Visual attention and representation switching during Java program debugging: A study using the Restricted Focus Viewer.

Pablo Romero, Richard Cox, Benedict du Boulay, and Rudi Lutz

Human Centred Technology Group,
School of Cognitive & Computing Sciences
University of Sussex, Falmer, Brighton,
East Sussex, BN1 9QH, UK

Abstract. Java program debugging was investigated in programmers who used a software debugging environment (SDE) that provided concurrently displayed, adjacent, multiple and linked representations consisting of the program code, a functional visualisation of the program, and its output.
A modified version of the Restricted Focus Viewer (RFV)[3] - 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. Other measures included debugging performance (number of bugs identified, the order in which they were identified, bug discovery latencies, *etc.*).
The aim of this investigation was to address questions such as 'To what extent do programmers use each type of representation?' and 'Are particular patterns of representational use associated with superior debugging performance?'.
A within-subject design, and comparison of performance under (matched) RFV/no-RFV task conditions, allowed the use of the RFV as an attention-tracking tool to be validated in the programming domain.
The results also provide tentative evidence that superior debugging using multiple-representation SDE's tends to be associated with a) the predominant use of the program code representation, and b) frequent switches between the code representation and the visualisation of the program execution.

1 Introduction

When trying to perform a programming activity in everyday settings, programmers normally work with a variety of external representations as well as the program code. Some of these external representations are used in debugging packages, prototyping and visualisation tools in software development environments, or are included as part of internal and external documentation. Therefore, programming normally requires the co-ordination of multiple representations.

Probably the most typical case, at least for beginner programmers, of co-ordination of external representations in programming is working with debugging packages, a common example of a visualisation tool. Novice programmers often spend a good amount of their learning time attempting to understand the behaviour of programs when trying to discover errors in the code. To perform this task, novices normally work with both the program code and the debugger output, trying to co-ordinate and make sense of these representations. Yet studies of program comprehension have not, to the best of our knowledge, addressed the issue of how multiple external representations are used for this kind of programming task.

We believe that the investigation of the co-ordination of multiple external representations in programming can be effectively supported by visual attention tracking methods, and that a tool like the Restricted Focus Viewer (RFV)[3] can be used for this purpose. The use of this experimental tool allows us to analyse the *process* of representational use in program debugging by addressing questions such as 'How much time do users spend using each representation?', 'Under what circumstances do programmers switch between representations' and 'Are particular patterns of representational use associated with superior debugging performance?'.

1.1 Co-ordination of multiple external representations in programming

Two important aspects to consider regarding the co-ordination of multiple representations in programming are *modality* and *perspective* [9]. The term 'modality' is used here to mean the representational forms used to present or display information, rather than in the psychological sense of sensory channel. A typical modality distinction here is between propositional and diagrammatic representations. Thus, the first aspect refers to co-ordinating representations which are basically propositional with those that are mainly diagrammatic. It is not clear whether co-ordinating representations in 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.

Modality Although programmers normally have to coordinate representations of different modalities, there has not been much research on these issues in the area of programming. One of the few examples is the GIL system [15], 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 employs mainly textual representations. In [15], it is claimed that this system is successful in teaching novices to program in this language; however, this work did not compare co-ordination of the same and different modalities.

Work in the algorithm animation area ([5]) has found advantages for the use of multiple representations of mixed modality. In [5], it was found that students might benefit from the dual coding 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 [16,18]. Those studies have offered conflicting results about the co-ordination of representations of different modalities. In [18], it was found that subjects working with representations of the same and different modalities had similar performance, while in [16], it was reported that the ones 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 by these studies was TPM [10]. While the performance of the participants of the former study [18] was similar for the textual debuggers and TPM, the subjects of the latter study [16] 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.

Perspective The second aspect refers to co-ordinating representations that highlight either the same or different programming information types. Computer programs are information structures that comprise different types of information [20], and programming notations usually highlight some of these aspects at the cost of obscuring others [12]. 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 [19]. Some of these different information types are: function, data structure, operations, data-flow and control-flow. It is an open issue whether co-ordinating notations that highlight different information types will be more beneficial to programmers than working with those that highlight the same ones.

Java debugging To date, there have been numerous investigations of debugging behaviour across a range of programming languages [4,11,22,25] and previous research has also examined the effect of representational mode upon program comprehension [13,15,16,18].

However, these studies were performed mainly in the context of procedural or declarative computer languages. It is not clear whether the results will generalise to the (currently popular) Object-Oriented paradigm. Research in program comprehension for Object-Oriented languages suggests that these kinds of language highlight functional information [6,26]. However, it is not clear whether novice programmers working with medium size programs find comprehending function in Object-Oriented languages an easy task [27], specially because as program size increases, functional information tends to become diffuse.

Furthermore, debugging studies have not tended to employ debugging environments that are typical of those used by professional programmers (*i.e.* multi-representational software debugging environments (SDE's)). 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. In [23], a comprehensive survey of external representations employed in object-oriented programming environments is provided.

1.2 Aims

The aim of this work was to conduct a small-scale, exploratory study as a first step towards the development of a descriptive model of representational behaviour in program debugging.

To date, the RFV has been validated in the context of reasoning about simple mechanical systems via the inspection of static diagrams [3]. A secondary aim, therefore, was to validate the RFV for use in an active debugging context (in which users make frequent switches between multiple, heterogeneous, interactive representations).

Additional aims were:

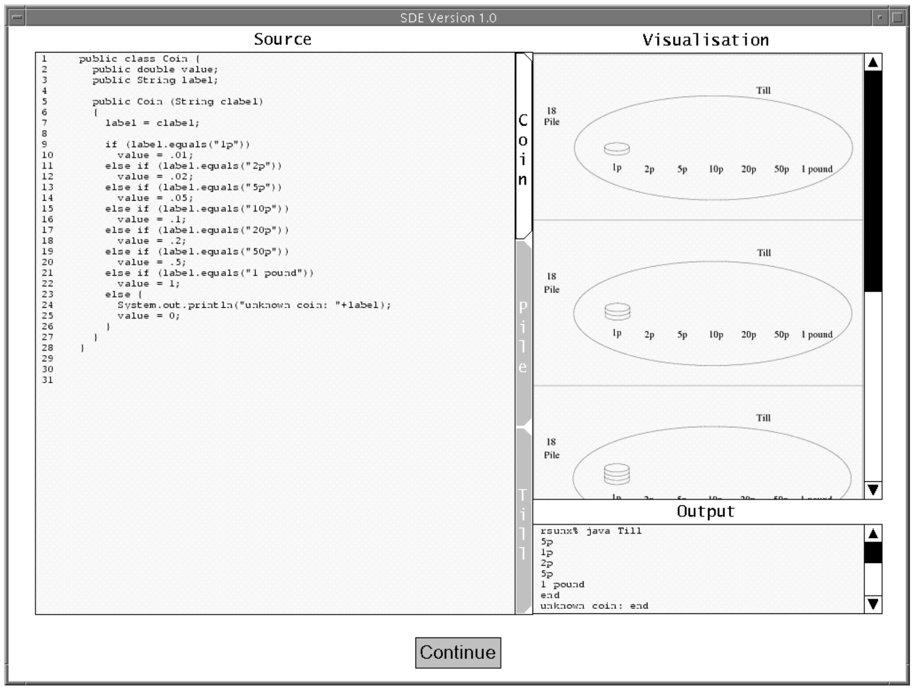
- to employ a multi-modal debugging environment; for multi-modality is a characteristic typical of those used in professional practice;
- to investigate debugging in the context of the Java programming language - a modern, Object-Oriented and widely used programming language;

2 Method, materials and procedure

2.1 The experimental debugging environment

The Java SDE enabled participants to see the program's code, its output for a sample execution and a visualisation of this execution in terms of the program's functionality. A screen shot of the system is shown in Figure 1. Participants were able to see the several program modules in the code window, one at a time, through the use of the side-tabs ('coin', 'pile', 'till'). Also, the visualisation window presented a functional visualisation of the program's execution similar to those found in code animation systems [14]. Functional representations were selected in preference to other program perspectives because research in Object-Oriented program comprehension has suggested that function is an important information type for these languages (see Section 1.1).

The SDE was implemented on top of a modified version of the Restricted Focus Viewer (RFV). The RFV has been reported in [3] as an alternative to eye-tracking devices. This program presents image stimuli in a blurred form (but note that none of the images in Figure 1 are so blurred). When the user moves the mouse to 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.



```
public class Coin {
  public double value;
  public String label;

  public Coin (String clabel)
  {
    label = clabel;

    if (label.equals("1p"))
      value = .01;
    else if (label.equals("2p"))
      value = .02;
    else if (label.equals("5p"))
      value = .05;
    else if (label.equals("10p"))
      value = .1;
    else if (label.equals("20p"))
      value = .2;
    else if (label.equals("50p"))
      value = .5;
    else if (label.equals("1 pound"))
      value = 1;
    else {
      System.out.println("unknown coin: "+label);
      value = 0;
    }
  }
}
```

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

Fig. 1. The debugging environment used by participants (with RFV switched off)

It allows visual attention to be tracked as the user moves an unblurred 'foveal' area around the screen. Use of the RFV enabled moment-by-moment representation switching between concurrently displayed, adjacent representations to be captured for later analysis. Here, we used a modified version of the original RFV to track visual attention and representation switches during program debugging.

The original RFV was modified for use in this study in several ways. First, stimulus images can be presented in a scroll or a tab pane. This allows us to present big images or more than one image in a specified display area. Second, the focused ('foveal') spot no longer follows the movement of the mouse. In our modified version, participants may click a mouse button to set the focused spot in the desired place. Every window image 'remembers' where its focused spot is, so when the user returns to that window the region in focus is the one that was set by the previous mouse click performed on that window image. This feature makes switching between stimulus images easier, because participants do not have to re-establish the place where they were looking at every time they switch their attention from one window image to another. Also, this change allowed us to distinguish between two kinds of mouse-usage - *i.e.* using the mouse to navigate among images versus using it to position the focused region.

Participant	Java experience (in months)	General programming experience (in years)
1	6	7
2	24	12
3	3	5
4	1.5	6.5
5	36	6

Table 1. Programming experience of participants

In order to assess the effect of using the RFV itself upon debugging performance, each participant also debugged an equivalently-matched program using the SDE with the RFV disabled. Experimental program versions were counterbalanced in RFV/non-RFV conditions across participants.

The data collected by the RFV consists of a log of mouse and keyboard actions, as well as the times taken by participants to perform the debugging sessions[1]. Additional data recorded included the number of program errors (bugs) identified by subjects, their description and the order in which they were identified.

2.2 Participants and procedure

The experimental participants were four DPhil students and one professional programmer. All of them knew Java and the four students were using it in coursework projects. Table 1 gives details of the participants programming experience.

Participants performed three debugging sessions. The first one was a warm-up session and it was performed under the restricted focus condition. The two main sessions followed — one with and the other without the restricted focus condition (order was counterbalanced across participants). Participants were allowed as much time as they needed in each of the sessions. They were instructed to find as many errors as they could in the programs. The debugging sessions 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 describing in plain English 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.

[1] These data are also used as input to a screen movie capture mode for 're-plays' post-session.

```
import    java.io.*;
public class Till {
    private MoneyPile[] piles;

    public Till () {
        piles = new MoneyPile[7];
        piles[0] = new MoneyPile("1p",.01);
        piles[1] = new MoneyPile("2p",.02);
        piles[2] = new MoneyPile("5p",.05);
        piles[3] = new MoneyPile("10p",.1);
        piles[4] = new MoneyPile("20p",.2);
        piles[5] = new MoneyPile("50p",.5);
        piles[6] = new MoneyPile("1 pound",1.0);
    }

    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 void count() {
        double total = 0;
        double pile_total;

        for (int i=0; i<piles.length; i++) {
            pile_total = piles[i].n_coins * piles[i].coin_value;
            System.out.println(piles[i].n_coins+" "+ piles[i].coin_type+
                                   " coins is "+ pile_total+ " pounds");
        }
        System.out.println("The total is: "+ total+" pounds");
    }

    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("Counting the till contents: ");
        myTill.count();
    }
}
```

Fig. 2. Code for the Till class.

In the second phase they were presented with three windows containing the program code, a sample interaction with the program and a visualisation which illustrated this interaction graphically. They were instructed to identify as many errors as possible in this program. When subjects reported that they thought they had detected all of the errors they moved on to the next debugging session.

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


Counting the 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%
```

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

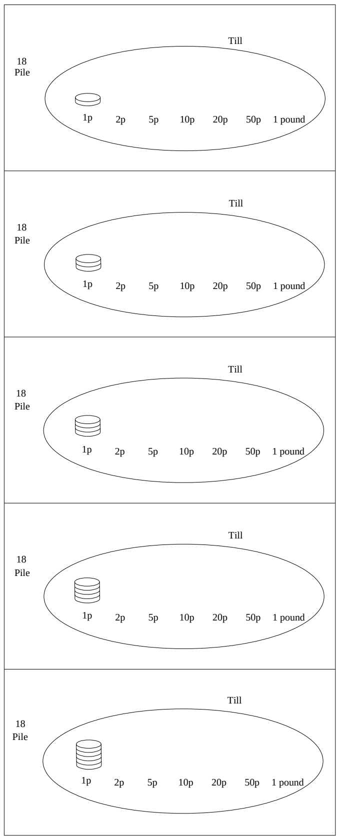


Fig. 4. Functional visualisation of a sample execution session of the *Till* program.

The target programs consisted of three short Java programs. The 'warm-up' session program detects whether a point is inside a rectangle, given the co-ordinates of the point and the vertices of the rectangle. The first experimental program prints out the names of the children of a sample family. The second experimental program ('Till') counts the cash in a cash register till, giving subtotals for the different coin denominations. Some of the code, output for a sample execution session and a functional visualisation to this execution for the *Till* program are shown in Figures 2, 3 and 4 respectively.

The programs of the two main debugging sessions were seeded with three 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 'functional', 'control-flow' and 'data structure'. In this classification, functional errors are those that occur in the line or lines in which the main computation of the program is performed [21]. For example, in the *Till* program of Figure 2, the functional error can be found in the method count, where the grand total of the money being counted is not computed.

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 Results

3.1 Debugging performance

The results of the experiment in terms of debugging performance are presented in Table 2. It can be seen that although the restricted view condition slowed down the debugging performance of some participants, the number of errors found did not seem to be affected by this experimental condition (participants tended to spot more errors working under the restricted view condition). This result to some extent replicates that of [3], which compared eye-tracking with the RFV and found that response times were generally slower for the RFV, but other aspects of performance were less affected. The results reported here represent a more controlled validation of the RFV than that of [3], since in that study RFV data from a diagram inspection task were compared to eye-tracking data of an earlier study conducted by another investigator. In the current investigation a within-subject design ensured that participants served as their own controls and parallel forms of the tasks were counterbalanced across subjects for the RFV/non-RFV conditions.

Table 2 shows that in terms of number of errors spotted and for the two main sessions, the most successful participant was number 2 (5 errors found), then participants 3 and 4 (4 errors spotted each), then 5 with 3 errors located and finally participant 1 with only one error found.

Participant	Warm-up		RFV on		RFV off	
	Errors found	Time	Errors found	Time	Errors found	Time
1	0/2	19.53	1/3	5.9	0/3	10.22
2	2/2	19.24	3/3	36.28	2/3	6.28
3	2/2	29.58	3/3	17.23	1/3	9.56
4	1/2	22.46	2/3	33.42	2/3	21.26
5	2/2	12.25	2/3	17.34	1/3	13.12

Table 2. Number of errors found and time taken for the three debugging sessions for each participant (the warm-up program was seeded with two errors and the other two with three errors)

Participant	Code	Visualisation	Output	Switches per minute
1	82.5	6.3	11.2	1.55
2	92.2	5.9	1.9	1.54
3	95.5	2.5	2.0	1.44
4	87.5	7.6	4.9	2.19
5	80.8	7.1	12.1	1.93

Table 3. Percentage of time spent in each representation and average number of switches per minute for each participant.

3.2 Debugging behaviour

The global experimental results in terms of debugging behaviour are shown in Table 3. This table presents the percentage of time that participants spent looking at each representation when working in the restricted view condition. Additionally, this table also presents the average number of switches per minute between the SDE representations.

The more successful participants (2,3 & 4) spent a longer amount of time focusing on the code representation compared to the less successful ones (1 & 5). No clear pattern seems to exist relating the average number of switches per minute to debugging performance in this representation of the data. However, when the averages are represented as annotations to cyclic directed graph depictions of switches between representations, a distinctive pattern emerges.

The different types of switches considered for this analysis are: a switch from the code representation to the visualisation, from visualisation to code, code to output representation, output to code, visualisation to the output and output to visualisation. Table 4 presents the results in these terms. For each participant, the number of switches per minute is reported. Figures 5, 6, 7, 8 and 9 present these data in the form of cyclic directed graphs for each participant, with the frequency of switches normalised. In each of the figures, the frequency of code-visualisation, visualisation-output, and code-output switches are represented by

Participant	Code to visualis.	Visualis. to code	Code to output	Output to code	Visualis. to output	Output to visualis.
1	1.54		1.54	2.31	0.77	
2	1.98	1.98	0.99	1.1	0.11	0.11
3	2.07	2.07	0.23	0.46	0.46	0.23
4	2.84	3.08	1.19	1.07	0.24	0.48
5	2.05	2.05	1.82	1.37	0.23	0.23

Table 4. Number of switches per minute for each switch type.

annotated directional arrows. From these Tables and Figures it can be observed that the most successful participants (2, 3 & 4) performed more frequent switches between the code and the visualisation than the less successful ones (1 & 5).

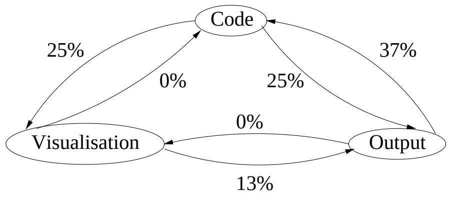


Fig. 5. Number of switches per minute for each kind of switch for participant 1.

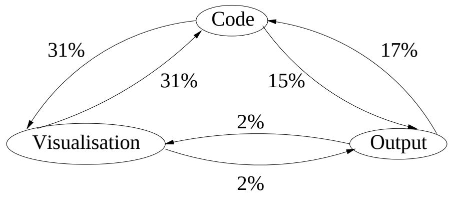


Fig. 6. Number of switches per minute for each kind of switch for participant 2.

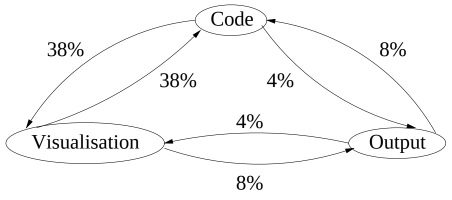


Fig. 7. Number of switches per minute for each kind of switch for participant 3.

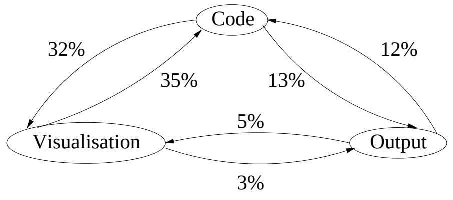


Fig. 8. Number of switches per minute for each kind of switch for participant 4.

4 Discussion

This investigation aimed to relate debugging performance to representation use in a multi-representational, multi-modal debugging environment similar to those found in commercial software development environments and software visualisation packages [23]. 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). Because software debugging environments are an important tool for novice programmers, modeling the process of representation use in this sort of environment can be of central relevance for educational purposes.

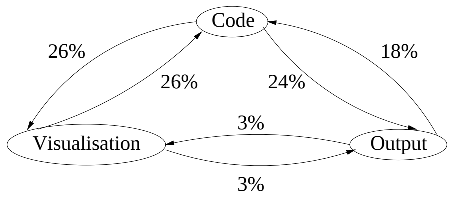


Fig. 9. Number of switches per minute for each kind of switch for participant 5.

Several hypotheses provide a basis for an initial descriptive model. First, better performing participants quickly identified the code representation as the central one and devoted a high percentage of inspection time to it. Secondly, the global frequency of switching *per se* does not seem to be related to good debugging performance. Successful and unsuccessful performance share similar levels of switching frequency among all the available representations. This differs from experimental results obtained in studies of representational behaviour in the domain of analytical reasoning [8, 7]. Those studies found that poor performers switched more frequently than successful ones. 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' [8]. In the present study, representations were complementary rather than alternative, therefore, switching did not represent discarding one representation for another, but complementing the information of one with another. Also, representations in the analytical reasoning study were presented serially, while the ones of the study reported here were displayed concurrently.

Our third hypothesis is related to the frequency of switching but it considers the different types of switches between the representations. A relatively high frequency of switches between the code and visualisation representations seems to be related to good debugging performance. This seems to be in agreement with findings in the program comprehension area that suggest that experienced programmers, when comprehending a program, are able to develop a mental representation that consists of different program perspectives, as well as rich mappings between them [19]. It also suggests that efficient debugging performance using SDE's is associated with the exploitation of representations that differ maximally in terms of their modality and expressiveness [24], *i.e.* in other words, good software bug diagnosticians tend to be heterogeneous reasoners [2].

The finding that performance, in terms of number of bugs detected, was superior in the RFV condition (compared to the non-RFV condition) was unexpected. This finding might be explained by at least two different causes (or by a combination of them). The first is that such a result might be explained by two features of the modified RFV which may have provided a degree of additional user-support. The amount of visual search performed by users in the RFV condition may have been reduced in two ways: a) by the RFV's blurring of unused parts of the display - thus reducing visual clutter, and, b) by the RFV's window-by-window location memory for the user's 'last-attended-to-position' that re-instated the 'fovea' on each switch back to a particular representation. The second explanation assumes that restricting the visual focus of the screen made local action more effortful. As suggested in [17], increasing the cost of performing the task could have increased the level of planning, which in turn could have enhanced the task performance.

5 Conclusions

This study investigated Java program debugging performance and behaviour 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 allowed us to propose several hypotheses which can be considered as a first step in building a preliminary descriptive model of the process of representation use for program debugging. Also, the modified version of the RFV was validated for use in the program representation domain[2].

The preliminary hypotheses about representation use in program bug diagnosis need to be further tested with a larger sample population and an experimental design that takes into account several important issues. For example, this investigation has only considered functional visualisations, while commercial software development environments also offer visualisations of data structure and control-flow [23]. Considering different kinds of visualisations would allow us to relate debugging performance of bug type to type of representation.

Another issue concerns the co-ordination of uni-modal versus multi-modal representations. Some evidence suggests that co-ordinating representations of different modality seems to be more complicated than co-ordinating those of the same perspective [1]. However, it is not clear how modality differences between the representations affect representational behaviour. These questions will be investigated in further work.

[2] Like the original RFV, our modified version has also been placed in the public domain for use by the diagrammatic research community. It can be downloaded from `http://www.cogs.susx.ac.uk/projects/crusade/`

Acknowledgments

This work is supported by the EPSRC grant GR/N64199.

The authors would like to thank the referees for their helpful comments and the participants for taking part in the study.

References

1. S. Ainsworth, D. Wood, and P. Bibby. Co-ordinating multiple representations in computer based learning environments. In P. Brna, A. Paiva, and J. Self, editors, *Proceedings of the 1996 European Conference on Artificial Intelligence on Education*, pages 336–342, Lisbon, Portugal, 1996.
2. J. Barwise. Heterogeneous reasoning. In G. Allwein and J. Barwise, editors, *Working papers on diagrams and logic.* Visual Inference Laboratory, Indiana University, 1993.
3. A. Blackwell, A. Jansen, and K. Marriott. Restricted focus viewer: a tool for tracking visual attention. In M. Anderson, P. Cheng, and V. Haarslev, editors, *Theory and Application of Diagrams. Lecture Notes in Artificial Intelligence 1889*, pages 162–177. Springer-Verlag, 2000.
4. D. Bergantz and J. Hassell. Information relationships in PROLOG programs: how do programmers comprehend functionality? *International Journal of Man-Machine Studies*, 35:313–328, 1991.
5. M. D. Byrne, R. Catrambone, and J. T. Stasko. Evaluating animations as student aids in learning computer algorithms. *Computers & Education*, 33:253–278, 1999.
6. C. L. Corritore and S. Wiedenbeck. Mental representations of expert procedural and object-oriented programmers in a software maintenance task. *International Journal of Human Computer Studies*, 50:61–83, 1999.
7. R. Cox. *Analytical reasoning with multiple external representations.* PhD thesis, University of Edinburgh, Edinburgh, Scotland, U.K., 1996.
8. R. Cox and P. Brna. Analytical reasoning with external representations: Supporting the stages of selection, construction and use. *Journal of Artificial Intelligence in Education*, 6(2/3):239–302, 1995.
9. T. de Jon, S. Ainsworth, M. Dobson, A. van der Hulst, J. Levonen, and P. Reimann. Acquiring knowledge in science and mathematics: The use of multiple representations in technology-based learning environments. In M. W. van Someren, P. Reimann, H. P. A. Boshuizen, and T. de Jon, editors, *Learning with Multiple Representations*, pages 9–40. Elsevier Science, Oxford, U.K., 1998.
10. M. Eisenstadt, M. Brayshaw, and J. Paine. *The Transparent Prolog Machine.* Intellect, Oxford, England, 1991.
11. D. J. Gilmore. Models of debugging. *Acta psychologica*, 78(1):151–172, 1991.
12. D. J. Gilmore and T. R. G. Green. Comprehension and recall of miniature programs. *International Journal of Man-Machine Studies*, 21(1):31–48, 1984.
13. J. Good. *Programming Paradigms, Information Types and Graphical Representations: Empirical Investigations of Novice Program Comprehension.* PhD thesis, University of Edinburgh, Edinburgh, Scotland, U.K., 1999.
14. S. P. Lahtinen, E. Sutinen, and J. Tarhio. Automated animation of algorithms with eliot. *Journal of Visual Languages and Computing*, 9:337–349, 1998.
15. D. C. Merrill, B. J. Reiser, R. Beekelaar, and A. Hamid. Making processes visible: scaffolding learning with reasoning-congruent representations. *Lecture Notes in Computer Science*, 608:103–110, 1992.

16. P. Mulholland. Using a fine-grained comparative evaluation technique to understand and design software visualization tools. In S. Wiedenbeck and J. Scholtz, editors, *Empirical Studies of Programmers, seventh workshop*, pages 91–108, New York, 1997. ACM press.
17. K. P. O'hara and S. J. Payne. Planning and the user interface: the effects of lockout time and error recovery cost. *International Journal of Human Computer Studies*, 50:41–59, 1999.
18. M. J. Patel, B. du Boulay, and C. Taylor. Comparison of contrasting Prolog trace output formats. *International Journal of Human Computer Studies*, 47:289–322, 1997.
19. N. Pennington. Comprehension strategies in programming. In G. M. Olson, S. Sheppard, and E. Soloway, editors, *Empirical Studies of Programmers, second workshop*, pages 100–113, Norwood, New Jersey, 1987. Ablex.
20. N. Pennington. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology*, 19:295–341, 1987.
21. R. S. Rist. Schema creation in programming. *Cognitive Science*, 13:389–414, 1989.
22. P. Romero. Focal structures and information types in Prolog. *International Journal of Human Computer Studies*, 54:211–236, 2001.
23. Romero, P., Cox, R., du Boulay, B. & Lutz, R. A survey of representations employed in object-oriented programming environments. (in press) *Journal of Visual Languages and Computing.*
24. K. Stenning and J. Oberlander. A cognitive theory of graphical and linguistic reasoning: logic and implementation. *Cognitive Science*, 19(1):97–140, 1995.
25. I. Vessey. Toward a theory of computer program bugs: an empirical test. *International Journal of Man-Machine Studies*, 30(1):23–46, 1989.
26. S. Wiedenbeck and V. Ramalingam. Novice comprehension of small programs written in the procedural and object-oriented styles. *International Journal of Human Computer Studies*, 51:71–87, 1999.
27. S. Wiedenbeck, V. Ramalingam, S. Sarasamma, and C. L. Corritore. A comparison of the comprehension of object-oriented and procedural programs by novice programmers. *Interacting with Computers*, 11:255–282, 1999.