Java debugging strategies in multi-representational environments

Pablo Romero, Benedict du Boulay, Richard Cox, and Rudi Lutz
Human Centred Technology Group,
School of Cognitive & Computing Sciences
University of Sussex, Falmer, BN1 9QH, UK.
juanr@cogs.susx.ac.uk

Abstract

This paper reports the *qualitative* analysis part of a Java debugging experiment. Java program debugging was investigated in computer science students who used a software debugging environment (SDE) that provided concurrently displayed, adjacent, multiple and linked representations consisting of the program code, a visualisation of the program, and its output.

The aim of this qualitative analysis was to characterise the debugging strategies employed by participants, both at the level of focus of attention and representation use as well as in terms of the general reasoning strategy deployed. A modified version of the Restricted Focus Viewer (RFV) - a visual attention tracking system - was employed to measure the degree to which each of the representations was used, and to record switches between representations.

The experimental results are in agreement with research in the area that suggests that people start a debugging session by trying to understand the code of the program before they attempt to locate any bugs. Two different strategies to locate bugs were detected: by spotting something odd in the program code and by comparing information from the different external representations available. These strategies may be linked to cognitive characteristics of the programmer such as level of programming skill and display modality preference.

1 Introduction

Professional programmers typically employ debugging packages, prototyping and visualisation tools in software development environments. These tools routinely provide a range of external representations of both the static and dynamic states of the program in addition to the code itself. A measure of professional expertise is the ability with which these representations are coordinated to form a multifaceted but coherent understanding of the program.

A similar situation applies to novice programmers. They often spend a large amount of their time attempting to understand the behaviour of programs when trying to discover errors in the code. To perform this task, they normally work with both the program code and the debugger output, trying to coordinate and make sense of these two representations.

Despite the importance of coordinating multiple representations in programming, little is known about how multi-representational systems are used for this kind of programming task.

This paper reports an investigation into multiple representation use in novice program debugging. The next section briefly describes research on working with multiple external representations, both inside and outside the programming area. After this, we present the experimental method we applied in an empirical study, and describe its main findings. Finally we relate these findings to similar work both inside and outside the area of computer programming.

2 Coordination of multiple external representations in programming

Modality and *perspective* are two important aspects to consider regarding the coordination of multiple representations in programming (de Jon, Ainsworth, Dobson, van der Hulst, Levonen, & Reimann, 1998).

2.1 Modality

The term 'modality' is used here to mean the representational form used to present or display information, rather than in the psychological sense of a sensory channel. A typical modality distinction is between propositional and diagrammatic representations.

Thus, this first aspect refers to coordinating representations which are basically propositional with those that are mainly diagrammatic. It is not clear whether coordinating representations with the same modality type has advantages over working with mixed multiple representations or whether including a high degree of graphicality has potential benefits for performing the task (Ainsworth, Wood, & Bibby, 1996).

According to (Ainsworth et al., 1996), in general, the more different the degree of graphicality external representations exhibit, the more difficult it is for students to coordinate them. However, this will also depend on individual ability and the support given by the software environment. According to Oberlander, Stenning, and Cox (1999), one view is that if the multimodal system involves diagrammatic representations, people who have a preference for the visual modality are those who will benefit more from it (the *visual preference* hypothesis). Another view is that people will benefit from multimodal systems to the degree that they are able to translate between modalities (the *transmodal* hypothesis).

This latter view seems to be in agreement with Ainsworth, Wood, and O'Malley's (1998) conclusion that the design of environments that require the coordination of different modalities has to pay special attention to providing cues to support the process of translation between the representations.

Although programmers normally have to coordinate representations of different modalities, there has not been much research on this topic in the area of programming. One of the few examples is the GIL system (Merrill, Reiser, Beekelaar, & Hamid, 1992), which attempts to provide reasoning-congruent visual representations in the form of control-flow diagrams to aid the generation and comprehension of LISP, a functional programming language which normally employs textual representations. Merrill et al. (1992) claim that the GIL system is successful in teaching novices to program in this language; however, this work did not compare coordination of the same and different modalities.

Work in the algorithm animation area (Byrne, Catrambone, & Stasko, 1999) has found advantages for the use of multiple representations of mixed modality. Byrne et al. (1999) found that students benefited from the dual mental encoding that results from presenting a graphical visualisation of the program together with a textual explanation of it.

Other studies in the area have been concerned with issues related to the format of the output of debugging packages (Mulholland, 1997; Patel, du Boulay, & Taylor, 1997). Those studies have offered conflicting results about the coordination of representations of different modality. Patel et al. (1997) found that subjects working with representations of the same and different modalities had similar performance, while Mulholland (1997) reported that those working with different modalities showed a poorer performance than those working with the same modality. In both cases, participants worked with the program code and with the debugger's output. The debugger notations used by both of these studies were mostly textual. The only predominantly graphical debugging tool used was TPM (Eisenstadt, Brayshaw, & Paine, 1991). While the performance of the participants of the former study (Patel et al., 1997) was similar for the textual debuggers and for TPM, the subjects of the latter study (Mulholland, 1997) found working with TPM more difficult. One important difference between these two studies is that while the former used static representations, the latter employed a visualisation package (dynamic representations). The additional cognitive load of learning and using a multi-representational visualisation package may explain the difference in findings.

2.2 Perspective

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, 1987b), 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.

Experienced programmers, when comprehending code, are able to develop a mental representation that comprises these different perspectives or information types, as well as rich mappings between them (Pennington, 1987a).

Perspective is orthogonal to modality. Each of these perspectives can be displayed in either a diagrammatic or textual modality and each modality can be employed to present a variety of perspectives of the program.

2.3 Java debugging

To date, there have been numerous investigations of debugging behaviour across a range of programming languages (Gilmore, 1991; Romero, 2001; Vessey, 1989), and previous research has also examined the effect of representational mode upon program comprehension (Good, 1999; Merrill et al., 1992; Mulholland, 1997; Patel et al., 1997).

Studies relating to debugging strategies are of special interest to this investigation (Jeffries, 1982; Gugerty & Olson, 1986; Carver & Klahr, 1986; Kessler & Anderson, 1986; Katz & Anderson, 1988). According to Katz and Anderson (1988), debugging strategies can be classified into those that reflect either *forward reasoning* or *backward reasoning*. The first category comprises those strategies in which programmers start the bug search from the program code, while the second involves starting from the incorrect behaviour of the program 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 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.

However, debugging studies have tended not to employ debugging environments that are typical of those used by professional programmers (*i.e.* multi-representational software debugging environments, SDEs). Such environments typically permit the user to switch rapidly between multiple, linked, concurrently displayed representations. These include program code listings, data-flow and control-flow visualisations, output displays, etc. So the issue of how multiple representations are used and coordinated in debugging and in an object-oriented paradigm is relatively unexplored.

The aim of this paper is to investigate the coordination of multi-representational environments for Java debugging. In particular, this work aims to characterise the debugging strategies employed by participants, both at the level of focus of attention and representation use as well as in terms of the general reasoning strategy deployed.

3 Method

The aim of the experiment reported here was to relate debugging behaviour, especially representation use and coordination, to debugging strategy and accuracy, and to representation modality and perspective.

This experiment considered three within subject independent variables: visualisation modality (textual or graphical), visualisation perspective (data structure or control-flow), and type of error (data structure or control-flow). The qualitative analysis reported here took into account the verbal utterances of participants as well as their debugging accuracy, accumulated fixation time on and switching frequency between the available representations. Accumulated fixation time refers to the total time participants spent focusing on each representation for each of the debugging sessions. Switching frequency refers to the total number of switches involving each possible pair of representations (code and visualisation, code and output and visualisation and code) for each of the debugging sessions.

All subjects participating in the experiment were pre-tested on a battery of individual difference tests. These comprised verbal, spatial and translation between representations ability tests.

3.1 The experimental debugging environment

Romero, Cox, du Boulay, and Lutz (2002a) showed that visual attention tracking methods, and more specifically a tool like the Restricted Focus Viewer (RFV) (Blackwell, Jansen, & Marriott, 2000) can be used to investigate issues related to the *process* of coordinating multiple external representations in program debugging. Research of this type can offer important clues about the relationship between representation use and programming information types, the issue of sentential versus graphical representations, and debugging performance.

The Java SDE, a modified version of the RFV that we employed in our experiment, enabled participants to see the program code, its output for a sample execution, and a visualisation of this execution. A screen shot of the system is shown in Figure 1. Participants were able to see the several program class files in the code window, one at a time, through the use of the side-tabs ('coin', 'pile', 'till' in the example shown). Additionally, the visualisation window presented a visualisation of the program's execution similar to those found in Object-Oriented software development environments. This visualisation highlighted either a data structure or a control-flow perspective. These representations were selected because research in Object-Oriented program comprehension has suggested that function and data element information is highlighted in languages of this programming paradigm while control-flow is obscured (see Section 3.1).

In our experiments, these representations, and the Java SDE, were static in that participants were presented with selected pre-computed information about the program execution. We chose to present information in this limited

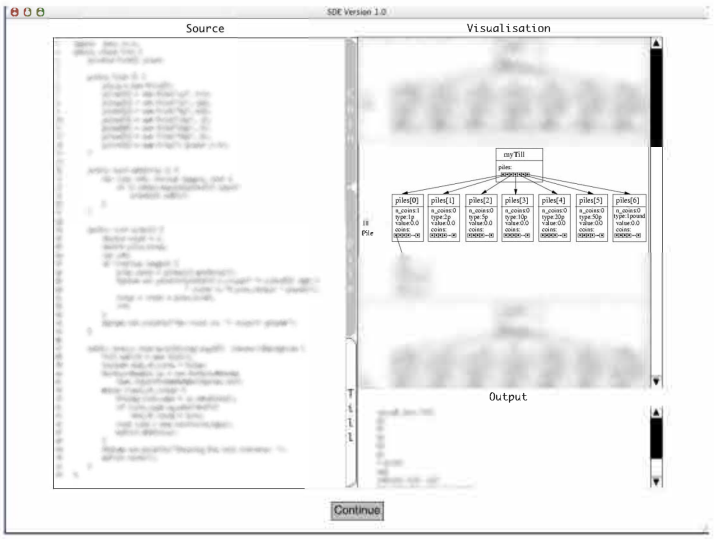

Figure 1: The debugging environment used by participants

way so that we could control for issues like the increased complexity of dealing with a full debugging environment and the ephemeral nature of the information presented by a dynamic debugging tool, which, as mentioned in Section 2.1, could have played a role in the discrepancy of results reported by Mulholland (1997) and Patel et al. (1997).

The SDE was implemented on top of a modified version of the Restricted Focus Viewer (RFV) (Blackwell et al., 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 allows visual attention to be tracked as the user moves an unblurred 'foveal' area around the screen. Use of the SDE enabled moment-by-moment representation switching between concurrently displayed, adjacent representations to be captured for later analysis.

A previous study which employed the SDE to validate the suitability of this technology to investigate Java program debugging offered promising results (Romero et al., 2002a). Specifically, it suggested that debugging performance is not affected by this method of tracking visual attention and that there might be fixation and switching patterns characteristic of superior debugging in this context.

3.2 Participants and procedure

The experimental participants were forty nine computer science undergraduate students from the School of Cognitive and Computing Sciences at Sussex University, U.K. All of the participants had taken a three month introductory course in Java, but their programming experience varied from having only taken this course to a few extra months of Java experience and even having worked as professional programmers. The *less experienced* programmers had on average 3 months of Java experience (basically the duration of the introductory Java course) plus 10.5 months of other programming experience, while the *more experienced* group had on average 1 year of Java and 13 months of other programming experience.

Participants performed five debugging sessions. The first one was a warm-up session and it employed a functional visualisation. The four main sessions followed, two of them using a data structure and the other two a control-flow visualisation. Also, two of them employed a textual and the other two a graphical visualisation. In this way, the main sessions' conditions comprised the four ways in which perspective and modality could be combined, and their order and combinations were counterbalanced across participants and target programs.

Each debugging session consisted of two phases. In the first phase participants were presented with a specification of the target program. This program specification consisted of two paragraphs that described, in natural language, the problem that the program was intended to solve, the way it should solve it (detailing the solution steps, specifying which data structures to use and how to handle them), together with some samples of program output (both desired and actual). When participants were clear about the task that the program should solve and also how it should be solved, they moved on to the second phase of the session.

In the second phase of a debugging session participants were presented with three windows containing the program code, a sample interaction with the program and a visualisation which illustrated this interaction. They were allowed up to ten minutes to debug each program. They were instructed to identify as many errors as possible in this program and to report them verbally by stating the class file and line number in which they occurred as well as a brief description of them. They were also encouraged, besides reporting the errors, to think aloud throughout this second phase. Some participants chose to speak much more than others.

The target programs consisted of five short Java programs. Functionally, the 'warm-up' session program detects whether a point is inside a rectangle, given the coordinates of the point and the vertices of the rectangle. The first and second experimental program prints out the names of the children of a sample family. The main difference between these two programs is that the second one is a more sophisticated version of the first one. The third and fourth experimental programs ('Till' programs) count the cash in a cash register till, giving subtotals for the different coin denominations. Again, the main difference between these two versions is that the fourth program is implemented in a more sophisticated way. Some of the code, output for a sample execution session and a control-flow graphical visualisation of this execution for one of the *Till* programs are shown in Figures 2, 3 and 4 respectively.

```
  public void add(Coin c) {
    for (int i=0; i<piles.length; i++) {
      if (c.label.equals(piles[i].coin_type))
        piles[0].add(c);
    }
  }

  public static void main(String args[])
                          throws IOException {
    Till myTill = new Till();
    boolean end_of_coins = false;
    BufferedReader in = new BufferedReader
            (new InputStreamReader(System.in));

    while (!end_of_coins) {
      String coin_type = in.readLine();
      if (coin_type.equals("end"))
        end_of_coins = true;

      Coin coin = new Coin(coin_type);
      myTill.add(coin);
    }
    System.out.println("Till contents:");
    myTill.count();
  }
}
```

Figure 2: Segment of the program code for the Till class.

The programs of the two main debugging sessions were seeded with four errors, and the 'warm-up' session's program was seeded with two errors. The errors of the main debugging sessions' programs can be classified as 'control-flow' and 'data structure'. In this classification, control-flow errors have to do with the execution of the program not following a correct path. For example, the control-flow error in the *Till* program is located in the two last lines of the *while* loop of its *main* procedure. These two lines should be included within an *else* structure, so that the execution of the program either acknowledges an end-of-coins case or adds the new coin to the till, but never follows both paths at the same time.

Data structure errors normally have undesired consequences for the program data structures. For the *Till* program of Figure 2, the data structure error is located within the only instruction of the *if* structure of the *add* method. This error consists of every coin added to the till being sent only to the first money pile, regardless of its type. In this way, the money pile receiving all coins is one which should only accumulate coins of a one-pence denomination.

3.3 Analysis of the representational system

This section offers a brief analysis of the representational system employed in the experimental task. The DeFT framework (Ainsworth & Labeke, 2002) provides a basis for exploring the space of interactions between the variables of interest. The DeFT framework has been proposed by Ainsworth and Labeke

```
rsunx% java Till
5p
1p
2p
5p
1 pound
end
unknown coin: end


Till contents:
5 1p coins is 0.05 pounds
0 2p coins is 0.0 pounds
0 5p coins is 0.0 pounds
0 10p coins is 0.0 pounds
0 20p coins is 0.0 pounds
0 50p coins is 0.0 pounds
0 1 pound coins is 0.0 pounds
The total is: 0.0 pounds

rsunx%
```

Figure 3: Output from a sample execution session of the *Till* program.

(2002) for systems that work with multiple external representations. Although this framework is concerned with learning, some of the issues it raises can be applied to multi-representational systems of other sorts. It comprises three fundamental aspects: the functions of the representations, the cognitive tasks that must be undertaken by a user of these systems, and the design parameters that are unique to learning with multiple external representations. For the purposes of this analysis, only functions and cognitive tasks will be taken into account.

The functions of the representations are the roles each representation and representation subsystem play within the whole system. For example, representations might complement each other because they encode different information or because they support different cognitive processes.

Cognitive tasks are the activities that users must undertake in order to, for example, understand how each representation in the system encodes information, how to select the appropriate representation to use at any given moment, and how to coordinate the representations in the system.

Generally speaking, the code represents the specification of the solution to a problem in the programming language. For the specific problems in the debugging exercise, the solution consists mainly of simulating the behavior of entities in the real world. In this way, the output represents some aspect of this behavior in symbolic terms. Finally, the visualisation represents certain aspects of the execution of the program.

The main purpose of the debugging environment is to help users to build a robust mental representation of the program under consideration so that they can discover and correct any potential errors. In this way, the functions of the external representations of this debugging environment were mainly to play

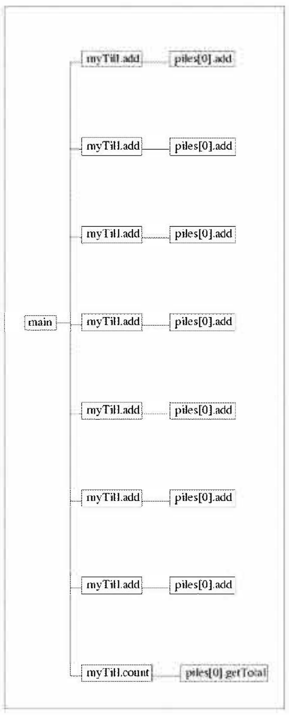

Figure 4: Control-flow graphical visualisation of a sample execution session of the *Till* program.

complementary roles and to assist in the construction of a deeper and more comprehensive understanding of the program.

There were two kinds of complementary roles played by these external representations: one concerned with processes, the other concerned with information. In the graphical visualisation condition, different comprehension processes can be brought into play because the code and output representations are mainly textual. The representations also provided additional information because although all information about the program is implicit in the code, the input for the sample interaction was only available in the visualisation and output representations.

The way in which the representations support the construction of a deeper understanding is by helping users to identify the different perspectives or information types comprised by the program. According to Pennington (1987a), developing a mental representation that comprises these different perspectives as well as to rich mappings between them is characteristic of good programmers.

3.4 Debugging accuracy scoring

Participants described aloud where the errors were located and their nature. The audio recordings of the debugging sessions were analysed to identify the participants' debugging accuracy. Each set of utterances reporting an error was scored according to whether participants identified the place and nature of the error correctly. The place of the error was considered correct if participants mentioned the line of code where the bug occurred, and partially correct if they mentioned only the Java method where it happened. Similarly, identifying the nature of the error was considered as correct if participants described it appropriately or if they proposed a correct fix for it. If, for example, they described an effect but not the cause of the error, the score for identifying the nature of the error was reduced.

4 Results

The quantitative analysis of this experiment has already been reported in Romero, Lutz, Cox, and du Boulay (2002b). This quantitative analysis suggested that graphical representations might be more useful than textual ones when the degree of difficulty of the debugging task poses a challenge to programmers. Additionally, the results of that analysis linked programming experience to switching behaviour, suggesting that although switches between the code and the visualisation are the most common ones, programming experience might promote a more balanced switching behaviour between the main representation, the code, and the secondary ones.

For the purposes of the qualitative analysis, the data for two participants only were taken into account. The reason for this was that although all participants were encouraged to verbalise their thoughts, this was not a compulsory condition, and only a small percentage of the total participant population did so. There were six students who talked the whole way through the experiment, and from these only the two more contrasting participants (in terms of the independent variables and the pre-test results) are described here. Throughout this analysis, these two participants will be referred as Participants 1 and 2.

Participant 2 had considerably more programming experience than participant 1. Participant 2 had worked as a professional programmer, knew at least three other programming languages apart from Java, had 48 months of general programming experience and 12 months of experience with Java. On the other hand, participant 1 had not worked as a professional programmer did not know any other programming languages apart from Java and had only 4 months of both general and Java programming experience. The results of the individual differences pre-tests were similar for these two participants, except for the case of the verbal ability test. The score for Participant 2 in this test was good while that of Participant 1 was poor.

This analysis compares verbal utterance and log files for these two participants to explore whether individual differences and different levels of experience were related to the information types referred to by their verbalisations as well as their general debugging strategy. In order to carry out this comparison, the utterances of these two participants were categorised both in terms of general strategy and the information types they referred to. The utterance categorisation scheme is similar to those applied in Mulholland (1997) and in Bergantz and Hassell (1991).

This verbal information was supported by synchronous data from the log file to create a better picture of their debugging strategy.

4.0.1 Utterance analysis

Tables 1 and 2 present the verbal utterances data for the two participants. Table 1 shows the relative percentages of the different types of utterances. The final row *Total number of utterances* shows the total number of utterances in each debugging session. It can be noticed that Participant 2 provided more utterances of the type *spotting suspicious code* than Participant 1. Also, Participant 1, unlike Participant 2, did not provide utterances of the type *communication of compliance*. On the other hand, Participant 2, unlike Participant 1, did not talk in terms of *agenda management*. This table does not exhibit any obvious pattern which characterises sessions by experimental condition.

Table 2 shows the percentages of the different information types referred to by the participants. Notice that the utterances taken into account for this table are a subset of the total number of utterances of participants; this table only considers those verbalisations referring directly to the program code. Most of these code references occurred under the utterance type *code description*, but also included some in the *hypothesis testing*, *error reporting* or *noticing inconsistency* types in table 1, among others. The column labeled *undetermined* is for those utterances which were describing the code superficially, almost reading it out loud, and therefore could not be classified as comprising a specific information type. It can be noticed that Participant 1 talks mostly in terms of data structure, while Participant 2 produces utterances of *undetermined* type.

4.0.2 Debugging strategy analysis

The debugging sessions analysed shared several characteristics. First, both participants started these sessions by making long fixations at the code window, reading the program almost like reading prose, from top to bottom. These initial *code browsing episodes* might have been necessary for them to familiarise themselves with the code. These code browsing episodes varied in length, sometimes they were relatively short, while at other times they extended to cover almost all the debugging session. Occasionally participants would discover a suspicious piece of code within these initial code browsing episodes. Sometimes this *spotting a suspicious piece of code* would prompt participants to report this piece of code as containing an error.

After these initial code browsing episodes, the referred participants would

Utterance type	Example	Participant 1				Participant 2			
		dg	dt	cg	ct	dg	dt	cg	ct
Hypothesis testing	Should be around line 20..22	5	11	8	5	5	3		2
Representation related	..I'll try the other screen		5	8	5	2	3		4
Confirmatory	yeah it was right what I said before...	6	2			8	3		4
Code description	so you've got three classes...got name age sex	63	48	43	63	37	55	62	40
Visualisation description	Ok all the inputted coins are going to pile zero		5	8	3				
Output description	just saying male, male male	9				5			
Agenda management	I'll come back and look at that	5	2	2	2				
Self-awareness of difficulty	I find it very difficult	6	14	14	7	15	8		5
Noticing inconsistency	which doesn't really make sense..		6	6		5	2		7
Point of insight	that's why it keeps on saying "oh its zero" in the visualisation	3			3		2		
Analogy	We've got the same things as before			2	7				2
Meta-cognitive	this output on the side is quite helpful		2						7
Communication of compliance	I'm just looking at the usual program interaction					10	12	9	14
Error reporting	For a start that 5 P shouldn't be 5 it should be 0.05	3	2	6	3	8	8	9	7
Spotting suspicious code	buffer reader equals new buffer reader, that does seem a bit odd		2	3	2	4	4	19	5
Total number of utterances		43	43	35	58	40	47	21	42

Table 1: Relative percentages of the different types of participants' utterances. dg = data structure graphical condition, dt = data structure textual condition, cg = control-flow graphical condition, ct = control-flow textual condition

Information type	Participant 1				Participant 2			
	dg	dt	cg	ct	dg	dt	cg	ct
Control-flow	23%	8%	9%	5%	15%	6%	28%	23%
Data structure	40%	52%	56%	59%	25%	42%	28%	23%
Undetermined	36%	40%	35%	36%	60%	52%	43%	54%
Total number of utterances	30	25	23	31	20	31	14	22

Table 2: Percentages of the different information types referred to by the participants. dg = data structure graphical condition, dt = data structure textual condition, cg = control-flow graphical condition, ct = control-flow textual condition

sometimes engage in several *coordination of representations episodes.* These episodes were characterised by frequent switches between the code and one of the other two representations. In these episodes, it seems that participants were trying to build a more robust understanding of the program by integrating information from different external representations.

Sometimes, errors were reported after a coordination of representations episode. Participants combined a forward and backward reasoning strategy in these episodes. Sometimes, by interpreting the code, they would create expectations about the content of one of the other representations. If these expectations were not met, the participant tried to locate the place in the code which might be responsible for this inconsistency, as this place could be the source of the error. On other occasions they would notice a deviation from the desired behaviour of the program in either the visualisation or the output window, and try to link it to the place in the code where it originated as this location could contain the error.

In some cases, participants could not identify the error after a coordination of representations episode. In these cases, an impasse was produced and they would normally return to a code browsing episode.

Table 3 shows the number of coordination of representations episodes and of suspicious piece of code spottings. The rows after these events show the number of times they prompted participants to report an error and also how many times these reports were correct. This table also presents the number of bugs detected and percentage of time devoted to code browsing episodes.

One important difference between Participants 1 and 2 was that Participant 2 devoted a high proportion of his debugging session time doing code browsing episodes and reported a high proportion of errors by spotting a suspicious piece of code. It is relevant here to note that Participant 2 showed a high level of skill when translating between representations as well as verbal skills in the experiment pre-tests. He also had more programming experience than

	Participant 1				Participant 2			
	dg	dt	cg	ct	dg	dt	cg	ct
Coordination episode	3	1	2	1	2		1	2
Episode leading to report	1		1		1			2
Successful episode	1		1		1			1
Spot suspicious code		1	1	1	2	2	4	1
Spotting leading to report		1	1	1	2	2	2	1
Successful spotting			1	1	1			1
Errors detected	1	0	2	1	2	0	0	2
Initial code browsing episode time percentage	35	33	36	42	63	99	76	41

Table 3: Number of coordination of representations episodes and of suspicious piece of code spottings. dg = data structure graphical condition, dt = data structure textual condition, cg = control-flow graphical condition, ct = control-flow textual condition

Participant 1. This seems to indicate that he chose to concentrate mainly on the code only, not because of a lack of ability or confidence to coordinate the other two representations, but because he preferred to work in a uni-modal, textual environment.

Despite this difference in strategy, the debugging accuracy of these two participants was similar.

5 Discussion

This investigation aimed to relate debugging strategy and performance to representation use in a multi-representational, multi-modal debugging environments similar to those found in commercial software development environments and software visualisation packages. These sorts of environment are characterised by having several concurrently displayed representations of the program. There is a central representation, the program code, and a series of secondary representations that support it (program output and execution visualisations).

The qualitative analysis of the logs and verbal utterances files for two vocal participants showed several common characteristics in these two debugging sessions. Both participants started their sessions with code browsing episodes in which they performed long fixations at the code window, reading the program almost like reading prose, from top to bottom. Occasionally participants would discover, within these code browsing episodes, a suspicious piece of code.

This initial code browsing episode to get familiar with the code is in agreement with studies that have suggested that when debugging someone else's code, programmers devote an initial period of time to do program comprehension (Jeffries, 1982; Kessler & Anderson, 1986; Katz & Anderson, 1988). Spotting suspicious pieces of code during these episodes could be classified as a *comprehension* debugging strategy (Katz & Anderson, 1988), in which participants find bugs while building a representation of the program.

After this initial code browsing episode, participants would sometimes engage in several coordination of representations episodes. These episodes were characterised by frequent switches between the code and one of the other two representations. Sometimes, errors were reported after a coordination of representations episode. These episodes seem to be a combination of *hand simulation* and *causal reasoning* debugging strategies (Katz & Anderson, 1988), because participants would reason backwards and forwards between the code and the other two available representations. These combination of strategies seem to be due mostly to the employment of a multi-representational debugging environment and in particular to the visualisation representation.

Previous studies (Jeffries, 1982; Gugerty & Olson, 1986; Katz & Anderson, 1988) distinguished clearly between hand (mental) simulation, in which the program was evaluated by the programmer as if she were the computer and causal reasoning, in which an error was spotted in the output of the program and then traced backwards to the code. It seems reasonable to assume that having a representation that could be considered as an *intermediate* type of output could promote a strategy in which the program would be mentally simulated and its expected behaviour verified step by step on the visualisation representation. Differences between this expected behaviour and the one reflected in the visualisation could prompt possible error hypotheses. But it is also possible that the visualisation could contain inconsistencies not related to the hand simulation expectations, but to those that have to do with the global functionality of the program (the price of an item suddenly changing to a negative number, for example). In this latter case, the programmer would probably reason backwards from this inconsistency to the code to discover a possible error.

The problem solving strategy of these two participants was different in that Participant 2 had long code browsing episodes and spotted suspicious pieces of code more frequently during these initial code browsing episodes. Participant 1, on the other hand, had relatively short code browsing episodes and engaged in more coordination of representations episodes. As pointed out in Section 4, he might have chosen to concentrate largely on the code, not because of a lack of ability or confidence to coordinate the other two representations, but because he preferred to work in a uni-modal, textual environment.

The finding that despite differences in strategy, the debugging accuracy of the two participants was similar, was also noteworthy. Marked individual differences in reasoning *strategy* associated with similarities in performance have been found on other computer-based tasks such as proof development in first-order logic (Oberlander et al., 1999). In this case, Participant 2 had considerably more programming experience than Participant 1. He also showed a higher level of skill when translating between representations as well as in verbal abilities in the experiment pre-tests. Taking this into account, it was surprising that he did not show a better debugging accuracy than Participant 1. One possible explanation for this is that his choices of debugging strategies were not optimal. For example, by choosing a comprehension debugging strategy he engaged in a relatively large number of reports of suspicious pieces of code. These reports were unsuccessful most of the times. The fact that the comprehension debugging strategy was not highly effective for him suggests that he might have been better off by engaging in more coordination of representations episodes instead.

A clear difference in the types of utterances of these two participants was that Participant 1 talked mostly in terms of data structure, while Participant 2 produced utterances of *undetermined* type. It seems reasonable to assume that this difference is related to their difference in the choice of debugging strategies. Possibly data structure utterances were preferred to control-flow ones given that Java as an Object-Oriented language would highlight function as well as static data element information whilst obscuring control-flow information (Corritore & Wiedenbeck, 1999; Wiedenbeck & Ramalingam, 1999).

It is worth noting that there were no noticeable differences either in debugging strategy or in utterance type due to the type of visualisation employed (control-flow and data structure, textual and graphical). However, it is clear that these results should be taken with caution as only a small proportion of the participants was taken into account.

6 Conclusions

This study investigated Java program debugging strategies through the use of a software debugging environment that provided concurrently displayed, adjacent, multiple and linked representations and that allowed visual attention switches of participants to be tracked.

The experimental results suggest that the employment of a multi-representational debugging environment and in particular of a visualisation representation might have promoted participants to use a debugging strategy that combined a forward and backward mode of reasoning about the program code and the rest of the available representations. In this debugging strategy, programmers performed frequent switches between the code and one of the other two representations. In these episodes, it seems that participants were trying to build a more robust understanding of the program by integrating information from the different external representations available.

The results confirm previous findings that suggest that when debugging someone else's code, programmers devote an initial period of time to do program comprehension (Jeffries, 1982; Kessler & Anderson, 1986; Katz & Anderson, 1988). This program comprehension episode seem to be characterised by long fixations at the code window, reading the program almost like reading prose, from top to bottom.

The results of the experiment reported here need to be reinforced by further empirical studies with different experimental settings. One experimental factor that is important to manipulate is the use of a dynamic debugging environment instead of, as in this case, a static one. The use of a dynamic debugging environment might impose an additional cognitive load on participants but will enhance the ecological validity of the experimental task by providing an interactive (and more authentic) SDE environment.

7 Acknowledgments

This work is supported by the EPSRC grant GR/N64199. The authors would like to thank the participants for taking part in the study.

References

Ainsworth, S., & Labeke, N. V. (2002). Using a multi-representational design framework to develop and evaluate a dynamic simulation environment. In *Proceedings of the 2002 Dynamic Information and Visualisation Workshop* Tuebingen, Germany.

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.

Ainsworth, S., Wood, D., & O'Malley, C. (1998). There is more than one way to solve a problem: evaluating a learning environment that supports the development of children's multiplication skills. *Learning and Instruction*, *8*(2), 141–157.

Bergantz, D., & Hassell, J. (1991). Information relationships in PROLOG programs: how do programmers comprehend functionality?. *International Journal of Man-Machine Studies*, *35*, 313–328.

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

Byrne, M. D., Catrambone, R., & Stasko, J. T. (1999). Evaluating animations as student aids in learning computer algorithms. *Computers & Education*, *33*, 253–278.

Carver, S. M., & Klahr, D. (1986). Assessing children's logo debugging skills with a formal model. *Journal of educational computing research*, *2*.

Corritore, C. L., & Wiedenbeck, S. (1999). Mental representations of expert procedural and object-oriented programmers in a software maintenance task. *International Journal of Human Computer Studies*, *50*, 61–83.

de Jon, T., Ainsworth, S., Dobson, M., van der Hulst, A., Levonen, J., & Reimann, P. (1998). Acquiring knowledge in science and mathematics: The use of multiple representations in technology-based learning environments. In van Someren, M. W., Reimann, P., Boshuizen, H. P. A., & de Jon, T. (Eds.), *Learning with Multiple Representations*, pp. 9–40. Elsevier Science, Oxford, U.K.

Eisenstadt, M., Brayshaw, M., & Paine, J. (1991). *The Transparent Prolog Machine*. Intellect, Oxford, England.

Gilmore, D. J. (1991). Models of debugging. *Acta psychologica*, *78*(1), 151–172.

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

Good, J. (1999). *Programming Paradigms, Information Types and Graphical Representations: Empirical Investigations of Novice Program Comprehension*. Ph.D. thesis, University of Edinburgh, Edinburgh, Scotland, U.K.

Gugerty, L., & Olson, G. (1986). Comprehension differences in debugging by skilled and novice programmers. In Soloway, E., & Iyengar, S. (Eds.), *Empirical Studies of Programmers, first workshop*, pp. 13–27 Norwood, New Jersey. Ablex.

Jeffries, R. (1982). A comparison of the debugging behaviour of expert and novice programmers. In *Proceedings of AERA annual meeting*.

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

Kessler, C. M., & Anderson, J. R. (1986). A model of novice debugging in lisp. In *Empirical Studies of Programmers, first workshop* Norwood, New Jersey. Ablex.

Merrill, D. C., Reiser, B. J., Beekelaar, R., & Hamid, A. (1992). Making processes visible: scaffolding learning with reasoning-congruent representations. *Lecture Notes in Computer Science*, *608*, 103–110.

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 Moss, L. S., Ginzburg, J., & de Rijke, M. (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. (1987a). Comprehension strategies in programming. In Olson, G. M., Sheppard, S., & Soloway, E. (Eds.), *Empirical Studies of Programmers, second workshop*, pp. 100–113 Norwood, New Jersey. Ablex.

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

Romero, P. (2001). Focal structures and information types in Prolog. *International Journal of Human Computer Studies*, *54*, 211–236.

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

Vessey, I. (1989). Toward a theory of computer program bugs: an empirical test. *International Journal of Man-Machine Studies*, *30*(1), 23–46.

Wiedenbeck, S., & Ramalingam, V. (1999). Novice comprehension of small programs written in the procedural and object-oriented styles. *International Journal of Human Computer Studies*, *51*, 71–87.