]. In this paper, we illustrate how we

978-1-4673-8159-8/15 $31.00 © 2015 IEEE
DOI 10.1109/ICPC.2015.36

have adapted eye movement-based metrics used originally to characterize natural language reading to describe how novice and expert programmers read source code. Specifically, we expanded on the concept of linearity in reading, and developed and tested several linearity metrics.

Linearity represents how closely readers follow a text's natural reading order — the top-to-bottom, left-to-right path in which words are written in Latin script languages like English.[1] Proficient natural language readers are known to maintain comprehension even though they skip words as they read and reread words less frequently than novice readers (i.e., children learning to read) [20]. Based on the observation that novice programmers seem to read source code "just as one might read a story in a natural language, starting at the beginning and reading through to the end" [26], we introduce the term *story reading* for this reading behavior.

The research questions we address in this paper are

- RQ1: Do the linear trends observed in natural language reading hold for novices reading source code?
- RQ2: With respect to reading linearity, how do novices reading source code differ from experts?

Here, we operationalize linearity using two sets of metrics based on gaze. The set of *local metrics* include the fraction of gaze locations that move with or against story reading order (measured in elements and lines). Our set of *global metrics* is based on fixation sequences and quantifies how closely a participant follows a particular reading pattern. These metrics assess the participants' degrees of alignment to two reading orders: story reading and execution order.

We evaluate these metrics in a study that had novice and professional programmers perform computer-based comprehension tasks on short English texts and also on programs written in Java and pseudocode, while being monitored with an eye tracking device.

Our results show that novice participants read code less linearly than they read English text. 80% of the the novices' eye movements were linear when reading natural language text and 70% when reading source code. For experts on the other hand, we found only 60% linear eye movements on source code. The experts' reading patterns can be characterized by a greater number of eye gaze movements that skip intermediate words and lines.

The contributions of this paper include:

1) Operationalization of linearity metrics for reading source code
2) Validation of these metrics for natural language reading and source code
3) Characterization of differences in reading behaviors between novice and expert programmers
4) Automatic detection of story and execution order source code reading patterns

[1]While many natural languages follow other word orders, programming languages almost always follow the Latin script reading direction.

II. Background

We present a brief description of eye tracking research necessary to understand the measures presented later and discuss eye tracking studies on reading natural language text and program comprehension with a focus on the latter.

Many previous eye tracking studies conducted on natural text reading provide abundant insights into the visual attention behavior. During reading, the eye stays upon one location for a few hundred milliseconds and then moves to the next location. This relatively steady state between eye movements is called a *fixation*. The amount of time spent in the location is the *fixation duration*. The movements that re-position the eyes' focus over the text are *saccades*. Processing of visual information occurs only during fixations. Backward movements in the text are called *regressions*; about 10 to 15% of fixations are regressive. Good readers are characterized by few regressions and short fixations. Difficult texts usually induce longer fixations, short saccades and frequent regressions. Fixation duration has been shown to be positively correlated with cognitive effort. See Rayner et al. [19] for a detailed description on eye movements in reading.

In the programming domain, however, there have been only a few notable works to report on the nuances of gaze in code reading. One of the pioneering studies of the role of gaze in programming [9] suggests that viewing strategies when reading short, but complex algorithms differ from those of natural language. Crosby et al. [8] also found that novices do not use beacons as discussed by Brooks [3], whereas experts focus mainly on them. Uwano et al. [30] identify a pattern called *Scan* during which programmers read the entire code snippet to get an idea of what the program does. They state that 70% of lines in source code were seen in the first 30% of time spent. A replication of this experiment confirmed the finding [24].

Both Fan [10] and Busjahn et al. [5] conducted eye tracking studies on short source code programs. Fan showed that code scanning patterns of programmers are related to the way the programs were commented i.e., programmers could chunk larger code blocks if comments were present. However, the presence of comments did not improve the identification of beacons. Busjahn et al. noted a few basic differences between reading source code and natural language text, e.g. increased fixation durations and regression rates for code.

Hansen et al. [12] investigated factors that impact code comprehension. They studied 10 Python programs with subtle differences between them with the task of predicting the output. They conducted a standard questionnaire-based online version and an eye tracking version of the study. Results from the questionnaire-based study indicate that even subtle notation changes can have a large effect on the performance of programmers. The results of the eye tracking part of their study are pending.

Turner et al. [29] used an eye tracker to compare short C++ and Python programs in a between-subjects study. The 38 students were asked to find a logical error in the programs.

Students spent significantly different amounts of time looking at the lines with bugs than the other code lines when comparing their behaviors between programming languages. Novices had higher fixation rates on buggy lines in Python, whereas non-novices had higher fixation rates on buggy lines in C++. The authors call for more studies to determine other differences between languages.

Bednarik et al. [2] found that repetitive gaze patterns were associated with less expertise when students were debugging Java programs using a program visualization system. Rodeghero et al. [21] found that 10 developers looked more at a method's signature than its body when they were asked to summarize Java methods while being recorded by an eye tracker. Fritz et al. [11] conducted a study with 15 developers and used signals from eye tracking, electroencephalogram (EEG), and electrodermal activity (EDA) sensors to predict if developers found a task to be difficult. The task was to read short C# programs and answer multiple choice questions about what the program did. Their results show that eye tracking measures such as fixations, saccades, and pupil size were important to predict task difficulty.

Siegmund et al. [25] conducted a study with 17 programmers inside an fMRI scanner. The programmers were asked to comprehend short source code snippets. They find a distinct pattern active in five brain regions, all necessary for program comprehension and that language processing is an essential part of program comprehension.

In our most recent work [6], we provide an introduction on how to use eye tracking to study programmer behavior while reading code. A tiered coding scheme was developed to further understand existing program comprehension strategies [1], [3], [16], [27]. Both objective and subjective behaviors were coded. The scheme includes codes based on the location of a single fixation as well as codes characterizing sequences of multiple fixations, called patterns. For example, some patterns introduced were *Linear Scan* (where gaze moves linearly through parts of the code) and *Jump Control* (where gazes follows code execution order).

None of the studies mentioned discuss the concept of linearity and whether or not the linearity effect in reading natural languages transfers to reading of source code. We believe this is an important step taken for further research using eye tracking in program comprehension.

III. Study

In this section, we describe our study: participants, tasks, experimental procedures, measures, and threats to validity.

A. Participants

Our study looked at the similarities and differences in two populations: novice and expert programmers.

We conducted a longitudinal study with 14 novices, who attended a Java beginner's course at Freie Universität Berlin. Participants were primarily recruited via flyers posted on the university's bulletin boards. With the exception of two participants, all the others were university students (none in computer science). We provided a participation statement if requested (no grade was given). During the weekly Java class they individually worked through an online course,[2] while there was a tutor present to provide assistance. Students were allowed and encouraged to seek help from their classmates.

```
public class Vehicle{
 String producer, type;
 int topSpeed, currentSpeed;
 public Vehicle(String p, String t, int tp){
   this.producer = p;
   this.type = t;
   this.topSpeed = tp;
   this.currentSpeed = 0;
 }
 public int accelerate(int kmh){
   if ((this.currentSpeed + kmh) > this.topSpeed){
      this.currentSpeed = this.topSpeed;
   } else {
     this.currentSpeed = this.currentSpeed + kmh;
   }
     return this.currentSpeed;
 }
 public static void main (String args[ ]){
   Vehicle v = new Vehicle("Audi","A6",200);
   v.accelerate(10);
 }
}
```

LISTING 1. Sample program presented to both novices and experts

The course consisted of six modules, each spanning several weeks. These modules covered objects, classes, fundamental data types, decisions, and loops. The students' eye movements were recorded after they finished each module. Several participants dropped the course eventually, leading to a sparse dataset in later modules.

We also recruited a group of six professional software engineers who worked at different software companies in Berlin. The participants were promised that all data and analyses about their participation would be anonymized and kept confidential from their employers. No one was remunerated for participating.

All study participants filled out a demographic questionnaire about their age, gender, proficiency with English, and their programming skills and experience. The novices, who were between 19 and 33 years old, included 7 females. The experts were between 26 and 49 years old and only included one female. While everyone had at least medium English proficiency, German was their primary language (except one novice who spoke French). The novices self-reported having little to no prior programming experience, experts' programming experience ranged from 5 to 28 years and all were proficient in several programming languages including Java. At the time of the recording, all were employed as professional programmers.

B. Experimental Procedure and Materials

The expert participants were studied individually at their offices or in another location of their choice. The novice

[2]http://www.udacity.com/course/cs046

participants were recorded after finishing each of the six modules directly in the classroom.

Each participant was set up with an SMI RED-m remote eye tracker set to sample at 120 Hz. All of the eye tracking data was recorded by the open-source tool Ogama (OpenGaze-AndMouseAnalyzer).[3] Once a participant ran the eye tracker's calibration routine, he or she was ready to begin. Before the first recording on SC, novices were asked to read three English language passages of four to five lines. Each participant was given the texts in a randomly selected order. After finishing the text in each trial, they had to answer a comprehension question.

Each module's recording session consisted of asking the novice participants to read a set of three programs ranging from a few lines to an entire screen full of text. Two of these programs were in English pseudocode, the others were complete Java classes. When required, the source code also included specific inputs to determine an execution order.

Experts looked at six programs in total, two of which were the same ones that novices saw in the latter weeks of the study. An example of one of these programs can be seen in Listing 1. The remaining programs were comparable in length to the novices' stimulus material, however, they were more complex and included concepts the beginner's course did not cover, in order to prompt experts to actually use their advanced programming skills.

Immediately after reading each program, the participants were asked one of three possible questions: (1) write a summary of the code, (2) write the value of a variable after program execution, or (3) answer a multiple-choice question about the algorithmic idea. The order of programs and tasks given to the participant were both shuffled for each trial to avoid any bias caused by coupling a certain program to a specific question.

In total, we recorded 17 trials of novices reading natural language texts, and 101 trials of novices reading source code. 35 of these 101 source code trials occurred just after participants finished their first module. The remaining 66 trials were conducted as the novices finished their next five modules.

Some experts did not finish reading the Java programs during their recording session. In total, we were able to record 21 trials. When answering RQ2, we compare these 21 trials against the 101 novice trials.

C. Measures

There are two independent variables in our experiment: whether the participant is from the novice or expert programmer sample population and whether they read a natural language text or source code.

We developed nine dependent measures (see Table I) and used them to analyze data records from an eye tracking device. These consist of a time-ordered sequence of gaze fixations, each containing a gaze location (combining both eyes to produce a single (X, Y) screen coordinate) and the fixation's duration (in milliseconds). The gaze coordinates are then mapped to line and word positions in the texts read by each participant using a tool called EyeCode.[4]

Participants do not always fixate on words on the screen; sometimes they look at empty space nearby. We were able to map 90.51% of the natural language fixations to a line of text. For the source code trials, we could map 82.88% of the novices' fixations and 72.72% of the experts' fixations to lines in source code.

The first five of our measures are computed from the fixated line and word positions. They represent the fraction of gaze records (per trial) where the participant moved their eyes with or against the linear reading order.

The next measure, **Saccade Length**, is the average distance between the participant's two consecutive fixations in each trial. It is commonly found in prior eye tracking studies that experts are more likely to make longer saccades than novices.

Element Coverage measures the percentage of words (i.e., source code elements) in the text that the participant looked at. Experts are known to be better able to focus on fewer, more relevant, words in the source code than novices can.

The last two measures indicate to what extent the participants' gaze followed the models **Story Order** and **Execution Order**. Story Order represents reading the program line by line, from top to bottom, the way natural-language text is generally read. Execution Order stands for reading the lines according to control flow. In order to compare how well the participants' actual gaze path matches these models, we employ an optimal string matching algorithm called Needleman-Wunsch (N-W), which has been applied to sequences of eye movements before by Cristino et al. [7]. This string algorithm computes a global similarity score, where a high score implies that the two sequences are close to each other. For instance, a person's score of 10 asserts a higher accordance with the order of lines specified by the model than a score of 1.

The N-W algorithm is often applied to compute the similarity of DNA and protein sequences, where both sequences possibly contain mutations. However, in our settings only one sequence can have mutations, namely the gaze, while the model presents the prototype. Subsequently, the goal is to determine how far the gaze is from the respective model. Mutations in the model are less acceptable than mutations in the gaze, so we need to keep the model in one piece as much as we can. Thus, we penalize a gap in the model more than a gap in the gaze sequence. Gaps will be preferentially inserted into the gaze sequence rather than the model. To operationalize this, we designed the following scoring scheme: +3 for a match, -3 for a mismatch, -1 for a gap in the participant's gaze, and -2 for a gap in the model.

The approach to compare the gaze to the exact model is however slightly naïve, since texts are usually read more than once. Thus, we adapted the alignment measure to identify where the participant reread the text in order to find the optimal alignment between model and gaze. We realize this

[3]http://www.ogama.net

[4]https://github.com/synesthesiam/eyecode

TABLE I

GAZE-BASED MEASURES. IN EACH TRIAL, F IS THE SET OF ALL RECORDED FIXATIONS. F_i (WHERE $i = \{1, \ldots, n\}$) IS THE FIXATION RECORDED AT TIME INDEX i. $L(F_i)$ IS THE LINE NUMBER OF THE FIXATION AT INDEX i. IN EACH TRIAL, W IS THE SET OF WORD INDICES IN THE TEXT. $W(F_i)$ IS THE WORD NUMBER OF THE FIXATION AT INDEX i.

	Measure	Definition	Computation
local	Vertical Next Text	% of forward saccades that either stay on the same line or move one line down.	% of all F_i, where $L(F_i) - L(F_{i+1}) = \{0, -1\}$
	Vertical Later Text	% of forward saccades that either stay on the same line or move down any number of lines.	% of all F_i, where $L(F_i) \leq L(F_{i+1})$
	Horizontal Later Text	% of forward saccades within a line.	% of all F_i, where $L(F_i) = L(F_{i+1}) \wedge W(F_i) \leq W(F_{i+1})$
	Regression Rate	% of backward saccades of any length.	% of all F_i, where $W(F_i) > W(F_{i+1})$
	Line Regression Rate	% of backward saccades within a line.	% of all F_i, where $L(F_i) = L(F_{i+1}) \wedge W(F_i) > W(F_{i+1})$
	Saccade Length	Average Euclidean distance between every successive pair of fixations.	$\frac{\sum_{i=1}^{n-1} \text{Distance}(F_i, F_{i+1})}{\lvert F \rvert - 1}$
	Element Coverage	Fraction of words the participant looked at.	% of W for $\frac{\lvert \text{Unique}(W(F_i)) \rvert}{\lvert W \rvert}$
global	Story Order	N-W alignment score of fixation order with linear text reading order.	$\text{Alignment}(L(F), \text{Story-Order-Pattern})$
	Execution Order	N-W alignment score of fixation order with the program's control flow order.	$\text{Alignment}(L(F), \text{Execution-Order-Pattern})$

by repeating the model and aligning this extended model sequence to the gaze. This dynamic approach finds the optimal number of repeats by iteratively appending a copy of the model and computing the similarity score until the lengths of the gaze and model sequences correspond with one another. In the end, the algorithm returns the alignment with the highest score together with the number of model repetitions used to achieve it. Each model *instance* in this final sequence denotes one pass through the text.

Table II illustrates the procedure for Story Order (Line 1, 2, 3, 4) and Execution Order (Line 1, 2, 3, 4, 2, 3, 4, 2) with the sample code in Listing 2 and a participant's actual gaze sequence (Line 1, 2, 3, 1, 2, 3, 2, 3, 2, 1, 2, 1, 3, 4, 3, 2). Repeated fixations within the same line have been removed.

When testing how close the participant's gaze is to Story Order (i.e. Line 1, 2, 3, 4), we get a *naïve* score of only -12. We can improve the alignment if we also assess how many times the participant read through the program. If the participant read the text 4 times, then we would get a *dynamic* score of 24. Looking at Execution Order, the naïve score is 2, while the dynamic score is 24. The participant followed the control flow twice. Looking at the exact models, this participant read the program more according to control flow than line-wise from top to bottom. When comparing the gaze to multiple instances of the model, both models achieve the same score, however with 4 repetitions for the shorter Story model and only 2 for Execution. Since the sequences are of comparable length now, there are less gaps reducing the score.

```
1   n = 3
2   while (n > 1):
3     print n
4     n = n - 1
```

LISTING 2. Code with Story Order: Line 1, 2, 3, 4 and Execution Order: Line 1, 2, 3, 4, 2, 3, 4, 2

D. Threats to Validity

As mentioned before, some novice participants dropped out of the course and did not finish all of the trials. This does not affect RQ1 (comparing novices reading natural language vs. source code in module 1), as we are only interested in the early novice.

When considering RQ2, we pool together novice trials across several modules, so some trials represent novices with up to 10 hours more coding experience than others. We highlight this explicitly, however we always compare novices only to the pool of expert programmers who have orders of magnitude greater and more frequent coding experiences. While the pooled data does not allow for fine-grained analyses of the novices' progress, it covers a broad range of novice behavior. According to a multinational study [18], student programmers show difficulty comprehending small programs like those we use even after a whole year of CS study.

In order to reduce bias by the stimulus programs, both sets of stimulus programs were designed to cover a range of concepts and varied in facets like identifier naming conventions.

TABLE II
SAMPLE ALIGNMENTS FOR LISTING 2 (EVERY SECOND MODEL REPETITION IS HIGHLIGHTED).

	Approach	Alignment	Score	Repetitions
Story Order	Naïve	model: – – – – – – – – – – 1 2 – 3 4 – – gaze: 1 2 3 1 2 3 2 3 2 1 2 1 3 4 3 2	-12	1
	Dynamic	model: 1 2 3 4 **1 2 3 4** 1 2 3 4 **1 2** – **3 4** – – gaze: 1 2 3 – 1 2 3 – – 2 3 2 1 2 1 3 4 3 2	24	4
Execution Order	Naïve	model: – – – 1 – – 2 3 – 4 2 – 3 4 – 2 gaze: 1 2 3 1 2 3 2 3 2 1 2 1 3 4 3 2	2	1
	Dynamic	model: 1 2 3 4 2 3 – 4 2 **1 2** – **3 4 2 3 4 2** gaze: 1 2 3 1 2 3 2 3 2 1 2 1 3 4 – 3 – 2	24	2

When mapping the gaze coordinates and the on-screen texts, we found that despite calibrating the eye tracker to each participant prior to every task, the recorded gaze location often appeared too low or too high. In much rarer cases, it was also shifted to the left or right. This kind of error can happen when the participant moves his or her head too far relative to the screen. We needed to correct these errors, but were wary of introducing additional bias. The authors worked together in pairs to collaboratively produce a corrected version for every trial. When there were disagreements, they rechecked the scanpaths and arrived at a consensus to determine the final corrected offset.

The first five measures can be computed directly from the screen coordinates, without requiring any corrections to the correspondence map. To further increase validity, we also filtered out fixations which were clearly outside the bounding box of the text (over 100 pixels outside) as unmapped. We compared our measures computed on this uncorrected data to the same measures computed on the corrected data and found that 98–100% of the measures were in agreement.

In our study, we operationalize reading linearity with seven fixation-based measures and two global measures. The linearity of natural language reading can be measured using just the regression rate [13]. However, this single measure misses relevant aspects of source code reading. For example, source code texts contain shorter horizontal passages than typical natural language texts, sometimes having just one source code element on a line. Our mix of horizontal and vertical measures provides a more detailed description of the reading patterns.

IV. RESULTS

A. Novices: Natural Language versus Source Code

We first report on the linearity measures for novices reading natural language text and source code at the beginning of their Java course (Module 1). We expected that since novices lack any specialized reading strategies for source code, they would make use of their usual linear reading approach that they apply to reading natural language. We should therefore see that the linearity measures are similar in the two conditions.

Considering that the participants were non-native English speakers, the regression rate of 15.62% for NT is well in accordance with the 10 to 15% reported for English text [19].

Notice in Figure 1 that the measures Vertical Next Text, Vertical Later Text, and Horizontal Later Text are higher for natural language text than for source code, since they indicate linearity. Regression Rate and Line Regression Rate, on the other hand describe non-linear reading. Consequently they show a lower rate.

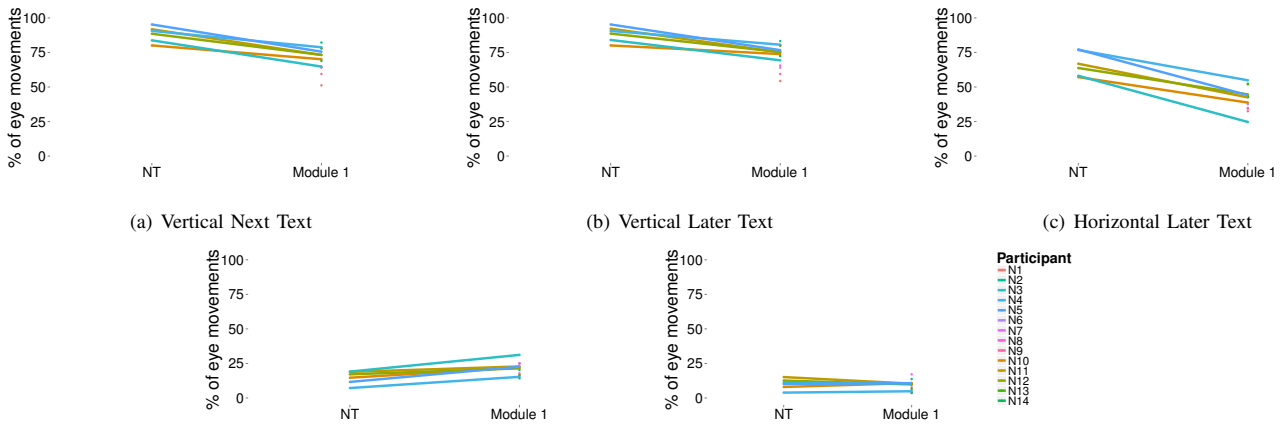
The novices followed the linear Story Order on natural language texts with approximately 80% of their eye movements (computed as the non-regressive portion of saccades), and Module 1 source code still with 75%.

Due to the sparseness of the data set, tests for normality are not appropriate, hence we employed the non-parametric Wilcoxon Signed Rank test for matched samples. We found that for the first four measures shown in Figure 1, Vertical Next Text, Vertical Later Text, Horizontal Later Text, and Regression Rate the differences are statistically significant (see Table III). Only Line Regression Rate was comparable for the two types of text. The test statistic W provides the sum of the positive ranks. The values of 21 and 0 respectively indicate that the direction of the differences was persistent for all pairs.

TABLE III
WILCOXON SIGNED RANK TEST FOR NATURAL LANGUAGE VS. SOURCE CODE (MODULE 1)

Vertical Next Text:	$W(6) = 21, Z = 2.20,$	$p = 0.03^*$
Vertical Later Text:	$W(6) = 21, Z = 2.20,$	$p = 0.03^*$
Horizontal Later Text:	$W(6) = 21, Z = 2.20,$	$p = 0.03^*$
Regression Rate:	$W(6) = 0,\ Z = -2.20,$	$p = 0.03^*$
Line Regression Rate:	$W(6) = 13, Z = 0.52,$	$p = 0.69$

Furthermore, we compared the average Saccade Length for natural language and the Java source code in Module 1. Novices have a significantly higher average saccade length



(a) Vertical Next Text (b) Vertical Later Text (c) Horizontal Later Text

(d) Regression Rate (e) Line Regression Rate

Fig. 1. Linearity measures for novices.

(2.28° of visual angle (98.13 pixels)) when reading natural language text than when reading source code (Module 1) (1.68° of visual angle (71.44 pixels)): $W(6) = 21, Z = 2.20, p = 0.03^*$.

When comparing Element Coverage, novices looked at an average of 83.46% of the words in the natural language texts, but only at 60.38% of the source code elements (i.e. keywords and identifiers) in the Java code for Module 1. The difference is statistically significant: $W(6) = 21, Z = 2.20, p = 0.03^*$.

TABLE IV
N-W RESULTS COMPARING THE STORY ORDER FOR NATURAL LANGUAGE TEXTS AND SOURCE CODE (MODULE 1).

		Natural Language	Source Code (Module 1)
Story Order	Naïve N-W Score	-29.82	-24.11
	Dynamic N-W Score	25.88	23.71
	Repetitions	6.35	3.89

Finally, we tested the alignment of reading natural language and source code (Module 1) to Story Order (see Table IV). We present two N-W global alignment scores. The first (naïve) compares the participant's line order with the story order sequence. The second (dynamic) allows the story order sequence to repeat a number of times. The participant's line order was compared with the story order sequences repeated an increasing number of times until it maximized the N-W score for the participant's fixations in that trial. This accounts for the fact that the text is read several times. Note that the scores for natural language and the source code in Module 1 are very similar to one another. There is no statistically significant difference between natural language and the source code in Module 1, not for the (naïve) score ($W(6) = 9.5, Z = -2.20, p = 0.91$), nor the dynamic score ($W(6) = 12, Z = 0.31, p = 0.84$). This supports our expectation that novices start out with a primarily linear approach to reading code. The results indicate that both the natural language text and source code were read multiple times, but participants read the natural language text 6.35 times vs. 3.89 times for source code. The greater the number of repeated read-through, the higher the N-W alignment score will be.

RQ1 asked if the story reading approach holds true for novices reading source code (Module 1). The N-W score, reflecting the general reading approach aligned to the linear Story Order, and Line Regression Rate are comparable for natural language and source code (Module 1). However, we find significant differences for the local measures Vertical Next Text, Vertical Later Text, Horizontal Later Text, Regression Rate, Saccade Length, and Element Coverage.

B. Novices vs. Experts

In this section, we report on the linearity measures that compare experts reading source code to novices (at any stage in the Java course) reading source code. We combined all of the novices' trials from every module in the class, and not just module 1 as we did for RQ1. We expect that since the novices rely on their natural language text reading behaviors when reading source code, there will be significant differences between them and the experts. We note that the novices and experts read different programs (for the most part; two programs were read by both sets of participants). Since the novices were enrolled in the Java class, their programs were tailored to be accessible to their (considerably lower) experience level. In this section, we report statistics that compare the novices to the experts reading their respective programs.

In Figure 2, we show the results of our first five measures comparing the pool of novices to the pool of experts (Figure 2). The measures for linearity Vertical Next Text, Vertical Later Text, and Horizontal Later Text are higher for novices than for experts. The same holds true when computing the non-regressive portion of the saccades, 70% linear eye movements for novices and 60% for experts. Since normality cannot

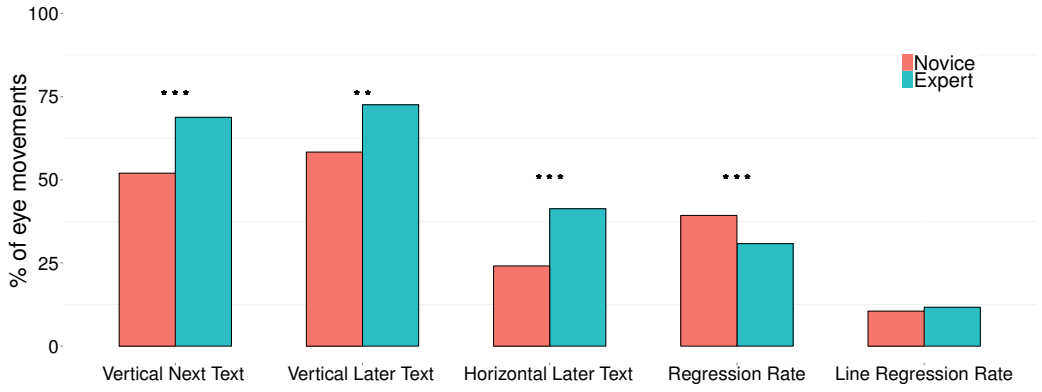


Fig. 2. Comparison of linearity measures for novices and experts reveals more linear behavior in the eye movements.

be assumed and we have two different samples, we tested these differences with the non-parametric Mann-Whitney test. We found that the differences are statistically significant for all measures except Line Regression Rate (see Table V), indicating that experts have a less linear reading behavior for code than the novices.

Next, we calculated the average Saccade Length for all modules and source codes. Novices' saccades had an average length of $1.86°$ of visual angle (80.26 pixels), while experts' saccade covered $3.12°$ of visual angle (133.90 pixels). A Mann-Whitney test indicates that average novices' saccades were significantly shorter than experts': $U = 105, p < 0.001^{***}$. This result matches our expectations that experts are able to make larger jumps during reading to focus on important source code elements.

With regard to Element Coverage, we found that novices directly looked at 52.42% of the source code elements, while experts looked at just 41.27% of the elements. Again, the difference is significant, $U = 84, p < 0.01^{**}$. This result implies that experts are better able to focus on the relevant source code elements than novices, and is in line with previous studies on expertise.

Finally, we compared the alignment of the novices' and expert reading order. The results are shown in Table VI. One can see that novices read source code much more linearly (i.e., in Story Order) than the experts did because the novices' higher scores indicate better alignment with the Story Order model. This difference is significant both for naïve and dynamic scores: $U = 82, p < .001^{***}$ and $U = 84, p < .0001^{***}$, respectively.

TABLE V
MANN-WHITNEY RESULTS FOR NOVICES VS. EXPERTS

Vertical Next Text:	$U = 81,$	$p < 0.001^{***}$
Vertical Later Text:	$U = 76,$	$p < 0.01^{**}$
Horizontal Later Text:	$U = 84,$	$p < 0.001^{***}$
Regression Rate:	$U = 0,$	$p < 0.001^{***}$
Line Regression Rate:	$U = 41,$	$p = 0.97$

TABLE VI
N-W RESULTS COMPARING NOVICES AND EXPERTS IN STORY AND EXECUTION ORDER.

		Novices	Experts
Story Order	Naïve N-W Score	-73.47	-161.52
	Dynamic N-W Score	-12.82	-113.62
	Repetitions	5.36	3.95
Execution Order	Naïve N-W Score	-77.27	-89.81
	Dynamic N-W Score	8.68	-46.50
	Repetitions	6.69	2.56

The low, negative Story Order global alignment scores for experts indicates that Story Order is not a good model for their reading behavior. Instead, notice that the Execution Order scores are higher, indicating a better fit with an order matching the control flow of the program.

Also, one can see that the number of repetitions of both the text and execution order line number sequences is much lower for experts than for novices. This indicates that the experts were more efficient in their reading of the source code than novices, agreeing with the Element Coverage finding above.

Somewhat counter-intuitively, experts have lower similarity scores for Execution Order than novices. This is due to the linear nature of the programs that the novices read. Conditions and loops were only introduced in later modules, therefore the execution orders of the programs were mainly linear. Comparing Story and Execution order with N-W illustrates this effect, with an average similarity score of -1 for novice programs and -71 for expert programs. Novices tend to read linearly and the execution order of their programs is rather linear, hence they get higher similarity scores for execution order than experts.

RQ2 asks how experts differ from novices in linear reading. For the measures representing the fraction of eye movements where the participant moved their eyes with or against the linear reading order, we find that except for the Line Regression Rate, experts exhibit significantly different reading behaviors

than novices. The same applies to Element Coverage, Saccade Length and the global alignment of the reading pattern to the linear Story Order. This result matches our expectations that the experts were better at reading code than the novices, but the additional details illustrated by the linearity measures we introduced in this study clarify much more precisely how these differences were composed.

V. Discussion

Programs are a particularly difficult form of text that requires problem solving skills. Often the viewing of algorithms appears closer to reading mathematical formulas or interpreting graphs than to reading a "story" form of text. Unlike natural language text, algorithms contain a wealth of information condensed within non-redundant text.

The difficulty of reading natural language text depends on the participant and the context. In many eye tracking studies, the majority of significant effects are often the result of participant variability. Studies from many domains such as reading and problem-solving, have shown that eye movements of experts differ from those of novices. For example, Kennedy and Murray [14] gave evidence that poor readers and good readers show different reading strategies for varying difficulties of text and that fixation time was longer for difficult text.

The research questions we addressed in this paper were

- RQ1: Do the linear trends observed in natural language reading hold for novices reading source code?
- RQ2: With respect to reading linearity, how do novices reading source code differ from experts?

Our results show that novice participants follow the linear Story Order on natural language texts with approximately 80% of their eye movements and on source code with 70%. While there are differences between reading natural language and source code, there is still a fairly strong linear character to novice source code reading. The non-linear portion might be caused by novices moving their gaze around in search for comprehension cues in the unfamiliar type of text. Expert programmers showed 60% linear eye movements, they adapt their natural language reading strategies when reading source code. We believe that experts are applying their knowledge of program execution to aid in comprehension.

The Execution Order model better explains the experts' reading approach than Story Order does. This suggests that experts tend to trace at least parts of the code. For novices on the other hand, none of the two models is predominant.

VI. Implications

The results presented here have implications both for practitioners and for programming educators.

A. Implications of findings for practitioners

The significance of this research for industry lies in the findings on the code reading behavior of the experts and in the development of measures which may provide the basis of diagnostic tools to improve the daily practices of programmers.

We find evidence that experts have more advanced skills in adapting their natural language reading strategies for computer programs. We find that they take advantage of non-linear reading orders and specific code reading strategies (e.g. code tracing). We feel that this insight could lead to improved models of program comprehension. In addition, tools that incorporate the measures we have introduced in this paper could provide the means to automatically recognize developer reading patterns and use these to infer comprehension strategies. Such tools could be deployed as diagnostic aids built into the programming environment to detect patterns which may indicate understanding, confusion or frustration. We envisage that subtle recommendation cues can be integrated into the modern IDEs to guide the attention of programmers and recommend certain actions without being excessively prescriptive or critical.

Code reading is obviously related to the styles used in code writing, both of which are dependent of the design of the programming language and libraries used by developers. Our algorithms and models could be used as diagnostics to evaluate the effect of coding standards and practices on the readability and understandability of source code [4].

B. Implications of findings for educators

Code reading skills are important for programmers and contribute to their ability to solve problems. Thus, the development of expert reading behavior should be an important goal of computer science curricula.

Learning to program is challenging, causing many students to experience difficulty carrying out basic programming tasks even at the end of a first programming course. Lister et al. [18] suggest that this is often a consequence not of poor problem-solving skills, as is commonly thought, but of a lack of knowledge and skills relating more to reading code than writing it. These elements are precursors to problem-solving. However there is little explicit focus on reading skills in programming textbooks and academic and industrial courses. Our findings show a differences in reading strategies between novices and expert programmers that could help guide curriculum design and teaching practice to explicitly support the development of reading expertise.

Unlike natural language text, programs have a duality consisting of the code and the dynamics of what happens when the code runs. Expert programmers rarely make this duality explicit because it is obvious to them, but it is not necessarily obvious to the novice [28]. Linear reading suggests the application of a mental model that does not include the dynamic side of this duality. Teachers and novice programming environments should therefore make explicit the ways in which machine behavior differs from human thought, and that program code is designed primarily to affect machine behavior. An early focus on program dynamics will support development of a valid mental model of what happens during program execution, or of the notional machine, an idealized abstraction that serves the purpose of understanding this. Teachers should be prepared to make use of metaphors and visualizations that

transfer program dynamics from the abstract to the concrete, and to use tools and development environments that provide program visualizations. Code reading and tracing skills should be explicitly taught and practiced in addition to code writing.

Eye tracking integrated into an IDEs or visualization tools can provide feedback to the student or the teacher. For instance, informing that the initial Scan of the program was not sufficient or that someone solving a bug finding task is concentrating on a code entirely unrelated to the bug, would be types of interventions that may promote early expert-like behaviors.

Code reading is, of course, not exactly the same as code tracing. While it can be useful in program comprehension to emulate the execution of code, humans can, once expertise has been gained, apply a range of strategies for assimilating programs. In an object-oriented program, for example, we can understand some things about the purpose of a class quite quickly by looking at the overall structure before tracing in detail what happens as particular methods execute. In contrast, the computer can only find the entry point for execution and proceed from there to execute code in class constructors and methods as they are called. Experts are able to recognize beacons that are typical indicators of the programs functionality [8], and patterns in code that typify, for example, the implementation of specific algorithms or class relationships. We usually do not teach comprehension of algorithms in the same way we teach reading comprehension, but such an approach should be built into CS curricula. Teachers should make the importance of beacons and patterns explicit at appropriate points in the development of program knowledge and give practice in identifying examples. Further study of the way in which experts make use of these may provide guidance in this.

At the same time we need to be realistic about what degree of expertise can be developed within a first programming course. The reading behavior of our novices even at the end of the course is not "expert" in nature. Expertise in programming will only develop over an extended period of study and sustained practice. However, our findings suggest that embedding strategies for code reading in the curriculum could provide a strong foundation for the development of expertise.

It has recently been shown that one of the effective ways to improve skill acquisition is to cue visual attention of novices to the locations that experts attend while performing a task [31]. If such intervention should be adopted in teaching program comprehension skills, knowledge about the expert behavior and significant differences between expert and novice programmers is very helpful, so that interventions can concentrate on these differences.

Instructors could use tools like these to investigate the effectiveness of new pedagogies. Novice programmers could monitor their own progress and judge whether they have achieved personalized learning goals. Additional applications are discussed in more detail in our prior work [6].

VII. Conclusions

Our results affect future research on the use of eye tracking to understand how source code reading occurs in various theoretical, behavioral, and practical ways. In applying a range of measures to eye tracking data for novice and expert programmers reading both natural language text and source code, our study has been able to determine the degree of linearity in both groups' reading strategies. We devised two distinct types of measures. Our local measures are based on the relationship between pairs of consecutive fixations. Our global measures are based on matching sequences of line fixations to model sequences based on idealized story and execution reading orders. Taken together, our measures give a more comprehensive picture of linearity within programmers' reading strategies.

References

[1] V. R. Basili and H. D. Mills, "Understanding and documenting programs," *IEEE Transactions on Software Engineering*, vol. 18, pp. 270–283, 1982.

[2] R. Bednarik, "Expertise-dependent visual attention strategies develop over time during debugging with multiple code representations," *International Journal of Human-Computer Studies*, vol. 70, no. 2, pp. 143–155, Feb. 2012.

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

[4] R. P. L. Buse and W. R. Weimer, "Learning a metric for code readability," *IEEE Transactions on Software Engineering*, vol. 36, no. 4, pp. 546–558, Jul. 2010.

[5] T. Busjahn, C. Schulte, and A. Busjahn, "Analysis of code reading to gain more insight in program comprehension," Koli, Finland: ACM, 2011, pp. 1–9. DOI: 10.1145/2094131.2094133. [Online]. Available: http://doi.acm.org/10.1145/2094131.2094133.

[6] T. Busjahn, C. Schulte, B. Sharif, Simon, A. Begel, M. Hansen, R. Bednarik, P. Orlov, P. Ihantola, G. Shchekotova, and M. Antropova, "Eye tracking in computing education," in *Proceedings of the Tenth Annual Conference on International Computing Education Research*, Glasgow, Scotland, United Kingdom: ACM, 2014, pp. 3–10.

[7] F. Cristino, S. Mathôt, J. Theeuwes, and I. D. Gilchrist, "ScanMatch: a novel method for comparing fixation sequences," *Behavior Research Methods*, vol. 42, no. 3, pp. 692–700, 2010.

[8] M. E. Crosby, J. Scholtz, and S. Wiedenbeck, "The roles beacons play in comprehension for novice and expert programmers," in *14th Workshop of the Psychology of Programming Interest Group*, 2002, pp. 58–73.

[9] M. E. Crosby and J. Stelovsky, "How do we read algorithms? a case study," *Computer*, vol. 23, no. 1, pp. 25–35, 1990.

[10] Q. Fan, "The effects of beacons, comments, and tasks on program comprehension process in software maintenance," PhD thesis, University of Maryland at Baltimore County, Catonsville, MD, USA, 2010.

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

[12] M. E. Hansen, R. L. Goldstone, and A. Lumsdaine, "What makes code hard to understand?" *CoRR*, vol. abs/1304.5257, 2013.

[13] J. Z. Jacobson and P. Dodwell, "Saccadic eye movements during reading," *Brain and Language*, vol. 8, no. 3, pp. 303–314, 1979.

[14] A. Kennedy and W. Murray, "The components of reading time: eye movement patterns of good and poor readers," in *Eye Movements: From Physiology to Cognition*, New York, NY, USA: ACM, 1987, pp. 509–529.

[15] W. Kintsch, *Comprehension: A Paradigm for Cognition*. Cambridge University Press, 1998.

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

[17] B. Liblit, A. Begel, and E. Sweetser, "Cognitive perspectives on the role of naming in computer programs," in *Proceedings of the 18th Annual Psychology of Programming Workshop*, Brighton, England, United Kingdom, Sep. 2006.

[18] R. Lister, E. S. Adams, S. Fitzgerald, W. Fone, J. Hamer, M. Lindholm, R. McCartney, J. E. Moström, K. Sanders, O. Seppälä, B. Simon, and L. Thomas, "A multi-national study of reading and tracing skills in novice programmers," in *Working Group Reports from ITiCSE on Innovation and Technology in Computer Science Education*, ser. ITiCSE-WGR '04, Leeds, United Kingdom: ACM, 2004, pp. 119–150.

[19] K. Rayner, "Eye movements in reading and information processing: 20 years of research.," *Psychological bulletin*, vol. 124, no. 3, p. 372, 1998.

[20] K. Rayner, S. P. Ardoin, and K. S. Binder, "Children's eye movements in reading: a commentary.," *School Psychology Review*, vol. 42, no. 2, pp. 223–233, 2013.

[21] P. Rodeghero, C. McMillan, P. W. McBurney, N. Bosch, and S. D'Mello, "Improving automated source code summarization via an eye-tracking study of programmers," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014, Hyderabad, India: ACM, 2014, pp. 390–401.

[22] C. Schulte, "Block model: an educational model of program comprehension as a tool for a scholarly approach to teaching," in *Proceedings of the Fourth International Workshop on Computing Education Research*, ser. ICER '08, Sydney, Australia: ACM, 2008, pp. 149–160.

[23] C. Schulte, T. Clear, A. Taherkhani, T. Busjahn, and J. H. Paterson, "An introduction to program comprehension for computer science educators," in *Proceedings of the 2010 ITiCSE Working Group Reports*, ser. ITiCSE-WGR '10, Ankara, Turkey: ACM, 2010, pp. 65–86.

[24] B. Sharif, M. Falcone, and J. Maletic, "An eye-tracking study on the role of scan time in finding source code defects," in *Proc. of the Symposium on Eye Tracking Research & Applications*, Santa Barbara, CA: ACM, 2012, pp. 381–384.

[25] J. Siegmund, C. Kästner, S. Apel, C. Parnin, A. Bethmann, T. Leich, G. Saake, and A. Brechmann, "Understanding understanding source code with functional magnetic resonance imaging," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014, Hyderabad, India: ACM, 2014, pp. 378–389.

[26] Simon, "Eye movements in programming education 2: analysing the novice's gaze," in *Eye Movements in Programming Education II: Analyzing the Novice's Gaze*, T. Busjahn, C. Schulte, S. Tamm, and R. Bednarik, Eds., Berlin, Germany, 2015, pp. 31–33.

[27] E. Soloway, J. Pinto, S. Letovsky, D. Littman, and R. Lampert, "Designing documentation to compensate for delocalized plans," *Communications of the ACM*, vol. 31, pp. 1259–1267, 11 1988.

[28] J. Sorva, "Notional machines and introductory programming education," *ACM Transactions on Computing Education (TOCE)*, vol. 13, no. 2, p. 8, 2013.

[29] R. Turner, M. Falcone, B. Sharif, and A. Lazar, "An eye-tracking study assessing the comprehension of C++ and Python source code," in *Proc. of the Symposium on Eye Tracking Research & Applications*, Safety Harbor, Florida: ACM, 2014, pp. 231–234.

[30] H. Uwano, M. Nakamura, A. Monden, and K.-i. Matsumoto, "Analyzing individual performance of source code review using reviewers' eye movement," in *Proc. of the Symposium on Eye Tracking Research & Applications*, San Diego, California: ACM, 2006, pp. 133–140.

[31] M. R. Wilson, S. J. Vine, E. Bright, R. S. Masters, D. Defriend, and J. S. McGrath, "Gaze training enhances laparoscopic technical skill acquisition and multi-tasking performance: a randomized, controlled study," *Surgical Endoscopy*, vol. 25, no. 12, pp. 3731–3739, 2011.