Visual attention switching patterns of programmers debugging with an IDE

Prateek Hejmady and N. Hari Narayanan
Intelligent & Interactive Systems Laboratory
Computer Science and Software Engineering Department
Auburn University
Auburn, AL 36849, USA
[ph, naraynh]@auburn.edu

Abstract

Integrated Development Environments (IDEs) generate multiple graphical and textual representations of programs. Co-ordination of these representations during program comprehension and debugging can be a complex task. In order to better understand the role and effectiveness of multiple representations, and to design better IDE interfaces in future, we conducted an empirical study of java program debugging with a multi-representation IDE. It was found that program code and dynamic representations (dynamic viewer, variable watch and output) attracted the most attention. Static representations like Unified Modeling Language (UML) and Control Structure Diagrams (CSD) saw significantly lesser usage. We analyzed gaze patterns by breaking down the debugging sessions into segments of three, five and fifteen minute intervals, and classifying gazes into short and long gazes. Novel data mining techniques were used to detect high frequency patterns from eye tracking data of participants. Significant visual pattern differences were found among the participants based on their programming experience, familiarity with the IDE and debugging performance.

CR Categories: H.5.2 [Information Interfaces and Presentation (e.g., HCI)]: User Interfaces Evaluation/methodology-Input devices and strategies;

Keywords: eye-tracking, psychology of programming, attention patterns, program comprehension, program debugging.

1 Introduction

With the widespread permeation of information technology into all facets of society, programming has become an important technical skill that not only computer scientists but professionals in other disciplines also seek to acquire. Integrated Development Environments (IDE) provide both novice and expert programmers with several functionalities to write, test and debug programs. IDE interfaces typically present multiple graphical and textual representations of programs during compilation and execution. Effective IDE use requires co-ordination of these representations during program comprehension and debugging, which can be quite a demanding task.

There are many popular IDE’s that provide multiple visualization and other sophisticated functionalities to facilitate program comprehension and debugging. These enable programmers to treat programs as a range of abstract entities, besides source code, that can be visualized according to different criteria or executed under a variety of conditions. These visualizations could be either graphical or textual, and present different programming perspectives of the same program [Romero et al. 2003]. For example, during program execution, a visualization could show the change in data elements, and another could simultaneously highlight the program flow. Thus, one of the central skills in effective IDE use is the ability to coordinate appropriate representations and functionalities of the IDE [Romero et al. 2007].

In particular, a programmer needs to be skilled in generating and testing hypotheses about the program’s execution from the various representations provided by an IDE. This ability differs from one programmer to another based on factors such as programming expertise, familiarity with the IDE and personal preference. Novice programmers face the additional challenge of mastering abstract concepts of programming as well as these IDE usage skills. It is hence of utmost importance to better understand the underlying processes and strategies at work during the complex problem solving tasks of program comprehension and debugging in a rich software development environment. Therefore, we conducted an empirical study of java program debugging using a professional IDE with multiple static and dynamic program representations. Results from this study add to the knowledge generated by recent research on this topic reported in the literature.

2 Related Work

Most of the early studies investigating cognitive and behavioral aspects of programming used verbal utterances of participants as data. It has been argued that verbalizing thoughts interfere with a participant’s natural processing, by adding an extraneous cognitive load. This could affect the results as this load may hinder or even alter the problem solving strategies of participants. Users may also skip critical utterances due to different reasons, such as not being aware of some aspects of their own behavior or being less vocal by nature. Since eye-tracking is non-intrusive, it has been suggested as a strong alternative to verbal protocols in capturing the cognitive processes involved in programming. As discussed below, researchers have successfully employed this technique in studies of programming to better understand the underlying cognitive processes.

Crosby and Stelovsky [1989] studied the visual patterns of programmers while reading a binary search algorithm. They found
that experienced users paid more attention to meaningful areas of source code and complex statements. Novice students paid more attention to comments and comparisons. Least attention was paid by both groups to keywords. The two groups did not exhibit any methodical differences in code reading strategies. Crosby and Stelovsky evaluated fixation durations and number of fixations with both qualitative approaches and parametric tests. A limitation of this study was that the only representation available to participants was the program code. This study did not employ any static or dynamic visualizations of the code.

Not many researchers investigated visual attention during programming following this early work, until 2002 when several experiments were conducted by Romero et al. [2002a; 2002b; 2003]. They evaluated co-ordination strategies of programmers while debugging in an environment that provided multiple visualizations. It was found that programmers frequently combined both forward and backward reasoning to debug a program. Frequent switches were made between code and output or code and static visualization of code. Balanced switching behavior was found among those with more programming experience. Statistical tests were used to analyze the visual data collected. The data analyzed was an aggregated average from the beginning to the end of a debugging session for each participant. Visual attention during the experiment was tracked by a Restricted Focus Viewer (RFV) [Blackwell, Jansen, & Marriott, 2000] where the programming environment was presented in a blurred format with a clear window at the location of the mouse cursor that the programmer could control. This way the RFV restricted the amount of stimulus shown to the programmer and facilitated tracking the visual attention of the programmer. It has however been pointed out that RFV can interfere with visual strategies and fixation duration, and that RFV data do not match eye tracking data [Bednarik and Tukiainen, 2007].

Nevalainen and Sajaniemi [2005] investigated the effect of graphical visualizations in the visual patterns of novice programmers. They implemented a within subject design where subjects used two different tools: a traditional text based environment and an environment that provided multiple graphical visualizations. Differences in visual patterns were found between these two tools. The visual attributes used in analysis were fixation durations and proportion of these durations over three different areas of interest (AOI). The data analyzed was an aggregated average from the beginning to the end of a debugging session for each participant.

A predominantly qualitative approach was taken by Uwano et al. [2006] to analyze visual patterns among intermediate programmers engaged in a code review task. They identified a particular pattern, called a scan, in the subjects' eye movements. A scan is an initial reading of the entire code prior to more targeted reading with the goal of finding defects. It was found that reviewers who spent a sufficient time scanning were more efficient at detecting defects than those who did not.

Bednarik et al. [2005; 2006] conducted studies to investigate the effects of experience on debugging strategies in a multi representation dynamic environment. They found that fixation counts and attention switching between representations (code and graphical representations) did not differ based on experience. An effect of experience was found, however, on overall strategies adopted to comprehend programs and on fixation durations. In these studies, average data from the entire session was analyzed.

In order to characterize and analyze coordination of multiple program representations during the program comprehension task, Bednarik and Tukiainen [2006] proposed a new methodological approach and conducted more detailed analyses. They argued that averaging data could result in loss of information regarding the temporal evolution of the strategies programmers employ during comprehension. Instead, they subdivided the comprehension process into meaningful segments and analyzed gradual changes in related eye-movement patterns. Instead of a repeated measures analysis, data values were coded as increasing or decreasing and a binomial analysis was applied. Besides proposing this novel methodology to analyze trends over time, they reported interesting novice-expert differences: (a) less experienced programmers viewed program animations more times and initially focused more on the visualization and later more on the code; (b) in contrast, more experienced programmers studied the code first and then viewed the animation once; and (c) attention switches between code and visualizations decreased over time for all participants. This research used the Jeliot IDE [Moreno et al. 2003].

In subsequent work, Bednarik and Tukiainen [2008] proposed the idea of segmenting a long session into shorter intervals to allow more fine-grained analyses. Ten minute debugging sessions of three Java programs were segmented into five 2-minute intervals for analysis. Participants were divided into two groups based on experience, fixation time ratios and attention switches were calculated, and attention on the three representations were plotted against time. Their findings are as follows. At the beginning, all programmers used code and visualizations with frequent switches between these representations. Later, more experienced programmers also attended to the output and coordinated the three representations. Experts, unlike novices, changed their strategies over time.

The programming environments used in prior research are not representative of the features available in professional IDEs. As far as we know, no prior research has analyzed visual strategies of programmers in the unrestricted environment of a full-fledged IDE. We overcame this limitation of previous studies by utilizing an IDE [JGRASP; Cross et al. 2009] that offers a plethora of representations, including static and dynamic visualizations along with program code, in our experiments. This IDE is used by over 300 academic institutions across the world and also by professionals (www.jgrasp.org).

3 Experiment

3.1 Participants

The participants chosen for this study were graduate and undergraduate students from the authors’ department, who had a minimum of 6 months programming experience in Java. All participants were volunteers and received $10 for each hour of their participation. We recruited 19 participants, 2 female and 17 male, all with normal or corrected vision. None of them had previously participated in an eye tracking study. Their level of programming experience varied, ranging from a sophomore in computer science having taken or being currently enrolled in a data structures class to graduate students who had substantial programming experience, with some who had professional experience in building enterprise applications in Java. The median and mode of their programming experience were in the range of 1 to 2 years. The median for Java programming experience in particular was 1 to 2 years, and the mode was 6 to 12 months.
Four of the participants had never worked with jGRASP before. Of those who had prior experience with jGRASP, all but one participant had used jGRASP for a period ranging 6 - 12 months.

### 3.2 Materials and Apparatus

A Bubble Sort Java program consisting of two classes (a client class with one method and 20 lines of code and a data structure class with 10 methods and 98 lines of code) was developed and seeded with three bugs. These bugs can be classified as control flow, data flow, and functional error. The input was a set of names stored in a doubly linked list. Participants were told that there were no syntactical errors in the program. The names of the methods, variables and classes were altered so that recognition of a program and the underlying data structure based on surface features would be difficult. On execution, the program was designed to print out the correct output (hard-coded) as the “expected output” as well as the actual output produced.

Figure 1 shows the jGRASP interface, with the following components:

- ‘1’ represents the dynamic viewer, which is a movable window, that shows changes to an underlying data structure in real time.
- ‘2’ depicts the Unified Modeling Language representation of class level relationships.
- ‘3’ stands for windows with the source code.
- ‘4’ represents a diagrammatic representation of code structure, unique to jGRASP, shown to the left of the source code. This is called the Control Structure Diagram (CSD) [Cross et al. 2009].
- ‘5’ is the program output.
- ‘6’ shows either a variable watch window or an expression watch window.
- ‘7’ is a menu bar with button controls for stepping through code during debugging.
- ‘8’ is the IDE’s top level bar with menu buttons and icon shortcuts.

![Figure 1 jGRASP IDE interface.](image)

During the debugging session, each participant’s eye movements were tracked. We used the Tobii T60 XL, a remote eye tracker with sampling rate set to 60Hz. This eye tracker was set up in a sound proof room with consistent fluorescent illumination. Participants were seated comfortably in an ordinary office chair, facing a twenty four inch TFT widescreen monitor and maintaining a viewing distance of 55-65cm. The screen resolution was set to 1920 x 1200. Tobii Studio™ 2.1 was used for setting up the experiment. The stimuli sequence was created by combining all the debugging tasks into one jGRASP project. Tobii Studio™ was also employed to create a holistic view of user behavior during debugging by integrating data captured from the recording of eye tracking data with user video, screen capture, sound, keystrokes and mouse clicks. During the experiment, user actions were supervised on a remote computer using Tobii Studio Logger™, which displayed the test screen with real time gaze data overlay.

### 3.3 Procedure

After becoming familiar with the experiment and signing a consent form, each participant was given 10 minutes to understand the functionalities of jGRASP IDE by working with a linear search program in Java that we developed as a warm up exercise. This was cut short if a participant had prior experience with jGRASP. The participant was then subjected to an eye calibration routine, which consisted of tracking their eyes as they followed nine points on the computer screen. This process was repeated if necessary, to achieve good calibration. On successful calibration, a natural language description of the problem statement (sorting) was presented on the computer screen. Next, the IDE was automatically brought up and the participant was asked to locate and fix the bugs in the code within a time limit of 15 minutes. They were allowed to use any of the visualizations available with the IDE. On completion of the debugging session, the participant was interviewed based on a semi structured interview protocol. A pilot study was conducted with three volunteers before the actual study. Minor issues were unearthed based on volunteer feedback and researcher’s observations. These issues were fixed before the actual experiment.

### 4 Data Analysis Methodology

To perform analysis of representation use, Areas of Interests (AOIs) were defined for eight representations provided by the IDE. These were Client Code, Client CSD, Data Structure CSD, Data Structure Code, Dynamic Viewer, Expression Evaluation Window, Output, and Variable Watch Window. During the debugging sessions, the AOI placement on the screen changed due to the movement of windows by programmers. Hence we manually segmented the complete debugging session for each participant into multiple segments, with each having a unique spatial arrangement of AOIs defined for the segment. Figure 2 is a snapshot of AOIs defined for a segment extracted from a debugging session, with the numbers representing the previously described interface components. Analysis was performed for representation use with three different visual attributes. Fixation Count, Dwell Time and Visit Count. Fixation Count of a participant for a representation is the total number of the participant’s fixations on that representation over his or her entire debugging session. Fixation detection was based on Tobii’s clear view fixation filter with fixation radius set to 50 pixels and minimum fixation duration of 100 milliseconds. Similarly, Dwell Time on a representation is the total time in milliseconds the participant spent on the representation, and the Visit Count for a representation is the total number of times the participant shifted his or her gaze from elsewhere to the representation. To analyze the visual patterns of programmers, attention on each AOI was categorized as a short duration gaze or a long duration gaze on Code (Client...
Attern Mining (SPAM) algorithm), Static Visualization (Client CSD or Data Structure CSD), Dynamic Visualization (Dynamic Viewer, Expression Evaluation Window, or Variable Watch Window) and Output, and coded using letters of the alphabet. Thus, we had eight categories of visual attention (Table 1). The visual pattern of each participant over his or her entire session was coded as a string made up of the characters A-H. For example, a string AFG represents a programmer spending a short duration of time on Code followed by a long duration on Dynamic Visualization, further followed by a short duration on Output. The string AEAEAF represents alternating short gazes on Code and Dynamic Visualization AOIs. In order to separate short and long durations, we used the threshold value of 500ms. Recurring patterns were algorithmically extracted from these strings. For instance, AE is a recurring pattern in the above example string.

![Figure 2 Eight AOI's defined for a segment with dynamic viewer in use.](image)

A utility program was developed to process the raw data, and based on the desired attributes (such as gaze duration and AOI dimensions) the program generated visual pattern sequences representing attention switches among Code, Static Visualization, Dynamic Visualization and Output coded as character strings. Once the visual pattern for all 19 participants was known, the Sequential P-Atern Mining (SPAM) algorithm (Ayres et al., 2002) was applied on these patterns to mine the frequently occurring visual pattern sequences. SPAM finds all frequent sequences within a transactional database. This format was created as a text file and then processed by SPAM to discover recurring patterns. As the SPAM algorithm produces only the recurring patterns and not their frequency, the utility program we developed was also used to perform a frequency count of the patterns discovered by SPAM. This operation was performed for each participant.

## 5 Results

We analyzed the data from all participants to detect differences in visual attention allocated to the different IDE representations. Results from this analysis are presented in sections 5.1, 5.2 and 5.3. In section 5.4, we present results from an analysis of patterns of visual attention switching among the various IDE representations.

### 5.1 Fixation Count

Per minute fixation count on a representation for a participant was computed by dividing the Fixation Count for that representation by the participant’s session length in minutes. Mean fixation count per minute for a representation is the average of per minute fixation count over the 19 participants. This was computed for each of the eight representations in use. A one-way ANOVA revealed that fixation counts differed significantly across the eight AOI’s, $F_{0.05}(7, 144) = 107.95, p<.001$.

![Figure 3 Mean fixation count](image)

Schefé’s post-hoc comparisons of the eight groups indicated that the Client Code received significantly higher number of fixations than any of the other seven representations ($M = 71.92, SD=11.3, 95\% CI [66.45, 77.38]$). This was followed by fixation counts on the Data Structure Code ($M = 32.47, SD=18.1, 95\% CI [23.73, 41.22]$), then the Dynamic Viewer ($M = 21.46, SD=14.14, 95\% CI [14.65, 28.28]$), and the Output ($M = 17.41, SD=9.59, 95\% CI [12.79, 22.04]$).

### 5.2 Dwell Time

Per minute dwell time on a representation for a participant was computed by dividing the Dwell Time for that representation by the participant’s session length in minutes. Mean dwell time per minute for a representation is the average of per minute dwell time over the 19 participants. This was computed for each of the

<table>
<thead>
<tr>
<th>Character Representation</th>
<th>Gaze Duration on AOI (Short)</th>
<th>Character Representation</th>
<th>Gaze Duration on AOI (Long)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Code</td>
<td>B</td>
<td>Code</td>
</tr>
<tr>
<td>C</td>
<td>Static Visualization</td>
<td>D</td>
<td>Static Visualization</td>
</tr>
<tr>
<td>E</td>
<td>Dynamic Visualization</td>
<td>F</td>
<td>Dynamic Visualization</td>
</tr>
<tr>
<td>G</td>
<td>Output</td>
<td>H</td>
<td>Output</td>
</tr>
</tbody>
</table>

Table 1 Dwell time based AOI coding

Data mining was necessitated by the fact that there are many different combinations of possible attention switches. The visual pattern of each participant was converted to a representation resembling a transactional database record. This was performed as a text file and then processed by SPAM to discover recurring patterns. As the SPAM algorithm produces only the recurring patterns and not their frequency, the utility program we developed was also used to perform a frequency count of the patterns discovered by SPAM. This operation was performed for each participant.
eight representations in use. A one-way ANOVA revealed that mean dwell times differed significantly across the eight AOI’s, \( F_{0.05} (7, 144) = 114.93, p < .001 \).

Scheffè’s post-hoc comparisons indicated that the Client Code attracted significantly higher dwell time than any of the other seven representations (\( M = 44.51, SD=6.34, 95\% \ CI [41.45, 47.56] \)). This was followed by the dwell time on the Data Structure Code (\( M = 21.6, SD=11.19, 95\% \ CI [16.24, 27.02] \)), then the Dynamic Viewer (\( M = 13.63, SD=8.72, 95\% \ CI [9.42, 17.8] \)), and the Output (\( M = 9.05, SD=4.7, 95\% \ CI [6.78, 11.31] \)).

Figure 4 Mean dwell time

5.3 Visit Count

Per minute visit count on a representation for a participant was computed by dividing the Visit Count for that representation by the participant’s session length in minutes. Mean visit count per minute for a representation is the average of per minute visit count over the 19 participants. This was computed for each of the eight representations in use. A one-way ANOVA revealed that visit counts differed significantly across the eight AOI’s, \( F_{0.05} (7, 144) = 43.55, p < .001 \).

\[ \text{Visit Count} = \frac{\text{Visit Count for that representation}}{\text{Participant's session length in minutes}} \]

Figure 5 Mean visit count

5.4 Visual Pattern Analysis

Based on the results generated from SPAM and subsequent frequency count over the entire session, the attention switching patterns shown in Figure 6 were prominent across all 19 participants. The pattern frequency (y-axis) is the average over 19 participants of the average per minute frequency of that pattern for each individual participant. The short gaze switch between Code and Dynamic Visualization saw the highest frequency, followed by short gaze switches between Code and Output and short gaze switch between Code and Static Visualization. These three patterns were followed by short gaze switches between Dynamic Visualization and Output. Nine of the most occurring patterns are listed in sorted order of their frequency in Table 2, with pattern one being the most frequent.

<table>
<thead>
<tr>
<th>Visual Pattern</th>
<th>Gaze Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>AE</td>
<td>Short gaze on Code followed by short gaze on Dynamic Visualization</td>
</tr>
<tr>
<td>AG</td>
<td>Short gaze on Code followed by short gaze on Output</td>
</tr>
<tr>
<td>AC</td>
<td>Short gaze on Code followed by short gaze on Static Visualization</td>
</tr>
<tr>
<td>EG</td>
<td>Short gaze on Dynamic Visualization followed by short gaze on Output</td>
</tr>
<tr>
<td>BE</td>
<td>Long gaze on Code followed by short gaze on Dynamic Visualization</td>
</tr>
<tr>
<td>BG</td>
<td>Long gaze on Code followed by short gaze on Output</td>
</tr>
<tr>
<td>CE</td>
<td>Short gaze on Static Visualization followed by short gaze on Dynamic Visualization</td>
</tr>
<tr>
<td>AEG</td>
<td>Short gaze on Code followed by short gaze on Dynamic Visualization and then short gaze on Output</td>
</tr>
<tr>
<td>ACE</td>
<td>Short gaze on Code followed by short gaze on Static Visualization and then short gaze on Dynamic Visualization</td>
</tr>
</tbody>
</table>

Table 2 Visual patterns based on short & long gaze duration

In addition, we explored the differences in visual strategies of participants based on three independent variables, programming experience (high vs. low), familiarity with jGRASP (high vs. low) and debugging performance (good vs. bad). The number of participants belonging to each of these categories and their relative performance are given in Table 3.
5.4.1 Based on Programming Experience

We first looked at pattern differences based on programming experience of participants. A frequency count of the three most prominent patterns discovered by the SPAM algorithm was performed and plotted against time. In the following figures, blue lines represent the pattern short gaze on Code followed by short gaze on Static Visualization, red lines represent the pattern short gaze on Code followed by short gaze on Dynamic Visualization, and green lines represent the pattern short gaze on Code followed by short gaze on Output. Figures 7 and 8 show the mean frequency of these patterns observed among the participants over the complete duration of the experiment. The vertical axis represents the frequency count of the pattern and the horizontal axis represents the time, with each data point representing a 15 second interval. Here again, the mean frequency count (y-axis) is the average over the 19 participants of the average per minute frequency of the corresponding pattern for each individual participant. Figure 7 represents the visual patterns of the less experienced programmers and Figure 8 represents the same for the more experienced programmers.

Session data over 15 minutes were divided into three five-minute intervals and pattern frequencies were calculated for each interval. The frequencies were compared between the more and less experienced programmers for each pattern. More experienced programmers exhibited a higher occurrence of short...
gaze attention switching between Code and Output than the less experienced ones, at a statistically significant level ($t(17) = 2.59$, $p < .05$) during the first interval. All 19 participants switched attention between Code and Dynamic Visualization, more than any other switching pattern, and this pattern’s frequency increased over time.

### 5.4.2 Based on Experience with jGRASP

Figures 9 and 10 are consistent with the previous figures. The pattern frequencies of high and low jGRASP experienced participants were compared for each pattern. For the complete duration of the experiment, less experienced jGRASP users exhibited more frequent pattern of switching between Short Code followed by Short Output at a statistically significant level ($t(17) = -3.24$, $p < .05$). More experienced users on the contrary accounted for higher frequency of visual pattern, Short Code followed by Short Dynamic Visualization during the same period. This was statistically significant ($t(17) = 2.88$, $p < .05$) when compared to less experienced participants.

![Figure 9: Visual patterns - low experience with jGRASP](image)

![Figure 10: Visual patterns - high experience with jGRASP](image)

When patterns were analyzed for 3 intervals of five minutes each, more experienced participants exhibited a higher occurrence of short gaze attention switching between Code and Output compared to less experienced participants at a statistically significant level, during the second ($t(17) = 2.254$, $p < .05$) and third interval ($t(17) = 2.795$, $p < .05$). However less experienced participants exhibited higher switches during first interval, at a significant level ($t(17) = 2.59$, $p < .05$).

### 5.4.3 Based on Debugging Performance

Figures 11 and 12 likewise show the dominance of Code - Dynamic Visualization attention switching for all 19 participants. In addition, for the third 5-minute interval, poor performers exhibited a higher occurrence of short gaze attention switching between Code and Dynamic Visualization than good performers at a statistically significant level ($t(16) = 3.14$, $p < .05$). This difference was close to being statistically significant for interval two as well ($t(17) = 1.96$, $p = .066$). It is also interesting to note from Figures 9 and 10 that during the final two 5-minute intervals good performers did hardly any attention switching between Code and Static Visualization (the blue line) and very little attention switching between Code and Output (the green line). Most of their attention was divided between Code and Dynamic Visualization. In contrast, poor performers exhibited all three of these attention switches throughout the session.

![Figure 11: Poor performers’ visual patterns](image)

![Figure 12: Good performers’ visual patterns](image)

### 6 Discussion

Research reported in this paper advance the knowledge about strategies of programmers engaged in the complex problem solving tasks of program comprehension and debugging. Our findings corroborate those from the literature, but within the context of a multi-representational IDE that is widely used both in education and by professional programmers. We analyzed attention allocation on eight different static and dynamic program representations, more than what has been done in existing research. Attention allocation patterns of programmers were then analyzed using a sequential pattern mining algorithm.

We found that mean fixation count, dwell time and visit count data are consistent with each other, and show that Code received the most attention, followed by Dynamic Visualization (Dynamic Window and Variable Watch) and Output. Static
Visualization did not receive comparable attention from programmers. We identified nine most commonly occurring attention switching patterns during the entire session over all participants. We further plotted mean frequencies of the most common three attention switching patterns over the entire session. These plots revealed that attention switches between Code and Dynamic Visualization was most frequent, and its frequency increased over time. We also found that good performers switched their attention between Code and Dynamic Visualization lesser towards the end of the session compared to poor performers. There was no difference in this pattern observed between the programmers with high and low experience, although the frequency of this pattern increased towards end overall. Less experienced IDE users exhibited higher frequency of this pattern compared to the more experienced IDE users.

More experienced programmers tended to look at the program output more frequently than less experienced programmers in the beginning of the debugging task. Participants with more experience with the jGRASP IDE looked more often at the dynamic visualizations and less at the program output when compared to participants with less experience in using jGRASP. Participants who performed better in debugging activity did not perform frequent switches with the static visualizations and output towards the end of the session when compared to poor performers. Poor performers switched more between the dynamic visualizations and code in the beginning of the debugging session when compared to participants with good performance.

These results further flesh out the emerging picture of program comprehension and debugging with multi-representational IDEs that has been reported in the literature. Consistent with previous studies [Bednarik, et. al., 2006 and Romero, et. al., 2002a], it was found that source code received the highest attention followed by dynamic visualizations and output. However, unlike [Bednarik and Tukiainen 2006], we found the frequency of attention switches between source code and dynamic visualizations increasing towards the end of the debugging session.

Several limitations of this work need to be noted. One limitation is that data, especially attentional patterns, may depend not only on the expertise and experience, or lack thereof, of the participants, but also on the nature and complexity of the program being comprehended/debugged and interface features of the IDE being used. So one must use caution in generalizing results of a study using a specific program and IDE. What is needed being comprehended/debugged and interface features of the IDE and an account of individual differences are quite difficult to achieve.

These plots revealed that attention switches between Code and Dynamic Visualization was most frequent, and its frequency increased over time. We also found that good performers switched their attention between Code and Dynamic Visualization lesser towards the end of the session compared to poor performers. There was no difference in this pattern observed between the programmers with high and low experience, although the frequency of this pattern increased towards end overall. Less experienced IDE users exhibited higher frequency of this pattern compared to the more experienced IDE users.

More experienced programmers tended to look at the program output more frequently than less experienced programmers in the beginning of the debugging task. Participants with more experience with the jGRASP IDE looked more often at the dynamic visualizations and less at the program output when compared to participants with less experience in using jGRASP. Participants who performed better in debugging activity did not perform frequent switches with the static visualizations and output towards the end of the session when compared to poor performers. Poor performers switched more between the dynamic visualizations and code in the beginning of the debugging session when compared to participants with good performance.

These results further flesh out the emerging picture of program comprehension and debugging with multi-representational IDEs that has been reported in the literature. Consistent with previous studies [Bednarik, et. al., 2006 and Romero, et. al., 2002a], it was found that source code received the highest attention followed by dynamic visualizations and output. However, unlike [Bednarik and Tukiainen 2006], we found the frequency of attention switches between source code and dynamic visualizations increasing towards the end of the debugging session.

Several limitations of this work need to be noted. One limitation is that data, especially attentional patterns, may depend not only on the expertise and experience, or lack thereof, of the participants, but also on the nature and complexity of the program being comprehended/debugged and interface features of the IDE being used. So one must use caution in generalizing results of a study using a specific program and IDE. What is needed being comprehended/debugged and interface features of the IDE and an account of individual differences are quite difficult to achieve.

7 Conclusion

Complex cognitive processes active during program debugging in a multi representation programming environment can be better understood by tracking visual attention.

This paper explored the differences in visual patterns of programmers based on their programming experience, experience with an IDE and debugging performance. The visual pattern also took into consideration the duration of gaze on each area of interest by analyzing both short and long gazes. As mentioned by Bednarik and Tukiainen [2006], in order to better understand programmer strategies and behavior, a finer analysis need to be undertaken involving finite points of user actions. This could be categorized based on for example type of bugs found or order of bugs found.

The results of the experiment reported here need to be reinforced by further empirical studies of programming under different experimental settings. Some of the settings to manipulate would be the program attributes like its architecture, number and type of bugs; programming environment; and restricting availability of representations use during debugging.

References


Romero, P., Cox, R., du Boulay, B., and Lutz, R. 2002a. Visual attention and representation switching during Java program de-


