# Debugging strategies and tactics in a multi-representation software environment

Pablo Romero, Benedict du Boulay, Richard Cox
Rudi Lutz and Sallyann Bryant
Department of Informatics
Sussex University, U.K.

August 23, 2007

**Abstract**

This paper investigates the interplay between high level debugging strategies and low level tactics in the context of a multi-representation software development environment (SDE). It investigates three questions. 1. How do programmers integrate debugging strategies and tactics when working with SDEs? 2. What is the relationship between verbal ability, level of graphical literacy and debugging (task) performance. 3. How do modality and perspective influence debugging strategy and deployment of tactics? The paper extends the work of Katz & Anderson (1988) and others in terms of identifying high level debugging strategies to include following execution. It also describes how programmers of different backgrounds and degrees of experience make differential use of the multiple sources of information typically available in a software debugging environment. Individual difference measures considered among the participants were their programming experience and their knowledge of external representation formalisms. The debugging environment allowed the participants, computer science students, to view the execution of a program in steps and provided them with concurrently displayed, adjacent, multiple and linked programming representations. These representations comprised the program code, two visualisations of the program and its output. The two visualisations of the program were available, in either a largely textual format or a largely graphical format so as to track interactions between experience and low level mode-specific tactics, for example.

The results suggest that i) additionally to deploying debugging strategies similar to those reported in the literature, participants also employed a strategy specific to SDEs, *following execution*, ii) verbal ability was not correlated with debugging performance, iii) knowledge of external representation formalisms was as important as programming experience to succeed in the debugging task, and iv) participants with greater experience of both programming and external representation formalisms, unlike the less experienced, were able to modify their debugging strategies and tactics effectively when working under different format conditions (i.e. when working with either largely graphical or largely textual visualisations) in order to maintain their high debugging accuracy level.

Keywords: multiple external representations, graphical reasoning, program debugging.

## 1 Introduction

Much computer programming is performed via the use of software development environments which provide a variety of external representations and other sophisticated functionality. These representations and functionality enable programmers to treat programs not just as code text, but also as a range of abstract entities which can be visualised according to different criteria or executed under a variety of conditions.

This means that the kinds of high level debugging strategy identified by (Katz & Anderson, 1988) will now be interwoven with low level tactics associated with choosing which representations and functionality to exploit as well as being extended at the high level by possibilities opened up by the new functionality.

These representations help the programmer to visualise the program through different perspectives or information types. For example, some perspectives highlight the transformations which data elements undergo as they are processed, while others show the sequence of actions that will occur when the program is executed. Visualisations can be presented in formats that range from mostly textual to mostly graphical (Romero, Cox, du Boulay & Lutz, 2003). Very frequently a number of these visualisations contain links to one another and are displayed concurrently and side by side.

In terms of debugging strategies and tactics, the step facility is one of the most helpful pieces of functionality of such environments. This facility allows programmers to execute and pause the program at different points. At these points they can inspect the visualisations provided to obtain information about various aspects of the execution state.

Such program visualisation and debugging facilities should, in principle, be especially helpful for novice programmers because they have the potential to enable them see the program not as a black box but as an abstract machine containing a set of elements and moving between states. However, their effective use requires strategic knowledge about how to generate and test debugging hypotheses from the evidence in the program's output and visualisations, knowledge about how to decode and coordinate the available representations as well as skill in operating the SDE itself. It is often assumed that novices possess this knowledge and these skills. Thus, novice programmers can face a double challenge. As well as trying to learn abstract concepts about programming, they have to master the decoding, representation coordination and step-and-trace skills required to use debugging environments.

This paper characterises the debugging strategies and tactics of Java programmers in terms of step-and-trace choices and representation usage in the multi-representation debugging environment, relating these aspects of their behaviour to debugging accuracy, experience and knowledge of external representation formalisms. Section 2 explores research in programming strategy focusing on the way programmers manipulate the tools and representations available. Section 3 describes the experimental design and method. Section 4 presents the results of this experiment and Section 5 discusses these results. Finally, Section 6 presents some conclusions and describes further work.

## 2 External representation usage in programming

Good performance in programming tasks is as much dependent on the strategies and tactics chosen to accomplish programming tasks as it is on the programmers' knowledge about the syntax and semantics of the programming language (Gilmore, 1990). In program debugging, strategy is usually related to the high level, systematic plan to identify program errors while tactics have to do with lower-level actions performed to, for example, coordinate and integrate multiple sources of information when using a SDE. According to Katz & Anderson (1988), bug finding strategies can be broadly classified into *forward reasoning* and *backward reasoning.* The first category comprises those strategies in which programmers start searching for bugs from the program code, while the second involves starting from the incorrect behaviour of the program (typically its output) and reasoning backwards to the origin of the problem in the code. Examples of forward reasoning include *comprehension*, where bugs are found while the programmer is building a mental representation of the program and *hand simulation*, where programmers evaluate the code as if they were the computer. Backward reasoning includes strategies such as *simple mapping* and *causal reasoning.* In simple mapping the program's output points directly to the incorrect line of code, while in causal reasoning the search starts from the incorrect output going backwards towards the code segment that caused the bug.

Related but lower-level tactics have to do with the coordination of the available representations and the operation the SDE itself (mainly of its step-and-trace facility). These tactical aspects are particularly important for novice programmers. An inability to cope with these demands, frequently due to cognitive overload (van Bruggen, Kirschner & Jochems, 2002), means that multiple sources of information, instead of improving performance and learning, can sometimes impede them (Bodemer, Ploetzner, Feuerlein & Spada, 2004).

The step-and-trace facility of the SDE is particularly important as it can transform a continuous animation of the program behaviour into a sequence of discrete steps. Animations are ephemeral and sometimes too quick to be accurately perceived, however judicious use of interactivity can help to avoid these difficulties (Tversky & Morrison, 2002).

When working with SDEs, high level debugging strategies need to be supported by the low level tactics used to coordinate the available representations and to operate the SDE itself. Although there have been studies that have looked at debugging strategies (Katz & Anderson, 1988; Mulholland, 1997; Prabhakararao, Cook, Ruthruff, Creswick, Main & Durham, 2003; Chintakovid, Wiedenbeck, Burnett & Grigoreanu, 2006; Grigoreanu, Beckwith, Fern, Yang, Komireddy, Narayanan, Cook & Burnett, 2006) and tactics (Romero, Cox, du Boulay & Lutz, 2002; Romero, Lutz, Cox & du Boulay, 2002; Bednarik & Tukiainen, 2004), it is not clear how programmers integrate them.

## 2.1 Factors affecting strategy and tactics

When using a SDE to debug a program, there are a number of factors that can influence the quality of the strategy and tactics deployed. Some of these are programming experience, the form and nature of the visualisations employed and individual differences associated with representational format.

Research on code generation has highlighted the reliance of experienced programmers upon external aids and the strategic knowledge required to make use of them. Davies (1993a), for example, has shown that experienced programmers, unlike novices, are strongly affected by restrictions in their normal working environment because they are forced to use their working memory to hold information that otherwise would be stored and accessed through the environment. Generally speaking, some forms of strategy can be explained in terms of the properties of the knowledge that programmers develop through experience, and this experience and associated strategies are related to improved performance (Davies, 1993b).

The form of the available visualisations can be an important factor in multi-representational environments. Here a common distinction is between propositional and diagrammatic representations. Research in this area has focused on the advantages and disadvantages of mixing modalities in multi-representational environments. According to Ainsworth, Wood & Bibby (1996), in general, the more different the degree of graphicality external representations exhibit, the more difficult it is for students to coordinate them. On the other hand, it might be that graphical representations, by constraining the interpretation of textual ones because of their weak expressiveness (Stenning & Oberlander, 1995), could promote improved understanding. Additionally, it is not clear is how modality influences task strategy.

While modality is concerned with form, perspective is concerned with content. Perspective refers to the programming information types that a representation highlights. Computer programs are information structures that comprise different types of information (Pennington, 1987), and programming notations usually highlight some of these aspects at the cost of obscuring others (the *match-mismatch hypothesis* (Gilmore & Green, 1984)). Some of these different information types are: function, data structure, operations, data-flow and control-flow. Program visualisations usually highlight some of these information types and knowing, for example, which visualisation to use for which kind of error is part of the programmer's strategic knowledge.

In the context of debugging with SDEs, individual differences associated with representational format preference are potentially important. People differ in terms of their preferences for particular forms of representation, their skill at decoding them, and educational background among other factors. As mentioned above, a typical distinction in representational format is usually between propositional and diagrammatic representations and a number of studies have focused on comparing verbal and diagrammatic ability. Individual differences in external representation use have been studied extensively in various domains (logic reasoning (Oberlander, Stenning & Cox, 1999), mechanical systems (Kriz & Hegarty, 2004) and HCI (Campagnoni & Ehrlich, 1989), among others). Recently, the amount of background knowledge people have of external representation formalisms, or 'graphical literacy', has been proposed as an important type of individual difference (Cox, 1996), and one that might have particular relevance for computer programming (Cox, Romero, du Boulay & Lutz, 2004). Although there seem to be advantages in having a high level of graphical literacy (Cox, 1999), it is not clear how this relates to task performance.

Although there has been some research into the strategies employed to understand and debug programs when working with computerised environments, this research has focused mainly on debugging performance (Mulholland, 1997; Patel, du Boulay & Taylor, 1997) or has relied on indirect accounts of the behaviour exhibited, for example through questionnaires and post-hoc interviews (Storey, Wong & Muller, 2000). There is a need for studies that present a more direct account of how people go about debugging using computerised environments. Some important questions to address with such studies are:

1. How do programmers integrate debugging strategies and tactics when working with SDEs?
2. What is the relationship between verbal ability, level of graphical literacy and debugging (task) performance.
3. How do modality and perspective influence debugging strategy and deployment of tactics (an important aspect of which is visual attention allocation in the SDE)?

The following sections describe an empirical study that addresses these questions.

## 3 Method

### 3.1 Aims

This study had three main aims, each aligned with one of the questions detailed above.

Regarding question one, we aimed to investigate the relationship between the debugging strategies employed and the programmers' tactical use of the representations and facilities made available in the SDE. This was a detailed, qualitative analysis of the fine-grained events that took place in the recorded debugging sessions. We expected the deployed tactics to support programmer's debugging strategies and these tactics and strategies to be similar to those reported in the literature (Katz & Anderson, 1988; Mulholland, 1997; Prabhakararao, Cook, Ruthruff, Creswick, Main & Durham, 2003; Romero, Lutz, Cox & du Boulay, 2002; Bednarik & Tukiainen, 2004).

Relating to question two, our aim was to identify the key relationships between graphical literacy, verbal ability and debugging performance. This was a quantitative analysis that looked for correlations between these individual differences measures. Based on related previous studies (Grawemeyer & Cox, 2003; Grawemeyer & Cox, 2004), we expected graphical literacy and debugging accuracy to be positively correlated.

Finally, with reference to question three, we aimed to investigate the relationship between experience, modality, perspective and debugging strategy and tactics. This was a quantitative analysis on the data logged during the debugging sessions. According to previous studies (Romero, Lutz, Cox & du Boulay, 2002; Romero, Cox, du Boulay & Lutz, 2002) we expected the choice of strategy and the tactics deployed to be associated with programming experience, knowledge of external representations and modality.

### 3.2 Design

The study was divided into three aspects, each related to the questions detailed at the end of Section 2. A description of the design of each of the three aspects of the study follows.

In order to address question 1, a detailed qualitative analysis of the events in the debugging sessions was performed. The events considered were participants verbalisations, the focus of their visual attention and their interaction with the SDE.

In order to address question 2 the study investigated the relationship between debugging accuracy and several individual differences measures: Object-Oriented and procedural programming experience, verbal ability and knowledge of external representations. The analysis of this part of the study computed the correlations between these performance and individual differences measures.

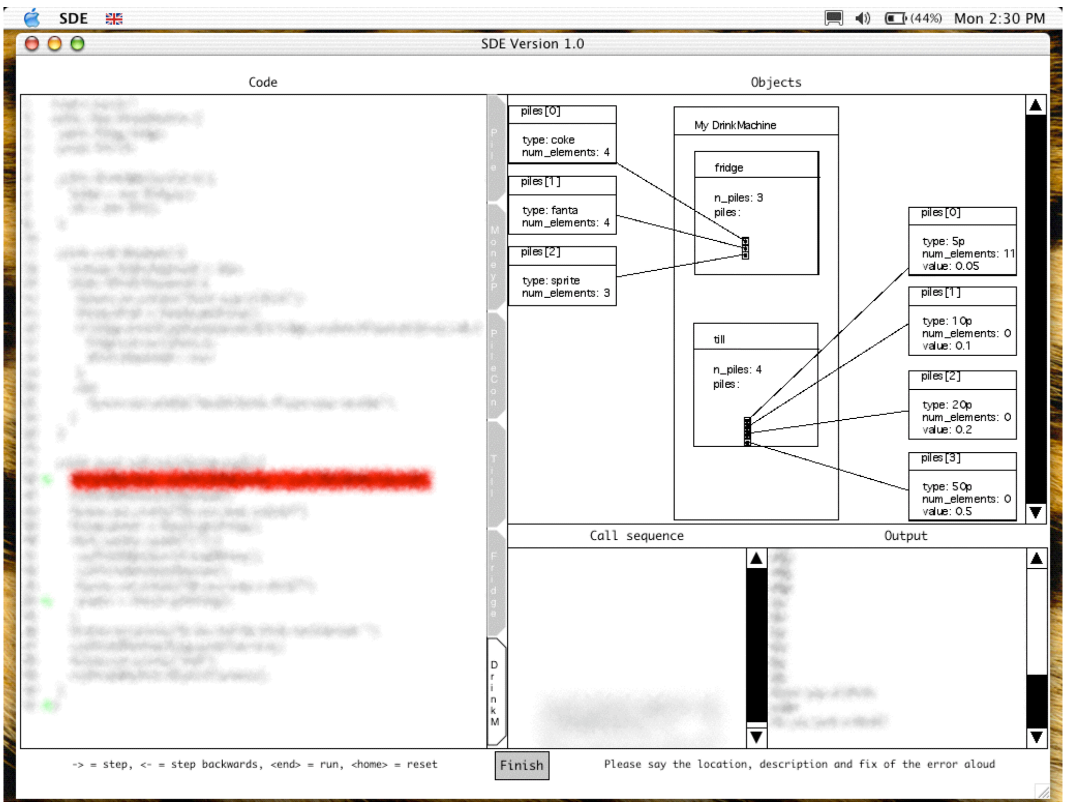
Regarding question 3, the investigation into the relationship between experience, modality, perspective and debugging strategy and tactics, considered four independent variables (two between subjects and two within subjects) and five dependent variables. The independent between subjects variables were procedural programming experience (PE) and knowledge of external representation formalisms (KER). The independent within subjects variables were type of error (data structure or control-flow) and modality (graphical or textual visualisations). The five dependent variables were debugging accuracy, inspection time for the available representations, switching frequency between these, inspection time at the different points of the program execution (*breakpoints*) and switching frequency between these. Inspection time for the available representations refers to the time participants spent focusing on each window of the SDE. Switching frequency between the available representations refers to the number of changes of focus between the SDE windows. Inspection time at the different breakpoints refers to the time participants spent focusing on each one of the breakpoints at which they chose to view the execution of the program. Finally, switching frequency between these breakpoints is the number of times participants changed from one breakpoint to another.

## 3.3 The experimental debugging environment

The SDE enabled participants to view the pre-computed execution of a Java program and presented, in addition to the code, its output and two visualisations of its execution. Participants were able to view the execution of the program by stepping between predefined *breakpoints* for a specific sample input. The SDE did not provide students with tools to edit, compile or re-execute the program with different input values or to reset breakpoints to other places in the code. The motivation to limit the functionality of the tool in this way was to ensure, as much as possible, that all participants saw the same information and to reduce the complexity of operating the debugging environment.

Participants were able to see the program code, its output for a sample execution, and two visualisations of this execution. A screen shot of the system is shown in Figure 1. Participants were able to see the program class files in the code window, one at a time, through the use of the side-tabs. The *objects* and *call sequence* windows presented visualisations of the program's execution similar to those found in Object-Oriented software development environments (Romero, Cox, du Boulay & Lutz, 2003). The objects window (top right) presented data structure aspects while the call sequence window (bottom middle) showed control-flow information.

The SDE is a modified version of the Restricted Focus Viewer (RFV), a visual attention tracking software environment (Blackwell, Jansen & Marriott, 2000). The SDE presents image stimuli in a blurred form. When the user clicks on an image, a section of it around the mouse pointer becomes focused. In this way, the program restricts how much of a stimulus can be seen clearly and thus indirectly allows visual attention to be tracked as the user moves an unblurred



piles[0]
type: coke
num_elements: 4

piles[1]
type: fanta
num_elements: 4

piles[2]
type: sprite
num_elements: 3

fridge
n_piles: 3
piles:

till
n_piles: 4
piles:

piles[0]
type: 5p
num_elements: 11
value: 0.05

piles[1]
type: 10p
num_elements: 0
value: 0.1

piles[2]
type: 20p
num_elements: 0
value: 0.2

piles[3]
type: 50p
num_elements: 0
value: 0.5

-> = step, <- = step backwards, <end> = run, <home> = reset

Please say the location, description and fix of the error aloud

Figure 1: The debugging environment used by participants

area around the screen. Use of the SDE enabled moment-by-moment representation switching between different program breakpoints and between concurrently displayed, adjacent representations to be captured for later analysis. The system was also able to digitally record audio and to replay sessions, showing what participants did as well as what they said. In this way, the SDE can allow both quantitative and qualitative analyses of the recorded data. The user-computer interaction data (window and breakpoint fixation time and switching) can be analysed in a quantitative way (for example writing programs to process the logged data) to compare switching and fixation behaviour among the different experimental conditions. Observing replays of experimental sessions, on the other hand, can be used to interpret intentions and behaviours of participants. The main difference between this environment and the one employed in our previous studies (Romero, Cox, du Boulay & Lutz, 2002; Romero, Lutz, Cox & du Boulay, 2002) is its capability to show the execution of the program in steps. A previous version of the environment presented users with visualisations comprising several static screen snapshots of the program execution. Employing an environment with dynamic visualisations enabled us to study not only representation usage but also how participants employed the step and trace facilities provided. More details about the system and methodology employed can be found in Romero, Cox, du Boulay, Lutz & Bryant (2007).

Previous studies (Romero, Cox, du Boulay & Lutz, 2002; Romero, Lutz, Cox & du Boulay, 2002) suggested that the restricted focus technology works best for program comprehension and debugging purposes if the unblurred area is of a size appropriate to cover entire representation units. In the case of the code, for example, these units can be equated to methods. The objects window represents an extreme case because the representation unit is the main object and therefore the unblurred spot covers the whole window.

Studies that have validated the use of this technology have found that it does not modify task performance significantly (Romero, Cox, du Boulay & Lutz, 2002; Jansen, Blackwell & Marriott, 2003). Studies that have compared visual attention behaviour using this technology and employing eye-tracking equipment have however found differences in these two conditions (Bednarik & Tukiainen, 2004). The central issue concerns the validity of eye-tracking as unequivocal measure of visual attention. One issue is that the two techniques work at different degrees of granularity, with eye-tracking capturing many more fleeting changes of gaze direction. Researchers have tended to interpret measurement differences between the two techniques as reflecting the superiority of eye tracking methods. However, recent evidence from the visual attention, change blindness and attention design literatures (Wood, Cox & Cheng, 2006) raises some questions in relation to this assumption.

## 3.4 Participants and procedure

The experimental participants were forty two computer science undergraduate students from the School of Cognitive and Computing Sciences at Sussex University, U.K. All had taken a three month introductory course in Java. Some of them had previous programming experience, in most cases a few extra months of academic programming experience.

Participants performed a verbal ability test, an external representation ('graphical literacy') decision task (Cox, Romero, du Boulay & Lutz, 2004), a program modification exercise, a program comprehension activity and six debugging sessions. The experiment was divided into two sessions of about one hour each which took place on different days. The verbal ability test was based on items from a commercial book of GRE practice examples (Brownstein, Weiner & Weiner-Green, 1990; Cox, Stenning & Oberlander, 1995). The items have a multiple-choice response format and are designed to measure the respondent's ability to, for example, compare several passages in terms of the similarity of their arguments, make valid inferences from narratively presented information passages, assess the relative strengths of arguments, judge whether alternative passages strengthen or weaken particular arguments and the identify assumptions underlying arguments.

The external representation decision task was a visual recognition activity requiring decisions as to whether a diagram was real or fake. A sequence of well-formed (real) and chimeric (fake) diagrams was presented to participants and they had to decide whether each one of these was real or fake (Cox,

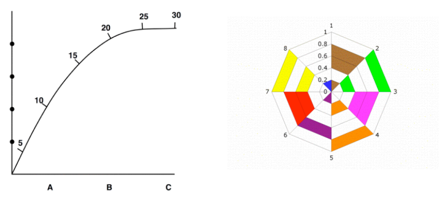


Figure 2: Examples of fake diagrams

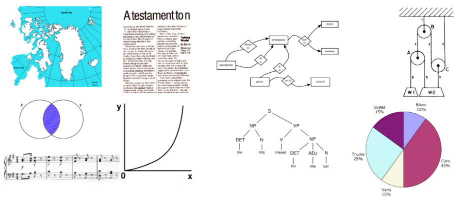


Figure 3: Examples of real diagrams

Romero, du Boulay & Lutz, 2004). Some of these diagrams are shown in Figures 2 and 3.

The program modification and comprehension tasks were intended to familiarise participants with the program they were going to debug and with the program visualisations that were going to be presented to them in the objects and call sequence windows. In the program modification task participants had to perform a simple modification to the program while in the program comprehension exercise participants had to answer a series of true/false questions about this same program and its associated visualisations.

Following the familiarisation tasks participants proceeded to the debugging part of the experiment.

### 3.4.1 The debugging session

In the six debugging sessions participants worked with buggy versions of the same program. Each version was seeded with one error and was also modified at a superficial level (variable and method names changed) to control for spotting errors by relying on memory alone.

The first debugging session was a warm-up exercise. The five main debugging sessions followed. One of these five sessions was used as a control and showed empty windows for the objects and call sequence visualisations in order to investigate whether the visualisations that would otherwise have been present in these windows were helpful to participants. The order of presentation of the four experimental sessions with 'normal' SDE as well as the single 'empty visualisations' condition was randomised, as well as the choice of which buggy program version to use in the warm up session.

Each debugging session consisted of two phases. In the first phase participants were presented with samples of program output, both desired and actual. When participants were clear about the difference between these two sample outputs they moved on to the second phase of the session.

In the second phase participants worked with the SDE. They were allowed up to ten minutes to debug each program. Following (Ericsson & Simon, 1984)'s recommendations, participants were instructed to think aloud throughout the session.

```
import  java.io.*;
public class Fridge extends PilesContainer {
    public Fridge(int number_of_drink_types) {
        super(number_of_drink_types);
    }

    public void load() {
        System.out.println("Loading the fridge");
        System.out.println("Do you want to add drinks?");
        String answer = EasyIn.getString();
        while (answer.equals("y") && !isFull()) {
            System.out.println("Enter the type of drink");
            String new_type = EasyIn.getString();
            System.out.println("Now enter the number of " + new_type + "s");
            int num_of_cans = EasyIn.getInt();
            add(new_type, num_of_cans);
            if (isFull())
                System.out.println("Fridge cannot take any more drink types");
            else {
                System.out.println("Do you want to add more drinks?");
                answer = EasyIn.getString();
            }
        }
        System.out.println("Finished loading the fridge");
    }

    public boolean typeExists(String drink_type) {
        boolean type_exists = true;
        if (piles.length < getIndex(drink_type))
            type_exists = false;
        return type_exists;
    }
```

Figure 4: Segment of the program code for the Fridge class.

The target program simulated the behaviour of a drink dispensing machine and was of medium size and complexity. This program loads the drink machine with cans of different drink types and also dispenses drinks after allowing the user to enter strings representing coins. The program is 201 lines long and comprises six classes linked by inheritance and composition relations. A typical execution of this program would create about 12 different objects, some of which are array data structures.

Some of the code, output for a sample execution session and objects visualisations for textual and graphical conditions for one of the buggy versions are shown in Figures 4, 5, 6 and 7 respectively.

Each version of the program was seeded with one error. This error was either data structure or control-flow related. The data structure errors were most

```
tsunx% java DrinkMachine
Loading the fridge
Do you want to add drinks?
y
Enter the type of drink
coke
Now enter the number of cokes
5
Exception in thread ‘main’
tsunx%
```

Figure 5: Output from a sample execution session of the *DrinkMachine* program.

easily seen in the objects view window, while the control-flow ones were most salient in the call sequence visualisation.

There were four predefined breakpoint lines in the code and different execution paths of the program generated different numbers of debugging steps or pauses. The average number of debugging steps for all the program versions was 5.5 (they ranged from 4 to 7). These predefined breakpoints were chosen because they were points in the execution where the arrays of the *DrinkMachine* object (the main data structure of the program) were updated.

The audio recordings of the debugging sessions were transcribed and analysed to score debugging accuracy. The score for each error was calculated on the basis of whether students reported the location, description and proposed fix of the error correctly. A score of one was assigned for each one of these aspects if it was correctly reported and zero otherwise, thus giving each session a score between zero and three for debugging accuracy.

## 4 Results

This section describes the experimental results and is divided into sub-sections: i) debugging accuracy and its relation to individual difference measures; ii) the results related to debugging tactics (debugging behaviour in terms of representation usage and controlling the execution of the program) and iii) a detailed analysis (combining qualitative and quantitative methods) of debugging strategy deployment and its relationship to tactics.

```
MyDrinkMachine :
   fridge :
      n_piles : 1
      piles :
         piles[0] :
            type : coke
            n_elements : 7
         piles[1] : null
         piles[2] : null
   till :
      n_piles : 4
      piles :
         piles[0] :
            type : 5p
            n_elements : 2
            value : 0.05
         piles[1] :
            type : 10p
            n_elements : 1
            value : 0.1
         piles[2] :
            type : 20p
            n_elements : 2
            value : 0.2
         piles[3] :
            type : 50p
            n_elements : 1
            value : 0.5
```

Figure 6: Textual objects view of the *DrinkMachine* program.

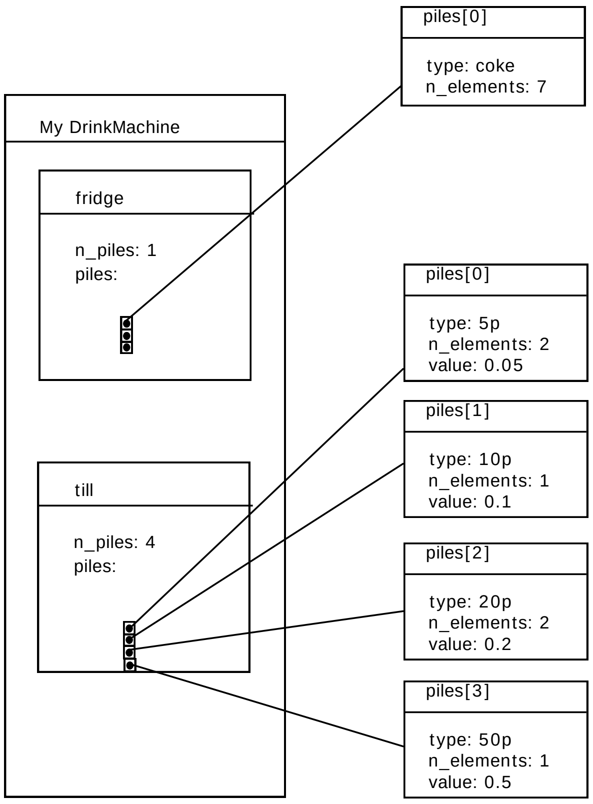


Figure 7: Graphical objects view of the *DrinkMachine* program.

### 4.1 Debugging accuracy and individual differences measures

This section reports on the results of three analyses, the first relating individual differences measures to debugging accuracy, the second comparing the normal and empty visualisations conditions and the third relating accuracy to the experimental factors considered. The second and third analyses were performed separately as some of the experimental factors (representation modality, for example) were not relevant in the empty visualisations condition.

#### 4.1.1 Individual differences

Table 1 presents a summary of the results that relate individual differences measures to debugging accuracy. Debugging accuracy was positively correlated

| | | OO Progr. Exp. | Proc. Progr. Exp. | Verbal Ability | ER Knowledge | Deb. Accuracy |
|---|---|---|---|---|---|---|
| OO Programming Experience | Pearson Correlation | 1 | -.035 | .224 | -.131 | -.112 |
| | Sig. (2-tailed) | . | .824 | .154 | .407 | .481 |
| Procedural Programming Experience | Pearson Correlation | - | 1 | -.049 | .068 | .358* |
| | Sig. (2-tailed) | - | . | .760 | .670 | .020 |
| Verbal Ability | Pearson Correlation | - | - | 1 | .248 | .167 |
| | Sig. (2-tailed) | - | - | . | .113 | .290 |
| ER Knowledge | Pearson Correlation | - | - | - | 1 | .321* |
| | Sig. (2-tailed) | - | - | - | . | .038 |
| Debugging Accuracy | Pearson Correlation | - | - | - | - | 1 |

Table 1: Correlations between pre-tests scores and debugging performance. (*) indicates significance at the .05 level

with both experience in procedural programming languages (C, Pascal, Basic, etc.) ($\sigma$ = .36 p < .05) and with the external representation decision test score ($\sigma$ = .32 p < .05). There were no significant correlations between these pre-test scores and any of the other individual differences measures.

The results of this analysis suggest that improved debugging performance was associated with programming experience in procedural languages and with knowledge about external representation formalisms but not with Object-Oriented programming experience and verbal ability. The lack of association between debugging performance and Object-Oriented programming experience might seem counter-intuitive at first. However if we consider that participants were novice Java programmers, with some of them having additional academic programming experience, mostly in procedural programming languages, then it makes sense that this extra procedural programming experience could have made the difference when solving a debugging problem. The analyses reported in the following sections consider these two factors, procedural programming experience (PPE) and knowledge of external representation formalisms (KER) as independent, between subject variables. The 42 participants were divided (post-hoc) by a median split on the basis of their scores for these two factors into high and low groups. There were 21 participants in each one of these groups and 10 or 11 in their intersection (10 in low PPE - low KER, 11 in low PPE - high KER, 11 in high PPE - low KER and 10 in high PPE - high KER).

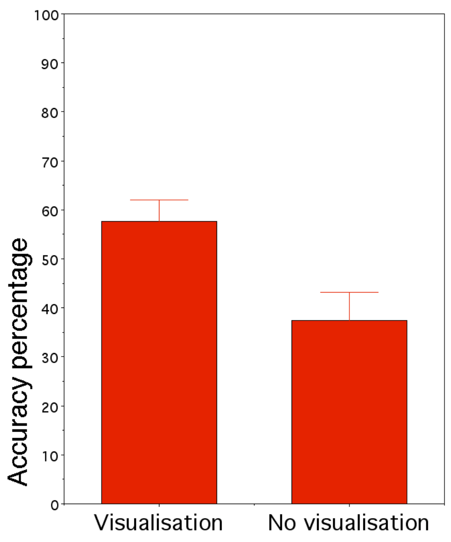


Figure 8: Debugging performance for normal and empty visualisations conditions.

#### 4.1.2 Normal and empty visualisations conditions comparison

The results of the experiment comparing debugging performance for the normal and empty visualisations conditions are illustrated in Figure 8. A repeated measures ANOVA with two between subjects variables (PPE and KER), one within subjects (visualisation) and one dependent variable (accuracy performance) was run. There were main effects for the visualisation condition (F(1,38) = 18.4, $p < .01$) only and no interaction effects. This result suggests that the visualisations were indeed helpful to students, they obtained better debugging scores with them regardless of their programming experience and their knowledge of external representations. The rest of the analyses will consider sessions in the normal condition only (those in which students had available visualisations in the debugging environment).

#### 4.1.3 Debugging performance and the experimental factors

The results of the experiment relating debugging performance to the experimental factors considered are illustrated in Figure 9. A repeated measures ANOVA with two between subjects variables (PPE and KER), two within subjects (representation modality and error type) and one dependent variable (accuracy performance) was run. There were no significant main effects but a significant interaction effect for PPE and KER (F(1,38) = 5.25, $p < .05$). Post hoc comparisons revealed a significant effect when comparing the group of high PPE and high KER with the rest of the participants (t(40) = -2.8, $p < .01$). This result suggest that superior debugging performance was associated with a high level of both programming experience and external representations knowledge. These two factors make separate contributions to

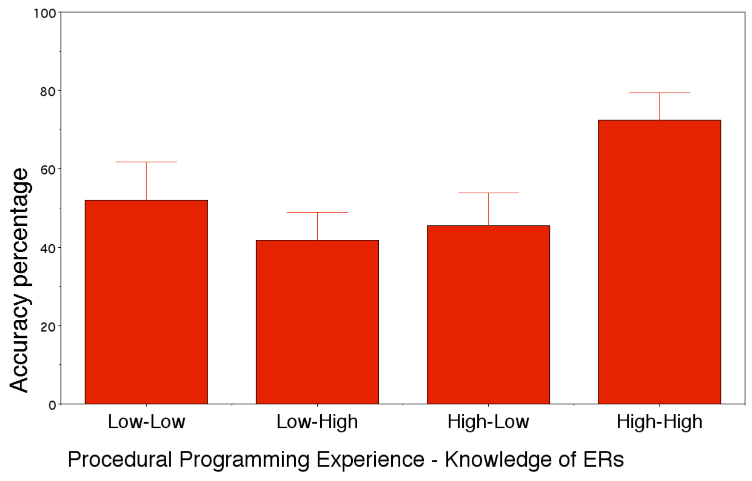


Figure 9: Debugging performance by PPE and KER.

debugging performance as they do not correlate to each other (see Table 1).

This part of the analysis revealed that visualisations were helpful and that debugging accuracy was positively correlated to individual differences measures such as programming experience and graphical literacy. It makes sense that if visualisations can indeed be helpful, knowledge about representation formalisms is key to take advantage of them.

## 4.2 Debugging tactics

This part of the analysis focuses on debugging behaviour in terms of representation usage and the way participants controlled the view of the program execution. The following subsections describe these two analyses.

### 4.2.1 Representation usage

The experimental variables considered in this analysis relate to the way visual attention was allocated during the debugging process. In particular, this analysis takes into account switches of visual attention between the different SDE windows and time spend inspecting each one of these windows. Thus, this part of the analysis relates switching frequency, accumulated fixation time and average fixation time for the available representations to the experimental factors (visualisation modality, type of error, PPE and KER). Three separate ANOVAs were computed; one for switching frequency between the available representations (the code, the objects, the call sequence and the output windows), another for accumulated fixation time within the available representations and the third for average fixation time within the available representations.

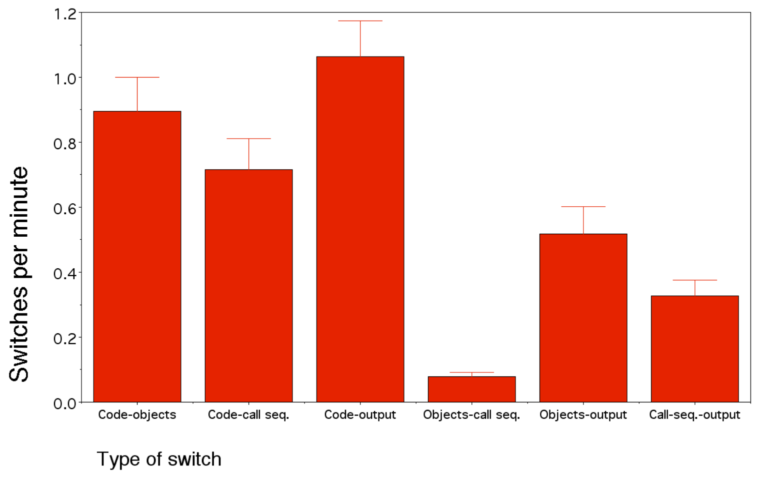


Figure 10: Window switching frequency by type of switch

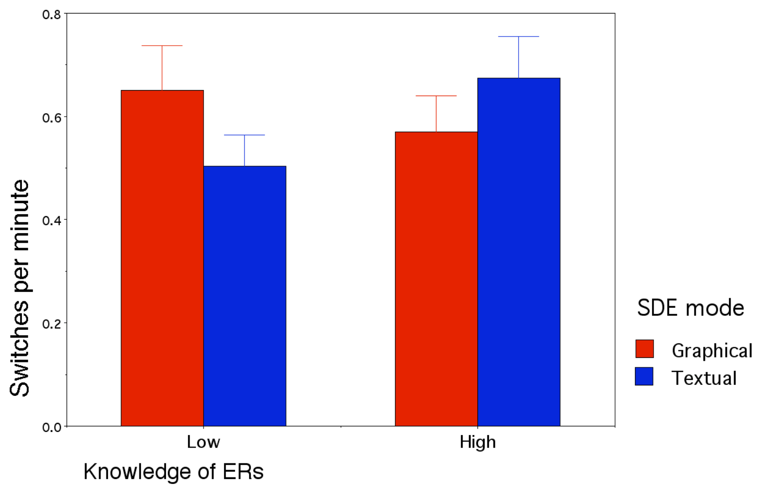


Figure 11: Window switching frequency by KER

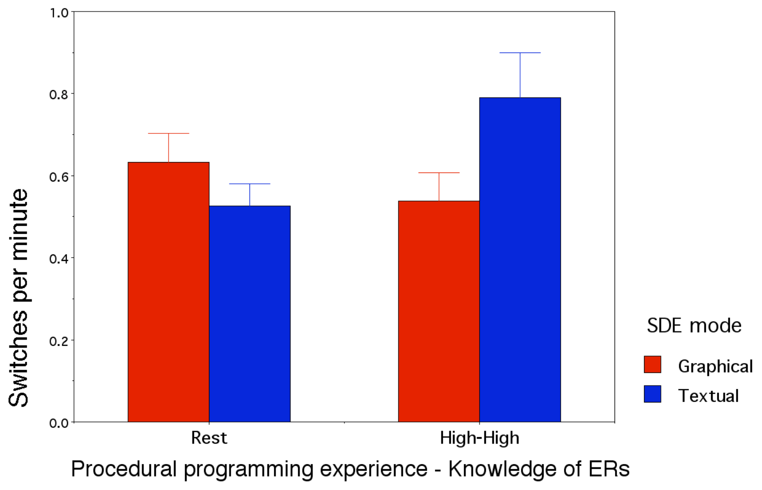


Figure 12: Window switching frequency by PPE and KER

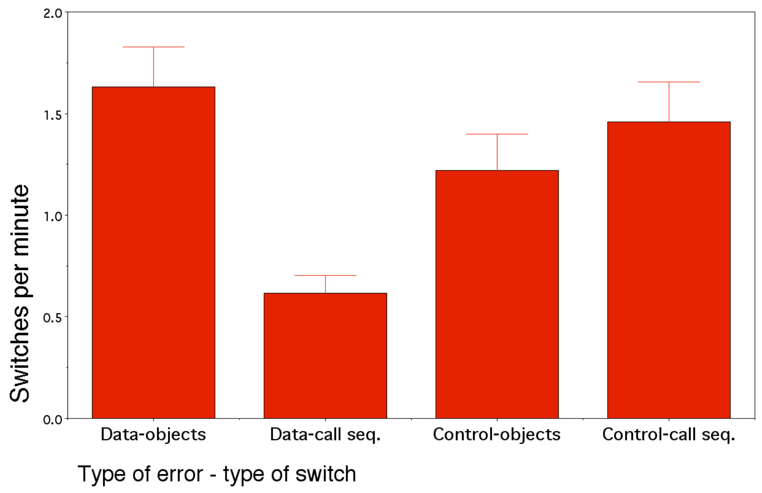


Figure 13: Window switching frequency by type of error and switch

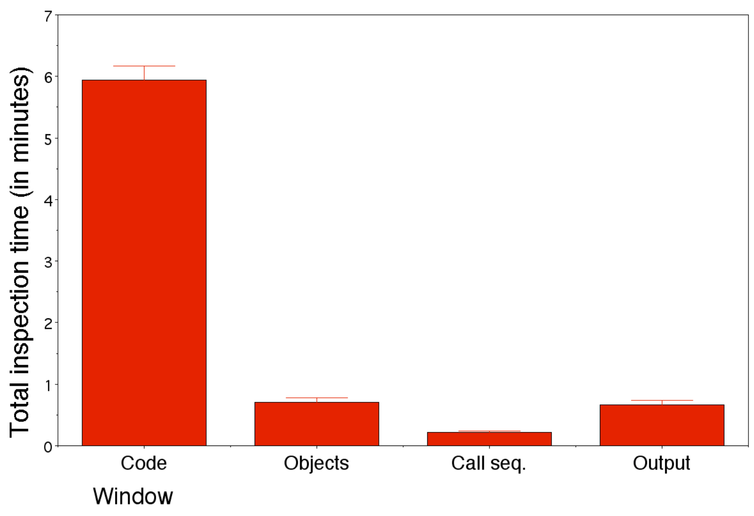


Figure 14: Accumulated inspection time for each window

The results for window switching frequency are illustrated in Figures 10, 11, 12 and 13. There were main effects for type of switch (F(5,34) = 36.41, p < .01) and interaction effects for type of error by type of switch (F(5,34) = 6.31, p < .01), for modality and KER group (see Figure 11) (F(1,38) = 9.28, p < .01) and for modality, PPE and KER groups (F(1,38) = 4.34, p < .05).

Planned comparisons revealed a significant effect when comparing the frequency of switching involving the code window against those between the other windows (t(41) = 12.54, p < .01) (see Figure 10). In the case of the interaction effect between modality, PPE and KER, planned contrast comparisons revealed a significant contrast when comparing the group of high knowledge in both PPE and KER with the rest of the participants (F(1,40) = 16.98, p < .01) (see Figure 12). Finally in the case of the interaction effect between type of error and type of switch, a planned test of within subject contrasts for the type of error by type of switch effect revealed a significant contrast when comparing switches including the objects window (switching between the objects and either the code or output windows) to switches including the call sequence window (switching between the call sequence and either the code or the output windows) (F(1,38) = 30.65, p < .01) (see Figure 13). These results suggest that switches involving the code window were more frequent than those involving any of the other windows, that differences in KER are associated with differences in the amount of switching in different modality conditions, that these differences are magnified when considering the group of high KER and high PPE, and that unsurprisingly, the frequency of switches involving the objects and call sequence visualisations varies according to the type of error at hand.

Regarding accumulated inspection time, there were main effects for window (F(3,36) = 179.68, p < .01) (see Figure 14) and interaction effects for modality and KER group (F(1,38) = 5.35, p < .05) and window, modality and KER

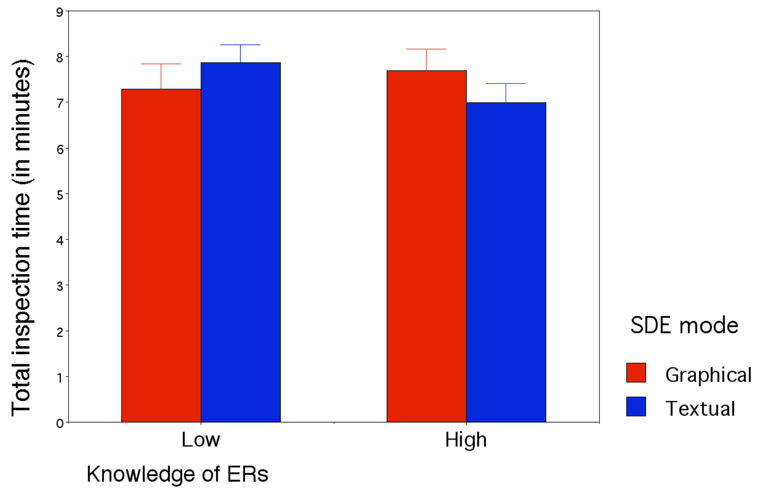


Figure 15: Accumulated inspection time by KER and SDE mode

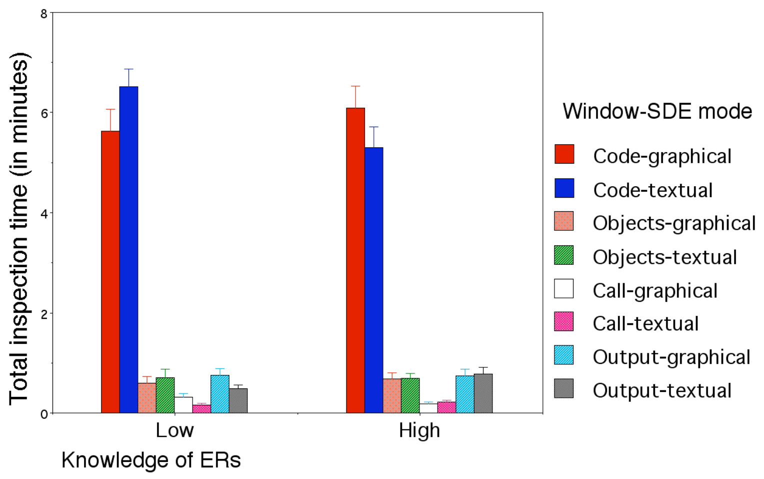


Figure 16: Accumulated inspection time by KER, window and SDE mode

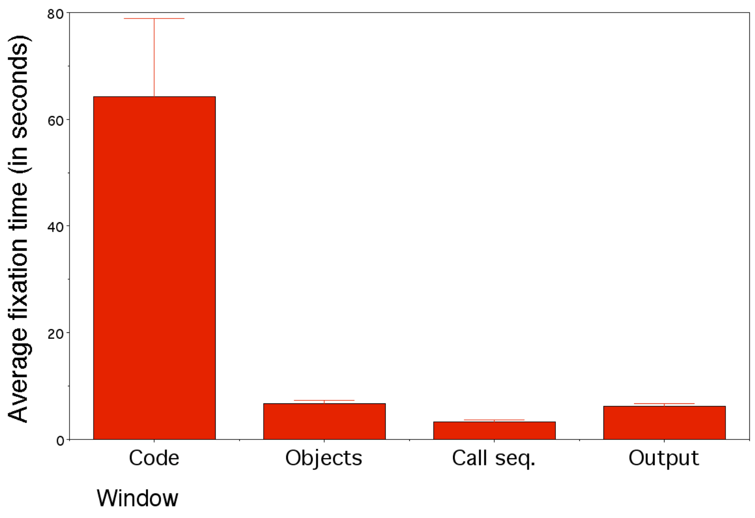


Figure 17: Average fixation time for each window

group (F(3,36) = 5.06, p < .01) (see Figures 15, and 16 respectively). For the interaction effect between window, modality and KER group, tests of within subjects contrasts revealed significant effects when comparing the differences between the KER groups in their the total fixation times for graphical and textual conditions, for the code window and the other windows (the difference in fixation times between the graphical and textual SDE modes of the KER groups for the code window was significantly different to those of the other windows) (F(1,38) = 8.65, p < .01). These results suggest that participants looked at the code much more than any other window (for about 80% of the time) and that while those participants in the high KER group inspected the code for a longer time when working in the SDE graphical mode the opposite was true for those in the low KER group (they looked at the code window for a longer time when working in the SDE textual mode). This difference seems to be responsible for the corresponding global difference in SDE mode for these two groups (see Figure 15).

The results for average fixation time per visit to the window are illustrated in Figure 17. There were main effects for window (F(3,36) = 22.43, p < .01) only and no interaction effects. Participants made average fixations of about one minute for the code window and of less than 10 seconds for the other windows. This result suggests that participants' average fixations were considerably longer for the code window but there were no significant differences for any of the other factors considered.

The results for representation usage therefore suggest that participants with both high KER and PPE had a high window switching frequency when working with textual visualisations and that there were dissimilar patterns for the amount of time participants inspected the code window for low and high KER groups under different SDE mode conditions. The low KER group looked at the code window longer when working under the SDE textual

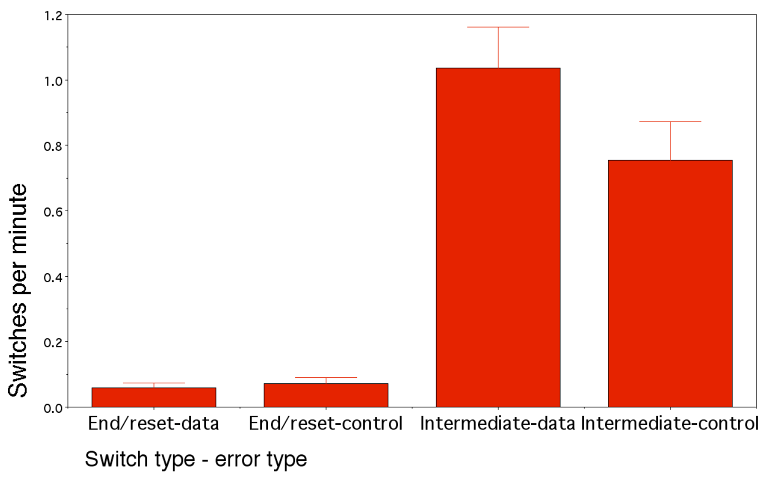


Figure 18: Breakpoint switching by types of switch (end/reset and intermediate switches) and error (data-flow and control-flow errors)

condition while the high KER group looked at the code window longer when working under the SDE graphical mode (see Figure 16). Regarding average fixation times, participants performed longer fixations when looking at the code window. There were, however, no other significant differences for this aspect either for KER, PPE, modality, perspective or their interactions.

One way to explain the reason for participants with both high KER and PPE switching more in the textual SDE condition would be to say that they made shorter fixations in this condition, however average fixation times were similar for different KER and PPE groups and for the different modality and perspective conditions (according to the results for average fixation times). Therefore if participants with both high KER and PPE were doing more switching it was not because they made shorter fixations but may be because of differences in total inspection times (differences in total time on task). The dissimilar patterns observed for total inspection times for low and high KER groups working under different SDE modes could be considered as evidence, however this result did not involve PPE groups so we cannot be conclusive.

### 4.2.2 Breakpoint usage

This part of the analysis relates accumulated fixation time and switching frequency for the program breakpoints to the experimental factors (visualisation modality, type of error, PPE and KER). Two separate ANOVAs were computed; one for switching frequency between the program breakpoints and another for fixation time within the different program breakpoints.

The analysis for breakpoint switching frequency compared switches between the beginning of the program execution and the last breakpoint (end/reset switches) with switches between intermediate breakpoints (intermediate

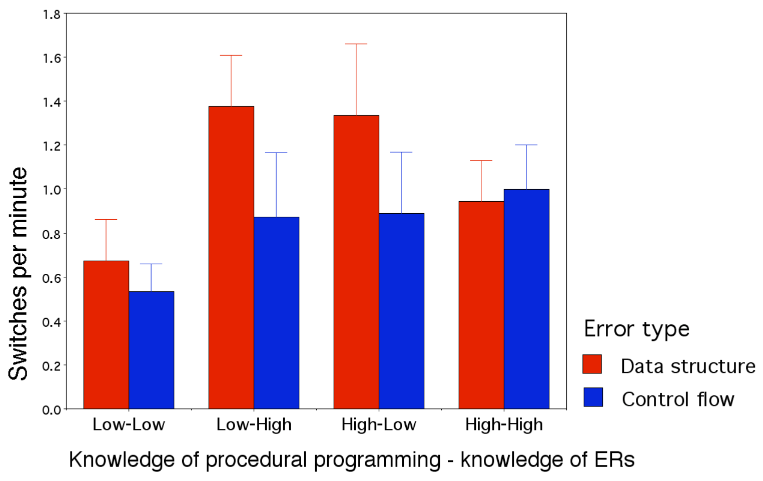


Figure 19: Breakpoint switching by PPE and KER

switches). This was to compare two typical debugging tactics: following the program execution step by step or analysing it 'post-mortem' by jumping from the the beginning of the program execution to the last breakpoint.

The results for breakpoint switching frequency are illustrated in Figures 18 and 19. There were main effects for switch (F(1,38) = 52.58, p < .01) and type of error (F(1,38) = 6.23, p < .05) and interaction effects for the combination of these two factors (F(1,38) = 6.99, p < .05) (see Figure 18), and type of error, PPE and KER (F(1,38) = 4.35, p < .05) (see Figure 19). Planned contrast comparisons failed to reveal significance for specific contrasts for the latter interaction effect. These results suggest that intermediate switches were more frequent than switches between the first and last breakpoints, that data structure errors promoted more switching than control-flow errors, but also for intermediate breakpoints, participants switched more for data structure than for control flow errors. Additionally, breakpoint switching frequency varies according to the type of error and differences in PPE and KER.

The analysis for breakpoint fixation time compared the relative time participants spent in the first, last and intermediate breakpoints. The results for breakpoint fixation are illustrated in Figures 20 and 21. There were main effects for breakpoint (F(2,37) = 3.7, p < .05) (see Figure 20) and interaction effects for breakpoint and error types (F(2,37) = 14.56, p < .01). Planned contrast comparisons revealed a significant contrast in this interaction when comparing intermediate and last breakpoints (F(1,38) = 29.54, p < .01) (see Figure 21. These results suggest that participants spent the most time on the last breakpoint and the least on the first in general but that this was also dependent on error type (this was not the case for data structure errors).

The global results for breakpoint usage suggest that differences in the control of the program execution viewing were related mainly to the type of error at hand. For data structure errors, participants spend longer in intermediate breakpoints, switching frequently between them. For control-flow errors

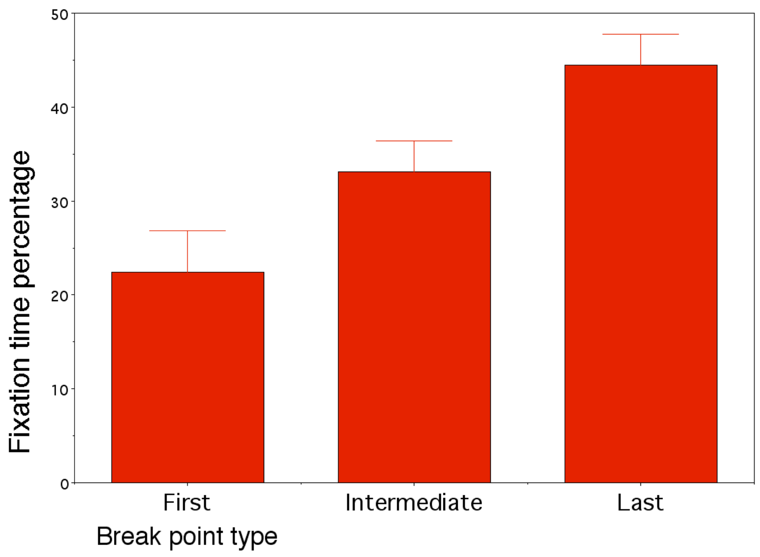


Figure 20: Fixation time by breakpoint type.

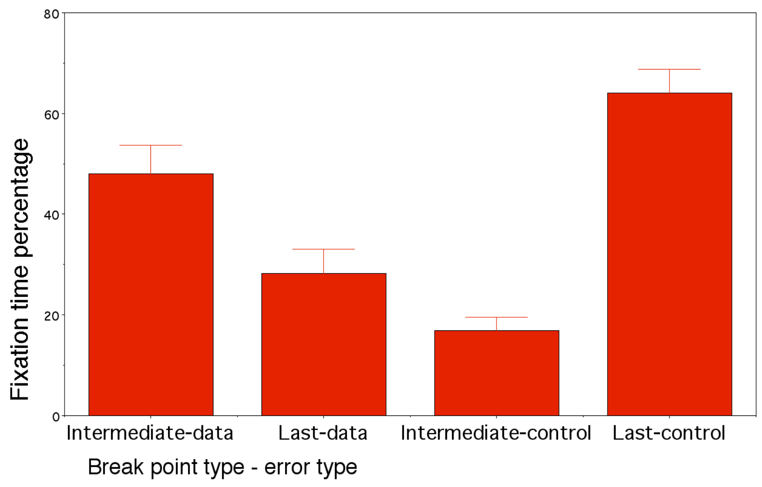


Figure 21: Fixation time by breakpoint and error types.

participants spend longer in the last breakpoint (the end of the program execution).

## 4.3 Debugging strategy

This section analyses the debugging strategies deployed by participants both in qualitative and quantitative ways. These debugging strategies were identified by interpreting detailed accounts of the behaviour of participants. These detailed accounts were obtained by watching replays of the debugging sessions and breaking them down into a sequence of discrete debugging events by interpreting three types of experimental data simultaneously. The three types of data considered were trace of focus of attention, control of the presentation of the program's execution and participants' verbalisations. A new event was defined by a change in the focus of attention, a command related to the presentation of the program's execution, participants' verbalisations or a mixture of these. Therefore events were bounded by pauses or changes of topic in programmers' verbalisations (utterances), inter-window switches of visual attention focus or breakpoint switches. A detailed description of this methodology can be found in Romero, Cox, du Boulay, Lutz & Bryant (2007).

This part of the analysis took into account only a subset of the experimental data. The debugging sessions for only one of the six target program versions were taken into account [1]. The program version chosen was not significantly different to the other versions in terms of the use of representations (code and visualisations) that participants displayed and was the one that showed the widest spread of debugging accuracy scores.

The following sections present qualitative and quantitative analyses of these debugging events and associated strategies.

### 4.3.1 Qualitative analysis

A detailed qualitative analysis of the debugging events identified specific debugging strategies by categorising each one of these events as the deployment of a specific strategy. The debugging strategies identified are shown in Table 2. Most of these debugging strategies correspond to those described in (Katz & Anderson, 1988). The only different strategy is *following execution.* There are not many references to similar strategies in the debugging literature, perhaps because only a few debugging studies have taken into account the programmer's interaction with computerised debugging environments and in particular with the visualisations provided by them. *Following execution* shares similarities with forward reasoning strategies as the

[1]This detailed analysis was an extremely time consuming process which took about 240 coder hours. It was not possible to involve multiple coders but we tried to maximise the quality of the coding process with a thorough training of the coder. First, the coder was briefed about the coding scheme. Then, the authors as well as the coder coded one debugging session. Codings were compared and discrepancies were resolved. This process was repeated until there was a high level of agreement in the codings. After this the coder coded the rest of the debugging sessions on his own.

| **Debugging behaviour** | **Description** |
|---|---|
| Following execution | Following the execution of the program for a specific example to understand some aspect of it or to identify the error. Utterances describe the behaviour of the program in terms of the changes to its data structures or real world objects |
| Causal reasoning | Homing in on an area of code after having uttered a debugging hypothesis. Participant is reading the code searching for the place responsible for the observed faulty behaviour |
| Comprehension | High level browsing of the code to build up a more complete picture of the program. Participant is reading the code for program comprehension purposes |
| Hand simulation | Talking about the program in terms of a dynamic view of it, but without stepping through it. Participant focuses on the code window commenting on dynamic aspects of the program without actually executing the program in steps |

Table 2: Debugging strategies observed in the detailed qualitative analysis of the debugging study

programmer starts the search by trying to understand what the program does, however, unlike forward reasoning, the comprehension process integrates visualisation and code information.

*Following execution* is related to *mapping* (Mulholland, 1997), cross-referencing information between visualisation and code. In *following execution* there are frequent visual attention switches between the code, the available visualisations and the output of the program as well as breakpoint switches.

In *causal reasoning* there are also visual attention and breakpoint switching, although these are not as frequent as in *following execution.* Occassionally, while trying to identify the piece of code responsible for the error, programmers might switch breakpoints or have a look at the output or the visualisations.

Finally, *comprehension* and *hand simulation* typically make no use of either visualisations or breakpoints. In these two strategies programmers concentrate on the code and do not try to execute the program in steps.

### 4.3.2 Quantitative analysis

This section analyses the debugging strategies identified in Section 4.3.1 in a quantitative way. The frequencies of the debugging events and associated strategies were computed and an ANOVA relating debugging strategy to the experimental factors (SDE mode, type of error, PPE and KER) was

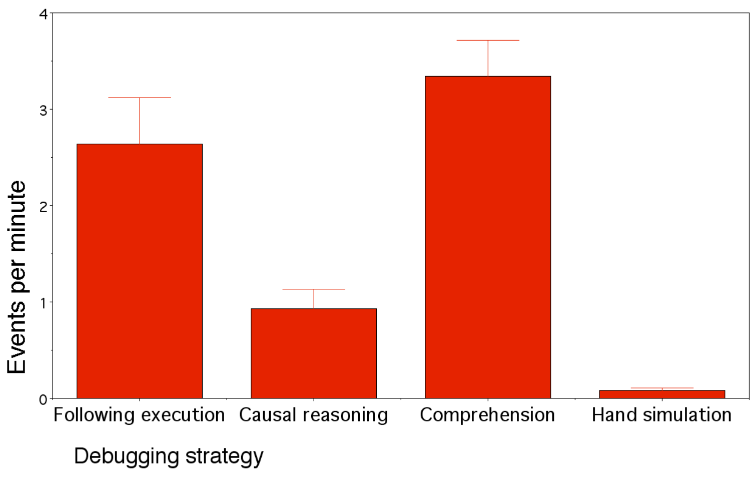


Figure 22: Frequency of debugging strategy deployment

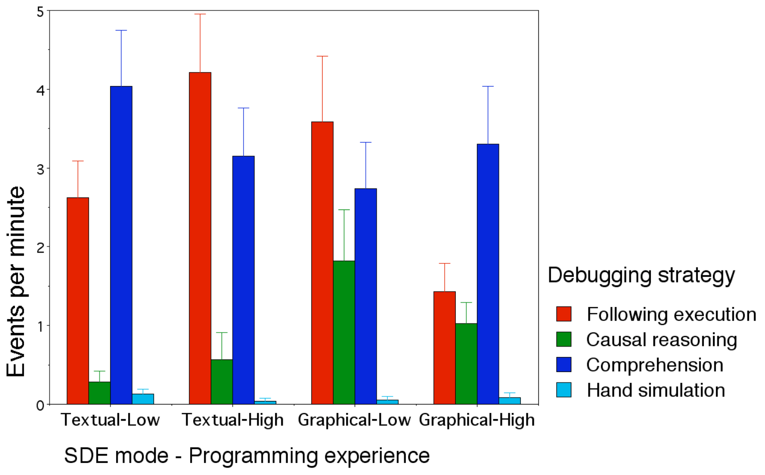


Figure 23: Frequency of debugging strategy deployment by programming experience and SDE mode

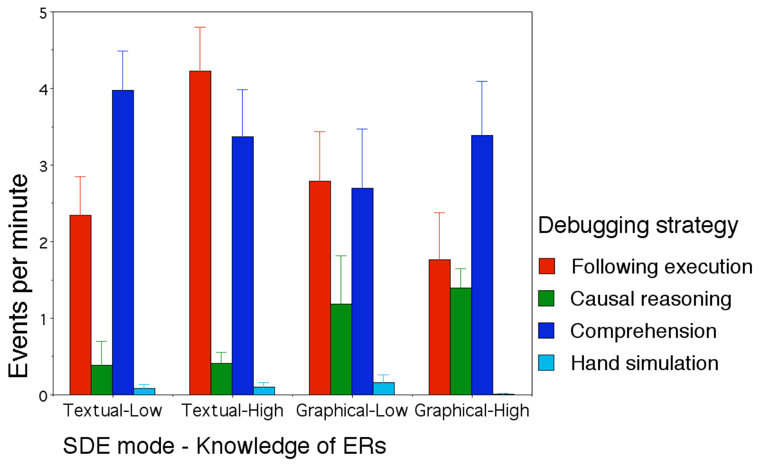


Figure 24: Frequency of debugging strategy deployment by knowledge of ERs and SDE mode

performed.

There were main effects for strategy (F(1,19) = 68.51, $p < .01$) and interaction effects for strategy, modality and programming experience (F(1,19) = 4.88, $p < .05$) and for strategy, modality and knowledge of external representations (F(1,19) = 3.37, $p < .05$) (see Figures 22, 23 and 24 respectively). Regarding the main effect, tests of within subjects contrasts revealed that *comprehension* and *following execution* did not differ significantly but that *comprehension* and both *causal reasoning* and *hand simulation* did (F(1,21) = 46.65, $p < .01$).

Regarding the interaction effect for strategy, modality and programming experience, tests of within subjects contrasts revealed that the only significant difference when comparing textual and graphical SDE mode for the high experience group was for *following execution* (F(1,21) = 20.03, $p < .01$). Regarding the interaction effect for strategy, modality and knowledge of ERs, planned contrast comparisons failed to reveal significance for specific contrasts. These results suggest that the most frequently deployed strategies were *following execution* and *comprehension* and that participants in the high and low groups for PPE and KER employed debugging strategies differently when working in different SDE modes. For the case of PPE, this difference seem to be associated with the low frequency of *following execution* performed by the high experience level group when working with graphical visualisations. For the case of KER it is not possible to be more precise.

## 5 Discussion

The discussion on the findings of this study is structured around the questions in section 2.

## 5.1 Relationship between debugging strategies and tactics

The first question is about the way in which programmers integrate debugging strategies and tactics. However before discussing this it is important to compare the strategies and tactics identified with those of previous studies.

Both the strategies and tactics observed were similar to those reported in studies looking at the use of visualisation tools in debugging (Katz & Anderson, 1988; Mulholland, 1997; Bednarik & Tukiainen, 2004). One important difference was that students employed, additionally to the strategies reported by Katz & Anderson (*causal reasoning*, *comprehension*, and *hand simulation*) a strategy perhaps specific to SDEs, *following execution.* In this strategy programmers try to comprehend the program or identify the error by running the program in steps and following its execution for a specific input example, integrating information from the program code, its visualisations and output. It can be categorised as a forward reasoning strategy, however, unlike other strategies in this category (such as *comprehension* or *hand simulation*), the search for the error takes into account information from sources other than the program code.

Tactics related to coordinating the available representations and operating the SDE step-and-trace facilities were particularly important for the *following execution* strategy. They were also employed, although to a lesser extent, in *causal reasoning.*

When deploying a *following execution* strategy, participants viewed the execution of the program in steps and made frequent visual attention switches between the code, the available visualisations and the output of the program. Frequently, once they had identified the program error in this way, they would switch to a *causal reasoning* strategy, concentrating on the program code but also making ocasional references to other representations.

The strategies more frequently deployed were *comprehension* and *following execution* while *hand simulation* was only infrequently deployed. This suggests that students made good use of the environment affordances and specifically of the facilities for browsing through the code text and of those for viewing the execution of the program in steps.

The other two debugging strategies observed, *comprehension* and *hand simulation* consisted almost entirely of reading the program code, switching between the different class files but making practically no reference to the other representations or to dynamic aspects of the program execution.

## 5.2 Relationship between graphical literacy and debugging performance

The second question refers to the relationship between verbal ability, level of graphical literacy and debugging performance.

The visualisations employed were helpful for students, debugging without visualisations decreased performance. However, in order to take advantage of the information in the visualisations both relevant programming experience and a good knowledge of representation formalisms is needed.

It seems counter-intuitive that relevant programming experience in this context means experience in procedural programming languages. However, if we consider that the participants' Object-Oriented programming experience was fairly homogeneous (mainly the undergraduate courses they had taken), what might have made the difference was the procedural programming experience they had accumulated elsewhere.

Verbal ability was not correlated with debugging performance. This result is in agreement with other studies that have found no significant correlation between verbal ability and programming performance (Mayer, Dyck & Vilberg, 1986; Tukiainen & Monkkonen, 2002). It may be that verbal ability as measured by the pre-tests applied is different from the skill required to read and interpret information in propositional form about computer programs.

## 5.3 Relationship between experience, modality and debugging behaviour

The last question refers to the way experience, modality and perspective influence debugging strategy and tactics deployment. An important finding here relates to the way in which experienced participants were able to modify their strategy and tactics according to changes in the format of the visualisations without altering their performance. Additionally, results also confirm the importance of the type of error in the debugging task.

Regarding the first finding, experienced participants (those who had both a high level of programming experience and a high level of knowledge of external representation formalisms) displayed a debugging behaviour different from the rest of the participants when dealing with different SDE modes. When working in a textual mode (with visualisations displayed in a textual format), experienced participants tended to switch their visual attention between the windows of the environment more than when working in a graphical mode. This difference implies that the reduced memorability of textual as opposed to graphical representations of data structures and flow of control was compensated for by more frequent visual cross-checks between the code and those representations. This difference seems related to corresponding differences in the debugging strategies employed in these two conditions. The following paragraphs elaborate on this point and offer a possible explanation for the reduced memorability of textual representations.

At least for participants with a high level of knowledge of external representation formalisms, there is a corresponding difference in the amount of *following execution*, a debugging strategy related to executing the program in steps and following its execution for a specific input example, integrating information from the program code, its visualisations and output. Participants working in textual mode tended to employ this debugging strategy more than in graphical mode. One possible explanation for these difference is that the textual condition imposed an additional burden which required cross-referencing the information in the different representations frequently, therefore requiring participants to spend longer in debugging strategies which rely on representation switching (such as *following execution*).

A comparison between Figures 6 and 7 illustrates the difference between graphical and textual representations. Both figures encode the same information. However by grouping certain elements in boxes, Figure 7 helps to identify meaningful structures in the visualisation (in this case the objects of the program execution). Participants working in the textual condition, on the other hand, had to perform this grouping and then keep a mental reference to these meaningful structures in working memory. These processing overheads can be crucial when dealing with dynamic representations, as participants also had to detect patterns of change through time in the visualisations. These results seem to confirm the view that diagrams, unlike propositional representations, exploit perceptual processes by grouping relevant information together and therefore make the search and recognition of information easier (Larkin & Simon, 1987).

Several studies have identified this grouping of relevant information into meaningful structures (chunking) as a crucial part of problem solving (Chase & Simon, 1973a; Chase & Simon, 1973b) and in particular of the programming skill (McKeithen, Reitman, Rueter & Hirtle, 1981; Brooks, 1983). This study seems to exemplify the way in which representation format can support chunking for a population that is presumably developing this programming skill. A graphical reference to the program's relevant structures gives better support as it enables a more direct identification of these structures.

The additional cognitive effort required to interpret the textual condition can be considered as an example of extraneous cognitive load. According to van Bruggen, Kirschner & Jochems (2002), extraneous cognitive load is the cognitive effort produced by the characteristics of the learning environment. This contrasts with germane cognitive load, which is the cognitive effort associated with storage and retrival of schemata in long term memory. According to Cognitive Load Theory (van Bruggen, Kirschner & Jochems, 2002), learning environments should attempt to decrease extraneous cognitive load and increase germane cognitive load. It seems that the graphical condition is closer than the textual condition to this aim.

Other studies have also highlighted differences in representation switching patterns between participants with different levels of skill. Cox (1997) and Cox & Brna (1995) reported that poor performers switched more frequently than successful ones in analytical reasoning tasks. However, there are several differences between those studies and the one reported here. Although analytical reasoning as a cognitive task might be remarkably similar to program comprehension, the analytical reasoning studies encouraged participants to build their own representations. Therefore, switching representations represented 'a strategic decision by the subject to abandon the current external representation and construct a new one' (Cox & Brna, 1995). In the present study, representations were complementary (and pre-constructed) rather than alternative, therefore, switching did not necessarily represent discarding one representation for another, but more likely complementing the information of one with another. The reason for switching in the present study had more to do with an inefficient use of the visualisations or with ineffective representations, rather than with giving up on specific representations.

The main results of this study suggest that, at least for the experimental conditions considered, graphical representations enabled a more direct understanding of the relevant structures in the problem space. However this does not mean that diagrams are superior to textual representations for every situation, or that they will provide a good level of support in all cases. One of the main issues to consider is scalability. Programs, even for small academic projects, very often involve dozens of objects. Presenting all of them on the screen can create layout difficulties for the designer of such a tool and probably cognitive overload problems for its users. More studies are needed to find out whether there are potential problems in using diagrammatic representations in this context and what their possible solutions might be.

## 6 Conclusions

This study has characterised the strategies and tactics deployed by novice programmers working in multi-representational software debugging environments. Additionally, it has investigated how factors such as experience, knowledge of external representation formalisms and the form and content of the representations employed in the software debugging environment influence both the choice of strategy and debugging performance. Although there have been studies that have looked at debugging strategies (Katz & Anderson, 1988; Mulholland, 1997; Prabhakararao, Cook, Ruthruff, Creswick, Main & Durham, 2003) and tactics (Romero, Cox, du Boulay & Lutz, 2002; Romero, Lutz, Cox & du Boulay, 2002; Bednarik & Tukiainen, 2004), the value of the present study resides in the fact that it offers an account of how programmers integrate them and about how different factors interact to influence the choice of these strategies and tactics and the accuracy of the debugging effort.

This study suggests that, at least for novice programmers working in multi-representational software debugging environments, knowledge of external representation formalisms is as important as programming experience to succeed in the debugging task. Visualisations of the program execution are helpful but only when students have enough programming knowledge to make sense of the information in them and enough knowledge about representation formalisms to decode this information. Students with these characteristics are able to modify their debugging strategies and tactics when working under different format conditions in order to retain a high accuracy level. Propositional representations are not as helpful as graphical ones in grouping meaningful elements of the representations and as a result these group of students had to perform frequent information cross-referencing between the available representations when working in the textual mode.

The results of this study raise several questions and more experimentation is needed, perhaps focusing only on representational format and for students with a high level of both programming experience and knowledge of external representation formalisms.

## Acknowledgments

This work was supported by the EPSRC grant GR/N64199 and the Nuffield Foundation grant URB/01703/G. The support for Richard Cox of the Leverhulme Foundation (Leverhulme Trust Fellowship G/2/RFG/2001/0117) and the British Academy is gratefully acknowledged. The authors wish to thank Stephen Grant for refining the coding categories of the detailed analysis of representation usage and for coding them. Finally the authors would like to thank the referees for their helpful comments and thorough review of earlier versions of the paper.

## References

Ainsworth, S., Wood, D., & Bibby, P. (1996). Co-ordinating multiple representations in computer based learning environments. In Brna, P., Paiva, A., & Self, J. (Eds.), *Proceedings of the 1996 European Conference on Artificial Intelligence on Education*, (pp. 336–342)., Lisbon, Portugal.

Bednarik, R. & Tukiainen, M. (2004). Visual attention and representation switching in Java program debugging: a study using eye-movement tracking. In Dunican, E. & Green, T. (Eds.), *Proceedings of the 16th annual workshop of the Psychology of Programming Interest Group*, (pp. 159–169).

Blackwell, A., Jansen, A., & Marriott, K. (2000). Restricted focus viewer: a tool for tracking visual attention. In M. Anderson, P. Cheng, & V. Haarslev (Eds.), *Theory and Application of Diagrams. Lecture Notes in Artificial Intelligence 1889* (pp. 162–177). Springer-Verlag.

Bodemer, D., Ploetzner, R., Feuerlein, I., & Spada, H. (2004). The active integration of information during learning with dynamic interactive visualizations. *Learning and Instruction*, *14*, 325–341.

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

Brownstein, S., Weiner, M., & Weiner-Green, S. (1990). *How to prepare for the GRE.* New York: Barron's Educational Series.

Campagnoni, F. R. & Ehrlich, K. (1989). Information retrieval using a hypertext-based help system. *ACM Transactions on Information Systems*, *7*, 271–291.

Chase, W. & Simon, H. (1973a). Perception in chess. *Cognitive Psychology*, *4*, 55–81.

Chase, W. G. & Simon, H. A. (1973b). The mind's eye in chess. In W. G. Chase (Ed.), *Visual Information Processing*. New York: Academic.

Chintakovid, T., Wiedenbeck, S., Burnett, M., & Grigoreanu, V. (2006). Pair collaboration in end-user debugging. In Grundy, J. & Howse, J. (Eds.), *2006 IEEE Symposium on Visual Languages and Human-Centric Computing*, (pp. 3–10)., Brighton, UK. IEEE press.

Cox, R. (1996). *Analytical reasoning with multiple external representations.* PhD thesis, University of Edinburgh, Edinburgh, Scotland, U.K.

Cox, R. (1997). Representation interpretation versus representation construction: a controlled study using switchERII. In du Boulay, B. & Mizoguchi, R. (Eds.), *Artificial intelligence in education: knowledge and media in learning systems (Proceedings of the 8th. World Conference of the Artificial Intelligence in Education Society*, (pp. 434–444)., Amsterdam. IOS.

Cox, R. (1999). Representation construction, externalised cognition and individual differences. *Learning and Instruction*, *9*, 343–363.

Cox, R. & Brna, P. (1995). Supporting the use of external representations in problem solving: The need for flexible learning environments. *Journal of Artificial Intelligence in Education*, *6*(2/3), 239–302.

Cox, R., Romero, P., du Boulay, B., & Lutz, R. (2004). A cognitive processing perspective on student programmers' 'graphicacy'. In Blackwell, A., Marriott, K., & Shimojima, A. (Eds.), *Diagrammatic Representation and Inference. Lecture Notes in Computer Science (LNCS) 2980.*, (pp. 344–346)., Berlin. Springer-Verlag.

Cox, R., Stenning, K., & Oberlander, J. (1995). The effect of graphical and sentential logic teaching on spontaneous external representation. *Cognitive Studies: Bulletin of the Japanese Cognitive Science Society*, *2*(4), 56–75.

Davies, S. P. (1993a). Expertise and display-based strategies in computer programming. In *Human Computer Interaction '93.*

Davies, S. P. (1993b). Models and theories of programming strategy. *International Journal of Man-Machine Studies*, *39*, 237–267.

Ericsson, K. A. & Simon, H. A. (1984). *Protocol Analysis: Verbal Reports as Data.* Cambridge, Massachusetts: The MIT Press.

Gilmore, D. J. (1990). Expert programming knowledge: a strategic approach. In J. Hoc, T. R. G. Green, R. Samurçay, & D. J. Gilmore (Eds.), *Psychology of Programming* (pp. 223–234). London, U.K.: Academic Press.

Gilmore, D. J. & Green, T. R. G. (1984). Comprehension and recall of miniature programs. *International Journal of Man-Machine Studies*, *21*(1), 31–48.

Grawemeyer, B. & Cox, R. (2003). The effects of knowledge of external representations and display selection upon database query performance. In *Second International Workshop on Interactive Graphical Communication (IGC2003).*

Grawemeyer, B. & Cox, R. (2004). The effect of knowledge-of-external-representations upon performance and representational choice in a database query task. In Blackwell, A., Marriott, K., & Shimojima, A. (Eds.), *Diagrammatic Representation and Inference, Third International Conference, Diagrams 2004*, (pp. 351–354).

Grigoreanu, V., Beckwith, L., Fern, X., Yang, S., Komireddy, C., Narayanan, V., Cook, C., & Burnett, M. (2006). Gender differences in end-user debugging, revisited: What the miners found. In Grundy, J. & Howse, J. (Eds.), *2006 IEEE Symposium on Visual Languages and Human-Centric Computing*, (pp. 19–26)., Brighton, UK. IEEE press.

Jansen, A. R., Blackwell, A. F., & Marriott, K. (2003). A tool for tracking visual attention: The restricted focus viewer. *Behavior Research Methods, Instruments, & Computers*, *35*(4), 57–69.

Katz, I. & Anderson, J. R. (1988). Debugging: an analysis of bug location strategies. *Human-Computer Interaction*, *3*, 359–399.

Kriz, S. & Hegarty, M. (2004). Constructing and revising mental models of a mechanical system: The role of domain knowledge in understanding external visualizations. In Forbus, K., Gentner, D., & Regier, T. (Eds.), *Proceedings of the 26th Annual Conference of the Cognitive Science Society*, (pp. 439–449)., Mahwah, NJ. Lawrence Erlbaum Associates.

Larkin, J. H. & Simon, H. A. (1987). Why a diagram is (sometimes) worth ten thousand words. *Cognitive Science*, *11*, 65–100.

Mayer, R. E., Dyck, J. L., & Vilberg, W. (1986). Learning to program and learning to think: what's the connection? *Commun. ACM*, *29*(7), 605–610.

McKeithen, K. B., Reitman, J. S., Rueter, H. H., & Hirtle, S. C. (1981). Knowledge organization and skill differences in computer programmers. *Canadian Journal of Psychology*, *13*, 307–325.

Mulholland, P. (1997). Using a fine-grained comparative evaluation technique to understand and design software visualization tools. In Wiedenbeck, S. & Scholtz, J. (Eds.), *Empirical Studies of Programmers, seventh workshop*, (pp. 91–108)., New York. ACM press.

Oberlander, J., Stenning, K., & Cox, R. (1999). Hyperproof: Abstraction, visual preference and modality. In L. S. Moss, J. Ginzburg, & M. de Rijke (Eds.), *Logic, Language, and Computation, Vol. II* (pp. 222–236). CSLI Publications.

Patel, M. J., du Boulay, B., & Taylor, C. (1997). Comparison of contrasting Prolog trace output formats. *International Journal of Human Computer Studies*, *47*, 289–322.

Pennington, N. (1987). Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology*, *19*, 295–341.

Prabhakararao, S., Cook, C. R., Ruthruff, J. R., Creswick, E., Main, M., & Durham, M. (2003). Strategies and behaviors of end-user programmers with interactive fault localization. In *HCC*, (pp. 15–22). IEEE Computer Society.

Romero, P., Cox, R., du Boulay, B., & Lutz, R. (2002). Visual attention and representation switching during java program debugging: A study using the restricted focus viewer. In Hegarty, M., Meyer, B., & Narayanan, N. H. (Eds.), *Diagrammatic Representation and Inference. Second International Conference, Diagrams 2002. Lecture Notes in Artificial Intelligence 2317*, (pp. 221–235).

Romero, P., Cox, R., du Boulay, B., & Lutz, R. (2003). A survey of representations employed in object-oriented programming environments. *Journal of Visual Languages and Computing*, *14*(5), 387–419.

Romero, P., Cox, R., du Boulay, B., Lutz, R., & Bryant, S. (2007). Methodologyforthecaptureandanalysisofhybriddata:acasestudyofprogramdebugging. *Behavior Research Methods*.

Romero, P., Lutz, R., Cox, R., & du Boulay, B. (2002). Co-ordination of multiple external representations during java program debugging. In S. Wiedenbeck & M. Petre (Eds.), *2002 IEEE Symposia on Human Centric Computing Languages and Environments* (pp. 207–214). Airlington, Virginia, USA: IEEE press.

Stenning, K. & Oberlander, J. (1995). A cognitive theory of graphical and linguistic reasoning: logic and implementation. *Cognitive Science*, *19*(1), 97–140.

Storey, A. D., Wong, K., & Muller, H. A. (2000). How do program understanding tools affect how programmers understand programs? *Science of computer programming*, *36*, 183–207.

Tukiainen, M. & Monkkonen, E. (2002). Programming aptitude testing as a prediction of learning to program. In Kuljis, J., Baldwin, L., & Scoble, R. (Eds.), *Psychology of Programming Interest Group 14th Workshop*, (pp. 45–57).

Tversky, B. & Morrison, J. B. (2002). Animation: can it facilitate? *International Journal of Human Computer Studies*, *57*, 247–262.

van Bruggen, J. M., Kirschner, P. A., & Jochems, W. (2002). External representation of argumentation in CSCL and the management of cognitive load. *Learning and instruction*, *12*, 121–138.

Wood, S., Cox, R., & Cheng, P. (2006). Designing for attention: Eight issues to consider. *Computers in Human Behavior*, *22*(1).