UNIVERSITY OF JOENSUU

DEPARTMENT OF COMPUTER SCIENCE

Roman Bednarik

Towards an Eye-tracking Methodology for Studies of Programming

Licentiate thesis

UNIVERSITY OF JOENSUU

2006

Supervisors	Markku Tukiainen Department of Computer Science University of Joensuu Joensuu, FINLAND
	Erkki Sutinen Department of Computer Science University of Joensuu Joensuu, FINLAND
Reviewers	Associate Professor Andrew T. Duchowski Department of Computer Science Clemson University Clemson, SC, USA
	Professor Kari-Jouko Räihä Department of Computer Science University of Tampere Tampere, FINLAND

Computing Reviews (1998) Classification: H.5.1, D.2, J.4, K.3.1.

Towards an Eye-tracking Methodology for Studies of Programming

Roman Bednarik

Department of Computer Science
University of Joensuu
P.O.Box 111, FIN-80101 Joensuu, FINLAND
bednarik@cs.joensuu.fi

University of Joensuu 2006, 100 pages

Abstract

Empirical studies of programming have previously employed approaches such as think-aloud or comprehension summary analysis. Despite its potentials, eye-movement tracking has not been widely applied to the studies of behavioral aspects of programming. In this thesis we study limitations and benefits of eye-movement tracking in the domain of psychology of programming. We argue for development of a methodological framework that would allow for safe and effective application of eye-movement tracking to study human behavior and cognition during programming tasks. As initial steps toward the methodology, we perform exploratory studies of programmers during program debugging and comprehension.

Keywords: Eye-movement tracking methodology, psychology of programming, program comprehension

Acknowledgements

” Cool, here I can write thanks to everyone.” In the template that I have used as a starting point, exactly this phrase is included as the only content of acknowledgments. With every other build of the manuscript, I saw it. Who are those everyone? Will I remember them all? The list would be long. I would forget somebody. Although I think it is too early to thank to everyone, here we go: thank you, everybody, with whom I could work during past years, who helped me and supported me.

List of original publications

P1. Bednarik, R., Tukiainen, M.: Visual Attention and Representation Switching in Java Program Debugging: A Study Using Eye Movement Tracking. In *Proceedings of 16th Annual Psychology of Programming Interest Group Workshop (PPIG'04)*, Institute of Technology Carlow, Ireland, April 5-7, 2004, pp. 159-169.

P2. Bednarik, R., Tukiainen, M.: Visual Attention Tracking During Program Debugging. In *Proceedings of NordiCHI 2004, The Third Nordic Conference on Human-Computer Interaction*, October 23-27, 2004, Tampere, Finland, ACM Press, pp. 331-334.

P3. Bednarik, R., Tukiainen, M.: Effects of Display Blurring on the Behavior of Novices and Experts during Program Debugging. In *CHI '05 extended abstracts on Human factors in computing systems, CHI 2005*, Portland, Oregon, USA, April 2-7, 2005, ACM Press, pp. 1204-1207.

P4. Bednarik, R., Myller, N., Sutinen, E., Tukiainen, M.: Effects of Experience on Gaze Behaviour during Program Animation. In *Proceedings of the 17th Annual Psychology of Programming Interest Group Workshop (PPIG'05)*, Brighton, UK, June 28 - July 1, 2005, pp. 49-61.

P5. Bednarik, R., Myller, N., Sutinen, E., Tukiainen, M.: Analyzing Individual Differences in Program Comprehension. *Technology, Instruction, Cognition and Learning (TICL)*, 3(3-4), 2006, pp. 205-232.

Contents

Chapter 1

Introduction

Program comprehension, the ability to understand programs written by others[1] , is widely recognized as central skill to programming. Being also a cognitively complex skill of acquiring mental models of structure and function of a program, program comprehension has been for many years a field of need to develop and apply methods and techniques to effectively capture and analyze the involved mental processes. Although originally centered around (professional) computer-programmers developing computer programs, studies of programming strategies nowadays extend far beyond these borders (Blackwell, 2002), both in terms of users and application domains.

Researchers interested in examining cognitive processing during problem-solving tasks have several possibilities of how to get insights into the behavior and strategies exhibited by the participants. Especially in the situations when the stimuli are visual and the reasoning is related and depended on it, eye-tracking systems have shown to be useful in revealing the patterns of visual attention during the task. The classical examples of successful applications of eye-movement tracking include studies relating the eye-movement patterns to cognitive processes (Just and Carpenter, 1976), research in reading (Rayner, 1998), or studies investigating differences between novices and experts in terms of the eye-movement patterns (e.g., Hyönä et al., 2002, Law et al., 2004). In the domain of studies of programming, instead, investigations have been mostly based on verbal-protocols, a well established - and probably the most popular - method used to capture and analyze the thought-processing. Despite its potentials, eye-movement tracking has not been widely applied in the domain.

[1]And, of course, also the programs written by oneself, after a prolonged period.

1.1 Motivation

It seems therefore obvious that eye-movement tracking could help us to shed a light onto the mental processes involved in programming. However, there are several important issues to be considered before applying an eye tracker. First is technological: the relatively high price of an accurate eye-tracking equipment and problems such as the need for calibration, drift or a certain level of obtrusiveness have prevented a wider application of the technology. To remedy some of the problems and limitations, researchers have sought to develop cheap and yet accurate alternatives to eye-trackers. This thesis will illustrate how well these alternatives deliver their promise to capture the patterns of visual attention in comparison with an eye-tracker.

Another issue is methodological: previous research using eye-movement tracking in other domains than studies of programming have established numerous measures and models of gaze behavior during certain tasks. For instance, Aaltonen et al. (1998) studied user behavior during menu selection. In usability studies, eye-tracking data can directly inform the evaluation (Goldberg and Kotval, 1999) and therefore also the design of computer interfaces. However, applications of eye-movement tracking to study the behavioral aspects of programming are still rare. At the moment, we have only a little knowledge how to apply an eye-tracking to study programmers. Therefore, we believe it is a high time to develop an eye-tracking methodology for this particular domain.

1.2 Research Questions

This thesis is written to start answering the question, how visual attention tracking, and eye-movement tracking in particular, can effectively be applied to record and analyze the cognitive processes involved in programming. More precisely, it starts to develop a methodological framework that will allow researchers from the domains of psychology of programming, and problem-solving in general, to safely *apply* eye-movement tracking and to *analyze* the resulting data to build proper conclusions about underlying behavior.

By comparing some of the alternative tools to track visual attention in a replication study we validate both their ability and usability to measure the visual attention patterns exhibited by programmers during debugging tasks. Further, we investigate what eye-movement measures developed in previous eye-tracking research can be used to characterize comprehension and debugging strategies. In addition, we investigate whether and what measures reflect a difference in expertise levels.

We approach the development process by answering the following questions:

- As some alternative tools to track visual attention have previously been developed, how do they compare to an eye-tracking device?

- Are expertise levels in programming reflected in some of the eye-movement data?
- Eye-tracking has not been widely applied to studies of programming, what previously developed measures can be applied to study cognitive processing of computer programmers? How do these measures relate to the underlying cognitive activities?

1.3 Research Methodology

In order to answer the research questions, we employ controlled empirical experiments conducted in a usability laboratory. We use within- and between-subject repeated measures designs with usually one and/or more factors. Independent variables controlled by the experimenter will be, for example, the technology to track the visual attention. Depended variables measured during the experiment will be performance (for example: time to complete the task, number of bugs discovered, cognitive model acquired), resulting visual attention patterns. In some cases we use also semi-structured interviews to evaluate the participants' subjective experience with the technology used in the experiments. Both exploratory and evaluative studies that would respectively generate and validate further hypotheses will be conducted. To analyze the resulting data from this research, we employ analysis of variance (ANOVA) techniques, pre-planned t-tests, and analysis of correlations.

1.4 Organization of the Thesis

The rest of this work is organized as follows: first, we discuss how visual attention focus is reflected in eye-movement patterns and the current state of visual attention tracking methodologies. At the second part of this thesis, we review previous research in programming behavior, in terms of the theoretical underpinnings of program comprehension and tools to study the behavioral aspects of programming. After presenting general discussion, we provide an overview of the results.

Chapter 2

Visual Attention and Eye-Movement Tracking Methodology

While the eyes of the readers of this thesis fixate on the words as they read this sentence, rather complex and complicated processes take part. The words have to be encoded, comprehended, integrated with the previous text, while the eyes have to be repositioned on the next word or, perhaps, on the previous, if an inconsistency appeared. As the stimuli in reading tasks are visual and therefore the underlying processes also visually driven, studies of reading, text comprehension, and language processing are a successful example of the benefits from applying eye-movement tracking. [1]

There are many reasons and advantages to employ eye-movement tracking as a research methodology. In past, eye-movements have been shown not only to provide a rich and relevant source of information about person's cognitive processes and intentions, but have been also employed as an interaction modality. Eye-movement tracking has been successfully employed in studies of reading (e.g., Just and Carpenter, 1980, Rayner, 1998), gaze-based interaction (e.g., Jacob, 1991, Jacob and Karn, 2003), eye typing (Majaranta and Räihä, 2002), menu selection (Crosby and Peterson, 1991, Aaltonen et al., 1998), usability (Goldberg and Kotval, 1998, 1999) or in virtual reality (Duchowski et al., 2000, 2002). Between many advantages of applying eye-movement tracking to study human behavior also belong that participants do

[1]Since this work is not primarily concentrated on eye-movement tracking and reading, but the research in the domain can serve as a good source of inspiration, for a detailed discussion of the previous research see (Rayner, 1998).

not have to be trained to exhibit their natural visual strategies and the most of the current eye-trackers are highly non-invasive. That makes the eye-tracking a usable, non-intrusive tool without causing any interferences with natural behavior.

In the following sections, we first present the main assumptions on which the research based on eye-movements rests and then focus on current methodologies to track and analyze the visual attention.

2.1 Visual Attention and Eye-Movements

Humans, as every vertebrate, have movable eyes. We move our eyes in order to bring an image of the inspected object onto the fovea, a small and high-resolution area of the retina where the cones are most densely packed. Once the image of the object is stabilized on the retina, the information can be extracted. This way the visual attention is linked with the current gaze direction and most of the time the visual information is processed it is also diverted to the point of visual inspection. In eye-movement tracking research, this principle is called *an eye-mind assumption* (Just and Carpenter, 1980). There are, however, situations when the visual attention and gaze direction are dissociated. Parafoveal or peripheral processing can be used to extract information from the environment. However, as also Duchowski (2003, p.14) points out: " *...in all eye-tracking work...we assume that attention is linked to foveal gaze direction, but we acknowledge that it maybe not always be so.*" This thesis rests on the assumption of Just and Carpenter, however, similarly as Duchowski, we acknowledge that the focus of visual attention and the direction of gaze might be, at times, dissociated.

Because of the limited size of the fovea, the gaze has to be re-directed to the new points of interest of the scene or object in order to allow for the new details to be perceived and processed. Once the eyes are directed and stabilized at the area of interest, the visual information can be extracted and encoded. The relatively stable position of the gaze direction is called *fixation*, while the shift of the gaze between two fixations is called *saccade*. A single saccade can last between 30 and 120 ms and can span over 1 to 40 degrees of visual angle (Sibert and Jacob, 2000), with velocities ranging up to 500 degrees per second (Rayner, 1998). It is assumed that a) during the saccades the vision is supressed and no information is extracted and processed; the phenomena known as saccadic suppression (Matin, 1974), and b) once initiated, a saccade's destination cannot be altered.

Fixations, on the other hand, are the movements of eyes during which the information can be extracted, encoded, and processed. Typically, the fixation duration ranges from 200 ms to 300 ms (Rayner, 1998) and can be thought as related to the processing required to extract and interpret the information (Just and Carpenter, 1976, Goldberg and Kotval, 1999). Following the eye-mind assumption, if we can track the movements of the eyes, we can also obtain good insights into and investigate the path and focus of visual attention during a task. Previous research has firmly confirmed this relation between eye movements, visual attention and underlying cognitive processes (e.g., Just and Carpenter, 1976, 1980, Rayner, 1998). Knowing which objects and elements have been visually fixated, in which order, frequency and context, we can attempt to infer what cognitive processes were involved in performing a task related to these objects.

Other types of eye-movements exist (Carpenter, 1988, Duchowski, 2003). *Smooth pursuits* occur when they eyes track a moving object, to match the velocity of the eyes with the target's and therefore reduce the target's retinal motion to minimum. *Nystagmus* eye movements serve to correct the motion of head or to attend repetitive patterns. *Miniature eye movements*, such as drifts and microsaccades, that might appear as a noise in the eye-movement signal, are executed to stabilize the retinal image during a fixation. Finally, the movements of the eyes that are not conjugate, executed for instance when attending an approaching object, are called *vergence.* For a more detailed review of eye-movements and their models, we refer an interested reader to (Carpenter, 1988).

Although not being eye-movements itself, *blinks* play an important role in vision and in eye-movement data analysis. Regarding the former role, blink is involuntary unconscious reflex that moves the eye lid, purposed to moist the cornea. As for the latter, a blink causes an instability and noise in the eye movement protocol and therefore has to be properly identified. Using blink rate as a physiological measure, Brookings and colleagues (1996) showed that blink rate decreases under higher cognitive load.

2.2 Visual Attention Tracking and Analysis

Eye-movement tracker is a device that records the point of gaze, i.e. the location of intersection of the line of sight and the surface of the observed object. Probably most of the current commercial eye-tracking systems are based on capturing infrared light reflections from the cornea and video image analysis of the center of the pupil. Unlike their first predecessors, the modern eye-trackers are relatively cheap and able to collect gaze data reliably and unobtrusively (Jacob and Karn, 2003). Two general classes of eye-tracking devices exist: a remote optics, table-mounted version and a head-mounted optics with a see-through mirror. The remote eye-movement trackers are usually used in experiments where the stimuli are presented at the video displays. They tolerate some head-movements, however, experimental participants are limited in their freedom of movements in the environment. On the other hand, the head-mounted eye-tracking systems allow the participants to freely move during the recording, but they impose certain discomfort by having the eye-tracking cameras mounted tightly on participants head. Regardless of the option, both types of eye trackers must be calibrated for each user before the first recording.

The data stream coming from an eye-tracker usually contains a timestamped pair of coordinates of current point of gaze, pupil diameter, and also some kind of validity codes. In order to identify fixations and saccades from the eye-tracking protocols, several algorithms to process the raw data were developed, for an overview see (Salvucci and Goldberg, 2000). These algorithms represent a middle layer between the raw data and higher level eye-tracking measures. As we mentioned above, fixation is a measure of highest interest in eye-movement based studies. In this work, we use the definition of fixation as in (Jacob and Karn, 2003): fixation is a stable position of the point of gaze within some threshold of dispersion over some minimum duration (100-200ms), and velocity below a defined threshold. Once the eye-tracking protocol is cleaned up from artifacts such as blinks and unwanted noise, fixations can be identified using the definition above.

In their summary of 21 usability studies, Jacob and Karn (2003) report on the most commonly used eye-tracking measures. Of these studies, 11 made use of *number of fixations*, 7 reported the *proportional time* spent looking on each area of interest. Another 6 studies used *mean fixation duration* and *number of fixations* on each area, 5 employed the *mean gaze duration* and 5 reported the *rate of fixations per second.* Other measures also exist, see (Jacob and Karn, 2003, Goldberg and Kotval, 1998) for more details and a review, respectively.

Jacob and Karn (2003) distinguish two ways of using the gaze data from an eye-tracking device: real time eye-tracking and retrospective analysis. The former approach involves gaze location as a direct interaction medium, so that the eyes of the user have a direct impact on the interaction with the interface in real time. The latter one, retrospective analysis of eye-movement data, normally, starts with a researcher defining so called *areas of interest*: (usually) rectangular areas covering the interface elements in question. The researcher then conducts an experiment, records the gaze data, and after the experiment tries to related the measures over the areas of interest to the manipulated variables and underlying cognitive processes, or to the eye control as such.

It turns out that one of the biggest challenges in any application of eye-movement tracking is to develop methods and techniques to effectively record, analyze, and interpret the gaze data. While the technical problems with eye-trackers, such as their relatively awkward application or low reliability, seem to be progressively solved by industrial vendors, methodological problems still persist.

2.3 Restricted Focus Viewer: An Alternative Tool to Track Visual Attention

To overcome some of the drawbacks of eye-movement tracking, certain alternatives to eye-movement tracking were developed. Recently, the Restricted Focus Viewer (Jansen et al., 2003), a tool that blurs the display and restricts users to only a small focused spot within an otherwise blurred stimulus, has been introduced as an alternative tool to track visual attention. When an experimenter employs the Restricted focus viewer (RFV) to track the visual attention, participants move the focused spot using a computer mouse to explore the visual representations in question. In order to get another part of the stimuli focused, a participant has to move the focused region using the computer mouse. The RFV then records the moves over the stimuli that are stored for later analysis. The tool collects the timestamped data for the mouse and the keyboard events, focused region's index, the total durations of sessions, and other events. Voice protocols can be recorded along the interaction data. The RFV tool, naturally, is not capable of collecting visual attention data when the blurring is turned off.

The Restricted Focus Viewer has been validated in two experiments run by Jansen et al.(2003); however, these validations involved only a relatively simple reasoning with visual stimuli. In other studies, the RFV-based technology has been applied 1) to discover strategies of participants debugging computer programs with an aid of multiple and linked visual representations of the programs (Romero et al., 2002a,b, 2003a,b), and for 2) investigating the issues of usability of hyper-linked documents (Tarasewich and Fillion, 2004, Tarasewich et al., 2005), or for 3) research of shifts of visual attention during integration of text and graphics (Futrelle and Rumshisky, 2001). Analysis and interpretation of the data recorded by RFV and their relation to the investigated task is, similarly as in eye-tracking studies, up to the researcher. Usually, the so-called areas of interest (AOI) are defined within the interface and several metrics can be computed, as for instance the total or proportional time spent on an AOI. Recently, the use of the RFV-based approach was questioned for the purposes of visual attention tracking during complex reasoning tasks such as computer-program debugging (**P2**, **P3**).

Chapter 3

Behavioral and Cognitive Aspects of Programming

Programming is a complex and cognitively demanding task due to the multiple interrelated components, trade-off decisions, and performance requirements that concern the whole process (Detienne, 2002). Traditionally, the studies of cognitive aspects of programming have been focused on expert computer-programmers.[1] Following the spread of computerized technologies, however, studies of programming strategies nowadays extend far beyond these borders (Blackwell, 2002).

One of the central tasks in programming, such as creating, maintaining and modifying a software product, is program comprehension, the ability to understand programs written by others (Pennington, 1987), and also own programs after an extended period. Program comprehension as a cognitively complex skill of acquiring mental models of structure and function of a program, is also a field of need to develop methods and techniques to effectively capture and analyze the involved mental processes. Despite its clear potentials, eye-movement tracking techniques have been only rarely applied in the domain. As the focus of the research of cognitive aspects of programming moves from the professional programmers to the end-users and novices in new domains such a program visualization, the traditional methodologies such as think-aloud might not be anymore as efficient as they showed to be in the past. In the following sections we review the previous research in program comprehension models and theories and in analysis of the cognitive aspects of programming.

[1]In many cases, the experts and novices alike were recruited from university students. Therefore, the experts shall rather be classified as intermediate programmers.

3.1 Theories of Program Comprehension

Studies of cognitive processes involved in program comprehension tasks are central to our understanding of software maintenance and development (von Mayrhauser and Vans, 1996). The theories that describe the program comprehension can be divided into *top-down*, *bottom-up*, or a combination of the two, according to how the comprehension is assumed to proceed (von Mayrhauser and Vans, 1996).

Top-down models propose that programmers use domain-knowledge and approach the comprehension by creating hypotheses and expectations that are later matched against the fractions of source code (Brooks, 1983). Brooks suggests that while verifying a hypothesis, programmers attempt to find a beacon (Wiedenbeck, 1986), a piece of code that would fit into the hypothesis. When such an information is not found, programmers either change their focus to another part of code, or modify the hypothesis or reject it. According to Brooks, the hypotheses are hierarchically organized and subdivided, until a certain level of fitness that the source code can explain and validate is found. Recently, different forms of top-down processes and sources of the actions that drive the programmers while approaching the comprehension have been studies by O'Brien et al. (2004).

Bottom-up models suggest that, as source-code is read, the high-level abstract representations and concepts are constructed from extracting and integrating the low-level information into the already formed model (Pennington, 1987, Shneiderman and Mayer, 1979). Pennington based her model on previous work on text comprehension (Pennington, 1987), distinguishing *program model* and *situation model* that are formed by a programmer. While the program model is constructed first and contains abstractions of information such as operations, control and data flow relations, the situation model is formed only after at least a part of the program model is constructed and contains more domain-oriented abstractions.

Based on observations of professional programmers during maintenance of large software products, Mayrhauser and Vans (1996, 1997) proposed so called *integrated program comprehension model.* The model proposes that programmers approach the comprehension using both top-down and bottom-up strategies, as described in previous research. When the programmers are familiar with the domain they tend to comprehend the program in a top-down manner. However, in unfamiliar domains, the comprehension is carried out in a bottom-up style to increase the domain and program knowledge. Therefore, (expert) programmers often switch between the two strategies.

Effect of expertise is an important issue not only in program understanding (Burkhardt et al., 2002, Pennington, 1987), but in any problem-solving domain. Therefore, as well as on program comprehension theories, the previous research has been focusing on differences between novice and skilled programmers, too. The studies have shown the superiority of expert programmers over novices in terms of domain knowledge, performance and strategies, both during comprehension and debugging of computer programs. Expert programmers found more bugs, found them faster and tended to spend more time on building a mental model of the problem (Gugerty and Olson, 1986). Experts are also more able to remember specific parts of the source code (Fix et al., 1993), they focus only on relevant information needed to solve the problem (Koenemann and Robertson, 1991), they are not committed to one interpretation as novices (Vessey, 1985) and are therefore able to change their strategies as needed.

3.2 Analyzing Cognitive Aspects of Programming

In past, program comprehension has been studied mostly using three main methodologies: cognitive processes during comprehension have been captured using think-aloud protocols and the outcomes of program comprehension have been analyzed using comprehension summaries and comprehension questions.

Particularly popular techniques to capture the thought-processes are different variations and derivations of the think-aloud methodology, either concurrent or retrospective, since the seminal work of Ericsson and Simon (1984). In the empirical studies of programmers, the think-aloud methodology was applied in numerous experiments, starting with the pioneering studies conducted by Soloway and others during 1980's, (e.g., Letovsky, 1986, Soloway et al., 1988, Littman et al., 1986, Pennington, 1987) through 1990's (e.g., von Mayrhauser and Vans, 1996, 1997, von Mayrhauser and Lang, 1999) and recently (Burkhardt et al., 2002, Ko and Uttl, 2003, O'Brien et al., 2004) and many others.

By analyzing verbal protocols, these and other studies attempted to get an access to cognitive processes and knowledge representation and thus provide insights into what strategies programmers of various expertise take while undergoing the tasks of program comprehension. To increase the validity of verbal reports, research practitioners often complement them with other concurrent behavioral data, such as direct observations, video or interaction protocols, as for example von Mayrhauser and Vans (1996) or Ko and Uttl (2003). An attempt to improve the methodology of think-aloud protocols for studies of programmers was proposed by Mayrhauser and Lang (1999). By unifying the terminology and improvements in coding of the verbal protocols, the proposal aims to reduce the efforts to conduct protocol analysis and provides researchers a way how to compare the results (Lang and Mayrhauser, 1997).

The approaches based on verbal reports, however, have been criticized widely (e.g., Branch, 2000, Nielsen et al., 2002, van den Haak et al., 2003), especially when used with complex tasks involving high cognitive load and requiring verbalizing visual information. As many of the studies of programming strategies are conducted with novice participants working with some programming environment, it might be hypothesized that indeed the tasks impose heavy cognitive load. Although not shown empirically, this might result in an interference with their normal problem-solving strategies and therefore biased results. In the context of usability studies, Goldberg and Wichansky (2003) also point out that some aspects of behavior, such as focus of attention during a task, have little awareness to an individual, and are thus hard to verbalize.

Besides the tools and techniques to record the human behavior during problem solving, also methodologies to analyze the cognitive processing and the resulting outcomes are needed. The outcomes of program comprehension are the mental models of program acquired by the programmer during the comprehension task. Good and Brna (2004) developed a schema based on the information types found in comprehension summaries. The schema allows a researcher to evaluate mental models acquired during comprehension in terms of proportions of different information types, such as statements related to control-flow, data-flow, function or operation. In addition, it allows to classify the statements according to the abstraction used to describe the objects. Recently, Byckling et al. (2004) suggested that also *irrelevant* category shall be included into the schema.

One of the traditional ways to analyze and evaluate the comprehension level and model are *comprehension questions*. After the comprehension task the participants are asked several questions that shall reflect the information that they have available. Comprehension questions were, for instance, employed in the studies of Pennington (1987) to validate her model.

3.3 Related Studies

Normally, programmers work within a computer-based (graphical) environments, such as program development, debugging or visualization tools. These environments often present some of the program representations in several adjacent windows and programmers have to coordinate these representations in order to construct a viable mental model. In these situations, when the problem-solving and formation of mental model is driven by visual information, such as during dynamic program visualization, it would be beneficial to capture and analyze the patterns of visual attention. As eye-movement data provide insight into visual attention allocation, it is also possible to infer underlying cognitive processes (Rayner, 1998).

Understanding that opportunity, Romero et al. (2002a, 2002b, 2003a, 2003b) conducted a series of experiments that involved the Restricted Focus Viewer (RFV)

(Jansen et al., 2003). In these studies, a Software Development Environment (SDE) was built on top of the RFV and employed to track visual attention to investigate the coordination strategies of programmers debugging Java programs, reasoning with multiple adjacent representations. Several eye-movement-like metrics were derived to identify superior debugging strategies of participants or to measure the effects of different visualizations on the coordination strategies. For example, balanced accumulated fixation times between different representations could reflect good debugging performance (Romero et al., 2003b). Another measure derived in the aforementioned studies is a number of switches per minute between the representations of a program. More experienced programmers were found to exhibit a higher switching frequency between the main representations.

This approach, however, was shown to be questionable. Using a remote eye-tracker, Bednarik and Tukiainen (2004a, 2004b, 2005) replicated one of the experiments in which the RFV was employed. They suggested that the blurring technique interferes with the natural strategies involved in program debugging, especially with those of more experience programmers. Similarly as in the influential studies of Petre (1995), the visual representations in experiments of Romero et al. were static. Modern program visualization tools, however, often present the concepts in form of dynamic animations. The ecological validity of these experiments could be, therefore, questioned too.

Eye-tracking as a research methodology in studies of programming has been previously applied to investigate how programmers read the code (Crosby and Stelovsky, 1990). Using an eye-tracker, patterns of programmers' visual attention were recorded while reading a binary search algorithm written in Pascal. Authors analyzed fixation times and number of fixations to reveal the strategies involved in reading source code. Crosby and Stelovsky argue, beside other findings, that while the subjects with greater experience paid attention to meaningful areas of source code and to complex statements, novice participants, on the other hand, visually attended comments and comparisons. Both groups paid least attention to the keywords and did not exhibit any difference in reading strategies. However, only one representation of program was used (the code) and the focus of the research was mainly on the critical, but surface features of code, not on the behavior while reasoning with multiple and dynamic visualizations.

Other studies using eye-movement tracking during program comprehension or debugging, such the one of Crosby and Stelovsky above, are infrequent and appeared only recently, (e.g., Bednarik and Tukiainen, 2004b, 2005, Nevalainen and Sajaniemi, 2005). Nevalainen and Sajaniemi (2004) compared three eye-tracking devices for the purposes of studying point of gaze of programmers using a program animation tool. The authors concentrated on practical issues of applying eye-tracking, such as the ease of use in terms of time needed to prepare a participant for a study, the accuracy measured as a mean distance of actual gaze from a requested point on the computer screen, and the validity of data obtained from an eye-tracker. Although somewhat vague in the analysis of the obtained results, the study suggests that a remote eye-movement tracker fits better for the given purposes than a head-mounted tracker.

Same authors (Nevalainen and Sajaniemi, 2005) later conducted an empirical experiment to investigate an effect of two program visualization tools on gaze patterns during comprehension. In the study, Nevalainen and Sajanimemi employed a remote eye-tracker and, not surprisingly, found some differences between the ways participants targeted their visual attention while working with a graphical program visualization tool to those gaze patterns exhibited while working with a traditional textual program visualization tool. As the depended eye-tracking measures, the authors used absolute viewing times (the sum of all fixations) and proportions of these times over three discrete areas. From a methodological point of view, it seems likely that the differences in the absolute viewing times might be related to or biased by the time needed to complete the animation task in each of the tools. Interestingly, with both tools the most of the time (about 58-64%) was spent looking at other area than at those containing code and visualization. In addition, no significant effect of a tool on the mental models created was found.

Bednarik et al. (2006, 2005) conducted an exploratory program-comprehension study where eye-movements of the participants were recorded using a remote eye-tracker. Authors reported that in terms of fixation counts and attention switching between main representations (code and graphical representation of execution) of a program during its animation, patterns of novice and more experienced programmers did not differ. An effect of experience was found, however, on overall strategies adopted to comprehend programs and on fixation durations.

In summary, although novel in employing a dynamic representation of program (Bednarik et al., 2005) or studying short-term effects of a visualization on comprehension (Nevalainen and Sajaniemi, 2005), researchers often approached the analysis of eye-movement data only from a long-term, global point of view: the data were treated as means over a whole comprehension session. In other words, a single measure, such as a mean fixation duration or total fixation time, are employed to represent a rather complex cognitive process of comprehending a computer program. We argue that these approaches are therefore oversimplification of the underlying processes and demonstrate a need to develop a more thorough methodological framework. To characterize and analyze the cognitive processes involved in programming in a greater detail, therefore, another approaches have to be involved.

Chapter 4

General Discussion

Thanks to the greater availability and improved usability, eye-movement tracking technologies become increasingly applied to study cognitive aspects of human behavior in various domains and psychology of programming is not an exception. Although the technical problems with applying eye-trackers are gradually disappearing, methodological issues and challenges remain.

Most of the previous research in the field of human behavior during programming tasks has been conducted using methodologies such as think-aloud or program summary analysis. In this thesis we introduce eye-tracking as an effective tool to study the behavioral aspects. We propose to build a solid methodological grounds, in terms of tools, measures, analysis methods, and relating the measures to the cognitive processes that shall allow researchers to safely apply eye-movement tracking.

As an initial step toward the methodological framework, we have validated an alternative tool, the Restricted Focus Viewer, that has been previously used to study human behavior in programming. We conducted a replication of a previous debugging experiment that employed RFV and we demonstrated that a remote eye-movement tracker is superior to RFV-based tracking for these complex domains. In addition, we revealed that the strategies of more experienced programmers were influenced by the blurring condition. Regarding the eye-tracking measures used in these studies, we found that the mean fixation duration can be used as an indicator of experience of the participants.

The initial studies were conducted using an artificial programming environment. The representations were static and precomputed, and the environment did not allow participants to interact with it. To increase the ecological validity of the eye-movement studies of programming behavior, we have conducted second experiment that employed a real program visualization environment. Participants of varying programming experience comprehended three Java programs while their eye-movements were recorded. In addition, we recorded interface-level events, such as mouse clicks, and application-level events such as start, pause or stop of the visualization. While the difference in expertise levels of the participants was demonstrated in the interaction protocols, only some of frequently used eye-tracking measures reflected the difference.

How much the eye-movements can reveal about the cognitive strategies of programmers? By conducting the exploratory experiments reported in this work and by reviewing some of the related studies, it seems that the adoption of the measures employed in previous eye-movement studies in usability domain (Goldberg and Kotval, 1999) can reveal only a fragment of the programmers' thought processes. Cognitive processes involved in programming are far more complex to be described by a single aggregated number, such as the proportion of time spent fixating on an area of interest. On the other hand, eye-movement patterns contain much more information than can be represented by a single aggregated measure. Our current efforts aim to address these issues.

4.1 Future Directions

Based on the results and experience gained from the presented studies, we suggest several research directions that seem to be worth of pursuing in the future.

In (**P4**, **P5**) we approached the analysis of eye-movement data only from a long-term, global point of view: the data were treated as means over a whole comprehension session. The experience gained from these studies indicates that to characterize the comprehension processes more completely, another, more detailed approach has to be taken. In future studies we plan to subdivide the comprehension process into meaningful pieces and study gradual changes in the related eye-movement patterns. By doing so, we believe to capture the changes in the role each of the representations take during the comprehension and therefore better characterize the construction of the mental model of the comprehended program.

Another step toward integration of eye-movement tracking into studies of programming is to investigate the relation of eye-movements to the cognitive activity and performance. Questions like "*can a strategy that leads to a successful comprehension be identified from eye-movement patterns?*" are to be answered in the future. Simultaneously, these questions have to be answered by using the currently available methodologies, such as information type analysis of comprehension summaries, verbal protocols, and interaction performance. The information of what part of the programming strategies and knowledge can be discerned using the various methods and how the data correlate will become an important contribution of the eye-tracking framework for studies of programming.

Finally, the previous research in program comprehension and debugging models established several models of program comprehension. Once the methodological grounds for applying eye-movement tracking are established, the previously created comprehension models can be validated using the technology or new models of comprehension can, possibly, be developed.

Chapter 5

Summary of Original Publications

Main results of the original publications (**P1**)-(**P5**) can be summarized and related to the research questions as follows:

In the first paper (**P1**), we conduct a small-scale empirical experiment and using a remote eye-tracker we replicate one of the previous studies that employed an alternative visual-attention tracking tool, the Restricted Focus Viewer (Jansen et al., 2003). We compare the measures obtained by the RFV to those recorded by a remote eye-tracker. Although based on a relatively moderate sample population, the results suggest that the data obtained from the eye-movement tracker differ from those obtained by the RFV. This work motivates us to concentrate on methodological issues of visual attention tracking in the domain of psychology of programming.

In the second paper (**P2**), we extend the study from (**P1**) to further validate our previous claim that the RFV-based systems do not seem to provide reliable information about gaze behavior. In comparison with a remote eye-tracker, the RFV-based system does not report the focus of visual attention as accurately. By observation of the video protocols with the gaze-direction superimposed, we support our empirical results that the focus of attention as measured by the eye-tracker is often displaced from the one measured by the Restricted Focus Viewer. Therefore, the conclusions based on the RFV-data can be subject of bias.

In the third paper (**P3**) we aimed to further delineate the limitations of the RFV-based visual attention. Rather surprisingly, we found that the effect of display blurring is more profound on more experienced participants, while their debugging performance is not influenced.

Altogether, the contributions (**P1**)-(**P3**) answer first research question. They demonstrate that there are limits on the generality of the conclusion that RFV is an equivalent substitute for eye-tracking in studies of problem solving strategies such as debugging computer programs. The blurring technique has been shown a) to not accurately collect visual attention, and b) to interfere with natural strategies of programmers. On the other hand, eye-movement tracking in its remote option, has been shown as highly non-invasive and superior tool for tracking visual attention of computer programmers. As the technique has not been widely applied to study the cognitive processes and related gaze patterns involved in programming, the published results represent the first steps toward an eye-tracking methodology to study behavioral aspects of programming.

Unlike in numerous previous investigations, **in the fourth and fifth papers** (**P4**, **P5**), we study computer programmers while working with a real, dynamic environment. We apply eye-movement tracking to record visual attention patterns of computer programmers during comprehension task aided by a Java program visualization tool. Programmers in our study were not limited in the ways they could interact with the environment; therefore some of them considered the visualization as guiding the comprehension process from beginning, while others used the visualization to confirm their hypotheses constructed from reading the source code. It turned out that the former pattern of interaction was characteristic to novice programmers, while the latter was exhibited by more experienced programmers. To allow for a fair comparison, in both reports, we analyzed allocation of visual attention only during program animation.

In the direction of second research question, in the paper (**P4**) we investigate differences in gaze behavior between novice and intermediate programmers. We concentrate on fixation count on each area, switching frequency between main representations, and mean fixation durations. We found that only mean fixation duration, whether measured on each of the important areas, or in general, reflects the differences in expertise of programmers. Other measures seem not to be affected by the experience levels.

There were two possible approaches to further de-construct the gaze behavior data during the complex tasks: spatial and temporal. We addressed the former one in (**P5**). Interface of a program visualization tool usually contains several linked representations displayed in adjacent windows, in which the tools provide different perspectives on program execution. During comprehension, programmers have to coordinate the representations and integrate with their mental models. As the tool employed in the experiment provides several representations (e.g. code, expression evaluation, instances of objects, and other) we analyzed the gaze patterns in terms of these representations, accordingly. Interaction events such as mouse clicks and states of the visualization tool were recorded, and analyzed. In addition, comprehension summaries were studied to evaluate the outcomes.

In regard of the third research question, we have found only a partial answer. Although capturing some of the differences caused by the experience levels and thus reflecting the differences in cognitive processing, the studies (**P4**) and (**P5**) demonstrate a partial inability of frequently used eye-tracking measures to distinguish novices from more experienced programmers. These findings motivate our future research to investigate the relation between cognitive processes involved in programming and eye-movement data.

References

Aaltonen, A., Hyrskykari, A., and Räihä, K.-J. (1998). 101 spots, or how do users read menus? In *CHI '98: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 132–139, New York, NY, USA. ACM Press/Addison-Wesley Publishing Co.

Bednarik, R., Myller, N., Sutinen, E., and Tukiainen, M. (2005). Effects of experience on gaze behaviour during program animation. In *Proceedings of the 17th Annual Psychology of Programming Interest Group Workshop (PPIG'05)*, pages 49–61, Brighton, UK.

Bednarik, R., Myller, N., Sutinen, E., and Tukiainen, M. (2006). Analyzing individual differences in program comprehension with rich data. *To appear in Technology, Instruction, Cognition and Learning*, 3(1).

Bednarik, R. and Tukiainen, M. (2004a). Visual attention and representation switching in java program debugging: A study using eye movement tracking. In *Proceedings of the 16th Annual Psychology of Programming Interest Group Workshop (PPIG'04)*, pages 159–169, Carlow, Ireland.

Bednarik, R. and Tukiainen, M. (2004b). Visual attention tracking during program debugging. In *Proceedings of The Third Nordic Conference on Human-Computer Interaction (NordiCHI'04)*, pages 331–334, New York, NY, USA. ACM Press.

Bednarik, R. and Tukiainen, M. (2005). Effects of display blurring on the behavior of novices and experts during program debugging. In *Proceedings of (CHI'05)*, Extended abstracts of the ACM Conference on Human Factors in Computing Systems, pages 1204–1207, Portland, OR, USA. ACM Press.

Blackwell, A. F. (2002). First steps in programming: A rationale for attention investment models. In *HCC '02: Proceedings of the IEEE 2002 Symposia on Human Centric Computing Languages and Environments (HCC'02)*, page 2, Washington, DC, USA. IEEE Computer Society.

Branch, J. L. (2000). Investigating the information-seeking processes of adolescents: The value of using think alouds and think afters. *Library and Information Science*

Research, 22(4):371–392.

Brookings, J. B., Wilson, G. F., and Swain, C. R. (1996). Psychophysiological responses to changes in workload during simulated air traffic control. *Biological Psychology*, 42:361–377.

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

Burkhardt, J., Détienne, F., and Wiedenbeck, S. (2002). Object-oriented program comprehension: Effect of expertise, task and phase. *Empirical Software Engineering*, 7(2):115–156.

Byckling, P., Kuittinen, M., Nevalainen, S., and Sajaniemi, J. (2004). An inter-rater reliability analysis of good's program summary analysis scheme. In *Proceedings of the 16th Annual Workshop of the Psychology of Programming Interest Group (PPIG 2004)*, pages 170–184, Institute of Technology Carlow, Ireland.

Carpenter, R. H. S. (1988). *Movements of the Eyes. (2nd ed.)*. Pion, London, UK.

Crosby, M. and Peterson, W. (1991). Using eye movements to classify search strategies. In *Proceedings of the Human Factors Society 35th Annual Meeting*, pages 1476–1480.

Crosby, M. E. and Stelovsky, J. (1990). How do we read algorithms? A case study. *IEEE Computer*, 23(1):24–35.

Detienne, F. (2002). *Software Design - Cognitive Aspects*. Springer-Verlag, Inc., London, UK.

Duchowski, A. T. (2003). *Eye Tracking Methodology: Theory & Practice*. Springer-Verlag, Inc., London, UK.

Duchowski, A. T., Medlin, E., Cournia, N., Gramopadhye, A., Melloy, B., and Nair, S. (2002). 3d eye movement analysis for vr visual inspection training. In *ETRA '02: Proceedings of the symposium on Eye tracking research & applications*, pages 103–110, New York, NY, USA. ACM Press.

Duchowski, A. T., Shivashankaraiah, V., Rawls, T., Gramopadhye, A. K., Melloy, B. J., and Kanki, B. (2000). Binocular eye tracking in virtual reality for inspection training. In *ETRA '00: Proceedings of the symposium on Eye tracking research & applications*, pages 89–96, New York, NY, USA. ACM Press.

Ericsson, K. A. and Simon, H. A. (1984). *Protocol analysis: Verbal reports as data.* Braford Books/MIT Press, Cambridge, MA.

Fix, V., Wiedenbeck, S., and Scholtz, J. (1993). Mental representations of programs by novices and experts. In *CHI '93: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 74–79, New York, NY, USA. ACM Press.

Futrelle, R. P. and Rumshisky, A. (2001). Discourse structure of text-graphics documents. In *Proceedings of 1st International Symposium on Smart Graphics*, New York, NY, USA. ACM Press.

Goldberg, J. and Kotval, X. P. (1998). Eye movement-based evaluation of the computer interface. In Kumar, S. K., editor, *Advances in Occupational Ergonomics and Safety*, pages 529–532. IOS Press.

Goldberg, J. and Kotval, X. P. (1999). Computer interface evaluation using eye movements: Methods and constructs. *International Journal of Industrial Ergonomics*, 24:631–645.

Goldberg, J. H. and Wichansky, A. M. (2003). Eye tracking in usability evaluation: A practitioner's guide. In Hyönä, J., Radach, R., and Deubel, H., editors, *The Mind's Eye: Cognitive and Applied Aspects of Eye Movement Research*, pages 493–516. Elsevier Science.

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

Gugerty, L. and Olson, G. M. (1986). Comprehension differences in debugging by skilled and novice programmers. In *First Workshop on Empirical Studies of Programmers on Empirical Studies of Programmers*, pages 13–27.

Hyönä, J., Lorch, R., and Kaakinen, J. K. (2002). Individual differences in reading to summarize expository text: Evidence from eye fixation patterns. *Journal of Educational Psychology*, 94:44–55.

Jacob, R. J. K. (1991). The use of eye movements in human-computer interaction techniques: what you look at is what you get. *ACM Transactions of Information Systems*, 9(2):152–169.

Jacob, R. J. K. and Karn, K. S. (2003). Eye tracking in human-computer interaction and usability research: Ready to deliver the promises (section commentary). In Hyönä, J., Radach, R., and Deubel, H., editors, *The Mind's Eye: Cognitive and Applied Aspects of Eye Movement Research*, pages pp. 573–605. Elsevier Science.

Jansen, A. R., Blackwell, A. F., and Marriott, K. (2003). A tool for tracking visual attention: The Restricted Focus Viewer. *Behavior Research Methods, Instruments, and Computers*, 35(1):57–69.

Just, M. A. and Carpenter, P. A. (1976). Eye fixations and cognitive processes. *Cognitive Psychology*, 8:441–480.

Just, M. A. and Carpenter, P. A. (1980). A theory of reading: From eye fixations to comprehension. *Psychological Review*, 87(4):329–354.

Ko, A. J. and Uttl, B. (2003). Individual differences in program comprehension strategies in unfamiliar programming systems. In *IWPC '03: Proceedings of the 11th IEEE International Workshop on Program Comprehension*, page 175, Washington, DC, USA. IEEE Computer Society.

Koenemann, J. and Robertson, S. P. (1991). Expert problem solving strategies for program comprehension. In *CHI '91: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 125–130, New York, NY, USA. ACM Press.

Lang, S. and Mayrhauser, A. V. (1997). Building a research infrastructure for program comprehension observations. In *Proceedings of the Fifth Iternational Workshop on Program Comprehension*, pages 165–169. IEEE Computer Society.

Law, B., Atkins, M. S., Kirkpatrick, A. E., and Lomax, A. J. (2004). Eye gaze patterns differentiate novice and experts in a virtual laparoscopic surgery training environment. In *ETRA'2004: Proceedings of the Eye tracking research & applications symposium*, pages 41–48, New York, NY, USA. ACM Press.

Letovsky, S. (1986). Cognitive processes in program comprehension. In *Papers presented at the first workshop on empirical studies of programmers on Empirical studies of programmers*, pages 58–79, Norwood, NJ, USA. Ablex Publishing Corp.

Littman, D. C., Pinto, J., Letovsky, S., and Soloway, E. (1986). Mental models and software maintenance. In *Papers presented at the first workshop on empirical studies of programmers on Empirical studies of programmers*, pages 80–98, Norwood, NJ, USA. Ablex Publishing Corp.

Majaranta, P. and Räihä, K.-J. (2002). Twenty years of eye typing: systems and design issues. In *ETRA '02: Proceedings of the symposium on Eye tracking research & applications*, pages 15–22, New York, NY, USA. ACM Press.

Matin, E. (1974). Saccadic suppression: a review and an analysis. *Psychological Bulletin*, 81(12):889–917.

Nevalainen, S. and Sajaniemi, J. (2004). Comparison of three eye tracking devices in psychology of programming research. In *Proceedings of the 16th Annual Workshop of the Psychology of Programming Interest Group (PPIG 2004)*, pages 151–158, Institute of Technology Carlow, Ireland.

Nevalainen, S. and Sajaniemi, J. (2005). Short-term effects of graphical versus textual visualisation of variables on program perception. In *Proceedings of the 17th Annual Psychology of Programming Interest Group Workshop (PPIG'05)*, pages 77–91, Brighton, UK.

Nielsen, J., Clemmensen, T., and Yssing, C. (2002). Getting access to what goes on in people's heads? Reflections on the think-aloud technique. In *Proceedings of The Second Nordic Conference on Human-Computer Interaction (NordiCHI'02)*, pages 101–110, New York, NY, USA. ACM Press.

O'Brien, M. P., Buckley, J., and Shaft, T. M. (2004). Expectation-based, inference-based, and bottom-up software comprehension. *Journal of Software Maintenance and Evolution: Research and Practice*, 16:427–447.

Pennington, N. (1987). Comprehension strategies in programming. In *Empirical studies of programmers: second workshop*, pages 100–113, Norwood, NJ, USA. Ablex Publishing Corp.

Petre, M. (1995). Why looking isn't always seeing: readership skills and graphical programming. *Communications of ACM*, 38(6):33–44.

Rayner, K. (1998). Eye movements in reading and information processing: 20 years of research. *Psychological Bulletin*, 124:372–422.

Romero, P., Cox, R., du Boulay, B., and Lutz, R. (2002a). Visual attention and representation switching during java program debugging: A study using the restricted focus viewer. In *DIAGRAMS '02: Proceedings of the Second International Conference on Diagrammatic Representation and Inference*, pages 221–235, London, UK. Springer-Verlag.

Romero, P., du Boulay, B., Cox, R., and Lutz, R. (2003a). Java debugging strategies in multi-representational environments. In *15th Annual Workshop of the Psychology of Programming Interest Group (PPIG'03)*, pages 412–434.

Romero, P., du Boulay, B., Lutz, R., and Cox, R. (2003b). The effects of graphical and textual visualisations in multi-representational debugging environments. In *Proceedings of 2003 IEEE Symposia on Human Centric Computing Languages and Environments*, pages 236–238, Washington, DC, USA. IEEE Computer Society.

Romero, P., Lutz, R., Cox, R., and du Boulay, B. (2002b). Co-ordination of multiple external representations during Java program debugging. In *HCC '02: Proceedings of the IEEE 2002 Symposia on Human Centric Computing Languages and Environments (HCC'02)*, pages 207–214, Washington, DC, USA. IEEE Computer Society.

Salvucci, D. D. and Goldberg, J. H. (2000). Identifying fixations and saccades in eye-tracking protocols. In *ETRA '00: Proceedings of the symposium on Eye tracking research & applications*, pages 71–78, New York, NY, USA. ACM Press.

Shneiderman, B. and Mayer, R. (1979). Syntactic/semantic interactions in programmer behavior: A model and experimental results. *International Journal of Computer and Information Sciences*, 8(3):219–238.

Sibert, L. E. and Jacob, R. J. K. (2000). Evaluation of eye gaze interaction. In *CHI '00: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 281–288, New York, NY, USA. ACM Press.

Soloway, E., Lampert, R., Letovsky, S., Littman, D., and Pinto, J. (1988). Designing documentation to compensate for delocalized plans. *Communications of ACM*, 31(11):1259–1267.

Tarasewich, P. and Fillion, S. (2004). Discount eye tracking: The enhanced restricted focus viewer. In *Proceedings of 2004 Americas Conference on Information Systems*, pages 1–9, New York, NY, USA.

Tarasewich, P., Pomplun, M., Fillion, S., and Broberg, D. (2005). The enhanced restricted focus viewer. *International Journal of Human-Computer Interaction*, 19(1):35–54.

van den Haak, M., Jong, M. D., and Schellens, P. J. (2003). Retrospective vs. concurrent think-aloud protocols: testing the usability of an online library catalogue. *Behaviour and Information Technology*, 22(5):339–351.

Vessey, I. (1985). Expertise in debugging computer programs: A process analysis. *International Journal of Man-Machine Studies*, 23(5):459–494.

von Mayrhauser, A. and Lang, S. (1999). A coding scheme to support systematic analysis of software comprehension. *IEEE Transactions on Software Engineering*, 25(4):526–437.

von Mayrhauser, A. and Vans, A. M. (1996). Identification of dynamic comprehension processes during large scale maintenance. *IEEE Transactions on Software Engineering*, 22(6):424–437.

von Mayrhauser, A. and Vans, A. M. (1997). Program understanding behavior during debugging of large scale software. In *ESP '97: Papers presented at the seventh workshop on Empirical studies of programmers*, pages 157–179, New York, NY, USA. ACM Press.

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

Original Publications

P1.

Bednarik, R., Tukiainen, M.: Visual Attention and Representation Switching in Java Program Debugging: A Study Using Eye Movement Tracking. In Proceedings of 16th Annual Psychology of Programming Interest Group Workshop (PPIG'04), Institute of Technology Carlow, Ireland, April 5-7, 2004, pp. 159-169.

Visual attention and representation switching in Java program debugging: A study using eye movement tracking

Roman Bednarik and Markku Tukiainen
Department of Computer Science
University of Joensuu
{roman.bednarik, markku.tukiainen}@cs.joensuu.fi

Keywords: POP-V.A. Attention Investment, POP-V.B. Eye Tracking, POP-II-B. Debugging

Abstract

This paper describes a study of Java program debugging using a multiple window software debugging environment (SDE). In this study we have replicated an earlier study by Romero et al. (2002a, 2002b, 2003), but with the difference of using both the Restricted Focus Viewer and the eye tracking equipment to track the visual attention of the subjects. The study involved ten subjects debugging short Java programs using the SDE. The SDE included three different representations of the Java programs, those of the program source code, a visualization of the program, and its output concurrently in three separate panels in SDE. We used the Restricted Focus Viewer (RFV) and a remote eye tracker to collect the visual attention of the subjects. A with-in subject design, similar to Romero et al., employing both RFV/no-RVF task conditions was used.

The overall results of the time distributions over three different representations of the programs agree with the study of Romero et al. But the results of visual attention switching raise some questions to be considered in later studies.

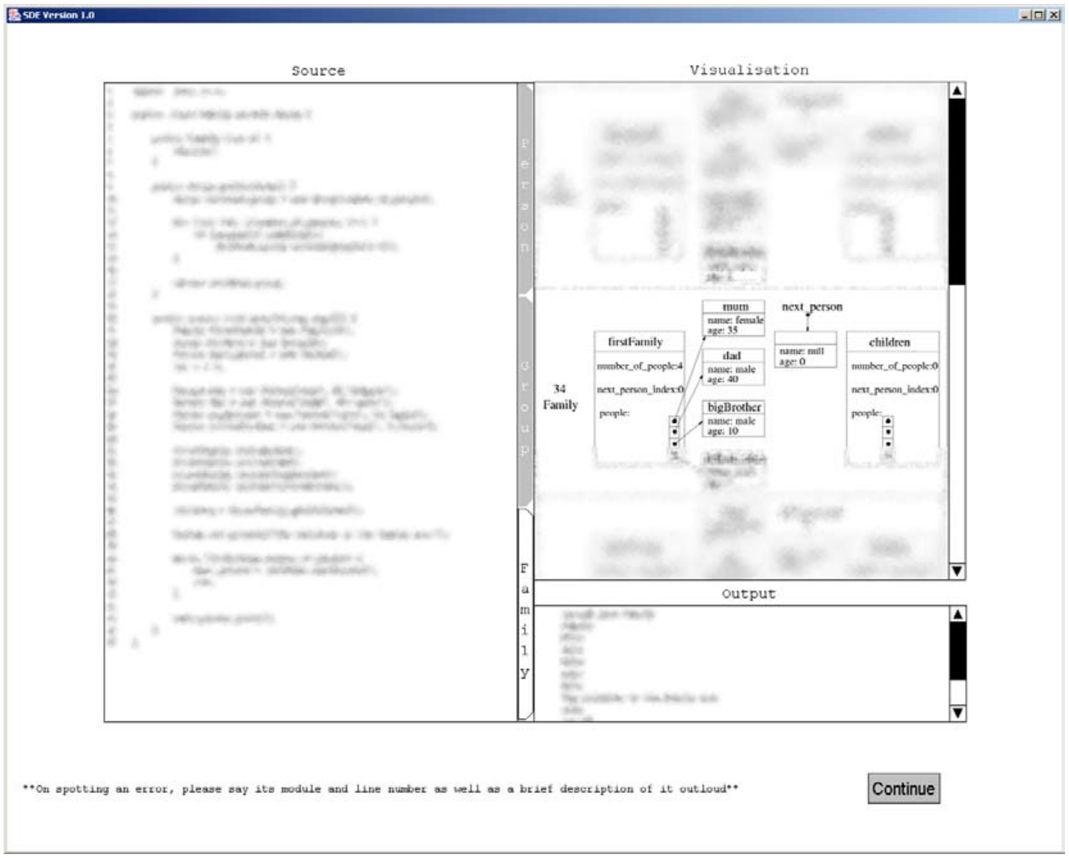
Introduction

Modern tools for software development usually include debugging environments, which typically consist of multiple mutually linked representations of the program under development. Programmers use these different representations to build up a mental model of a program. While discovering the errors in the source code, programmers have to perform a large number of mental tasks and at the same time have to coordinate different representations offered by the debugger. Typically, the program code, the debugger output, and some other kinds of representation of the program are displayed simultaneously. One of the aims of the research related to debugging is to understand, how programmers coordinate multiple representations, and what debugging strategies they exhibit.

Previous work on visual attention and representation switching in debugging

One way to investigate the coordination of multiple representations employed in debugging tasks is to use a tool to track the visual attention. For tracking visual attention eye tracking equipment has become more common in recent years. The Restricted Focus Viewer (RFV) has been developed as an alternative to eye tracking (Blackwell, 2000). One of the main advantages of RFV is that it enables automated collection of subjects' focus of attention. The RFV blurs the stimuli image and displays it on a computer screen, therefore allowing the subject to see only a limited focused region at a time. In order to get another part of the stimuli focused, subject has to move the focused region using the computer mouse. The RFV then records the moves over the stimuli, which are stored for later analysis. The tool collects the data for both mouse and keyboard events together with the timestamps, focused region's index, total durations of debugging sessions, and other events.

A special software debugging environment (SDE) on top of the RFV has been developed for the purpose of tracking the switches between the representations (Romero et al. 2002a, 2002b). For that purpose, the RFV has been modified so that the subject has to click a mouse button to set the place of the focused region. The SDE also remembers the positions of the focused regions for each of the displayed panels. When the subjects return their attention pointer to the panel with a representation, the SDE automatically focuses the last region of interest. The SDE shows several representations of a program in the adjacent windows, namely the source code, the output, and the visualization. While working under the restricted focus condition, only a small region of the SDE window is shown in focus. Figure 1 shows a typical screen shot of the sample interaction with SDE under the restricted condition. The focused region following the mouse pointer is shown over the visualization.



On spotting an error, please say its module and line number as well as a brief description of it outloud

Figure 1 Software debugging environment used in the experiment

Although the RFV-based debugging environment allows for easy tracking of attention, it also introduces several constrains. First, the environment is limited to the static pre-computed stimuli images, thus it does not allow us to track the visual attention in unlimited dynamic environments. Second, it requires subjects to move frequently the computer mouse to see the areas of interest focused. Finally, the temporal resolution of the RFV-based measurement is limited to the movement-by-movement level of detail. To our best knowledge, eye tracking has not been applied to research related to debugging strategies of programmers.

Eye movements

While searching for a visual object people make rapid eye movements, called saccades, shifting the point of gaze. It is supposed that no information is collected during the saccade since vision is suppressed during the shifts. This phenomenon is known as saccadic suppression. Once the object is positioned on the high acuity fovea, the information from the stimuli is extracted during the fixation. Jacob and Karn (2003) define a fixation as “a relatively stable eye-in-head position within some threshold of dispersion (typically ~2°) over some minimum duration (typically 100-200ms), and with a velocity below some threshold (typically 15-100 degrees per second).”

Eye movements and visual attention are coupled in the sense that the shifts in the foveal gaze direction are linked with the voluntary intention to change the focus of visual attention. Although the link is vivid and confirmed across the research in the recent decades, we acknowledge that it might not be always so, since we may sometimes attend to an object without moving our eyes.

While comparing the gaze and computer mouse as the input modalities, gaze is considered to be a more natural and intuitive way of input, since we move our eyes mostly unconsciously. Because the eye movements are rapid, the gaze is also significantly faster than the mouse (Sibert&Jacob, 2000); however, due to the limitation of human fovea it is less accurate, with the effective range being 1 degree of the visual field (Zhai et al., 1999).

For a comprehensive review of eye movement research see e.g. (Duchowski, 2003; Rayner, 1998; Jacob, 1995), for the overview of the research related to the relationship between eye movements and attention, see e.g. (Godijn&Theeuwes, 2003).

Eye tracking

Up to date, eye tracking techniques have been involved in many different kinds of studies. Their wider application, however, has been limited by many factors. The earlier eye trackers were expensive, required frequent recalibration, were awkward for subjects due to having mechanical contact with their eyes, and the data analysis was time consuming and tedious. Recently, the price of the technology dropped and enabled more research laboratories to acquire the eye tracking system. The ability to track subjects' eyes has also been significantly improved in comparison with the systems from the recent decades. Modern, commercially available eye trackers are usually based on video images of the eye (Jacob&Karn, 2003; Duchowski, 2003). A ray of light, usually from an invisible infrared source of illumination, is shone at the subjects' eye, and multiple reflections from the eye are captured together with the geometrical properties of the eye. Multiple reflections are used to dissociate minor head movements from the rotations of the eyes. Subjects are thus allowed to move their head freely, only with minor spatial restrictions. The table-mounted, remote eye trackers, do not make any contact with subjects, some even do not introduce any visible interference with the working environment. While using a remote eye tracker the subjects' head movements are restricted to approximate spatial cube of edge about 25cm. The head-mounted version of eye tracking optics, however, poses a need for a headband firmly mounted on the subject’s head. The disadvantage is counterbalanced by the fact that the head movements are not limited in space or speed.

The spatial accuracy of the modern eye trackers ranges around the half of visual degree, and the temporal resolution is often above 50 Hz, which allow tracking the eyes with the high spatial and temporal precision. In their recent study, Nevalainen and Sajaniemi (2004) confirm that eye tracking can be used in the research of psychology of programming, with a relatively high accuracy of point of gaze recording. Most of the current eye-tracking systems include a software package, which makes the collection and analysis of the data easy and fast.

Aims of the study

The goal of the current study was to validate the previous results in representation switching obtained using the RFV against results measured by the remote-mounted eye tracker. We especially aimed to set up an environment and stimuli programs as close as to those reported in (Romero et al. 2002a, 2002b; Romero et al 2003), but in addition, we used eye-movement tracking. Therefore, for the same settings we could compare the focus of attention measured by the RFV to that measured by the eye tracker. The RFV reports the focus of attention in form of the index of stimuli image and the position of mouse, the eye tracker reports the position of the gaze and corresponding area of interest index. As a part of the study, the investigation also aimed at validation of visual attention focus as measured by RFV and by eye tracking.

Method

The experimental setting

For eye-movement tracking we used the remote Tobii ET-1750 eye tracker with the sampling-rate set to 30Hz. Participants were seated comfortably in an ordinary office chair, facing the seventeen inch TFT computer screen from a distance of about one meter.

The software debugging environment (SDE) originally utilized e.g. in (Romero et al. 2002a, 2002b; Romero et al 2003) was used in the experiment. In these studies and also in the current experiment, the representations, i.e. the program code, visualization, and output of the program, were pre-computed and static. In the current study, all of the debugging sessions employed only graphical functional visualization.

The eye-tracking data and the audio protocol were collected throughout the whole experiment. Thus, for the same instant the three sources of data could be analyzed: the focus of attention measured by the RFV, the focus of attention measured by the eye tracker, and the audio protocol. The eye tracking protocol was processed by the automatic fixation detection algorithm, with the thresholds set as follows: the minimal fixation duration was 10 ms and the fixation radius limited to 50 pixels. The coordinates of the areas of interest corresponded to the corners of the windows presented by SDE.

Participants and procedure

The subjects in the experiment were teachers and students from the Department of Computer Science at the University of Joensuu. All subjects were volunteers and they received a lunch ticket for their participation. In the study were the total of ten participants, one female and nine males, all with perfect or corrected-to-perfect vision. None of the subjects had previously participated in eye tracking study. The level of programming experience varied, ranging from undergraduate students who had just passed an introductory course in Java, through postgraduate students with substantial programming experience, to experienced programmers with a long history of using Java. Some of the participants were Java teachers or even worked as professional programmers. Table 1 gives the background information about the participants' programming experience.

Participant	1	2	3	4	5	6	7	8	9	10
Java experience	24	4	5	6	3	10	60	12	5	6
Programming exp.	36	120	108	36	24	96	60	120	84	36
Professional exp.	No	No	No	Yes	No	Yes	Yes	Yes	Yes	No

Table 1. Programming experience of participants in months

Prior to the debugging sessions, each participant had to pass an automatic eye-tracking calibration routine, which consisted of tracking the eyes as they followed sixteen shrinking points across the computer screen. This procedure ensured the accuracy of the eye tracking and accommodation of the eye tracker's parameters to the personal characteristics. After the calibration, participants read the description of the experiment, specifying the environment used, its control and the setting of the study. Then they performed three debugging sessions. The first one was performed under the restricted focus condition and served as a warm-up, however, the participants were not aware of that fact. After the warm-up, the two main debugging sessions were performed. One of these was under the restricted focus condition, the other without the restriction. The order of the target programs and restricting conditions were counterbalanced.

Each debugging session consisted of two phases. In the first, the participants were presented with the short specification of the target program, which included a description of the problem, the approach to the solution, and two sample outputs of the interaction with the program. One output presented the actual, erroneous behaviour of the program, and the other one presented the desired, correct behaviour of the program.

In the second phase, participants were given ten minutes to debug each program; a sound signalled two minutes till the end of each session. Participants were instructed to find as many errors as possible and to report them aloud by stating the error itself, the class and line number in which it occurred, and how the error could be corrected. They were also encouraged, but not forced, to think aloud while debugging.

The target programs used in this study were identical to those used in (Romero et al. 2002b). The warm-up program inspected whether a point was inside a rectangle. The first program ('Family' program) prints out the names of the children of a sample family and the second program ('Till' program) counts the cash in a cash-register till, giving subtotals for the different denominations of coins. In their study (2002b), Romero et al. had two versions of the target programs; the main difference between these versions was that the second one was a more sophisticated version of the first one. We used the less sophisticated versions of the programs and the graphical functional representations in visualizations. Altogether with and without the restricted focus imposed by the RFV, there were four different experimental conditions.

The two main experimental programs contained four errors each; the warm-up program was seeded with two errors. Following the classification of the errors established in (Romero et al. 2002a, 2002b; Romero et al 2003) the errors in the target programs can be classified as functional, control-flow, and data-structure. There was no syntactical error in the programs (all programs could be compiled) and participants were notified of this.

In order to collect the additional subjective experiences, each participant was debriefed after the debugging session.

Results

Debugging performance

The results related to the debugging performance are presented in Table 2, which summarizes the number of errors spotted for the two main sessions. There were four errors in each of the target source codes. The results show that the most successful participants were number 9, who discovered all errors, then number 6 who spotted seven errors, and number 2, who located six errors. Participants 3, 8, 10 and 4, 7 achieved average scores of five and four, respectively. Participant 1 located three bugs, and participant 5 discovered one error.

The total number of errors found for all participants in restricted view condition was 22 and in unrestricted view condition was 26 out of a maximum of 40 errors. There seems to be no difference in distribution of errors spotted between the conditions of restricted and unrestricted view. In general, more experienced programmers found more errors (participants 2, 3, 6, 8 and 9, general programming experience of 6 years or more), U-value $U(5,5) = 24$, $p<0.05$ for the effect of experience on errors found.

Participant		1	2	3	4	5	6	7	8	9	10
Errors found	RFV on	2	3	3	3	0	3	2	3	4	3
	RFV off	1	3	2	1	1	4	2	2	4	2
	Total	3	6	5	4	1	7	4	5	8	5

Table 2. Number of errors found (each condition contained 4 errors)

Eye tracking results and debugging behaviour

The global experimental results for debugging behaviour are shown in Tables 3 and 4. Table 3 presents the percentage of time that participants spent looking at each representation obtained by the eye tracker. The results for time distribution between the windows in the SDE agree with the results obtained by Romero et al. (2002a, 2002b, 2003); most of the time is spent on the code window, then on the visualisation window, and least time is spent on the output window. In our experiment, there was nearly significant effect ($t(9)=1.895$, $p<0.05$) of proportional time spent on the visualization between the restricted view condition (RFV on) and the unrestricted view condition (RFV off). There were no significant effects for proportional time spent on the code and the output for the conditions.

Participant		1	2	3	4	5	6	7	8	9	10
RFV on	Code	92.7	86.8	87.2	91.4	71.8	89.8	75.1	92.7	93.1	73.0
	Visualization	4.1	4.9	9.8	4.7	27.6	4.6	22.8	6.4	5.8	16.0
	Output	3.2	8.3	3.0	3.9	0.6	5.6	2.1	0.9	1.1	11.0
RFV off	Code	93.6	91.5	79.2	73.6	62.9	88.3	81.0	85.9	84.4	75.1
	Visualization	5.7	2.8	17.1	17.5	33.0	7.0	15.3	9.6	9.0	22.5
	Output	0.7	5.7	3.7	8.9	4.1	4.7	3.7	4.5	5.6	2.4

Table 3. Percentage of time spend in each representation measured by the eye tracker

Table 4 presents the average number of switches per minute measured by both the RFV tool and the eye tracker. By the term “switch” we mean the change of focus between the areas of interest, here between the code window, visualization window and output. The results obtained from the RFV tool protocol in restricted view condition (RFV on) are slightly higher than those in previous works (total average of 2.56 switches per minute for nine subjects, Romero et al. (2002a) total average of 1.73 switches per minute for five subjects), but in line with results from Romero et al (2002a). The results under the restricted condition obtained from the eye tracker, however, differ significantly from those obtained by the RFV ($t(9) = 3.49$, $p<0.01$). The participants switch their visual attention more often than they change the focused area with the mouse pointer in the RFV. This is also evident from looking into the eye movement protocols. From the eye movement protocols, we can see that the participants look at the blurred areas quite often and switch their gaze point even between the totally blurred windows in SDE. There is also a significant difference in switches per minute between the RFV on and RFV off conditions measured by the eye tracker ($t(14)=2.32$, $p<0.05$).

Participant		1	2	3	4	5	6	7	8	9	10
RFV on	RFV	1.2	1.69	3.01	7.2	2.0	4.11	3.91	2.0	1.1	4.01
	Eye Tracker	5.31	6.55	8.81	4.4	5.61	6.34	6.71	4.01	3.01	8.03
RFV off	Eye Tracker	4.11	6.08	4.92	7.81	6.51	13.09	9.35	10.74	12.14	10.13

Table 4 Average numbers of switches per minute measured by RFV and eye tracking

The debugging behaviour measured by the eye tracker of all the participants in terms of switches per minute between each of the representations is shown in Table 5. For both of the conditions there are six types of switches and a corresponding number of switches per minute: a switch from code to visualization (the upper number) and a switch from visualization to code (the number at the bottom), a switch from code to output and an inverse switch from output to the code, and a switch from visualization to output and from output to visualization.

The effect of the restricted view condition on switching between the code and visualization representations and between the visualization and output was not significant. The effect of the restricted view on the total number of switches per minute from the code to output and from output to code was significant (t(9) = 2.39, p<0.05).

Participant		1	2	3	4	5	6	7	8	9	10
RFV on	Code <-> visual.	2.0 1.9	1.58 1.48	2.9 3.11	1.1 0.9	2.5 2.5	2.23 1.71	1.9 2.0	1.2 1.4	1.1 0.9	2.51 2.51
	Code <-> output	0.3 0.4	1.37 1.48	1.2 1.1	0.7 1.0	0 0.1	0.51 0.86	0.4 0.3	0.5 0.4	0.2 0.3	1.2 1.2
	Visual. <-> output	0.4 0.3	0.42 0.21	0.2 0.3	0.5 0.2	0.3 0.2	0.69 0.34	1.0 1.1	0.2 0.3	0.3 0.2	0.3 0.3
RFV off	Code <-> visual.	1.5 1.4	0.9 1.01	1.01 0.9	2.2 1.8	1.6 1.7	3.83 4.03	2.11 2.21	3.51 3.61	2.91 2.61	4.41 4.31
	Code <-> output	0.2 0.4	1.58 1.58	1.11 1.21	1.1 1.5	0 0	2.01 1.81	1.51 1.41	1.1 1.1	1.4 1.81	0.2 0.3
	Visual. <-> output	0.4 0.2	0.45 0.56	0.4 0.3	0.8 0.4	1.6 1.6	0.6 0.81	1.01 1.11	0.7 0.7	1.91 1.51	0.5 0.4

Table 5 Number of switches per minute for each switch type. Upper number corresponds with the rate from code or visualization; bottom corresponds with the rate from visualization or output.

Figure 2 and Figure 3 summarize the data from Table 4 and Table 5 in the form of graphs. The debugging behavior of participants 10 (less experienced) and 9 (most successful), respectively, is shown. The diagrams on the left describe the behavior under the restricted condition; the diagrams on the right describe the behavior under the unrestricted setting. The direction of arrows follows the types of switches introduced above.

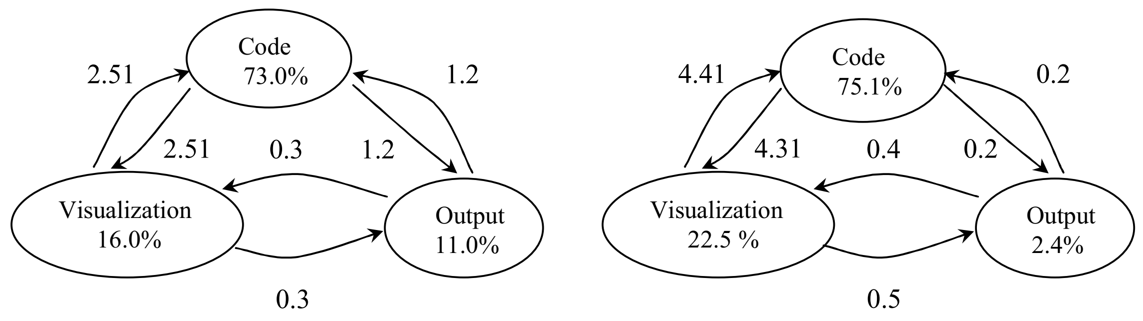


Figure 2 Number of switches per minute and percentage of time spent on each of the representation for participant 10.

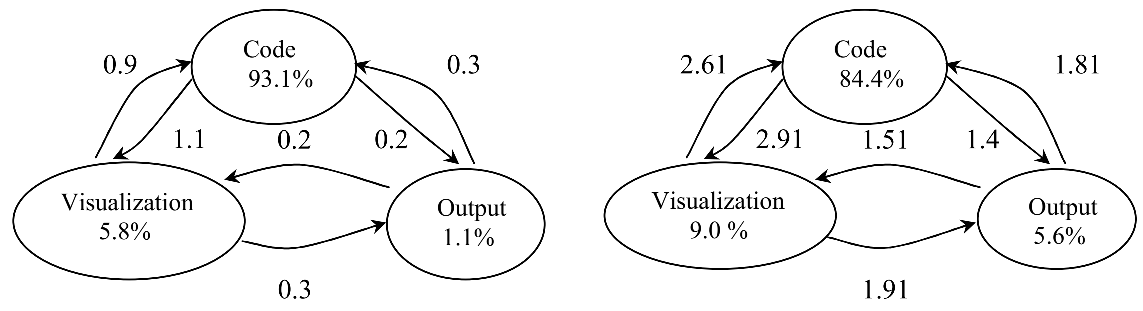


Figure 3 Number of switches per minute and percentage of time spent on each of the representation for participant 9.

From the Figure 2 and Figure 3 it can be observed that the more experienced participant (9) exhibited more balanced behavior in terms of switches per minute between all the representations than less experienced participant (10). These two participants were selected also as the representatives of their experience groups.

Subjective experiences reported by participants

Each subject was debriefed after the experiment. The majority of the experienced participants have stated that the restrictive condition imposed by the RFV does influence their ability to extract all the information needed. For instance, the most successful participant (9) hypothesized that he "was not good with the blur" although he spotted all the errors under both conditions. On the other hand, the majority of novices expressed the feeling that the restricted focus allows them to concentrate more on the particular piece of the code, while they are not influenced by the surrounding lines. Novices often expressed the opinion that using the RFV's restricting condition might bring efficiency into their debugging performance.

Almost all of the participants asked during the warm-up session whether it is possible to modify the program and to correct the error they found. Some of the participants asked whether they could take the notes using pen and paper.

Discussion

This experiment was done in order to verify the results obtained from earlier studies (Romero et al. 2002a, 2002b; Romero et al 2003) using two different tools. The Restricted Focus Viewer is a tool which relates the focus of visual attention to the location of a fully focused area within the blurred stimuli images. The eye tracker measures the location of gaze, which is thought to be tightly connected with the locus of visual attention.

Ten volunteers participated in the experiment, half of them with the experience outreaching six years of active programming. The four experimental conditions used in the study consisted of two target Java programs and two settings of RFV (restricted focus on and off). Each of the programs was seeded with four errors. Participants debugged two different programs, being allowed to debug each program for ten minutes.

The results of debugging performance agree with the findings from earlier studies, the error-finding performance is related to programmers' experience with programming. The more experienced programmers were more successful than the less experienced programmers in finding errors in the programs. The general programming experience determined the expertise of the participants better than their experience with Java programming.

Similarly to earlier studies (Romero et al. 2002a, 2002b; Romero et al 2003), the results for visual attention show that participants spent most of the time looking at the code window. This is not surprising, because the code contains most of the information and is clearly the primary representation of the program. However, our results differed from the earlier results of Romero et al. concerning the distributions of the times between the windows of the SDE. Our participants spent more time on the visualisation window (on average 12.5 % of the total time compared to an average of 8 % of the total time for the data structure visualisation, estimated from Romero et al. 2002a and 2002b). This could be explained by at least two causes. The first is that our results are for a small number of subjects (only 10 subjects compared to 49 subjects in Romero et al. 2002b). We are going to run the experiment with more subjects to find out if this is the case. The second explanation is that the RF tool itself might affect the behaviour of the subjects. In our study we were able to measure the time distributions using the actual eye movements from the participants, the results showed that our participants spent significantly more time on the visualisation window without using the RFV, than using RFV.

The experimental results for switching behaviour agree with the findings of the earlier studies in principle, that the most common type of switch is between the code window and the visualisation window, the second most common between the code window and the output window, and the least common between the visualisation window and the output window. However, when we study the switches at the level of the visual attention-tracking tool used, we see first a significant difference in the average number of switches per minute and second, a significant increase in the number of switches between the visualisation window and the output window in the unrestricted view condition (RFV off) compared to the restricted view condition (RFV on).

The first difference is due to fact that the RFV tool and an eye tracker measure the visual attention, and thus the attention switching, differently. This is illustrated in Figure 4. The blurred image of the visualisation of the data structure used, or the output of the program, could serve as a memory aid for the programmer to recognize the specific working of the program once he/she comprehends it in the unblurred mode. This could explain why the programmer does not have to shift the unblurred spot to the window but can just look at the blurred image.

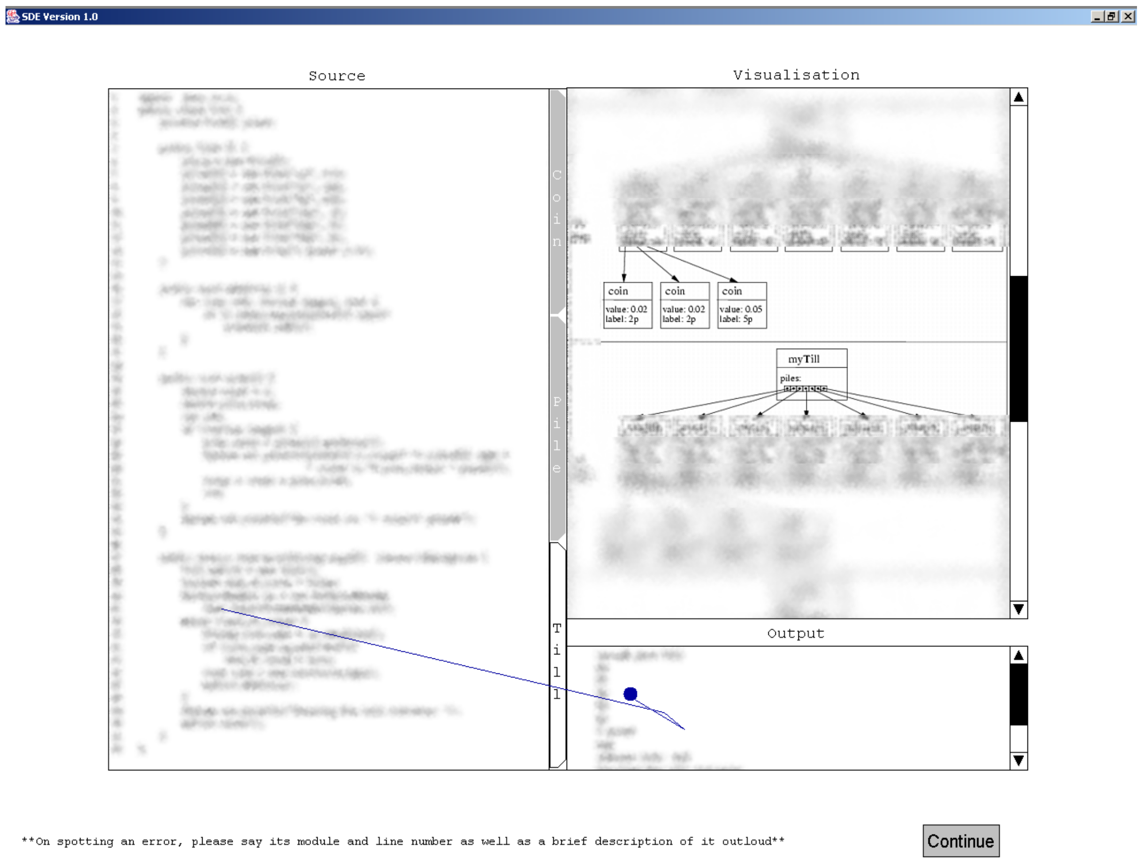
The second difference could have at least two possible explanations. The first one is that the switching has a lower cognitive cost in unblurred mode. The fact that the average number of switches per minute increases in the unrestricted view condition compared to restricted view condition could be seen as support for this. The second explanation is that the RFV tool has an effect on debugging strategy. For some reason, the unrestricted view allures the programmer to test an error hypothesis between the output representation and the visualisation representation. However, the results do not show better debugging performance for the unrestricted view condition than for the restricted view condition.

Several methodological reasons can be found for the differences between the RFV-based results and eye tracking results. The temporal resolution of RFV-based measurement is limited to the movements of the computer mouse. In our experiment, during the ten minutes of debugging a program the RFV collected about 400 data points, while the number of fixations was around 7000. Thus, the eye-tracking equipment provides a finer level of temporal details.

Furthermore, the spatial resolution of fixated areas as measured by RFV can be discussed. As seen from Figure 1 and Figure 4, the size of the focused region ranges far beyond the two degrees of visual angle as limited by the fovea. Therefore, it allows a subject to move his/her eyes within the focused region of RFV without moving the mouse. These changes in focus location are not recorded by RFV.

It has often been the case that the participant working under the restricted focus view has set the focused region onto the visualization and then returns the attention to the code or output without moving the mouse. Figure 4 shows the screenshot of such a situation: the trace of the gaze corresponds roughly to an interval of one second and the diameter of the fixation matches with about 50ms and increases. This behavior consisting of two or more switches between representations is not registered by the RFV.

We also observed that subjects could be divided into two groups when use of the mouse during debugging is considered. Some participants used the mouse as a pointer while reading the source code, while the others had some problems coordinating the hand (mouse-controlled focused spot) and gaze as required by the restricting view. In these subjects the gaze replay reveals that the destinations of the mouse cursor and the gaze are almost always exclusive. This is especially true for large eye movements, when the saccade precedes the movement of the hand. Several observations from the eye movement protocol recorded during the debugging show that mouse position can also precede the gaze. We maintain that the requirement of coordinating hand and gaze during debugging might be one of the reasons why the RFV-on condition affected switching behavior.



On spotting an error, please say its module and line number as well as a brief description of it outloud

Figure 4 The eye gaze trace and a fixation superimposed over the SDE with the restricting focus on

Acknowledgments

The authors would like to thank Jorma Sajaniemi for his deep insight into programmer studies and all kinds of help with this experiment.

References

Blackwell, A. F., Jansen, A.R. and Marriott, K. (2000): Restricted Focus Viewer: A tool for tracking visual attention. In M. Anderson, P. Cheng & V. Haarslev (Eds.), Theory and Applications of Diagrams. Lecture Notes in Artificial Intelligence 1889, pp. 162-177. Berlin, Springer Verlag.

Duchowski, A. T. (2003): Eye Tracking Methodology: Theory & Practice. Springer-Verlag, London, UK.

Godijn, R. & Theeuwes, J.(2003): The relationship between exogenous and endogenous saccades and attention. In J. Hyönä, R. Radach, & H. Deubel (Eds), The Mind's Eyes: Cognitive and Applied Aspects of Eye Movements. Elsevier Science.

Jacob R.J.K. (1995): Eye Tracking in Advanced Interface Design. In W. Barfield and T.A. Furness (Eds.), Virtual Environments and Advanced Interface Design, pp. 258-288, Oxford University Press, New York, (1995).

Jacob, R. J. K., & Karn, K. S. (2003): Eye tracking in human-computer interaction and usability research: Ready to deliver the promises (Section commentary). In J. Hyona, R. Radach, & H.

Deubel (Eds.), The Mind's Eyes: Cognitive and Applied Aspects of Eye Movements. Elsevier Science.

Nevalainen S., Sajaniemi J. (2004): Comparison of Three Eye Tracking Devices in Psychology of Programming Research. Accepted to the 16th Annual Psychology of Programming Interest Group Workshop (PPIG'04).

Rayner, K.(1998): Eye movements in reading and information processing: 20 years of research. Psychological Bulletin 124, 3, pp. 372-422.

Romero, P., Cox, R., du Boulay, B. and Lutz, R. (2002a): Visual attention and representation switching during Java program debugging: A study using the Restricted Focus Viewer. Diagrammatic Representation and Inference : Second International Conference, Diagrams 2002 Callaway Gardens, GA, USA, April 18-20, 2002. Lecture Notes in Artificial Intelligence, 2317, pp. 221-235. Berlin, Springer Verlag.

Romero, P., Lutz, R., Cox, R. & du Boulay, B.(2002b): Co-ordination of multiple external representations during Java program debugging. Empirical Studies of Programmers symposium of the IEEE Human Centric Computing Languages and Environments Symposia, Arlington, VA.

Romero, P., du Boulay, B., Cox, R. & Lutz, R.(2003): Java debugging strategies in multi-representational environments. 15th Annual Workshop of the Psychology of Programming Interest Group (PPIG), Keele University, UK.

Sibert L.E. and Jacob R.J.K. (2000): Evaluation of Eye Gaze Interaction. Proceedings of ACM CHI 2000 Human Factors in Computing Systems Conference, pp. 281-288, Addison-Wesley/ACM Press.

Zhai, S., C. Morimoto & S. Ihde (1999): Manual and Gaze Input Cascaded (MAGIC) Pointing. Proceedings of ACM CHI'99 Conference on Human Factors in Computing Systems, pp. 246-253.

P2.

Bednarik, R., Tukiainen, M.: Visual Attention Tracking During Program Debugging. In Proceedings of NordiCHI 2004, The Third Nordic Conference on Human-Computer Interaction, October 23-27, 2004, Tampere, Finland, ACM Press, pp. 331-334.

Visual Attention Tracking During Program Debugging

Roman Bednarik and Markku Tukiainen
Department of Computer Science, University of Joensuu
P.O. Box 111, FI-80101, Joensuu, FINLAND
{roman.bednarik, markku.tukiainen}@cs.joensuu.fi

ABSTRACT

This paper reports on a study which compared two tools for tracking the focus of visual attention - a remote eye tracker and the Restricted Focus Viewer (RFV). The RFV tool blurs the stimuli in order to simulate human vision; the user controls the portion of the screen which is in focus with a computer mouse. Both tools were used by eighteen participants debugging three Java programs for ten minutes each. The results in terms of debugging accuracy and debugging behavior were compared using the restricting view condition of the RFV and a measuring tool as factors.

The results show that while the debugging performance and the distribution of the time spent on areas of interest (AOI) are not influenced by the restricting view condition, the dynamics of programming behavior is different. The number of switches between the AOIs as measured by the RFV significantly differed from those measured by the eye tracker. Also the number of switches under the restricted and unrestricted RFV condition was significantly different. We maintain that the RFV must be used with caution to measure the switches of visual attention.

Author Keywords
Eye-movement tracking, visual attention, psychology of programming.

ACM Classification Keywords
H5.1. Information interfaces and presentation (e.g., HCI): Evaluation/methodology.

INTRODUCTION

The study of visual attention shifts is one of the approaches to understanding the cognitive processes and reasoning. Computer programmers are a typical class of computer users whose mental processing is influenced by the visual inputs their development environments provide. Programmers use debugging tools which provide several representations of a program in adjacent areas of interest. During program comprehension they use these representations to build up a mental model and during debugging they have to coordinate these representations which typically include the source code of a program, some kind of visualization, and the output of the debugger or program execution. Once we can track the visual attention of programmers interacting with the debugger, we can investigate how they coordinate multiple representations, what strategies they adopt, and what their behavioral patterns are. To arrive at the proper conclusions about behavior, it is important to study the limitations and possibilities of the technologies available.

Related Work

Currently, few tools to track visual attention exist. In recent years, eye-trackers have become common tools for visual attention tracking. Some alternatives to eye trackers, however, have been developed. The Restricted Focus Viewer (RFV) [2] is a visual attention tracking system which displays visual stimuli in a blurred form and allows only a small region to be seen in focus. To get a portion of the stimuli in focus, users have to use a computer mouse to move the focused spot; the focus of visual attention is then thought to be linked to the position of the spot. The RFV tracks the movements of a computer mouse over the stimuli and records them together with the indices of AOIs and timestamps.

In research of tracking the visual attention of programmers, a modified version of RFV was employed in various studies [4, 5, 6]. A Software Development Environment (SDE) was built on top of the RFV and used to track visual attention and investigate the coordination of multiple representations of programmers debugging Java programs. Figure 1 shows a screenshot of the SDE when the restricted condition is on. The code is on the left, the visualization on the top-right, and the output is on the bottom-right. The focused region is displayed over the middle part of the visualization panel.

In our previous study [1], two tools for visual attention tracking were compared: the RFV and the remote eye tracker. Preliminary conclusions indicated that the use of the RFV might affect the behavior of participants, while the debugging performance seemed to be unaffected. Further, our results indicated that for the visual attention tracking in multiple-representation displays, the RFV might not accurately measure the data as expected.

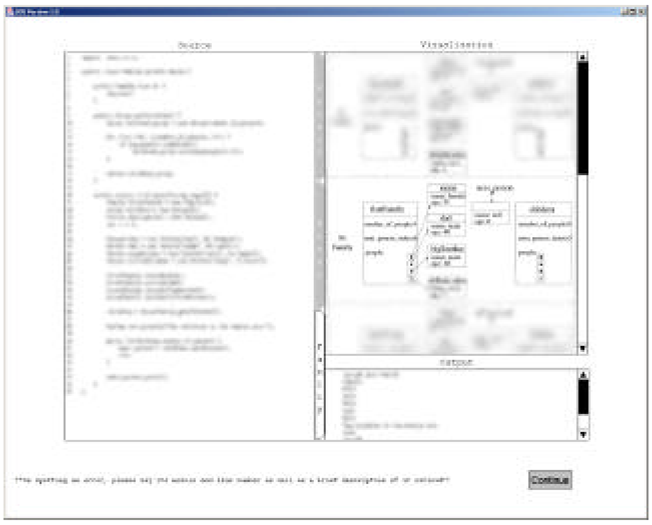


Figure 1. Software debugging environment using RFV.

The aim of the present study was to closely compare two tools for measuring visual attention, the RFV and the eye tracker. Besides, we aimed to further verify the results obtained in a previous study [1], using more participants. Our working hypotheses were: *using the restricted focus view, the RFV-based tool can change the strategies and behavior of a programmer* and *that the RFV does not accurately measures all visual attention switches.*

METHOD

We analyzed and compared the locations of the focus of visual attention as measured by the eye tracker and by the RFV. The RFV-tool links the visual attention focus to the position of a mouse-controlled focused spot in the otherwise blurred stimuli; the eye tracker reports the point of gaze, thought to be linked to the focus of visual attention.

Design and Participants

A totally-within subject design was used with two factors (*RFV restricting condition* and *measuring tool*) and four dependent variables (*errors spotted*, *accumulated fixation time*, *mean fixation duration*, and *switching frequency as measured by RFV and by eye tracker*). The accumulated fixation time is the total time spent during a session the participant is fixating an AOI. For an AOI, all fixations are summed and the number is divided by the total fixation count throughout the experiment, giving the mean fixation duration. The switching frequency refers to the average number of switches per minute between each of the AOIs, as measured by a tracking tool. Most of the results were analyzed by performing ANOVA or paired samples t-tests.

In the study a total of 18 participants were recruited from population of students, researchers, and teachers from the authors' department. All subjects had normal or corrected-to-normal vision and never had taken part in an eye tracking experiment. The average age was 25.3 (SD=4.4) years. Three of participants were females. The programming and Java experience varied from just passing a Java course and having little experience to professionals working in programming related careers. The average programming experience in months was 78.7 (SD=34.7), and the average Java experience was 11.5 (SD=14.8) months.

Procedure

Before the experiment, participants had to pass an automatic eye-tracking calibration procedure. After that, the participants read detailed instruction about the experiment and the environment used. Three programs were debugged. The first warm-up session was performed under the RFV restricted view condition (RFV-on) so that the participant could become familiar with controlling the focused spot and operating the debugging environment. Then, the two main debugging sessions were performed; one session was performed under the RFV-on condition, the other session was performed under the RFV-off condition. The order of the programs and conditions was counterbalanced.

Each session had two phases. First, the specification of the program was displayed which described the problem the program was supposed to solve and the approach to the solution. Two sample interactions were provided - the desired behavior and actual behavior of the program. Second, the participants were given ten minutes to debug the program and were instructed to find as many errors as possible and to report them aloud.

Materials and Setup

The target programs were identical to those used in [4]. The object of the warm-up program was to determine whether a point was inside a rectangle. The first program printed out the names of the children of a sample family and the second program counted the cash in a register till which gave subtotals for the different denominations. In the previous study [4], two versions of the target programs and several visualizations were used. In our experiment, we used the less sophisticated versions of the programs and graphical functional representations. The two main target programs were seeded with four errors each; the warm-up program contained two errors. The programs contained no syntactical error and participants were notified of this.

For the eye tracking, the remote Tobii ET-1750 (sampling at 30Hz) eye tracker was used. The eye tracking data were collected throughout the whole experiment; the RFV collected data only in the RFV-on condition. The AOIs were defined to correspond with the three main panels in the SDE window: the code, visualization, and the output panel.

The software debugging environment (SDE) used in the previous studies [4, 5, 6] was employed for the experiment as a source of stimuli. In these studies and in the present experiment, the program code, the visualization, and output were pre-computed and static.

RESULTS

Debugging performance

The results in terms of debugging performance show that the total number of errors spotted under the RFV-on condition was 46 (mean=2.56, SD=1.10) and under the RFV-off condition was 44 (mean=2.44, SD=1.04) out of maximum 72 errors. There was no significant difference in the average number of errors found between the conditions of restricted and unrestricted view (t(17) = 0.44, *ns*), the grand mean was 2.5 (SD=1.06).

Debugging behavior

The debugging behavior was measured by the eye tracker under the RFV-on/RFV-off and by the RFV-tool under the RFV-on condition. Figure 2 presents the proportion of accumulated time spent on fixating the AOIs as measured by the eye tracker. The effect of the RFV condition on the proportional times spent on the areas of interest was not significant: there was about the same distribution of proportional times regardless of whether the stimuli were blurred or not.

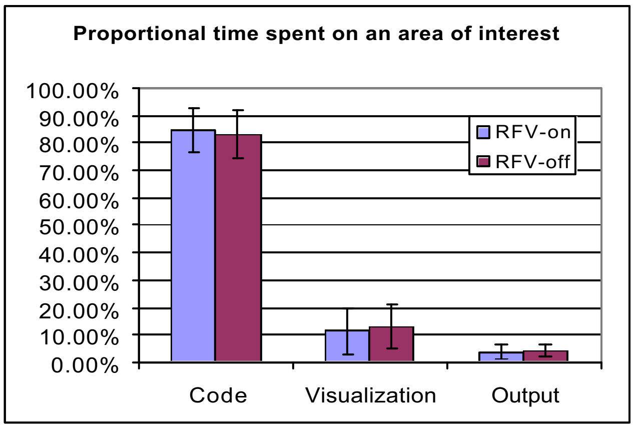


Figure 2. Proportion of time spent on the areas of interest

Figure 3 presents the behavior in terms of number of switches between the AOIs per minute. The results under the RFV-on condition were measured by the RFV-tool. The results under both conditions were measured by the eye tracker. The term "switch" refers to the change of focus between the AOIs, here between the code, visualization, and output panels. Single factor ANOVA was run for the average number of switches. The average number of switches measured significantly differed (F(2,17)=18.37, *p*<.001). Comparing the average number of switches as reported by the measuring tool used under the restricted view condition (RFV-on), there was a significant difference (t(17) = 5.51, *p*<.001) between the RFV and eye tracker. The difference of the number of average switches per minute regarding the condition (RFV-on/RFV-off) was also significant, t(17) =2.42, *p*<.001.

To further study the differences, we analyzed the switching behavior in terms of switches per minute between each of the AOIs. Three two-way ANOVAs revealed a significant effect of the measurement and RFV condition to the number of switches between each AOI (F(2,5)=38.2, *p*<.001), a significant effect of the tool used under the RFV-on condition (F(1,5)=50.7, *p*<.001), a significant interaction between the tool used and the number of switches between each of the AOIs under the RFV-on condition (F(1,5)=5.3, *p*<.001), and a significant effect of the restricting view condition to the number of switches between each of the AOIs as measured by the eye tracker (F(1,5)=9.3, *p*<.01). The numbers of switches per minute between each of the AOIs under as measured by the RFV-tool and by the eye tracker revealed were not correlated.

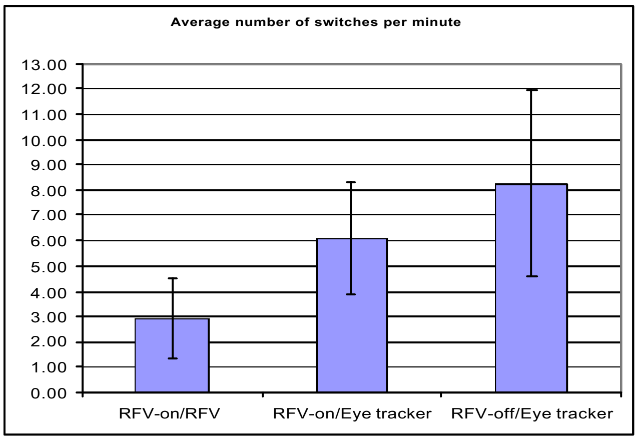


Figure 3. Average number of switches as measured by the RFV under RFV-on, and as measured by the eye tracker under RFV-on and RFV-off

The mean fixation durations measured by the eye tracker are shown in Figure 4. They were obtained by dividing the accumulative duration by the fixation count for each AOI. The two-way ANOVA revealed a significant difference in the mean fixation duration between the individual AOIs (F(1,2)=6.2, *p*<.01); however, the effect of the condition was not significant (F(1,2)=1.9, *ns*). All mean fixation durations under the RFV-off condition were lower than under the RFV-on condition, the effect of RFV condition was nearly significant for the output panel (t(17)= 1.62, *p*<ns) and significant for the overall mean fixation duration (t(17) = 2.09, *p*<.01). The overall mean fixation durations were also significantly correlated (r(18) = 0.857, *p*<.01) between the conditions.

DISCUSSION

The purpose of this experiment was to compare two tools for tracking visual attention measured by the eye tracker and by the RFV tool. The RFV relates the focus of visual attention to the location of a fully focused area within the blurred stimuli images; the eye tracker reports the coordinates of point of gaze which is thought to be connected to the focus of visual attention.

The results show that the error-finding performance and distribution of the fixation times is not influenced by presenting the stimuli in a blurred form. The distributions of the times spent over AOIs as reported by the eye tracker show the consistent pattern of behavior and confirm the results obtained in other studies.

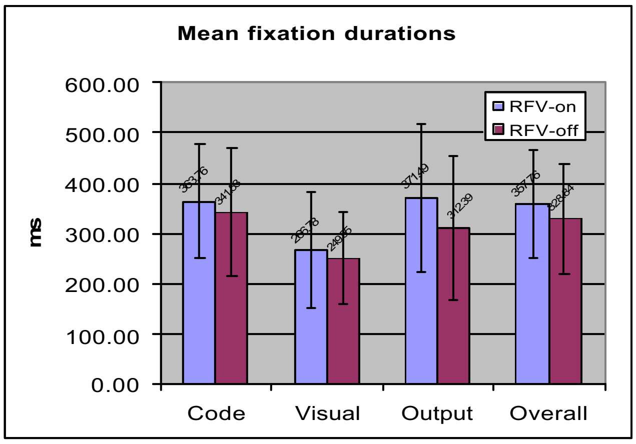


Figure 4. Mean fixation duration for each area of interest

The dynamics of the switching behavior as reported by the RFV tool and by the eye tracker differ significantly. The eye movement protocols may indicate where these differences stem from. Our explanation is that the participants often briefly look onto the blurred areas without moving the focused spot. Therefore, the RFV can not register these changes which most probably serve as a refresher of the mental images of the stimuli. A typical situation of the problem is shown in Figure 5. The participant has changed the visual focus and is fixating on the blurred right-bottom output panel, as indicated by two saccades and following fixation, while the focus as measured by the RFV is in the middle of the code on the left. The second difference is clear when comparing the number of switches with and without the restricted focus view. Participants changed the focus of visual attention more often when the stimuli were presented in focus.

The distribution of mean fixation duration shows a decrease in the duration while the stimuli are unrestricted and a significant decrease of fixation duration over the AIO containing the visualization. In some studies, e.g. [3], the fixation duration mean is thought to be related to the participants' difficulty with extracting the information. This result needs to be analyzed in further studies.

CONCLUSION

We compared two tools for tracking visual attention, the Restricted Focus Viewer and the remote eye tracker, to investigate their possibilities and limitations in the context of software debugging environment. We investigated the data provided by these tools and analyzed the changes in the behavior of participants when the stimuli are presented in a blurred form.

The results indicate that blurring of the stimuli does not influence the error-finding performance of participants and the distribution of times spent on the areas of interest. However, the results show that there are changes in the dynamics of the switching behavior and mean fixation duration. Under the restricted view condition the RFV-tool reports different dynamics in the attention switching than that is measured by the eye tracker

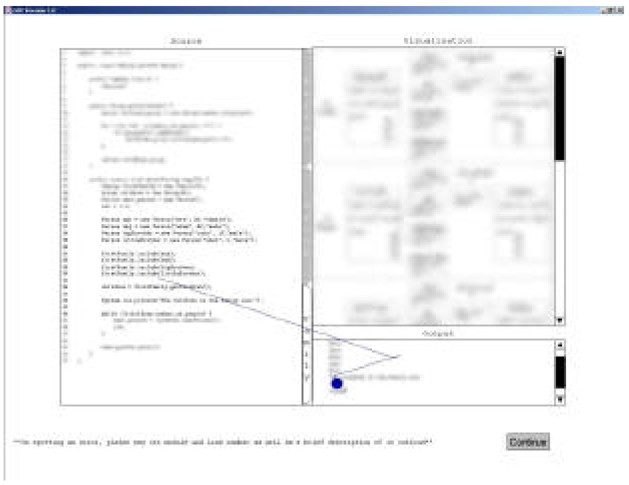


Figure 5. Debugging under RFV-on

REFERENCES

1. Bednarik, R., Tukiainen, M. Visual attention and representation switching in Java program debugging: A study using eye movement tracking. *16th Annual Psychology of Programming Interest Group Workshop (PPIG'04)*, pp.159-169, 2004.
2. Blackwell, A. F., Jansen, A.R., Marriott, K. Restricted Focus Viewer: A tool for tracking visual attention. In M. Anderson, P. Cheng & V. Haarslev (Eds.), *Theory and Applications of Diagrams*. Lecture Notes in Artificial Intelligence 1889, pp. 162-177, Springer Verlag, 2000.
3. Goldberg, J. H. and Kotval, X. P. Eye Movement-Based Evaluation of the Computer Interface. In Kumar, S. K. (Eds.), *Advances in Occupational Ergonomics and Safety*, Amsterdam: IOS Press, pp. 529-532, 1998.
4. Romero, P., Cox, R., du Boulay, B., Lutz, R. Visual attention and representation switching during Java program debugging: A study using the Restricted Focus Viewer. *Diagrammatic Representation and Inference: Second International Conference, Diagrams 2002* Callaway Gardens, USA. Lecture Notes in Artificial Intelligence, 2317, pp. 221-235, Springer Verlag, 2002.
5. Romero, P., Lutz, R., Cox, R., du Boulay, B. Co-ordination of multiple external representations during Java program debugging. *Empirical Studies of Programmers symposium of the IEEE Human Centric Computing Languages and Environments Symposia*, Arlington, VA, 2002.
6. Romero, P., du Boulay, B., Cox, R., Lutz, R. Java debugging strategies in multi-representational environments. *15th Annual Workshop of the Psychology of Programming Interest Group (PPIG'03)*, 2003.

P3.

Bednarik, R., Tukiainen, M.: Effects of Display Blurring on the Behavior of Novices and Experts during Program Debugging. In CHI ’05 extended abstracts on Human factors in computing systems, CHI 2005, Portland, Oregon, USA, April 2-7, 2005, ACM Press, pp. 1204-1207.

Effects of Display Blurring on the Behavior of Novices and Experts during Program Debugging

Roman Bednarik
Department of Computer Science
University of Joensuu
P.O. Box 111, FIN-80101, Joensuu, Finland
bednarik@cs.joensuu.fi

Markku Tukiainen
Department of Computer Science
University of Joensuu
P.O. Box 111, FIN-80101, Joensuu, Finland
mtuki@cs.joensuu.fi

ABSTRACT
The Restricted Focus Viewer (RFV) relates a small part of an otherwise blurred display to the focus of visual attention. A user controls which part of the screen is in focus by using a computer mouse. The RFV tool records these movements. Recently, some studies used the RFV to investigate the cognitive behavior of users and some others have even enhanced the tool for research of usability issues.

We report on an eye-tracking study where the effects of RFV's display blurring on the visual attention allocation of 18 novice and expert programmers were investigated. We replicated a previous RFV-based study and analyzed attention switching and fixation durations reported by an eye tracker. Our results indicate that the blurring interferes with the strategies possessed by experts and has an effect on fixation duration: however, we found that debugging performance was preserved. We discuss possible reasons and implications.

Author Keywords
Eye-movement tracking, visual attention, psychology of programming, experimental tools

ACM Classification Keywords
H5.1. Information interfaces and presentation (e.g., HCI): Evaluation/methodology

INTRODUCTION

In fields of HCI research such as usability, the psychology of programming, or diagram-understanding, it is important to investigate the limitations and impacts of tools used for collecting user behavior as indicators of the cognitive processes of users. We are highly interested in whether the actual use of tools does interfere with the (otherwise unaltered) behavior of participants in an experiment.

Computer programming and research into the related cognitive processes is typically a domain where researchers benefit from applying tools to investigate visual attention allocation. A great amount of research in the past has consistently confirmed the relations between shifts of visual attention focus and movements of the eyes; see [4] for a review.

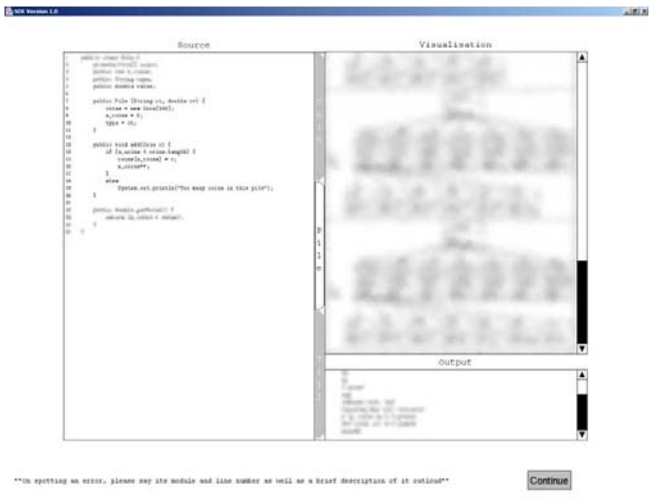
Related Work

The Restricted Focus Viewer (RFV) [6, 2] is a visual attention tracking system which displays visual stimuli in a blurred form and allows only a small region of the screen to be seen in focus. To get a portion of the stimuli in focus, users have to move the computer mouse over the area that they want to come into focus ; the focus of visual attention is then thought to be linked to the position of the spot. The RFV tracks the movements of a computer mouse over the stimuli and records them together with the indices of areas of interest (AOI, e.g. the predefined areas of display stimuli) and timestamps.

In the context of Java program debugging, a modified version of RFV was employed in various studies [7, 8, 9]. In these studies, a Software Development Environment (SDE) was built on top of the RFV and used to track visual attention and to investigate the coordination of multiple representations of programmers debugging Java programs. Figure 1 shows a screenshot of the SDE when the restricted condition is on. The SDE's interface contains three AOIs: the code is on the left, the visualization on the top-right, and the output is on the bottom-right. The focused region, as set by user's mouse, is located over the code in left panel.

The RFV was also employed in the research of shifts of visual attention during integration of text and graphics [3]. Other researchers used the RFV idea for usability studies of hyperlinked documents [10].

In our previous report [1], we showed that RFV's blurring condition introduces interference to the cognitive strategies of programmers: representation (attention) switching was inhibited and the mean fixation duration decreased. An interesting and important question could be raised: "What is the effect of this intervention on different classes of experimental participants, when the experience is considered as a study criterion?" Our hypothesis is that with *greater experience the actual usage of a tool does not interfere with strategies possessed.*



On spotting an error, please say its module and line number as well as a brief description of it outloud

Figure 1. A screenshot of SDE used in experiment. Focused spot is located in the code on left.

METHOD

We analyzed and compared the locations of the focus of visual attention between two conditions and two levels of experience. The SDE interface, based on the RFV-tool, was used to present stimuli blurred (RFV-on) and to present stimuli unblurred (RFV-off). Two groups of programmers (a novice and an expert group) worked with the environment to debug three Java programs. The visual attention was recorded using an eye-tracker.

Design and Participants

A mixed one within-subject (*RFV restricting condition*), one between-subject (*a level of experience*) design was used with four dependent variables (*number of errors spotted*, *accumulated fixation time, mean fixation duration*, and *switching frequency as measured by eye tracker*). The accumulated fixation time is the total time a participant spent during a session fixating an AOI. For an AOI, all fixations were summed and the number was divided by the total fixation count throughout the experiment, giving the mean fixation duration. The switching frequency refers to the average number of switches per minute between each of the AOIs. Most of the results were analyzed by performing ANOVA and/or planned paired t-tests.

In the study a total of 18 participants were recruited from a population of students, researchers, and teachers from the authors' department. All subjects had normal or corrected-to-normal vision according, by their own report, and had never taken part in an eye-tracking experiment. The average age was 25.3 (SD=4.4) years. Three participants were females. The programming and Java experience varied from having just passed a Java course and having little experience to professionals working in programming-related careers. The less-experienced group consisted of 10 programmers, who had an average of 63 months of programming experience, 8.13 months of which were Java programming. No novice participant had ever worked as a professional programmer. The expert group was formed from the remaining 8 participants, whose programming experience was 96 months, whose Java experience was 16.25, and who all, except one, had professional experience with programming.

Procedure

Before the experiment, participants had to pass an automatic eye-tracking calibration procedure. After that, the participants read detailed instructions about the experiment and the environment used. Three programs were debugged. The first warm-up session was performed under the RFV restricted view condition (RFV-on) so that the participants could become familiar with controlling the focused spot and operating the debugging environment. Then, the two main debugging sessions were performed; one session was performed under the RFV-on condition, the other session was performed under the RFV-off condition where the whole display was presented in focus. The order of the programs and conditions was counterbalanced.

Each session had two phases. First, the specification of the program was displayed. It described the problem the program was supposed to solve and the approach to the solution. Two sample interactions were provided - the desired behavior and actual behavior of the program. Second, the participants were given ten minutes to debug the program and were instructed to find as many errors as possible and to report them aloud.

Materials and Setup

The target programs were identical to those used in [7]. The object of the warm-up program was to determine whether a point was inside a rectangle. The first program printed out the names of the children of a sample family and the second program counted the cash in a register till which gave subtotals for different denominations. In a previous study done by Romero et al. [7], two versions of the target programs and several visualizations were used. In our replication of the experiment, we used Romero's less sophisticated versions of the programs and graphical functional representations. The two main target programs were seeded with four errors each; the warm-up program contained two errors. The programs contained no syntactical error and participants were notified of this.

For eye-tracking, the remote Tobii ET-1750 (sampling at 30Hz) eye tracker was used. The eye tracking data were collected throughout the whole experiment; the RFV collected data in the RFV-on condition, but for the purposes of this study this data was not used. The AOIs were defined to correspond with the three main panels in the SDE window: the code, visualization, and output panel.

The Software Debugging Environment (SDE) used in the previous studies [7, 8, 9] was employed for the experiment as a source of stimuli. In these studies and in the present experiment, the program code, the visualization, and output were pre-computed and static.

RESULTS

Debugging performance

The debugging performance was measured by the number of errors spotted. Under the RFV-on condition, the less experienced group found 2.1 (SD=1.10) errors on average and the more experienced group spotted 3.125 (SD=0.84) errors on average: t(7) = 2.53, p<*.05*. Under the RFV-off condition, the less experienced group found 2.1 (SD=0.88) errors on average, and the more experienced group spotted 2.88 (SD=1.13) errors on average. The effect of the restricted view condition on debugging performance was not significant.

Gaze related behavior

Figure 2 presents the mean fixation durations for each of the three main areas of interest and the overall mean fixation duration. A two way ANOVA revealed an effect of RFV condition on mean fixation duration (F(1,16) = 4.45, p<*.051*) and no interaction between level of experience and RFV condition (F(1,16) = 0.26, *ns*). The planned paired t-tests revealed that, for experts, the overall mean fixation duration and the mean fixation durations over the code AOI significantly differed between RFV-on and RFV-off conditions (t(7) = 2.80, t(7) = 2.66, respectively, all p<*.05*). The overall mean fixation durations of the experts were 308.82 ms (SD=83.95) and 263.09 ms (SD= 70.60) under RFV-on and RFV-off, respectively. For the code panel, the mean fixation durations of the expert group were 312.44 ms (SD=85.69) and 268.23 ms (SD= 73.64) under RFV-of and RFV-off, respectively. Considering novices, there was no significant difference in fixation durations between RFV-on and RFV-off conditions according to pair-wise tests. However, the mean fixation duration between the areas was significantly different, F(2,16) = 10.13, p<*.005*).

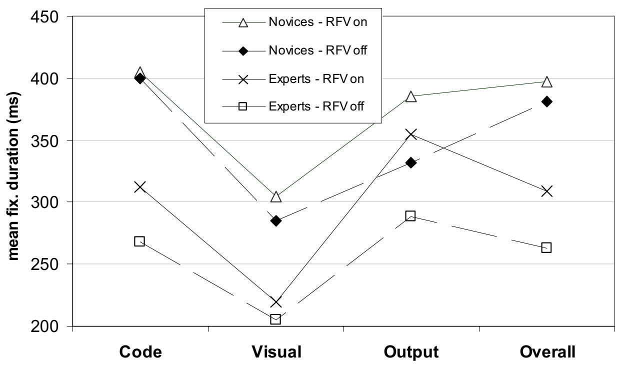


Figure 2. Mean fixation durations over the main panels of interface and overall mean fixation duration.

The distribution of relative accumulated fixation time over the areas of interest was not affected by the RFV condition for either of two experimental groups. Novice participants spent on average 82% of whole time fixating on the code panel, 14% over visualization, and 4% of total time over the output area of interest. For experts the relative accumulated fixation time followed distribution 87%, 10%, and 3%.

The dynamics of attention switching behavior was measured by the average number of switches per minute between any two of all areas of interest (Figure 3). The effect of RFV condition was significant, F(1,16) = 7.82, p<*.05*), and the interaction between the level of experience and RFV condition was significant at an alpha of 0.92, F(1,16) = 3.59, p<*.08*). We observed a decrease in the number of switches per minute, which was significant for experts (t(7) = 2.53, p<*.05*). More over, the average number of switches per minute of novices was significantly correlated under RFV-on and RFV-off (r(10) = 0.642, p=*.046*), while the same correlation for experts was low and not significant (r(8) = 0.068, p=*.873*).

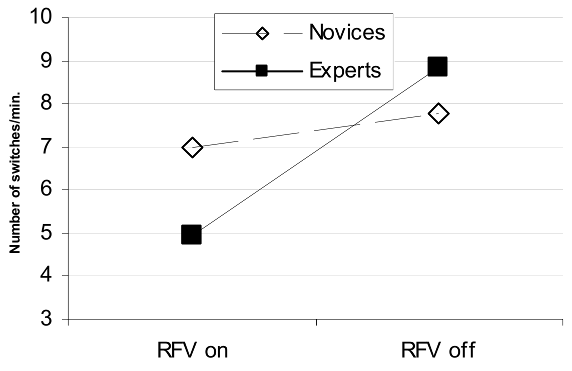


Figure 3. Number of switches per minute.

DISCUSSION AND CONLUSIONS

The purpose of this experiment was to investigate the effects of RFV's display blurring on the behavior of experimental participants. The RFV displays a focused spot within otherwise blurred stimuli images; the spot is controlled using a computer mouse. An eye tracker reports the coordinates of point of gaze which is thought to be connected to the focus of visual attention. We used an eye tracker to measure (1) the visual attention location of participants, (2) the accumulated fixation time over areas of interest, and (3) the fixation duration while debugging using the RFV based environment. We replicated one of the previous studies which used the RFV as a tool to measure visual attention switching.

The accumulated fixation time distributions of either novices or experts were not affected by the RFV's restricted view. This indicates that an RFV-based tool does not interfere with this measure; moreover, it means that participants spend the same amount of time fixating the areas of interest.

The mean fixation durations of all participants were increased under the RFV-on condition; for experts the effect led to a significant increase under the RFV-on condition. In eye movement based studies, the mean fixation duration is a measure of processing, which is related to the depth of required processing [5]. The RFV's blurred display caused our study's experts to process visual information longer that it was under the unrestricted view.

The RFV makes switching a manual task rather than a perceptual one. When a display is blurred the natural switching frequency decreases. In our study, the effect of the blurring was more significant for experts than for novices. As also seen from the correlations, novices' strategies seemed to be almost unaffected, while experts' behavior was different when the display was blurred.

These results indicate that experts are most probably processing much information through peripheral vision during debugging and the blurring is creating an obstacle causing the processing to take longer. This hypothesis is supported by a look into the video protocols. It is common that a participant places the focused point over the investigated piece of interface, while visually attending to some other, blurred part of the interface. The focused spot is therefore functioning as a kind of bookmark, but not as a single spot through which the information is exclusively extracted.

Despite the fact that RFV does not interfere with debugging performance, we conclude that it creates several effects on the behavior of experimental participants. The effects, in our study, were more serious for a group consisting of participants with higher experience levels. Since some researchers continue the idea of measuring visual attention allocation by display blurring, our results provide a warning: some conclusions based on a behavior measured using the RFV might be inaccurate.

ACKNOWLEDGMENTS
The work of first author was supported by a grant of Faculty of Science, University of Joensuu, Finland.

We would like to thank Pablo Romero for sharing the materials used in this study.

REFERENCES

1. Bednarik, R., Tukiainen, M. Visual attention tracking during program debugging. In *Proceedings of NordiCHI 2004, The Third Nordic Conference on Human-Computer Interaction*, October 23-27, 2004, Tampere, Finland, ACM Press, pp. 331-334.
2. Blackwell, A. F., Jansen, A.R., Marriott, K. Restricted Focus Viewer: A tool for tracking visual attention. In M. Anderson, P. Cheng & V. Haarslev (Eds.), *Theory and Applications of Diagrams*. Lecture Notes in Artificial Intelligence 1889, pp. 162-177, Springer Verlag, 2000.
3. Futrelle, R. P. & Rumshisky, A. Discourse Structure of Text-Graphics Documents. *1st International Symposium on Smart Graphics* Hawthorne, NY. ACM Press, 2001.
4. Godijn, R. & Theeuwes, J. The relationship between exogenous and endogenous saccades and attention. In Jukka Hyönä, Ralph Radach & Heiner Deubel (Eds). *The Mind's Eyes: Cognitive and Applied Aspects of Eye Movements*, pp. 3-26, 2003.
5. Goldberg, J. H. and Kotval, X. P. Eye Movement-Based Evaluation of the Computer Interface. In Kumar, S. K. (Eds.), *Advances in Occupational Ergonomics and Safety*, Amsterdam: IOS Press, pp. 529-532, 1998.
6. Jansen, A.R., Blackwell, A.F. and Marriott, K. A tool for tracking visual attention: The Restricted Focus Viewer. *Behavior Research Methods, Instruments, and Computers,* 35(1), 57-69, 2003.
7. Romero, P., Cox, R., du Boulay, B., Lutz, R. Visual attention and representation switching during Java program debugging: A study using the Restricted Focus Viewer. *Diagrammatic Representation and Inference: Second International Conference, Diagrams 2002* Callaway Gardens, USA. Lecture Notes in Artificial Intelligence, 2317, pp. 221-235, Springer Verlag, 2002.
8. Romero, P., Lutz, R., Cox, R., du Boulay, B. Co-ordination of multiple external representations during Java program debugging. *Empirical Studies of Programmers symposium of the IEEE Human Centric Computing Languages and Environments Symposia*, Arlington, VA, 2002.
9. Romero, P., du Boulay, B., Cox, R., Lutz, R. Java debugging strategies in multi-representational environments. *15th Annual Workshop of the Psychology of Programming Interest Group (PPIG'03)*, pp. 421-434, 2003.
10. Tarasewich, P., Fillion, S. Discount eye tracking: The Enhanced Restricted Focus Viewer. In *Proceedings of AMCIS*, August 2004, New York, pp. 1-9.

P4.

Bednarik, R., Myller, N., Sutinen, E., Tukiainen, M.: Effects of Experience on Gaze Behaviour during Program Animation. In Proceedings of the 17th Annual Psychology of Programming Interest Group Workshop (PPIG'05), Brighton, UK, June 28 - July 1, 2005, pp. 49-61.

Effects of Experience on Gaze Behavior during Program Animation

Roman Bednarik, Niko Myller, Erkki Sutinen, and Markku Tukiainen

Department of Computer Science, University of Joensuu,
P.O. Box 111, FI-80101 Joensuu, FINLAND
`firstname.lastname@cs.joensuu.fi`

Abstract. The purpose of program visualization is to illustrate some aspects of the execution of a program. A number of program visualization tools have been developed to support teaching and learning of programming, but only few have been empirically evaluated. Moreover, the dynamics of gaze behavior during program visualization has not been investigated using eye movements and little is known about how program animation is attended by learners with various levels of experience. We report on an empirical study of the gaze behavior during a dynamic program animation. A novice and an intermediate group, a total of 16 participants, used Jeliot 3, a program visualization tool, to comprehend two short Java programs. Referring to previous literature, we hypothesized that the performance as well as the gaze behavior of these two groups would differ. We found statistically significant differences in performance measures and in fixation durations. Other commonly used eye-tracking measures, the fixation count and the number of attention switches per minute, seem to be insensitive to the level of experience. Based on the results, we propose further directions of the research into gaze behavior during program visualization.

1 Introduction

Program visualization is used to illustrate visually the run-time behavior of computer programs. These systems can be utilized, for example, in programming courses to support teaching of programming concepts to novice programmers. Jeliot 3 is an interactive program visualization system that automatically visualizes data and control flows of Java programs. It has been successfully used in classroom settings to teach programming to high school students [1].

Although several program visualization tools exist, only few have been evaluated and little knowledge is available about the aspects of gaze behavior during a dynamic program visualization. It is not clear how different users attend the animation and what cognitive efforts they have to exercise in order to comprehend the dynamic visualization. Therefore, in order to improve program visualization systems to fit their users best, it is a crucial issue to investigate the visual attention paths of users while visualizing a program. If a purpose of program visualization is to support the novices in their understanding, it is reasonable to study how their behaviors differ from the behaviors of intermediates. In other domains, eye-movement tracking has been successfully applied to investigate the gaze patterns of participants while performing their tasks. However, no eye-movement based analysis of the gaze behavior during a dynamic program visualization has been conducted yet.

We report on an initial study in which we have employed a remote eye tracker to measure the gaze behavior of programmers during program comprehension facilitated by an animation tool, Jeliot 3.

The rest of the paper is arranged as follows. In Section 2, we review some related work in eye tracking research and program visualization, and Jeliot 3 is introduced. The experiment and results are described in Sections 3 and 4, respectively, and discussed in Section 5. Conclusions and future work are presented in Section 6.

2 Related Work

2.1 Eye Tracking

Humans move their eyes in order to bring an inspected object or a portion of it onto fovea, the high-resolution area of retina. This way the visual attention is closely linked with the direction of the eye-gaze, and most of the time it is also diverted to the point of visual inspection. Following this assumption, if we can track the movements of eyes, we can also get insights into and investigate the path and focus of attention during a task such as program comprehension. Furthermore, knowing which objects have been visually inspected and in which order and context, we can attempt to infer what cognitive processes were involved to perform the task related to these objects.

Eye tracker is a device that records eye movements. Most of the current eye trackers use infrared light emitters and video image analysis of the corneal reflections and pupil center to relate them to the direction of gaze. Typically, the accuracy of current eye trackers ranges around 1 degree, while the data is sampled at rates of 50–500Hz. Current eye trackers are relatively cheap and able to reliably and unobtrusively collect gaze data.

From the signal obtained from an eye tracker, two most important types of eye movements are usually identified: saccades and fixations [2]. *Saccades* are rapid ballistic movements of eyes that are executed to reposition the eyes from one location of attention to another one. A single saccade can last between 30 and 120 ms, can span over 1 to 40 degrees of visual angle [2], with velocities ranging up to 500 degrees per second [3]. No visual information is extracted during a saccade, a phenomena called saccadic suppression [4]. *Fixations* are eye movements stabilizing the image of an object on the retina. Typical fixation duration ranges between 200–300 ms [3]. It is assumed that during the period of a single fixation the information is extracted, decoded, and interpreted. The fixation duration can be therefore thought to be related with a required processing to extract and interpret the information [5, 6]. An accurate measurement and analysis of eye movements in terms of saccades and fixations provide researchers with the details of cognitive processing and related visual attention allocation within a performed task. For instance, the fixation count or sum of fixation durations on a certain element can be related to the importance of the element. In the context of program visualization interfaces, the relative fixation count measure can correspond with the relative importance of a representation (e.g. a code or a state diagram) of a program.

It is a well-known fact that eye movement patterns of experts and novices differ. Previous eye movement studies in other domains than program visualization have shown, for instance, that (1) search strategies differ between novice and expert radiologists [7], (2) expert-pilots' eye movement patterns were better defined and the dwell times were significantly shorter than those of novices [8]. A common denominator in these and other reports is that domain knowledge and experience of participants seem to be the main factors influencing not only the performance, but also the related gaze behavior.

Visual attention tracking during program comprehension has been previously studied by Crosby and Stelovsky [9]. They used an eye tracker to discover the relationship between cognitive styles and individual differences, and code-reading patterns. In their study, novices and experts were eye tracked during an algorithm comprehension. However, only one representation of program was used (the code) and the focus of the research was mainly on the critical, but surface features of code, not on the behavior during a dynamic program visualization.

In the direction of investigating issues such as visual attention switching or a multiple-representation use during program comprehension or debugging, previous studies involved only a static precomputed stimuli and the analysis was based on a recording of mouse movements over a blurred interface [10, 11]. The validity of such an approach was shown to be questionable [12, 13]. To our knowledge, no eye movement based analysis of behavior during program animation has been conducted yet. This is certainly surprising, considering the importance of knowledge how the visual attention and cognitive processes involved in program comprehension are influenced by program animation.

2.2 Program Visualization

A number of program visualization systems have been developed over the previous years to teach programming or to visually debug programs. Here we will briefly review those systems that in some aspects are similar to Jeliot, the program visualization tool employed in the present experiment.

Javavis [14] is a tool that visualizes automatically the runtime behavior of the Java programs. It shows changes in the state of the program during execution using animated UML-like object and sequence diagrams. DDD [15], a debugging front-end, uses diagrams to illustrate the references between data structures during program execution. The diagram can be seen as graphs where nodes are the separate data structures (e.g. struct in C) and vertices are the references between them. The DDD does not explicitly visualize the control flow of the program. Jive [16] uses a similar approach to Javavis and DDD to visualize the program state using diagrams. The references, primitive values and variables are visualized similarly in Jeliot 3 and these systems. However, only Javavis visualizes control flow, but in less detail compared to Jeliot 3.

PlanAni [17] is a program visualization system that illustrates the data flow of a program during its execution. The use of variables in different purposes is illustrated through the roles of variables. The expression evaluation and control flow are also visualized. Currently, the animations must be programmed beforehand by an instructor and the visualization of object-oriented concepts is not supported. The organization of the user interface in PlanAni is similar to Jeliot. However, Jeliot does not visualize the roles of variables as PlanAni and PlanAni does not visualize the control flow.

2.3 Jeliot 3

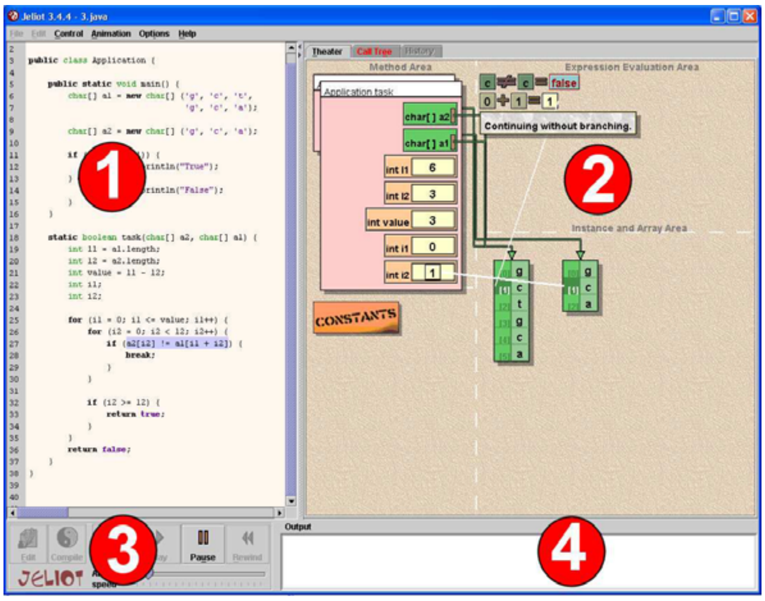
Moreno at al. [18] have developed a program visualization system, called Jeliot 3. Its predecessor, Jeliot 2000, has been successfully used to improve the teaching of introductory programming helping the novices to acquire vocabulary to explain programming structures and concepts [1]. Jeliot 3 retains the novice-oriented GUI and animation display of Jeliot 2000. Jeliot 3 introduced a new design in order to make the system extensible and to allow for adding new features into the visualization. It visualizes automatically the execution of user-written Java programs by illustrating the data and control flow and object-oriented features of the program. Jeliot 3 can visualize a large subset of novice-level Java programs (see `http://cs.joensuu.fi/jeliot/`). The user interface of Jeliot 3 is shown in Figure 1.

The interface consists of four discrete areas. A code editor on the left hand side shows the program code, and during program visualization, the currently executed statement or expression is highlighted. A control panel in the bottom left corner is used to control the animation with VCR-like buttons. The largest area of the user interface of Jeliot is occupied by the visualization view showing the execution state of the program on the right hand side of the window. Visualization consists of method frames, local variables, expression evaluation, static variables, objects and arrays. Finally, an output console lies in the bottom right corner of the window, showing the output of the executed program. To sum it up, Jeliot provides four different areas of interest to the user: code view, animation view, control panel, and output console. Moreover, animation view is further divided into four different areas of interest: method, expression evaluation, constant, and object and array areas. Furthermore, there are separate specialized visualizations where only the call tree of the program or the execution history are shown.

In a typical session with Jeliot, a user either writes or loads a previously stored program. User can compile the program through the user interface of Jeliot. When the program is compiled, a visualization view, where the user can see the animation of the program execution, is opened. Jeliot shows the execution either step by step or continuously. User can control the speed of the animation and stop or rewind the animation at any point. User can select the current visualization with the tabs on top of the visualization view.

3 Experiment

The present research investigates the differences in the gaze behavior during program animation of participants with different levels of programming experience. Based on the results from available literature, our hypothesis was that the performance and gaze behavior of novices and intermediates differ during the program animation. In other words, our aim was to answer the question, whether intermediates and novices pay attention to the animation in a similar or different way. Our hypothesis is not surprising, since we naturally assume that a different level of experience shall result into a different gaze behavior and performance, as it has been found in other domains. More experienced programmers are expected to form better hypotheses about the problem and this knowledge should guide them to use the available representations in a distinct


Jeliot 3.4.4 - 3.java
File Edit Control Animation Options Help
Theater Call Tree History
Method Area
Expression Evaluation Area
Application.task
Continuing without branching.
Instance and Array Area
CONSTANTS
Edit Compile Pause Rewind
Animation speed
Output
1
2
3
4

```
public class Application {

    public static void main() {
        char[] a1 = new char[] {'g', 'c', 't',
                                'g', 'c', 'a'};

        char[] a2 = new char[] {'g', 'c', 'a'};

        if [illegible])) {
            [illegible]rintln("True");
        }
            [illegible]rintln("False");
        }
    }

    static boolean task(char[] a2, char[] a1) {
        int l1 = a1.length;
        int l2 = a2.length;
        int value = l1 - l2;
        int i1;
        int i2;

        for (i1 = 0; i1 <= value; i1++) {
            for (i2 = 0; i2 < l2; i2++) {
                if (a2[i2] != a1[i1 + i2]) {
                    break;
                }
            }

            if (i2 >= l2) {
                return true;
            }
        }
        return false;
    }
}
```

Fig. 1. User interface of Jeliot 3. Area 1 is code editor, area 2 is animation frame, area 3 is control panel and area 4 is output console.

way, compared to novices. We had a further assumption that novices would rely more on the visualization than code and the other way around for intermediates.

To validate these hypotheses, we conducted an empirical experiment where we used a remote eye-tracker to record the gaze behavior of the participants during program comprehension task aided by an animation. Two groups of participants with different level of experience used Jeliot 3 to comprehend three short Java programs while their eye movements were simultaneously tracked.

3.1 Method

We used a between-subject design with experience (novice or intermediates) as the factor. The depended variables were: relative fixation count over the areas of interest, number of switches per minute and mean fixation duration over the areas of interest and in overall. The fixation count is a measure related to the level of participant's interest in an area. The number of switches per minute is a measure of attention allocation dynamics. The mean fixation duration is associated with the depth of processing required to understand an attended element. Only the gaze data during the program animation were used in this analysis because that is the only time when all the representations were available concurrently and the selection of the attended representation would make a difference in understanding the program. Most of the analysis was carried out using ANOVA and planned comparisons based on t-test.

3.2 Participants

Eighteen participants were recruited from high-school students attending a university-level programming course, undergraduate and graduate computer science students from local university. Due to technical problems with the eye tracking, data from two participants had to be discarded. Therefore, the results are based on the data collected from 16 subjects (13 male, 3 female). Participants were divided into two groups according to their level of programming experience. Participants with less than 24 months of programming experience were regarded as novices and above 24 months as intermediates. The characteristics of the two groups are presented in Table 1. Groups' mean values for programming experience (in months) and Java experience (in months) and counts for previous experience with Jeliot 3 (yes=1, no=0) and previous experience as professional programmer (yes=1, no=0) are shown. Standard deviations are shown in parentheses.

Table 1. Characteristics of the groups. * marks a significant difference between groups in two-tailed t-test (interval values) or χ^2-test (nominal values) with $p < 0.05$

Experience level	Count	Prog. exp.*	Java exp.*	Jeliot exp.	Prof. exp.
Novices	8	12.8 months (6.9)	6.4 months (4.6)	3	1
Intermediates	8	85.5 months (56.4)	19.8 months (15.0)	2	1

3.3 Materials and Apparatus

Three short Java programs, factorial computation, recursive binary search, and naïve string matching were presented to the participants. The lengths of the programs in lines of code were 15, 34, and 38 respectively. Each of the programs generated only one line of output and did not require any user input. The names of methods and variables were altered so that the recognition of a program based on these surface features would be difficult.

In our study, we used an adapted version of Jeliot 3 which logged all the user actions and all the changes in the visualization of the programs to be compared with the eye tracking data. However, this material is not used in this analysis. The specialized visualizations, the execution history and the call tree, were disabled to avoid problems in interpreting the gaze behavior.

The remote Tobii ET-1750 (sampling rate 50Hz) eye tracker making no contact with participants was used to track eye movements; the eye tracker is built into a TFT panel so no moving part is visible and no sound can be heard during the recording. Only a computer mouse was available during the experiment to interact with the tool. The interaction protocols (such as mouse clicks) were collected for all the target programs, and audio and video were recorded for a whole session. Fixations shorter than 100 ms were disregarded from analysis. We have defined four main areas of interest matching the four main areas in the Jeliot interface: the code, the animation, the control, and the output area. Figure 2 illustrates the experimental settings used in the study.

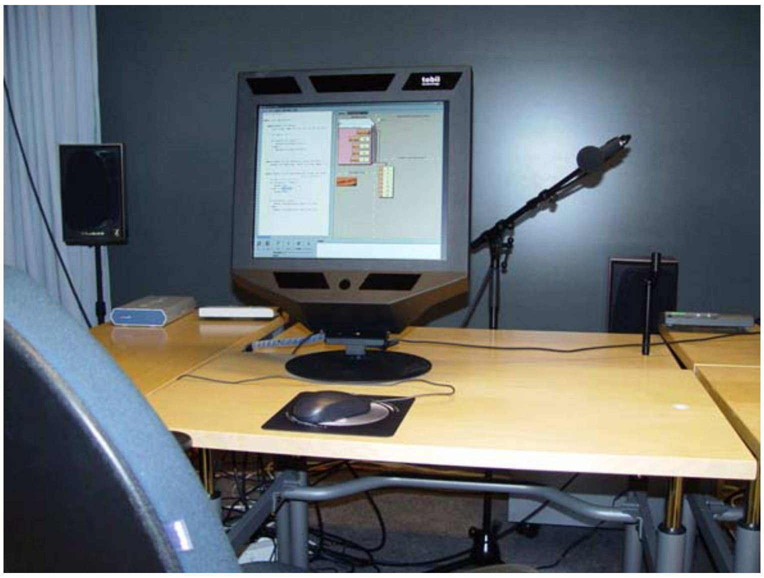


Fig. 2. Experimental settings.

3.4 Procedure and Design

The experiment was conducted in a quiet usability lab. Participants were seated in an ordinary office chair, near the experimenter, and facing a 17” TFT display. Every participant then passed an automatic eye-tracking calibration. During the calibration procedure, a participant had to follow sixteen shrinking points appearing one by one across the screen. If needed, the calibration was repeated in order to achieve the highest possible accuracy.

After a successful calibration, participants performed three sessions, each consisting of a comprehension phase using Jeliot 3 and a program summary writing phase. Participants were instructed to comprehend the program as well as possible and they could interact with Jeliot as they found it necessary. The target programs contained no errors and were always preloaded into Jeliot and compiled. The duration of a session was not limited.

The first program was factorial computation and it was used as a warm-up and the resulting data were discarded. The order of the two actual comprehension tasks was randomized so that half of the participants started with the recursive binary search and other half with naïve string matching.

4 Results

4.1 Completion and Animation Times

Mean completion times for the comprehension phase were 17.6 minutes (SD= 10.0) for novices, and 9.8 minutes (SD=2.6) for intermediates; the difference was statistically significant according to a two-tailed t-test $(t(7) = 2.48, p < .05)$. From that time, novices spent on average 85.4% (SD=9.6) animating the program whereas intermediates spent 52.9% (SD=20.0) of their time to animation; the difference was statistically significant according to the two-tailed t-test $(t(7) = 5.38, p < .01)$.

4.2 Fixation count distribution

Figure 3 shows a relative fixation count distribution over the areas of interest during the animation. Both groups spent most of the viewing time fixating the animation area, 57.4% (SD=11.9) novices, and 54.8% (SD=15.2) intermediates, of all fixations during the program animation. Next, 39.4% (SD=11.2) and 43.3% (SD=14.5), novices and intermediates, respectively, of all fixations was paid to the code area. No significant effect of experience on the distribution of fixations was found, without any interaction between the area of interest and experience. The fixation count has significantly differed between all four areas of interest, $F(3, 42) = 105.75, p < .001$. The planned comparison revealed a significant difference in the fixation count between the two most attended areas, the code and the animation $(t(15) = 2.29, p < .05)$.

4.3 Switching Behavior

Figure 4 illustrates the switching behavior as expressed by the number of switches per minute between the different areas of interest. The average number of switches per minute was 30.15 (SD=10.66) and 27.57 (SD=8.04) for novices and intermediates, respectively. The analysis of the effect of experience on the switching behavior discovered no significant change in the number of switches per minute, $F(1, 14) = 0.004, ns$. The switch between the code and the animation areas was far most common, $F(5, 70) = 145.25, p < .001$. Finally, the interaction effect between type of switch and experience was not significant, $F(5, 70) = 0.421, ns$.

4.4 Fixation Durations

Figure 5 shows the mean fixation durations during animation for the four main areas of interest and the overall mean fixation duration. These have been computed as a sum of durations of all fixations landing at an area of interest divided by number of the fixations. Since the programs did not generate an extensive output, some of the participants were not gazing to this area of interest. For the analysis, the missing values were replaced by the mean value of a group.

The overall mean fixation duration was 406.49 ms (SD=81.40) and 297.26 ms (SD=80.52) for novice and intermediate group, respectively. The effect of area of interest on the mean fixation duration was nearly significant, $F(3, 42) = 2.79, p = .052$.

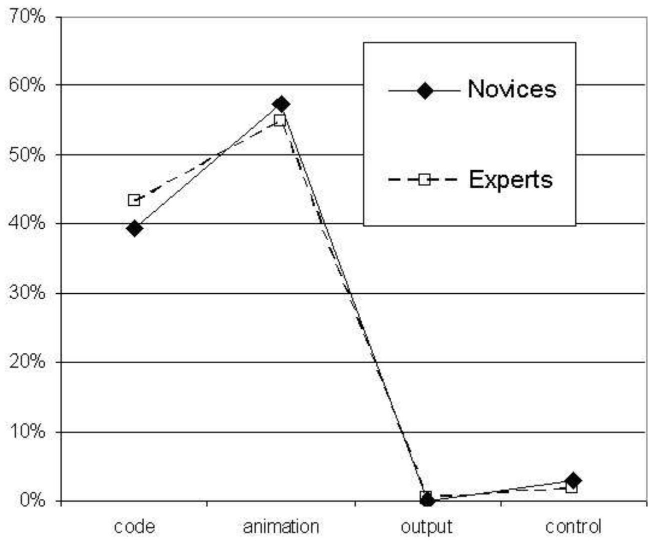


Fig. 3. Relative fixation count distribution during animation.

We also found an interaction between the fixation durations on the areas of interest and the level of experience, $F(3, 42) = 2.87, p = .048$. The effect of experience on the mean fixation durations was significant, $F(1, 14) = 8.98, p = .01$. Moreover, the effect of experience on overall fixation duration, $F(1, 14) = 7.16, p = .018$, was also significant.

5 Discussion

Intermediates completed the comprehension phase much faster than novices. Intermediates also spent significantly less time animating the programs which was in agreement with the hypothesis that intermediates would concentrate more on the code reading. This happened, however, only before they began and after they stopped visualizing the program. Both times can be kept as measures of performance. The initial code-reading episodes could have affected the behavior of the intermediates during the program animation compared to novices. Sajaniemi and Kuittinen [17] reported that during exercise sessions, students using PlanAni did not pay attention to the program code as much as to the visualization. Our results agree with this observation. Although both areas were attended with high fixation counts, it was more common to use the visualization than the code area during program animation, in our study, regardless the experience.

Analysis of the comprehension summaries have been done elsewhere in Bednarik et al. [19] with the program summary analysis by Good and Brna [20]. In this analysis, the summaries of intermediate subjects were found to be slightly better in the quality, but

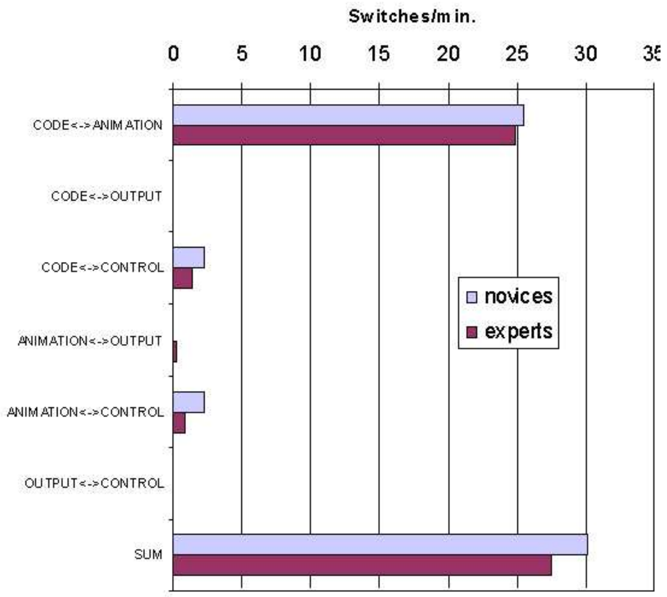


Fig. 4. Number of switches per minute between the main areas of interest.

there were no statistically significant differences found. Intermediates used higher level of abstraction than novices but again there were no statistically significant differences.

The results of this experiment related to the gaze behavior during program animation show that the relative fixation counts and the switching behavior between the areas defined in this study are insensitive to the level of experience. The distribution of fixations between code and animation was slightly more balanced for more experienced participants, but did not significantly differ from the distribution of novice fixations. With respect to these measures, we have to reject our hypothesis. Most of the animation time was spent on viewing the visualization part of the Jeliot interface.

The switches between code and animation areas were the far most common during the animation and therefore the sum of all switches is mostly composed by this type of switch. The code-control and animation-control switches were higher for novices. This is probably due to the fact that novices were interacting more with the tool during the animation than intermediates and therefore attending the control panel more often [19]. In terms of the total number of switches per minute, the two groups exhibited about the same behavior.

With respect to previous eye movement studies investigating the relationship between gaze behavior and expertise, these result are rather surprising. Several factors could, however, explain the results. One explanation seems to be that the features of animation attract equally novice and intermediate programmers to attend the animation in similar patterns. The visualization environment restricts the access to the elements of the graphical representation to only a short period of time, therefore the effects of

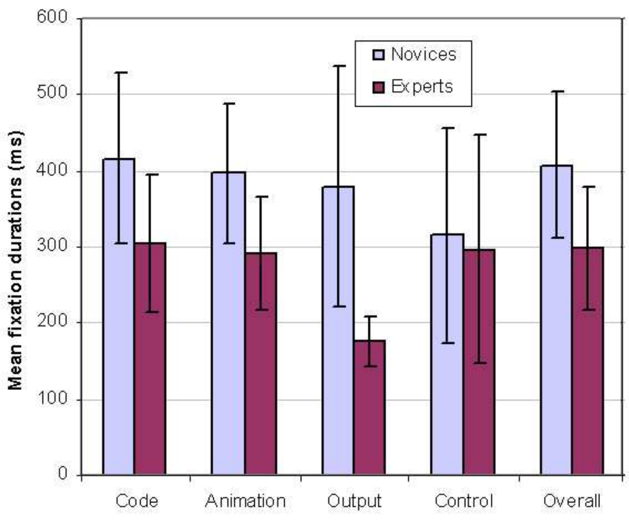


Fig. 5. Mean fixation duration during animation.

experience cannot materialize in the gaze measures used in this experiment. We also believe that more accurate measures have to be developed to reveal the differences between these two groups. For example, we could measure the disassociation between the current animation step and the gaze of the subject. Another possibility for not observing differences in the gaze behavior could be the number of subjects involved in the study and this will be taken into the consideration in the further studies. Finally, the gap between the skills of the two groups involved in this experiment might not be big enough to yield statistically significant differences in gaze behavior during animation.

Despite not finding differences in fixation count distribution and switching behavior, we did find a significant effect of experience on the mean fixation duration. For all the main areas (except for the control area) of the display and in overall, the mean fixation duration of intermediates was shorter than that of novices. This supports the results from previous studies and could be explained by at least two facts or a combination of both. One possibility is that, during the animation, intermediates might have an advantage of already formed hypothesis about the visualized problem. This hypothesis would be formed during the initial code reading before animating the program. The second explanation could be the available domain-knowledge and programming experience of the intermediates which would enable them to interpret the animation faster. From the mean fixation duration over the control panel, we can observe that novices and intermediates alike needed about the same time (300 ms) to decide what buttons they are going to use in order to control the flow on the ongoing animation.

Altogether, these findings could indicate that a difference in the programming experience can be seen in the mental efforts paid while attending the animation, while it does not affect the general patterns how the animation is attended. Both groups attend the suggested attention loci in about same way, but the more experienced programmers extract the information faster and, most probably, are therefore able to pay attention to the surrounding context. When a consecutive attention switch is suggested by the animation, both groups will follow it and thus exhibit similar switching behavior.

6 Conclusion and Further Work

We have conducted an empirical experiment to discover the aspects of gaze behavior during the dynamic program visualization. We employed a non-intrusive remote eye tracking equipment to record the eye movements of programmers with various level of experience. Our results, in terms of the attention switches between different program representations and the distribution of fixations, show no difference in the gaze behavior between novice and intermediate group of programmers during program animation. In other words, the focus of visual attention seems to be distributed in time and space evenly regardless of the experience in programming. When the level of processing required to attend the animation is measured as a mean duration of fixations over the main areas of interest and in overall, our results show that novice programmers spend significantly more time on extracting the features of animated concepts. We propose this difference to be linked to the experience level and with a pre-established model of the algorithm being animated. The performance measures seem to support this hypothesis.

Our initial experiment provides a take-off mark for further studies investigating gaze behavior related to the dynamic program visualizations. Several directions for future research can be taken. Based on the general, macro-level patterns presented in this paper, we aim to deconstruct the behavior into more micro-level sequences. Between our next aims belong to investigate the effects of the discrete animation elements on the gaze behavior as well as the changes in the behavior in a course of time. Among the questions raised by the present study belong, what kind of suggested switches are consumed during the animation and whether the decision differs given the level of experience.

To answer the questions, we plan to develop a methodological framework for a reliable application of eye-movement tracking in the context of program visualization. These studies shall provide us with a deeper understanding about the cognitive processes involved in program comprehension during program visualization.

Acknowledgments

We would like to thank all participants for taking part in this study. We acknowledge Andrés Moreno for a help with preparation of this study.

References

1. Ben-Bassat Levy, R., Ben-Ari, M., Uronen, P.A.: The Jeliot 2000 program animation system. Computers & Education **40** (2003) 15–21
2. Sibert, L.E., Jacob, R.J.K.: Evaluation of eye gaze interaction. In: CHI 2000, ACM Press (2000) 281–288

3. Rayner, K.: Eye movements in reading and information processing: 20 years of research. Psychological Bulletin **124** (1998) 372–422
4. Matin, E.: Saccadic suppression: a review and an analysis. Psychological Bulletin **81** (1974) 889–917
5. Carpenter, P.A., Just, M.A.: Eye fixations during mental rotation. In Senders, J.W., Fisher, D.E., Monty, R.A., eds.: Eye movements and the higher psychological functions. Erlbaum, Hillsdale, NJ (1997) 115–133
6. Goldberg, J.H., Kotval, X.P.: Eye Movement-Based Evaluation of the Computer Interface. In Kumar, S.K., ed.: Advances in Occupational Ergonomics and Safety. IOS Press, Amsterdam (1998) 529–532
7. Nodine, C., Mello-Thoms, C.: The nature of expertise in radiology. In Beutel, J., Kundel, H., Metter, R.V., eds.: Handbook of Medical Imaging. SPIE Press (2000)
8. Kasarskis, P., Stehwien, J., Hickox, J., Aretz, A., Wickens, C.: Comparison of expert and novice scan behaviors during VFR flight. In: The 11th International Symposium on Aviation Psychology. (2001)
9. Crosby, M., Stelovsky, J.: Subject Differences in the Reading of Computer Algorithms. In Salvendy, G., Smith, M.J., eds.: Designing and Using Human-Computer Interfaces and Knowledge-Based Systems. Elsevier (1989) 137–144
10. Romero, P., du Boulay, B., Cox, R., Lutz, R.: Java debugging strategies in multi-representational environments. In: The 15th Annual Workshop of the Psychology of Programming Interest Group (PPIG'03). (2003) 421–434
11. Romero, P., Lutz, R., Cox, R., du Boulay, B.: Co-ordination of multiple external representations during Java program debugging. In: Empirical Studies of Programmers symposium of the IEEE Human Centric Computing Languages and Environments Symposia, Arlington, VA (2002) 207–214
12. Bednarik, R., Tukiainen, M.: Visual attention tracking during program debugging. In: NordiCHI'04, ACM Press (2004) 331–334
13. Bednarik, R., Tukiainen, M.: Effects of display blurring on the behavior of novices and experts during program debugging. In: CHI '05: CHI '05 extended abstracts on Human factors in computing systems, ACM Press (2005) 1204–1207
14. Oechsle, R., Schmitt, T.: JAVAVIS: Automatic Program Visualization with Object and Sequence Diagrams Using the Java Debug Interface (JDI). In Diehl, S., ed.: Software Visualization. Volume 2269 of Lecture Notes in Computer Science., Springer-Verlag (2002) 176–190
15. Zeller, A., Lütkehaus, D.: DDD — A Free Graphical Front-End for UNIX Debuggers. ACM SIGPLAN Notices **31** (1996) 22–27
16. Gestwicki, P., Jayaraman, B.: Interactive visualization of Java programs. In: IEEE Symposia on Human Centric Computing Languages and Environments. (2002) 226–235
17. Sajaniemi, J., Kuittinen, M.: Program animation based on the roles of variables. In: ACM symposium on Software visualization, ACM Press (2003) 7–16
18. Moreno, A., Myller, N., Sutinen, E., Ben-Ari, M.: Visualizing Programs with Jeliot 3. In: Advanced Visual Interfaces (AVI 2004). (2004) 373–376
19. Bednarik, R., Myller, N., Sutinen, E., Tukiainen, M.: Analyzing Individual Differences in Program Comprehension with Rich-Data Capture. Submitted (2005)
20. Good, J., Brna, P.: Program comprehension and authentic measurement: a scheme for analysing descriptions of programs. International Journal of Human-Computer Studies **61** (2004) 169–185

P5.

Bednarik, R., Myller, N., Sutinen, E., Tukiainen, M.: Analyzing Individual Differences in Program Comprehension. Technology, Instruction, Cognition and Learning (TICL), 3(3-4), pp. 205-232, 2006.

Tech., Inst., Cognition and Learning, Vol. 3, pp. 205-232
Reprints available directly from the publisher
Photocopying permitted by license only

Analyzing Individual Differences in Program Comprehension

ROMAN BEDNARIK*, NIKO MYLLER, ERKKI SUTINEN
AND MARKKU TUKIAINEN

Department of Computer Science
University of Joensuu
Joensuu, Finland

Programming is a complex problem-solving domain often involving many dependent entities which may be even hidden or latent. Novice programmers have little knowledge about program execution and may see it as an abstract and non-deterministic process. To support novices, Jeliot was developed to visualize program execution, and thus help in specifying viable program development models. This paper reports on an empirical experiment in program comprehension where 16 subjects used Jeliot to comprehend two Java programs. The experiment focused on how the experience level and complexity of the program affected (a) the patterns of interaction with the tool, (b) the gaze behavior, (c) the use of visualization, and (d) the cognitive processes related to program comprehension. This was done by investigating the protocols obtained from an eye-tracker, interaction logging, and comprehension summaries. An interaction between experience and behavior was found. Experts read the whole code first, constructed a hypothesis, and tested it against the animation. Novice programmers did not read the code first. They animated the program directly, and replayed the animation several times focusing on the difficult sections. The results reveal the potentials of gaze as an additional modality in an adaptive tool for program visualization.

Keywords: Complex systems; Empirical programmer studies; Eye tracking; Program comprehension; Program visualization.

*Corresponding author: Tel: +358 13 251 7977; Fax: +358 13 251 7955; Email: bednarik@cs.joensuu.fi

INTRODUCTION

Programming is a complex and cognitively demanding task due to the multiple interrelated components, tradeoff decisions, and performance requirements that concern the whole process (Detienne, 2002; Hoc, 1990). One of the central parts in the processes of programming, such as creating, maintaining and modifying a software product, is program comprehension. Therefore, the ability to comprehend computer programs is essential and should be learned and supported with proper tools. In order to aid the process of program comprehension and its learning, several tools have already been developed. However, little is known about the user interaction with the tools and about the effects of these systems on the program comprehension and the underlying cognitive processing.

The present study addresses the questions of whether and how a dynamic visualization of program is used during comprehension processes, and how the experience of programmers and complexity of the target programs are reflected in eye-movement patterns and in patterns of interaction. Knowledge of these aspects may help us to create better and more personalized tools and methods for aiding program comprehension.

The complexity of programming

A complex system consists of a large number of components whose interrelations are difficult if not impossible to trace. We can call the complexity of these systems *substantial*. A computer-aided system is supposed to clarify these interrelations and help a human to understand the dynamics of the system, most often by the means of simplifying the interrelations and visualizing the system.

In a sequential program, the dependencies between its structures are – unlike those of a typical complex system – well defined and need no simplification or reduction. The value of a certain variable depends on the predetermined sequence of statements, which make use of other data structures of the program.

However, the complexity lies in the human process of comprehending a program, especially at the novice level. The learner needs to take into account diverse aspects of a program, such as its I/O, control flow, data management, and memory allocation, and to grasp all this information simultaneously makes understanding difficult. Individual preferences, like appropriate representation, complicate things even further: if a learner is exposed to a visualization of, for instance, a variable and its contents that s/he has difficulty to interpret, the representation may mislead her/him even more. Thus, we can characterize the complexity of a sequential program as *cognitive*.

At the same time, depending on the values of the inputs, even a short piece of code might have several different manifestations. This is particularly apparent in the case of short string algorithms, like in the variations of Knuth-Morris-Pratt or Boyer-Moore string searching algorithms. The call tree of a simple recursive program, for example, one operating on a binary search tree, might also result in complex-to-understand structures.

To summarize, the complexity related to programming and its learning is more cognitive than that in a conventional complex system where the complexity is substantial. Therefore, the fundamental challenge of any system that helps a learner to understand or, as a programmer, to control the internal dynamics or operations of a sequential program, is to lessen the cognitive load involved in elaborating the program. This means that the learner should get closer to the actual, often relatively simple, idea of the program. One approach to make the inherent simplicity of any program understandable to a novice programmer is to clarify its operations by visualization. Jeliot, a tool used in the present study, is one of several solutions to this challenge.

Currently, there are many visualization engines or systems available. Most of them have also been evaluated, at least partly; and there are even meta-studies, like that of Hundhausen (2002). However, careful analyses of how a user browses an animation are still mostly lacking. These analyses are essential for developing the visualization environments further, so that they can help the learner to focus on a program's essentials as efficiently as possible. In other words, future systems should uncover the cognitive complexity and give way for the substantial simplicity of the program.

It is worth noting that there are also programs which behave like a complex system and could hence be categorized as substantially complex systems. For example, it is not possible to predict the next step of a concurrent program; the same applies to randomized algorithms, as well as to multithreaded or event-driven programs. However, tools for these kinds of environments are not discussed in the current study.

Program comprehension

A number of studies have been carried out in the field of program comprehension. The theories of program comprehension can be divided roughly into three categories: bottom-up, top-down and mixed models.

The *bottom-up* model of program comprehension was proposed by Shneiderman and Mayers (1979), and Basili and Mills (1982). Pennington (1987) reported that programmers approached a comprehended program in a bottom-up manner from the control structures to the functional structure of the program. However, this model has not received strong support as a comprehensive model for describing the comprehension process, but it is often incorporated as a part of the model.

Brooks (1983) presented a model in which the program was comprehended in a *top-down* manner. The central idea of the theory was that programmers generate hypotheses about the code, using their programming and domain knowledge, and try to verify them. Brooks treated the bottom-up comprehension process as a degenerated special case of the top-down strategy. Letovsky (1986) proposed a similar model and verified the model in an empirical experiment. The conclusion was that the comprehension process is guided by the hypotheses. However, Letovsky also recognized the importance of the bottom-up approach in program comprehension.

The *mixed model* was supported by the findings of von Mayerhausen, Vans and Somlo (1999), who studied professional programmers in authentic software development projects. They proposed an integrated code comprehension model that combines the results from previous research. The top-down strategy is commonly used, but when the program or the domain is unfamiliar, the comprehension is carried out in a bottom-up manner to gather the program and domain knowledge. With these strategies and using their knowledge base, the programmers build and relate the domain, program and situational models to each other. The proper combination of these three models determines how well the programmer understands the program.

The methodology for studying and analyzing the program comprehension process relies mostly on three approaches: comprehension questions, comprehension summaries and think-aloud protocols.

Comprehension questions are used to determine how well the user has comprehended the program and can remember some aspects of it (Pennington, 1987). These can be used as a measure of comprehension performance. Although program summaries can also be evaluated for analysis of the performance, they are commonly used to investigate the mental models of the program acquired by the programmers during the comprehension task. Previously, *comprehension summaries* have been analyzed from two different standpoints (Pennington, 1987; Good & Brna, 2004). One focus is on the *information types* that are used in the summaries, wherefore ten categories were discovered for different information given from the program. The categories describe several dimensions of the comprehension outcome. First of all, categories can be divided roughly into four knowledge types, namely *functional*, *state*, *data flow* and *control flow* knowledge. Furthermore, functional and state knowledge have different abstraction levels. Functional knowledge has three abstraction levels: *function*, for what the program is used, *action*, function of a part of the program, and *operation*, a statement level description. State knowledge has two levels *state-high* and *state-low* depending on how high is the abstraction level in the description. In addition to the knowledge types, the information types contain four categories for *elaboration*, *meta*-cognition, *unclear* and *incomplete* statements and Byckling et al. (2004) proposed also an *irrelevant* category. The other focus of the summary analysis scheme is on *object descriptions*. They are classified according to their level of description in the summary. For instance, the descriptions are classified in different categories depending on if the objects are described in program or domain specific terms.

The thought sequences and cognitive processing during problem solving are often analyzed with *think-aloud protocol analysis* (Ericsson & Simon, 1984). Resting on an assumption that the sequence of thoughts is not altered during verbalization, protocol analysis has been used successfully in several domains as well as in analysis of the program comprehension processes.

Program Visualization

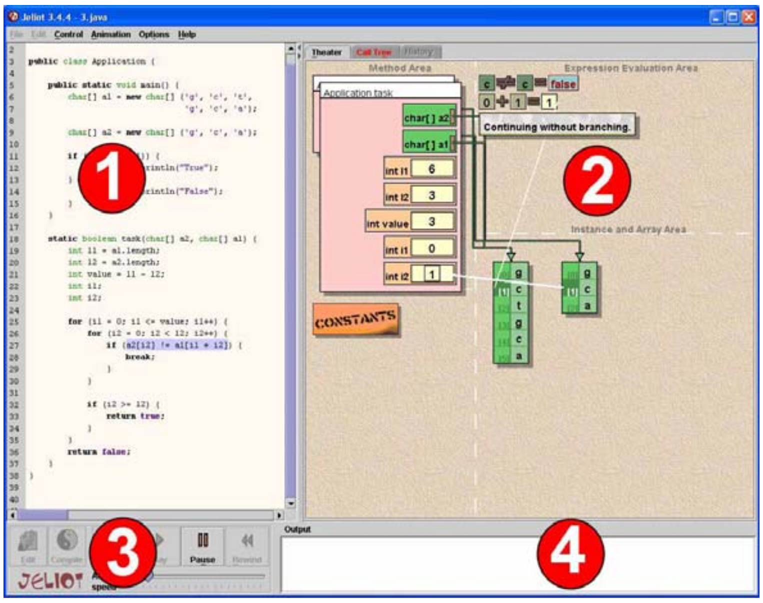
Many systems have been developed in the field of Software Visualization (SV) in the last two decades. Algorithm Visualization (AV), a subset of SV systems, has received the most attention and has been studied relatively intensively. However, the results have been inconclusive (Hundhausen et al., 2002). Program Visualization (PV) is another subfield of SV where visualization is closely coupled with the program and some aspects of the programs execution are visualized either during run-time or post-mortem. These systems are used, for example, to analyze the performance of the software, to debug programs visually or to teach programming concepts. Here we will briefly review systems that are similar in some aspect to Jeliot, the tool used in this experiment.

Javavis (Oechsle & Schmitt, 2002), Jive (Gestwicki & Jayaraman, 2002) and DDD (Zeller & Lütkehaus, 1996) are tools that automatically visualize the programs data flow and part of the control flow during program execution. Javavis and Jive are educational tools whereas DDD is a visual front end to a debugger. To visualize Java programs, Javavis uses UML-like diagrams in their visualizations. It illustrates the run-time behavior of the program through animated object and sequence diagrams. Jive uses a modified contour diagrams and shows the different contexts (i.e. static or dynamic) in which the program is executed together with the source code and local variables. DDD uses its own graph-like format to lay out the data structures and references between them on the screen. Jeliot combines this work to show objects and their fields in a UML-like notation of class diagram. References to the objects are treated as other variables to illustrate the reference semantics of Java language.

The visualization in Dynalab (Boroni et al., 1996) and PlanAni (Sajaniemi & Kuittinen, 2003) concentrates on variables. Dynalab shows the values of variables in a textual format whereas PlanAni shows the variables as graphical objects. Moreover, PlanAni visualizes the variables and operations on them differently, depending on the role (Sajaniemi & Kuittinen, 2003) they are assigned. Dynalab visualizes the programs automatically and can also animate the programs backwards whereas the animations in PlanAni need to be written manually beforehand. Both of these programs are currently developed to visualize only procedural programs. Dynalab has full support for Pascal and restricted support for C and Ada. PlanAni does not restrict the visualized language as long as the animation scripts are described in Tcl/Tk. Jeliot visualizes variables in different scopes separately. For example, method frames contain the local variables of the method, objects contain the fields and static variables are separated in their classes. Jeliot also does not make any difference between the roles of variables. However, it can automatically visualize object-oriented programs.

Jeliot 3

The Jeliot family is a collection of program visualization systems that have been developed over the last ten years (Ben-Ari et al., 2002). The latest version, Jeliot 3, which has been developed at University of Joensuu (Moreno et al., 2004), is designed to help teach novices the programming concepts and to aid in program comprehension and debugging. Its predecessor, Jeliot 2000, has been successfully used to improve the teaching of introductory programming courses, helping novices to acquire vocabulary to explain programming structures and concepts (Ben-Bassat Levy et al., 2003). This might be due to the fact that Jeliot can help the learner to build a viable mental model of the computer executing the program and use it to verbalize the execution.



```
public class Application {

    public static void main() {
        char[] a1 = new char[] {'g', 'c', 't',
                                'g', 'c', 'a'};

        char[] a2 = new char[] {'g', 'c', 'a'};

        if [illegible])) {
            [illegible]rintln("True");
        }
            [illegible]rintln("False");
        }
    }

    static boolean task(char[] a2, char[] a1) {
        int l1 = a1.length;
        int l2 = a2.length;
        int value = l1 - l2;
        int i1;
        int i2;

        for (i1 = 0; i1 <= value; i1++) {
            for (i2 = 0; i2 < l2; i2++) {
                if (a2[i2] != a1[i1 + i2]) {
                    break;
                }
            }

            if (i2 >= l2) {
                return true;
            }
        }
        return false;
    }
}
```

FIGURE 1
User interface of Jeliot 3. (1 = *code editor*; 2 = *visualization view*; 3 = *control panel*; 4 = *output console.*)

Jeliot 3 retains the novice-oriented GUI and animation display of Jeliot 2000 by only adding new menus to expose more functionality to the user, especially when it is used during lectures. In order to make the system extensible and to allow adding new features into the visualization, Jeliot 3 introduced a new design. It automatically visualizes the execution of user-written Java programs by illustrating the data and control flow and object-oriented features of the program. Jeliot 3 can visualize a large subset of novice-level Java programs and it is freely distributed under GPL (see http://cs.joensuu.fi/jeliot/). The user interface of Jeliot 3 is shown in *Figure 1* (1 = *code editor*; 2 = *visualization view*; 3 = *control panel*; 4 = *output console*).

The interface consists of four different areas. A *code editor* on the left side shows the program code, and during program visualization, the currently executed statement or expression is highlighted. A *control panel* in the bottom left corner is used to control the animation with VCR-like buttons. On the right side of the window, the largest area of the Jeliot's window is occupied by a *visualization view* showing the execution state of the program. Moreover, the animation view is further divided into four different areas:

- *the method area* showing the currently executed method and local variables;
- *the expression evaluation area* where the expressions are evaluated step-by-step and messages are shown to the user;
- *the constant area* containing classes together with their static variables and the constant box from which the literal constants appear; and,
- *the instance and array area* showing the visualization of the arrays and instances containing their fields.

Finally, an *output console* is located in the bottom right corner of the window, showing the output of the executed program. Furthermore, there are two specialized visualizations, where only the call tree or the execution history of the program is shown, on separate tabs of the tabbed pane. These views reduce the complexity and the amount of information shown to the users and thus help them to concentrate on the relevant parts. In the call-tree visualization, the previous method calls are shown and the currently active methods are highlighted. This allows following the program execution in the level of method calls and further reduces the complexity. In the history view, users can analyze the previous stages of the execution in a step-by-step manner and thus are able to reason about how the current execution step was reached.

In a typical session with Jeliot, the user either writes or loads a program. The user can then compile the program by using the user interface of Jeliot. When the program is compiled, a visualization view, where the user can see the animation of the program execution, is opened. Jeliot shows the execution either step by step or continuously. The user can control the animation with the buttons, for instance by stopping the animation and continuing in a step-wise manner. Reverse execution is not possible; but the user can view the execution history which is a stepwise recording of the current execution.

Eye-movement tracking

We move our eyes in order to bring an image of the inspected object onto the fovea, a small and high-resolution area of the retina. Once the image of the object is stabilized on the retina, the information can be extracted. This way the visual attention is linked with the current direction of eye gaze and most of the time it is also diverted to the point of visual inspection. Following this eye-mind assumption, if we can track the movements of the eyes, we can also obtain good insights into and investigate the path and focus of visual attention during a task. Previous research has firmly established this relation between eye movements, visual attention and underlying cognitive processes (Just & Carpenter, 1976; 1980; Rayner, 1998). Knowing which objects have been visually inspected and in which order and context, we can attempt to infer what cognitive processes were involved in performing a task related to these objects.

An eye tracker is a device that records eye movements. To estimate the direction of gaze, most of the current eye trackers use infrared light emitters and video image analysis of the center of the pupil and reflections from the cornea. Typically, the accuracy of the eye trackers currently available commercially is around 1 degree, while the data are sampled at rates of 50–500Hz. Modern eye-trackers are relatively cheap and able to collect gaze data reliably and unobtrusively. Two general classes of eye tracking devices exist: a remote optics, table-mounted version and a head–mounted optics with a see-through mirror. Regardless of the option, both types of eye trackers must be calibrated for each user before the first experiment.

From the signal obtained from an eye tracker, the two most important types of eye movements usually identified are saccades and fixations (Salvucci & Goldberg, 2000; Sibert & Jacob, 2000). Saccades are rapid ballistic movements of eyes that are executed to reposition the eyes from one location of attention to another. A single saccade can last between 30 and 120 ms and can span over 1 to 40 degrees of visual angle (Sibert & Jacob, 2000), with velocities ranging up to 500 degrees per second (Rayner, 1998). No visual information is extracted during a saccade; this is called saccadic suppression (Matin, 1974). Fixations are the movements of eyes stabilizing the image of an object on the retina, providing the human visual system a possibility to extract the features of the object. Typically, the fixation duration ranges from 200 ms to 300 ms (Rayner, 1998). It is assumed that during the period of a single fixation the information is extracted, decoded, and interpreted. The fixation duration can therefore be thought to be related to the processing required to extract and interpret the information (Just & Carpenter, 1976; Goldberg & Kotval, 1998; 1999).

Accurate measurement and analysis of eye movements in terms of saccades and fixations therefore provide us with the details of cognitive processing and related allocation of visual attention within a performed task.

For instance, the fixation count or the sum of fixation durations on a certain element can be related to the importance of that element. The fixation duration might also be seen as a measure of cognitive workload. As Goldberg and Kotval (1999) reported, the fixation duration increased when the task required difficult cognitive processing. In the context of program visualization interfaces, the measure of relative fixation count can, for instance, correspond with the relative importance of a representation (e.g. a code or a state diagram) of a program to the current user of the tool.

Previous research of eye movement tracking

In many fields of HCI and in other domains, studies of eye-movement tracking have significantly contributed to the body of available knowledge. Eye-movement tracking has been successfully employed in studies of reading (Just & Carpenter, 1980, 1984; Rayner, 1994), gaze-based interaction (Jacob, 1993; Karn & Jacob, 2004), eye typing (Majaranta & Räihä, 2002), menu selection (Crosby & Peterson, 1991; Aaltonen, Hyrskykari & Räihä, 1998), usability (Goldberg & Kotval, 1998; 1999) or in virtual reality (Duchowski et al., 2000, 2002).

Possible differences in the eye movement patterns of experts and novices have been of great interest. Previous studies of eye movement patterns in domains other than program visualization have shown that (a) search strategies differ between novice and expert radiologists (Nodine and Mello-Thons, 2000), and (b) the eye movement patterns of expert-pilots were better defined and the dwell times were significantly shorter than those of novices (Kasarskis et al., 2001). The common conclusion of these and other reports is that the domain knowledge and experience of participants seem to be the main factors influencing both the performance and the related gaze behavior.

In the context of the psychology of programming, however, only a few attempts to utilize eye-movement tracking have been made. Crosby and Stelovsky (1989, 1990) studied aspects of visual attention during reading the programs and program comprehension. In these studies, an eye tracker was used to discover the relationship between cognitive styles and individual differences, and code-reading patterns. In Crosby and Stelovsky's studies, novices and experts were eye tracked during algorithm comprehension. With the help of eye tracking metrics, complex statements in the source code of a variation of binary search algorithm have been found and related to beacons (Crosby et al., 2002). Moreover, differences in the programming experience of participants were reflected in the times they spent viewing different areas of particular complex statements. However, only one representation of a program was used (the code), and the focus of the research was mainly on the critical, but surface features of the code, not on the behavior during dynamic visualization of the program.

Program visualization often involves simultaneous presentation of several different representations of a program in adjacent views. Previous studies, investigating issues such as visual attention switching or a representation use during program comprehension aided by visualization, involved only static pre-computed stimuli and the analysis was based on a recording of mouse movements over a blurred interface (Romero et al, 2002; 2003). In comparison to the eye movement tracking, the validity of such an approach was shown to be questionable (Bednarik & Tukiainen, 2004). Although there is an apparent need for a deeper investigation of gaze behavior and the underlying cognitive processes during program visualization (Chandler, 2004), no attempts were made to fill this gap.

EXPERIMENT

The present research investigates differences in gaze behavior of participants with different levels of programming experience during program animation. In a preliminary study (Bednarik et al., 2005) we discovered that, in terms of global attention allocation between code and visualization, novice and expert gaze patterns do not differ during comprehension aided by program animation. This result was quite surprising, since in other domains experts' and novices' gaze patterns differ significantly. However, we found some differences in the mean fixation durations. To further investigate these findings, we deconstructed the interface of the visualization tool and compared the performance of novice and expert participants in terms of their gaze behavior. This provides us with specific knowledge of how explicit areas are used by different programmers and whether their use interacts with performance and cognitive models.

To examine this hypotheses, we conducted an empirical experiment using a remote eye tracker to record the gaze behavior of participants during a program-comprehension task aided by an animation. Two groups of participants with different levels of experience used Jeliot to comprehend three short Java programs while their eye movements were simultaneously tracked.

METHOD

A mixed one-between-subject (experience) with two levels and one-within-subject (program) with two levels design was used. The dependent variables were relative fixation count over the areas of interest, number of switches per minute, mean fixation durations over the areas of interest and overall. The fixation count is a measure related to the level of the participant's interest in an area and therefore related to the importance of that area. The number of switches per minute is a measure of attention allocation dynamics. The mean fixation duration is associated with the depth of processing required to understand an attended element. The analysis of eye movement protocols used only data recorded during the program animation because that was the only time when all the representations were shown concurrently and selection of the attended-to representation could make a difference in understanding the program.

The process of program comprehension was recorded with a video camera and the interaction protocol with Jeliot was logged. Program summaries were analyzed using the program summary analysis scheme of Good and Brna (2004) and revised by Byckling et al. (2004). The summaries were scored from 0-3 according to the completeness and correctness criteria. Most of the statistical analysis was carried out using repeated measures ANOVA and planned comparisons based on t-test.

Participants

Eighteen participants were recruited from high-school students attending a university level programming course, undergraduate and graduate computer science students from the local university; each received a lunch ticket. Due to technical problems, data from two participants was discarded. The results are based on data collected from 16 subjects (13 male, 3 female).

Participants were divided into two groups according to their level of programming experience. Those with less than 24 months of programming experience were regarded as novices and those with more than 24 months as experts. The characteristics of the two groups are presented in *Table 1*. Groups' mean values for programming experience (in months) and Java experience (in months) and counts for previous experience with Jeliot (yes=1, no=0) and previous experience as a professional programmer (yes=1, no=0) are shown. Standard deviations are in parentheses.

TABLE 1
Characteristics of the groups. * marks significant difference between groups in two-tailed t-test (interval values) or χ^2-test (nominal values) with $p<0.05$.

Experience level	Count	Prog. exp.*	Java exp.*	Jeliot exp.	Prof. exp.
novices	8	12.75 months (6.90)	6.38 months (4.60)	3	1
intermediates/ experts	8	85.50 months (56.44)	19.75 months (15.00)	2	1

Materials and apparatus

Three short Java programs - factorial computation (program 1), recursive binary search program (program 2) and naïve string matching (program 3) - were presented to participants. The lengths of the programs (lines of code) were 15, 34, and 38, respectively. Each of the programs generated one line of output and did not require user input. The names of the methods and variables were altered so that recognition of a program based on surface features was difficult. To allow for comparison with the eye tracking data, we used an adapted version of Jeliot 3 that logged all the user actions and all the changes in the visualization of the programs in this study.

The remote Tobii ET-1750 (50Hz) eye tracker, which made no contact with participants, was used to track eye movements; the eye tracker is built into the TFT panel so no moving part is visible and no sound can be heard during recording. The interaction protocol (such as key-strokes and mouse clicks) was collected for all target programs, and audio and video were recorded for a whole session. The minimal duration of fixation for the algorithm processing eye-data was set at 100ms. Seven static areas of interest (AOI), matching the seven main areas in the Jeliot interface, were defined: the code, the expression evaluation area, the method area, the instances area, the constants, the control, and the output area.

Procedure and design

The experiment was conducted in a quiet usability laboratory. Participants were seated in an ordinary office chair near the experimenter and facing a 17" TFT display. Every participant then passed an automatic eye-tracking calibration. The calibration required the participants to follow sixteen shrinking points that appeared one by one across the screen. If needed, the calibration was repeated in order to achieve the highest possible accuracy. The settings of the experiment are shown in *Figure 2*.

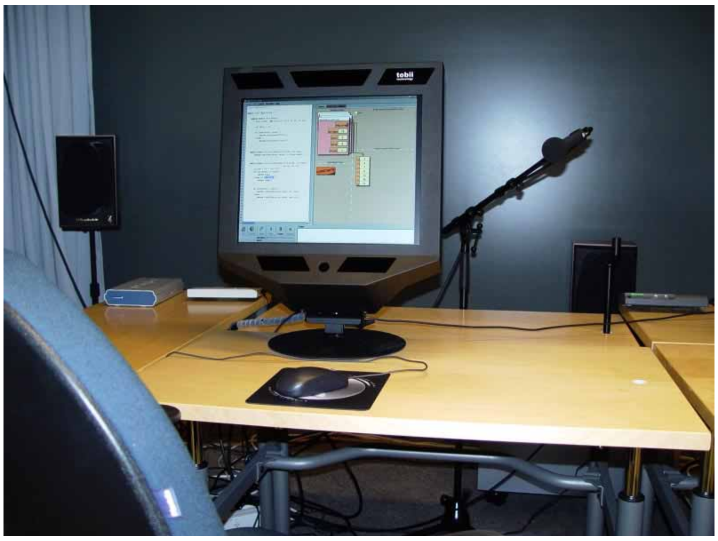


FIGURE 2
Experimental settings in the laboratory.

After successful calibration, participants performed three sessions, each consisting of a comprehension phase using Jeliot 3 and a program summary writing phase. Participants were instructed to comprehend the program as well as possible, and they could interact with Jeliot as they found necessary. The target programs contained no errors and were always preloaded into Jeliot and compiled. The duration of a session was not limited.

The first program was a factorial computation that was used as a warm-up; the resulting data were discarded. The order of the two comprehension tasks (program 2 and program 3) was randomized so that half of the participants started with the recursive binary search and the other half with naïve string matching.

RESULTS

Completion and animation times

Mean completion times for the comprehension phase were 17.6 minutes (SD = 10.0) for novices and 9.8 minutes (SD = 2.6) for experts; according to a two-tailed t-test, the differences were statistically significant (t(14) = 2.23, p<*.05*). From that time, novices spent, on average, 85.4% (SD = 9.6) of their time animating the program whereas experts spent 52.9% (SD = 20.0) of their time animating the program; according to a two-tailed t-test, the difference was statistically significant (t(14) = 4.13, p<*.01*).

Interaction patterns

The mean number of clicks on each of the buttons on the control panel and the mean number of animation replays between groups are show in *Table 2*. Novices interacted with the user interface more than experts did. Novices played, rewound and replayed the animation significantly more.

TABLE 2
Mean values of interaction with the user interface (standard deviations in parentheses).

Button	**Pause**	**Play**	**Rewind**	**Step**	**Overall**	**# of replays**
Novices	3.22 (3.55)	5.28 (3.69)	1.83 (1.09)	39.56 (57.20)	49.89 (63.42)	2.722 (0.939)
Experts	1.25 (1.00)	2.19 (1.19)	0.63 (0.69)	2.50 (7.07)	6.57 (7.55)	1.625 (0.694)
t-value (df=14)	1.395 *ns*	2.144 p=*.05*	2.824 p=*.01*	1.761 p=*.10*	1.91 p=*.09*	2.71 p=*.01*

The interaction patterns of a typical comprehension session of program 3 for a novice and expert (*Figure 3*) show the states of the visualization tool when used by a typical novice and by a typical experienced programmer. In both of the comprehension tasks, the experts spent significantly more time on initial code reading (on average, 173 seconds) before they animated a program for a shorter time (on average, 340 seconds) and usually at high speed. On the other hand, the novices paid little attention to the code (on average, 45 seconds) and it took them significantly more time to view the animation than it took the experts (on average, 857 seconds). In addition, behavior of a novice in our study was characterized by frequent use of a pausing/stepping approach combined with more replays of the animation. From the protocols and gaze-recordings it is clear that the stepping and pausing occurred around the central parts of the currently comprehended algorithm.

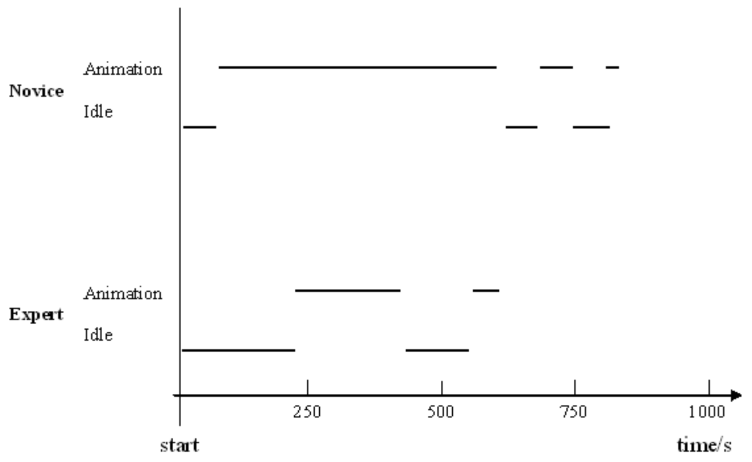


FIGURE 3
The phases of Jeliot animation engine during the comprehension of the binary search program.

Analysis of comprehension summaries

The comprehension summaries were graded in a scale from 0 to 3 and the averages of the two groups are shown in *Table 3*. The results indicate that experts received more points than novices did; however, the differences were not statistically significant.

TABLE 3
Points received from the summary evaluation (standard deviations in parentheses).

	program 2	**program 3**
both groups	1.81 (0.75)	1.88 (0.96)
novices	1.63 (0.52)	1.50 (1.07)
experts	2.00 (0.93)	2.25 (0.71)

In order to understand the qualitative differences in the summaries better, they were analyzed according to the scheme described by Good and Brna (2004). We analyzed only the information types found in the summaries (*Table 4*). The first eleven rows describe the standard categories of the analysis scheme. The repeated measures analysis of variance revealed no significant effect of program (F(1,14) = 2.32, *ns*) or experience (F(1,14) = 2.32, *ns*) on the information types contained in the summaries. The interaction between program and experience was not significant (F(1,14) = 2.32, *ns*). However, an effect of the information type was discovered (F(9, 126) = 6.82, p<*.001*), but had no interaction with the level of experience (F(9, 126) = 0.71, *ns*). The interaction effect between program and information type was significant (F(9, 126) = 2.45, p=*.013*).

To investigate this difference, pairwise comparisons were run. According to these comparisons, the action, and the irrelevant categories were present significantly more in the summaries of program 3 (t(14) = 2.38, p<.05) and t(14) = 2.45, p<.05, respectively), while the control related statements were present more in the summaries of program 2 (t(14) = 2.36, p<.05). No other significant difference was found, however, the state-low statements were more often present in program 2 summaries, with probability approaching a significance level of 0.05, (t(14) = 1.99, p=*.066*).

TABLE 4
Information type analysis (standard deviations in parentheses; unused categories omitted).

	program 2		**program 3**	
	novices	**experts**	**novices**	**experts**
function	5.68 % (5.71)	5.96 % (4.59)	5.23 % (4.86)	6.17 % (4.84)
action	6.71 % (6.27)	5.68 % (6.78)	13.61 % (13.71)	9.87 % (9.49)
operation	17.75 % (19.73)	10.63 % (17.17)	15.69 % (13.09)	16.28 % (14.37)
state-high	5.70 % (6.36)	8.13 % (7.47)	6.73 % (3.09)	15.91 % (10.10)
state-low	12.59 % (15.36)	12.83 % (11.80)	11.99 % (13.07)	6.78 % (5.90)
data	8.84 % (7.43)	22.90 % (8.11)	10.47 % (8.36)	9,91 % (8,67)
control	23.65 % (21.73)	21.00 % (5.11)	10.57 % (6.57)	12,33 % (10.25)
elaborate	16.45 % (12.41)	10.78 % (12.20)	16.22 % (13.13)	19,63 % (15.47)
meta	0.96 % (2.72)	0.83 % (2.36)	5.21 % (9.76)	0,00 % (0,00)
irrelevant	0.00 % (0.00)	1.25 % (2.48)	4.28 % (5.50)	3.11 % (4.58)
info-high	20.22 % (12.70)	36.99 % (13.50)	22.42 % (8.72)	32.00 % (12.94)
info-low	53.99 % (25.53)	44.46 % (16.46)	38.25 % (17.88)	35.38 % (22.14)

The last two rows of *Table 4* contain two aggregate values that describe the level of abstraction used in the summaries. The value *info-high* is a measure of high abstraction and is a sum of function, state-high and data statements. On the other hand, *info-low* is a sum of operation, state-low and control and indicates a low level of abstraction in the summary. The abstraction levels were used differently (F(1,14) = 5.37, p<*.05*). Experts used more *info-high* statements and less *info-low* statements, but there was no interaction between abstraction level and experience (F(1,14) = 2.21, *ns*).

Eye-gaze related data

Since the programs did not generate an extensive output, some of the participants were not gazing at the output area. Therefore, only the data of those participants who produced at least one fixation to each of the areas were included in the analysis. As one experimental participant did not perform any switching during comprehending the program 2, data for only fifteen participants were used in this analysis of eye-gaze data.

Distribution of Fixations

The distribution of the fixations over different areas of interest (AOI) during animation is illustrated in *Figure 4*, the value of the visualization AOI is composed as a sum of the method, expression, instance, and constants areas of interest. Only the data of those participants who produced at least one fixation to each of the areas were included in the analysis.

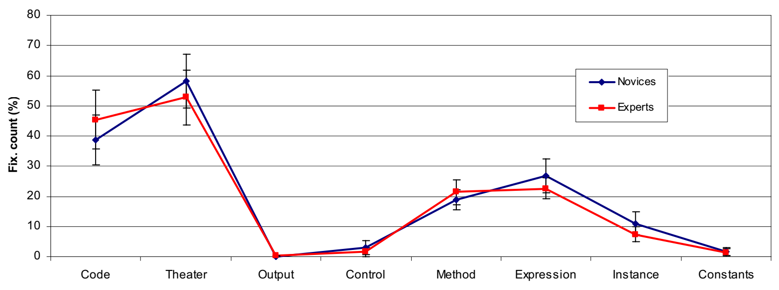


FIGURE 4
Relative fixation-count distribution during animation. The value of Visualization column represents a sum of fixations on the method, expression, instance, and constants areas.

According to the repeated measures ANOVA, the use of areas of interest was significantly different (F(5,65) = 62.89, p<*.001*). The effect of program was not significant (F(1,13) = 0.001, *ns*) and the effect of experience to the fixation count distribution was also not significant (F(1,13) = 0.46, *ns*). However, the interaction between program and AOI was significant (F(5,65) = 9.00, p<*.001*). Other interactions were not significant.

Distribution of the fixations over the user interface during comprehension of program 3 is shown in *Figure 5* as hotspot visualization, where the areas with more fixations are in darker colors. The distributions were similar for both groups. During both programs there were two areas that got the highest attention from both groups. One of the areas was in the code where the central idea of the program lay, and the other was in the expression area of the visualization frame.

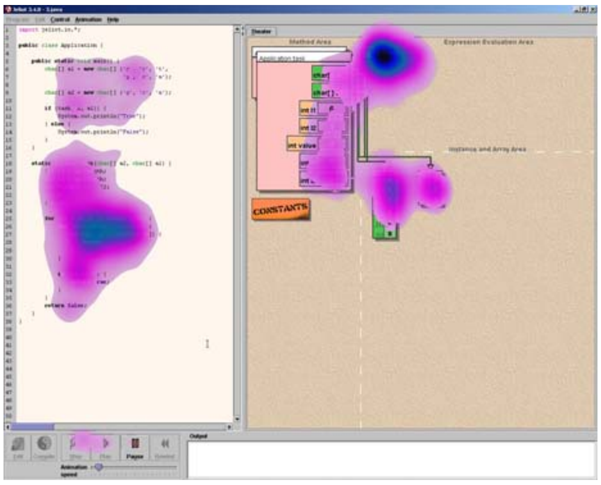


FIGURE 5
Visualization of the fixation distribution for the string matching program; the more attended areas are shown in darker colors.

Switching behavior

Figure 6 shows the mean number of switches per minute during animation. A switch was measured every time a gaze location changed between any two of the seven areas of interest. As one experimental participant did not perform any switch with the program 2, there were only data for fifteen participants used in this analysis. For the data during the animation, a 2 x 2 x 21 (experience x program x switch type) ANOVA showed a significant main effect of switch type (F(20, 260) = 68.44, p<*.001*), and no effect of program (F(1,13) = 1.04, *ns*) on the switching frequency. The effect of experience was not significant, however, approaching near significance (F(1, 13) = 3.37, p = *.089*). The interactions between program and experience, and between switch type and experience were not significant (F(1, 13) = 0.50, *ns*)*,* and (F(20, 260) = 1.02, *ns,* respectively). However, the interaction between program and switch type was significant (F(20,260) = 11.38, p<*.001*).

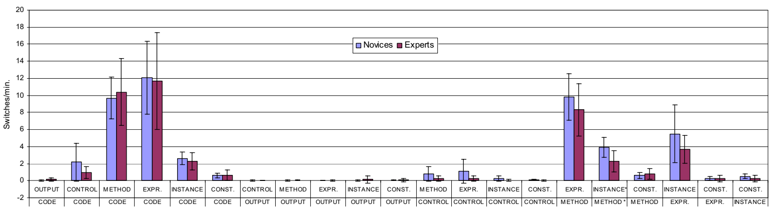


FIGURE 6
Visual attention switching during animation; * indicates significant difference in the two-tailed t-test (p<.05).

The pairwise comparisons revealed the sources of the differences: the switches between the code and the method AOIs and the code and the instances AOIs differed significantly between the programs (t(13) = 3.07, p<*.01*)*,* and (t(13) = 3.62, p<*.005*, respectively). Some switches within the animation frame, particularly those between the expression and the method areas, between the instances and method areas, and between the expression and instances areas, differed significantly (t(13) = 2.24, p<*.05*), (t(13) = 2.63, p<*.03*), and (t(13) = 6.97, p<*.001*, respectively).

Fixation durations

Figure 7 shows the mean fixation durations during animation for the seven main areas of interest (AOI) and the overall mean fixation duration.

These have been computed as the sum of durations of all fixations landing at an area of interest divided by the number of fixations. Since the programs did not generate an extensive output, some of the participants were not gazing at the output. The data on the areas affected during the animation, the method, the expression evaluation, the instance, and the constants areas, were included into the analysis.

According to repeated measures ANOVA, the durations of fixations on different AOIs were significantly different (F(4,52) = 16.66, p<*.001*). On almost all areas, experts seemed to fixate significantly shorter (F(1,13) = 7.27, p=*.018*) and an interaction effect between experience and fixation duration on the areas was significant (F(4,52) = 3.88, p<*.01*). No effect of program on the mean fixation duration was found (F(1,13) = 0.495, *ns*), as well as no interaction between program and AOI (F(4,52) = 1.59, *ns*).

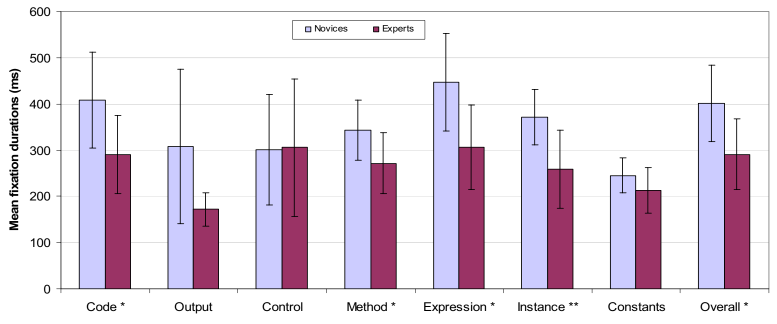


FIGURE 7
Fixation duration during animation. * indicates significant difference between groups in the two-tailed t-test (p<.05) and ** the two-tailed t-test (p<.01).

DISCUSSION

In our study, expert programmers completed the comprehension phase faster than novices did. This can be kept as one measure of performance. From the analysis of the summaries it can be seen that, although the differences were not statistically significant, experts performed better than novices. In the qualitative analysis of the summaries, it was found that, compared to novices, experts used higher level of abstraction in their summaries, but also this difference was not statistically significant. However, according to previous studies, the use of high level references is a sign of expertise (Hoadley et al., 1996; Pennington, 1987). Together with the finding that experts comprehended the programs faster, these indicate better performance of the experts compared to the novices.

From the interaction protocols analysis we found that the experts usually started their comprehension task by reading the code and spending significantly larger proportion of the session on constructing hypotheses. This finding is in agreement with results obtained in previous studies (Romero, 2003; 2002). After these initial code-reading episodes, the experts spent significantly less time than novices on animating the programs. Novices interacted with the user interface more than the experts did. This is due to the fact that novices spent more time on visualizing the program than experts did, and novices also replayed the animation more, compared to experts. The number of replays indicates that novices relied on visualization whereas experts used it as an additional source of information. Experts spent less time animating the programs which was in agreement with the hypothesis that experts would concentrate more on code reading. This happened only before and after they had been visualizing the program. The code-reading episodes could have affected the gaze behavior of the experts during the program animation compared to novices. However, we have not found such projection into the fixation patterns in terms of fixation distribution during the animation.

The results of this experiment related to gaze behavior during animation show that the level of experience does not affect the distribution of fixation counts, while the switching behavior is affected slightly. This result is inconsistent with previous eye movement studies where the differences between novice and expert gaze behavior were investigated. The animation attracts novice and expert programmers almost equally to attend it in quite similar patterns. Most of the animation time was spent on viewing the visualization part of the Jeliot's user interface. For more experienced participants, the distribution of fixations between code and animation was slightly more balanced, in other words, the novice programmers relied more on visualization.

During animation, the attention switch between the code and the expression evaluation areas was most common, followed by the switch between the code and the method area. The switches to and from the control area were higher for novices. We connect this difference to the fact that during the animation novices were interacting with the tool more than experts were. According to the analysis, the effect of experience on the switching behavior was only a nearly significant.

Unlike the fixation counts and attention switching, we found a significant effect of experience on the fixation duration. For all the main areas and also overall, the fixation duration of experts was shorter than that of novices. The fixation duration when participants gazed at the control buttons was the same regardless of experience. Altogether this means that novices needed to devote significantly more time to comprehending a currently animated feature, which is naturally linked to expertise. This difference was greatest when participants were attending to the instance area and smallest for the area where constants were appearing.

Considering the gaze behavior as related to a target program, we found that the use of the discrete areas and therefore the type of the attention switch differed as the comprehended program changed. In previous studies on the coordination of multiple representations during a debugging task (Romero et al., 2003; 2003) it has been suggested that the balanced use of different representations of a program during comprehension might be linked to superior programming experience. As seen from the fixation count distribution and switching behavior, both groups used the provided representations in about the same balanced way. Therefore, it could be, provocatively, suggested that using the Jeliot also made the novice programmers to behave in patterns similar to those of experts. This view could be supported by the difference found in the mean fixation durations: although the animation was attended in similar patterns, novices spent significantly more time to process the animation. However, the performance, as measured by the comprehension summaries, was different but not significantly.

FUTURE WORK

The results obtained in this initial experiment need to be confirmed further and extended. Future research can take several directions. Our aim is to investigate the effects that a single animation element and the difficult sections of code have on gaze behavior as well as the changes in gaze behavior over time and with increasing experience. The ultimate goal is to support learning with dynamic program visualizations that could adapt to the needs of the users. We believe that using the eye-tracker to collect the actual gaze directions and use it to adapt visualization is a way to get closer to our goal.

In addition, we plan to develop better methodological grounds for applying the eye-tracking in the context of program visualization. As seen from the present study, some of the widely used eye-tracking metrics are not sufficiently able to discover the links between experience, produced mental models and gaze behavior. Therefore, available measures need to be related to the cognitive processes and outcomes and new measures have to be developed. For example, we plan to examine, whether and how attending a certain element or area of animation correlates with the produced mental models captured in program summaries. Another direction could be taken, for instance, when eye tracking is applied to program visualization in real time: the level of dissociation between the attention focus of a programmer and the currently animated location should be investigated. These studies will provide a deeper understanding of the cognitive processes involved in program comprehension during program visualization and will further stimulate the research on the adaptivity of visualization tools.

Adapting program visualization

When visualizing a program, users of the visualization tools have different needs and levels of knowledge. Currently, a user can change some aspects of the visualization manually, but there are only few systems that would automatically adapt to the user's previous experience or needs. This would be beneficial for the user because her cognitive load could be reduced by hiding details that are not relevant to the current task of users or are already known to the users.

There are several scenarios how Jeliot could be made more adaptive. First of all, we could collect data from a learning environment for programming and use user modeling to analyze which programming concept the user currently does or does not understand. With the information on user knowledge, we could change the granularity of the visualization to emphasize those concepts that the users are currently struggling with and minimize the details of those that they already understand (Brusilovsky & Su, 2002).

The visualization could be adaptive in real time, based on the data collected from the user, for example, via an eye tracker. With an eye tracker, one can track the gaze path during a comprehension task and therefore can obtain insights into the user's allocation of visual attention and attention switching between different areas in an interface. Knowledge of gaze-related patterns provides us with important aspects of the underlying cognitive processes and could be a hint as to the needs of the users. As it has also been argued elsewhere (Cross et al., 1999), supporting those cognitive processes that are beneficial to program visualization, and therefore also to comprehension, can yield better effects on learning. Thus, one of the aims of the research presented in this paper was to discover the basic patterns of eye movements during program comprehension supported by dynamic visualization and thereby provide the research community with a starting point and means of adapting and tailoring the visualizations to personal needs. Till now, the eye- movement research has mainly concentrated on utilization of voluntary eye movements as an alternative gaze-based selection and pointing technique for HCI, but little has been done to exploit the gaze direction during complex dynamic visualizations.

CONCLUSION

We conducted an empirical experiment to discover the similarities and differences between experienced and novice programmers in interaction, in the comprehension process, and in the gaze behavior during program comprehension aided by a visualization tool. We employed non-intrusive remote eye tracking equipment to record the eye movements of programmers.

Our results in terms of attention switches between representations and distribution of fixations show no significant difference in behavior between novice and expert groups of programmers during program animation. The focus of visual attention seems to be distributed in space and also most of the time evenly, regardless of previous experience with programming. When the level of processing required to attend to the animation is measured as the duration of fixations over the main areas of interest and overall, our results show a significant difference.

Acknowledgements

We thank all participants for taking part in this study. We acknowledge Andrés Moreno for help with preparation of this study. We thank the reviewers for comments to improve the manuscript.

REFERENCES

Aaltonen A., Hyrskykari A., Räihä, K.-J. (1998). 101 Spots, or How Do Users Read Menus? In *Proceedings of the CHI'98* (pp.132–139), NY, ACM.

Basili, V. R., Mills, H. D. (1982). Understanding and Documenting Programs. *IEEE Transactions on Software Engineering*, 8(3), (pp. 270–283).

Bednarik, R., Tukiainen, M. (2004). Visual attention tracking during program debugging. In *Proceedings of NordiCHI'04* (pp. 331–334), NY: ACM.

Bednarik, R., Myller, N., Sutinen, E., Tukiainen, M. (2005) Effects of Experience on Gaze Behaviour during Program Animation. *Proceedings of the 17th Annual Psychology of Programming Interest Group Workshop* (PPIG'05) (pp. 49-61), Brighton, UK, June 28 - July 1, 2005.

Ben-Ari, M., Myller, N., Sutinen, E., Tarhio, J. (2002). Perspectives on Program Animation with Jeliot. In S. Diehl (Ed.), *Software Visualization* (pp. 31–45), Berlin: Springer-Verlag.

Ben-Bassat Levy, R., Ben-Ari, M., Uronen, P.A. (2003). The Jeliot 2000 Program Animation System. *Computers & Education*, 40(1), (pp. 1–15).

Boroni, C. M., Eneboe, T.J., Goosey, F.W., Ross J. A., Ross, R. J. (1996) Dancing with Dynalab - Endearing the Science of Computing to Students. In *Proceedings of the 27th SIGCSE Technical Symposium on Computer Science Education* (pp. 135–139), NY: ACM.

Brusilovsky, P. and Su, H.-D. (2002). Adaptive Visualization Component of a Distributed Web-based Adaptive Educational System. In *Proceedings of 6th International Conference on Intelligent Tutoring Systems 2002* (pp. 229–238), Berlin: Springer-Verlag.

Byckling, P., Kuittinen, M., Nevalainen, S., Sajaniemi, J. (2004). An Inter-Rater Reliability Analysis of Good's Program Summary Analysis Scheme. In *Proceedings of the 16th Annual Workshop of the Psychology of Programming Interest Group* (pp. 170–184).

Crosby, M. and Peterson, W., (1991). Using eye movements to classify search strategies. In *Proceedings of the Human Factors Society 35th Annual Meeting* (pp. 1476–1480).

Crosby, M. E., Scholtz, J., Wiedenbeck, S. (2002). The Roles Beacons Play in Comprehension for Novice and Expert Programmers. In J. Kuljis, L. Baldwin, R. Scoble (Eds.), *Proceedings of the 14th Annual Workshop of the Psychology of Programming Interest Group* (pp. 58–73).

Crosby, M. and Stelovsky, J. (1989). The influence of user experience and presentation medium on strategies of viewing algorithms. In *Proceedings of 22nd Annual Hawaii International Conference on System Sciences* (pp. 438–446).

Crosby, M. and Stelovsky, J. (1990). How do we read algorithms? A case study. *IEEE Computer*, 23(1), (pp. 24–35).

Cross, J. H., Hendrix, T. D., Mathias, K. S., Barowski, L. A. (1999). Software Visualization and Measurement in Software Engineering Education: An Experience Report. In *Frontiers in Education Conference FIE '99* (vol. 2, pp. 12B1/5–12B110).

Detienne, F. (2002). Software Design – Cognitive Aspects. Springer-Verlag London.

Duchowski, A.T., Shivashankaraiah, V., Rawls, T., Gramopadhye, A. K., Melloy, B., Kanki, B. (2000). Binocular eye tracking in virtual reality for inspection training. In *Tracking Research & Applications Symposium 2000* (pp. 89–96).

Duchowski, A. T., Medlin, E., Cournia, N., Gramopadhye, A. K., Melloy, B., Nair, S. (2002). 3D eye movement analysis for VR visual inspection training. In *Tracking Research & Applications Symposium 2002* (pp. 103–110).

Ericsson, K. & Simon, H. (1984). *Protocol Analysis*. Cambridge, MA: MIT.

Gestwicki, P., Jayaraman, B. (2002). Interactive visualization of Java programs. In *Proceedings of IEEE Symposia on Human Centric Computing Languages and Environments* 2002 (pp. 226–235).

Goldberg, J. H., Kotval, X. P. (1998). Eye Movement-Based Evaluation of the Computer Interface. In S. K. Kumar (Ed.), *Advances in Occupational Ergonomics and Safety* (pp. 529–532), Amsterdam: IOS.

Goldberg, J. H., Kotval, X. P. (1999). Computer Interface Evaluation Using Eye Movements: Methods and Constructs. *International Journal of Industrial Ergonomics*, 24(6), (pp. 631–45).

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

Hoadley, C. H., Linn, M. C., Mann, L. M., Clancy, M. J. (1996). When, Why and How Do Novice Programmers Reuse Code? In W. D. Gray and D. A. Boehm-Davis (Eds.), *Empirical Studies of Programmers: Sixth Workshop* (pp. 109–129), NJ: Ablex.

Hoc, J.-M., Green, T.R.G., Samurcay, R., Gilmore, D.J. (Eds.) (1990). Psychology of programming. Academic Press.

Hundhausen, C. D., Douglas, S. A., Stasko, J. T. (2002). A Meta-Study of Algorithm Visualization Effectiveness. *Journal of Visual Languages & Computing* 13(3), (pp. 259–290).

Jacob, R. J. K. (1993). Eye Movement-Based Human-Computer Interaction Techniques: Toward Non-Command Interfaces. In H. R. Hartson, D. Hix (Eds.), *Advances in Human-Computer Interaction* (Vol. 4, pp. 151–190).

Just, M. A. and Carpenter, P. A. (1976). Eye fixations and cognitive processes. *Cognitive Psychology*, 8, (pp. 441–480).

Just, M. A. and Carpenter, P. A. (1980). A theory of reading: From eye fixations to comprehension. *Psychological Review*, 87(4), (pp. 329–354).

Just, M. A. and Carpenter, P. A. (1984). Using Eye Fixations to Study Reading Comprehension, in D. Kieras & M. Just (eds), *New Methods in Reading Comprehension Research*, Lawrence Earlbaum Associates, Hillsdale, New Jersey, (pp. 151-182).

Kasarskis, P., Stehwien, J., Hickox, J., Aretz, A., and Wickens, C. (2001). Comparison of expert and novice scan behaviors during VFR flight. In *Proceedings of the 11th International Symposium on Aviation Psychology*.

Majaranta, P., Räihä, K-J. (2002). Twenty Years of Eye Typing: Systems and Design Issues. In *Eye Tracking Research & Applications Symposium 2002* (pp. 15–22).

Matin, E. (1974) Saccadic suppression: a review and an analysis. *Psychological Bulletin*, 81(12), (pp. 889–917).

Moreno, A., Myller, N., Sutinen, E., Ben-Ari, M. (2004). Visualizing Programs with Jeliot 3. In *Proceedings of Advanced Visual Interfaces, AVI 2004* (pp. 373–376).

Nodine, C., Mello-Thoms, C. (2000). The nature of expertise in radiology. In J. Beutel, H. Kundel, R. Van Metter (Eds.), *Handbook of Medical Imaging*, WA: SPIE.

Oechsle, R., Schmitt, T. (2002). JAVAVIS: Automatic Program Visualization with Object and Sequence Diagrams Using the Java Debug Interface (JDI). In S. Diehl (Ed.), *Software Visualization*, Vol. 2269 of LNCS, (pp. 176–190). Berlin: Springer-Verlag.

Pennington, N. (1987). Comprehension Strategies in Programming. In G. M. Olson, S. Sheppard, E. Soloway (Eds.), *Empirical Studies of Programmers: Second Workshop* (pp. 100–113), NJ: Ablex.

Rayner, K. (1998). Eye movements in reading and information processing: 20 years of research. *Psychological Bulletin*, 124(3), (pp. 372–422).

Romero, P., du Boulay, B., Cox, R., Lutz, R. (2003). Java debugging strategies in multi-representational environments. In *Proceedings of the 15th Annual Workshop of the Psychology of Programming Interest Group* (pp. 421–434).

Romero, P., Lutz, R., Cox, R., du Boulay, B. (2002). Co-ordination of multiple external representations during Java program debugging. In *Proceedings of the Empirical Studies of Programmers symposium of the IEEE Human Centric Computing Languages and Environments Symposia 2002* (pp. 207–214).

Sajaniemi, J., Kuittinen, M. (2003). Program animation based on the roles of variables. In *Proceedings of the 2003 ACM symposium on Software visualization* (pp. 7–16), NY: ACM.

Shneiderman, B., and R. Mayer. (1979) Syntactic Semantic Interactions in Programmer Behavior: A Model and Experimental Results. *International Journal of Computer & Information Sciences*, 8(3), (pp. 219–238).

Sibert, L. E., Jacob, R. J. K. (2000). Evaluation of eye gaze interaction. In *Proceedings of CHI 2000* (pp. 281–288), NY: ACM.

Zeller, A. and Lütkehaus, D. (1996). DDD, A Free Graphical Front-End for UNIX Debuggers. *ACM SIGPLAN Notices*, 31(1), (pp. 22–27).