Multiple Visualizations and Debugging: How do we co-ordinate these?

Abstract
There are many popular Integrated Development Environments (IDE) that provide multiple visualizations and other sophisticated functionalities to facilitate program comprehension and debugging. To better understand the effectiveness and role of multiple visualizations, we conducted a preliminary study of java program debugging with a professional, multi-representation IDE. We found that program code and dynamic representations (dynamic viewer, variable watch and output) attracted the most attention of programmers. Static representations like Unified Modeling Language (UML) Diagrams and Control Structure Diagrams (CSD) saw significantly lesser usage. Interesting eye gaze patterns of programmers were also revealed by the study.

Keywords
Eye-tracking, psychology of programming, attention patterns, program comprehension and program debugging.

ACM Classification Keywords
H.5.2 [Information Interfaces and Presentation (e.g., HCI)]: User Interfaces Evaluation/methodology-Input devices and strategies
General Terms
Human factors, Experimentation, Measurement

Introduction
IDE interfaces typically present multiple graphical and textual representations of programs during compilation and execution. These visualizations could be either graphical or textual, and present different programming perspectives of the same program [4]. Effective IDE use requires co-ordination of these representations during program comprehension and debugging, which can be quite a demanding task.

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 [jGRASP; 3] that offers a plethora of representations, including static and dynamic visualizations along with program code, in our experiments. jGRASP is used by over 300 academic institutions across the world and also by professionals (www.jgrasp.org).

Figure 1 shows the jGRASP interface, with the following components. '1' represents the dynamic viewer, which is a movable window that displays 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' also contains 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) [3]. '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.

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 to the program 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.

Method
Participants
The participants were graduate and under-graduate students from the authors’ department, and had a minimum of 6 months programming experience in Java. 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. 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.
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. 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.

**Procedure**

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. Participants’ eyes were calibrated with the eye tracker and presented with a problem description. Following this, the buggy Bubble Sort program was presented and the participant was instructed to find the bugs within 15 minutes.

**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 that segment. Analysis was performed for representation use with two visual attributes, Dwell Time and Visit Count. 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 that representation.

**Results**

We first analyzed the data from all participants to detect the visual attention allocated to each of the different IDE representations.

**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 eight representations in use as shown in figure 2. A one-way ANOVA revealed that mean dwell time differed significantly across the eight AOI’s, $F_{0.05}(7, 144)= 114.93$, $p<.001$. Scheffe’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]).

**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 total number of visits the participant made to that representation in the session, averaged over all segments. A two-way ANOVA with session length and representation as factors revealed a significant main effect of representation, $F_{0.05}(7, 144)= 114.93$, $p<.001$. Scheffe’s post-hoc comparisons indicated that the Client Code was visited significantly more often than any of the other seven representations ($M = 19.78$, $SD=8.0$, 95% CI [16.7, 22.8]). This was followed by the dwell time on the Data Structure Code ($M = 16.24$, $SD=11.19$, 95% CI [11.82, 16.66]), 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]).
representation is the average of per minute visit count over the 19 participants. This was computed for each of the eight representations in use as shown in figure 3. A one-way ANOVA revealed that visit counts differed significantly across the eight AOI’s, $F_{(7, 144)} = 43.55$, $p<.001$. Scheffe’s post-hoc comparisons indicated that the Client Code attracted significantly higher number of visits than any of the other seven representations ($M = 8.8$, $SD=2.15$, 95% CI [7.84, 9.92]). This was followed by the visit count on the Data Structure Code ($M = 4.67$, $SD=2.56$, 95% CI [3.44, 5.91]), then the Dynamic Viewer ($M = 4.41$, $SD=2.71$, 95% CI [3.1, 5.72]) and the Variable Watch Window ($M = 3.26$, $SD=1.91$, 95% CI [2.33, 4.19]).

**Visual Pattern Analysis**

The visual pattern of each participant was converted to a representation resembling a transactional database record in the form of a string of characters. Each character represented an individual AOI. This string sequence was created as a text file and then processed by the Sequential PAttern Mining (SPAM) [1] data mining algorithm to generate recurring patterns from this data. SPAM finds all frequent sequences within a transactional database. As the SPAM algorithm produces only the recurring patterns and not their frequency, a utility program was used to perform a frequency count for these patterns for each participant. This operation was performed for multiple combinations of AOIs that we were interested in, as explained below.

The first combination consisted of 4 AOI’s Code, Static Visualization, Dynamic Visualization and Output. Dynamic Visualization was a combination of AOIs for Dynamic Viewer, Expression Evaluation Window and Variable Watch Window whereas Static visualization was a combination of CSD and UML diagrams. Code was a combination of all AOIs containing source code. The frequency of common visual patterns thus discovered are presented in figure 4.

![Figure 4. Mean frequency count with 4 AOI’s](image)

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. X-axis shows the patterns. For example, Code-Static Visualization is the pattern indicating an attention switch between any source code window and any static visualization window on the IDE interface. This remains true for figure 5 and figure 6 as well.

Next, we separated the dynamic and static visualizations into their constituent AOIs, (Code, CSD, UML, Variable Watch, Dynamic Viewer, Evaluation Window and Output). The resulting patterns are shown in figure 5. In third analysis, gazes on each individual AOI were categorized as short or long. Gaze shorter than threshold value of 500ms was classified as short.
and others as long. Most frequent short and long gaze patterns on the 4 combined AOI’s (Code, Dynamic, Static and Output) are shown in figure 6.

![Figure 5](image1.png)
**Figure 5.** Mean pattern frequency with seven AOI’s

![Figure 6](image2.png)
**Figure 6.** Mean pattern frequency with dwell time based AOI’s

Findings 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 seven 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 dwell time and visit count data are consistent with each other, and show that Code received the most attention, followed by Dynamic Visualization (Dynamic Viewer and Variable Watch) and Output. Static Visualization did not receive comparable attention from programmers. These results flesh out, and are consistent with, the emerging picture of program comprehension and debugging with multi-representational IDEs that has been reported in the literature [2, 4].

Visual attention switching patterns were further analyzed. It was found that in general Code and Dynamic Visualization saw the most switches followed by Code and Output. Further analysis found that most switches were between the Variable Watch window and Code followed by Code and Dynamic Viewer switches and Code and Output attention switches. Gaze duration based analysis found that there were frequent short gaze switches between Code and Dynamic Visualization suggesting that programmers were engaged in a verification task during debugging. This was followed by short gaze switches between Code and Output and between Code and Static Visualization, further suggesting that programmers are verifying program
behavior through these attention switches between source code representations and representations of the results of executing the code.

Conclusions and future work
Complex cognitive processes active during program debugging in a multi representation programming environment can be better understood by tracking the visual attention of programmers. This paper explored the visual patterns of programmers during a program debugging task. The analysis also took into consideration the duration of gazes on each area of interest (each representation) on an IDE interface. This study provided some interesting observations in terms of programmer preferences for various IDE representations and their patterns of visual attention switching during program debugging activity.

One limitation of this work is that data, especially visual attention 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 is a large collection of studies generating converging lines of evidence supporting common conclusions. Therefore, findings from the exploratory experiment reported here need to be reinforced by further empirical studies of programming under different experimental settings. Some of the variables to manipulate would be program attributes like its architecture, number and type of bugs, programming environment, and restricting the availability of various representations during debugging. As Bednarik and Tukiainen [2] suggest, finer analysis need to be undertaken in order to better understand programmer strategies and behavior. This is part of our future work.

References