Towards a Debugging Tutor for Object-Oriented Environments

Benedict DU BOULAY, Pablo ROMERO, Richard COX, and Rudi LUTZ
IDEAs Laboratory,
Human Centred Technology Research Group,
School of Cognitive & Computing Sciences,
University of Sussex, Brighton, BN1 9QH, UK.
bend@cogs.susx.ac.uk

Abstract. Programming has provided a rich domain for Artificial Intelligence in Education and many systems have been developed to advise students about the bugs in their programs, either during program development or post-hoc. Surprisingly few systems have been developed specifically to teach debugging. Learning environment builders have assumed that either the student will be taught these elsewhere or that they will be learnt piecemeal without explicit advice.

This paper reports on two experiments on Java debugging strategy by novice programmers and discusses their implications for the design of a debugging tutor for Java that pays particular attention to how students use the variety of program representations available. The experimental results are in agreement with research in the area that suggests that good debugging performance is associated with a balanced use of the available representations and a sophisticated use of the debugging step facility which enables programmers to detect and obtain information from critical moments in the execution of the program. A balanced use of the available representations seems to be fostered by providing representations with a higher degree of dynamic linking as well as by explicit instruction about the representation formalism employed in the program visualisations.

Introduction

The task of comprehending a piece of code is central to programming and comprehension is central to the task of debugging. Comprehension itself is multi-faceted and includes understanding the various relationships between the following (very partial list of) representations of a problem and its program:

(i) the code text itself as a specific implementation of the solution to the problem;

(ii) the input/output behaviour of the code when run;

(iii) dynamic representations of different aspects of the code, e.g. control flow and data-structure representations, produced by the debugging environment and the incidental output produced by deliberately inserted print statements and error messages;

(iv) the representation of the real world problem domain that the program is designed to address.

In debugging the programmer will normally have to reconcile his or her understanding of the code as a piece of text with the other representations given above. When debugging within the context of modern, multi-representational debugging environments, a special emphasis is normally placed on the relations between all of these representations and the programmer's expectation of the input/output behaviour of the program in comparison with how it actually behaves.

This paper is in three further parts. The next section offers a brief review of the area of debugging as a cognitive activity. The paper continues by describing two experiments where computer science students tried to find errors in Java programs using a software debugging environment (SDE) that provided them with concurrently displayed, adjacent, multiple and linked representations. These programming representations comprised the program code, visualisations of it and its output.

The first experiment was performed with a static SDE while the second one employed a dynamic, more interactive SDE. An important question addressed in these experiments had to do with characterising the debugging strategies of good programmers as a way to analyse the components of the debugging skill in the context of modern, multi-representational programming environments. Finally the experimental results are used to map out design issues for a tutor for Java debugging.

1. Teaching debugging

An interesting question concerns whether novices can be taught to become better debuggers by direct instruction rather than simply by amassing debugging experience in an ad hoc fashion as part of their general increase in programming skill.

Teaching debugging can be done at two levels: teaching debugging strategy or as instruction at the level of structural knowledge of programming.

1.1 Debugging strategy

Research on debugging strategy has concentrated on characterising the strategies of experienced programmers or/and comparing these to the strategies employed by novices, hoping to obtain in this way a list of good and poor debugging strategies, see e.g. [6, 7, 8, 21]. This research has found a set of debugging strategies that can be classified roughly into those that reflect either *forward reasoning* or *backward reasoning* [10]. 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 *program slicing* [22]. In simple mapping the program's output points directly to the incorrect line of code, while in program slicing the programmer concentrates on the output behaviour of the program and works back through the code to see which parts of it would have been responsible for that output and ignoring the rest (for the moment).

1.2 Structural knowledge

Comprehension is central to debugging and a crucial characteristic of comprehension is that it involves the coordination of several, complementary aspects of the code [15]. One way to explain the different strategies promoted by programming experience is in terms of the differences in the mental representations that programmers have built of the programs they are trying to understand [4].

One way to improve debugging performance would be to improve the quality of the mental representation produced by the program comprehension process. The question is therefore how to promote a robust, multi-faceted mental representation of the program in hand. A possible approach to promote this multi-faceted internal representation would be to employ multiple external representations.

Work in Software Visualisation (SV) has started to take advantage of multiple external representations in programming [3, 5, 13, 14]. In SV, the program code is complemented by a representation (normally a graphical representation) that highlights important aspects of its execution. The hope of SV is that if students can *see* the internal workings of the program then this can serve as instruction, strengthening their mental representations and possibly fixing any misconceptions. However, there are two assumptions made by this approach: a) that programmers will indeed use (i.e. *see*) the visualisation and b) that they will be able to decode the information comprised in it.

It is not clear how many of these findings generalise across language paradigms. Research in the area of program comprehension, suggests, for example, that the mental representations of programmers are affected by notational properties of the language [16]. In the case of Java, for example, one infers the behaviour of a program from the code but a variety of language features (e.g. polymorphism and threads) make the balance between complex inference and surface analysis different compared to procedural languages such as Pascal.

1.3 Intelligent tutoring systems for debugging

Programming has provided a rich domain for Artificial Intelligence in Education systems and many of them have been developed to advise students about the bugs in their programs, either during program development [1] or post-hoc [9, 12]. Surprisingly few if any systems have been developed specifically to teach debugging, and environment builders have assumed that either the student will be taught these elsewhere or that they will be learnt piecemeal without explicit advice. Some progress has been made in Prolog [11, 12, 20, 23]. In [2] it is reported, to the best of our knowledge, the only attempt to build a tutoring system to teach debugging skills explicitly. Unfortunately, the scope of this project was quite modest and also specific to Prolog.

Our approach to designing a tutor for debugging focuses on both strategic and structural knowledge. These two aspects of programming knowledge can be considered as two sides of the same coin. Changes in the organisation of programmers' mental representations can promote changes in their program comprehension and debugging strategies [4, 16]. It seems reasonable to assume an inter-play between these two aspects of programming knowledge; for example, changes in the debugging environment that promote a change in strategy might produce modifications in the organisation of the programmers' mental representations.

Multi-representational debugging environments seem specially suitable to explore this approach. They normally include program visualisations which might help to promote a multi-

faceted mental representation of the program. On the other hand, they provide functionality intended to support the implementation of sound debugging strategies. Unfortunately, most studies about program comprehension and debugging have not employed these sorts of environments. We started by performing empirical studies in a multi-representational debugging environment and then used the results as desiderata towards the design of a tutor for general debugging skills in Java. A brief description of the method and results of these studies follows.

2. Debugging strategies in Java

The aim of the experiments reported here was to relate debugging behaviour to accuracy. In the first experiment, the focus was on relating the use of the representations provided to debugging accuracy while in the second one the focus was on relating the use of the debugging step facility as well as the available representations to debugging performance. The first experiment was performed with a static version of the SDE while the second employed a dynamic version of this tool. The following sections describe this tool and the two empirical studies.

2.1 The software development environment

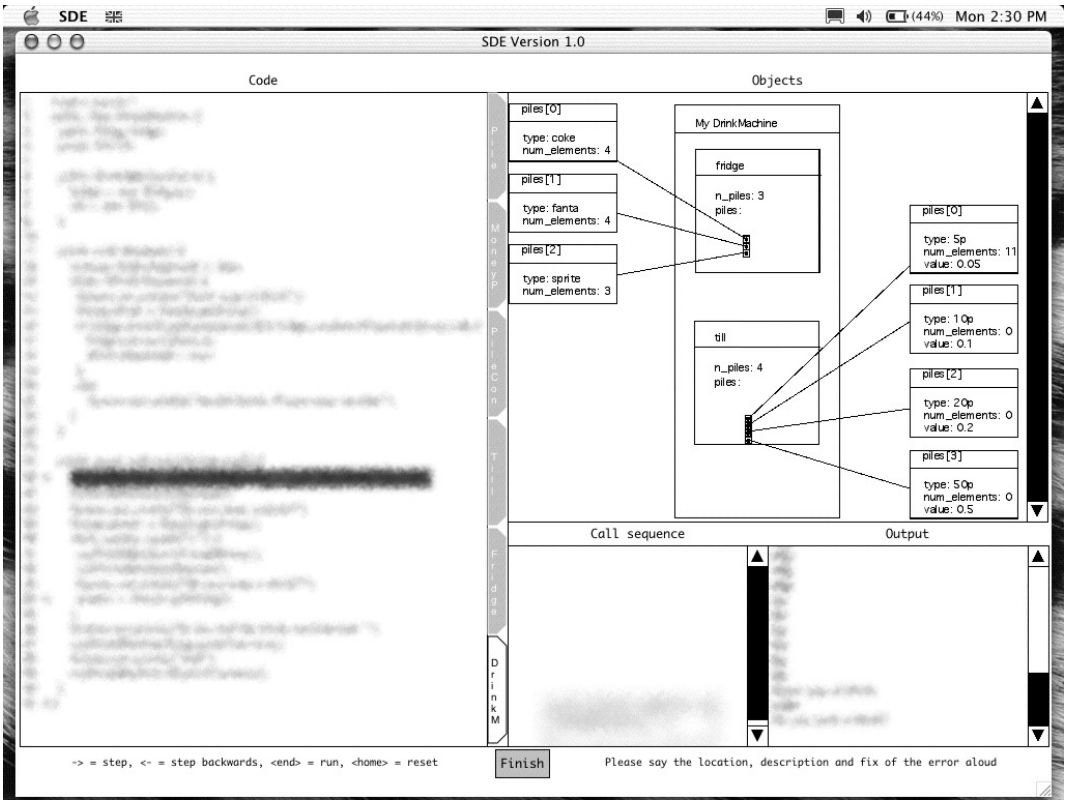

Figure 1. The debugging environment used by participants.

In both experiments participants tried to find errors in Java programs using a multi-representational SDE, see Figure 1. This tool enabled them to see the program code, visu-

alisations of it and its output. It also allowed visual attention to be tracked by presenting visual stimuli in a blurred form. When the user clicks on any part of the screen, a section of it around the mouse pointer becomes focused (see top right portion of Figure 1). In this way, the program restricts how much of a stimulus can be seen clearly 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.

2.2 Using a static set of representations

In the first experiment the SDE enabled participants to see the program code, its output for a sample execution, and a visualisation of this execution which presented information in either graphical or textual form. Forty nine computer science undergraduate students from the School of Cognitive and Computing Sciences at Sussex University, U.K. participated in this experiment. They debugged several Java programs presented to them in the SDE. This tool recorded their computer interactions (mouse movements and keyboard actions) as well as their verbalisations. For a full description of the method of this experiment see [19].

Experiment participants were categorised post-hoc into less and more experienced in terms of their background in programming. More experienced participants were more successful than the less experienced participants in detecting errors in the programs. Also, it seemed that graphical visualisations were more helpful when programmers faced a debugging task with an intermediate degree of difficulty.

Additionally, the results of this experiment link 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.

A qualitative analysis of two contrasting participants of this experiment [17] suggested that debugging sessions start with an initial program comprehension episode which varied in length. Sometimes this comprehension episode was relatively short, while at other times it extended to cover almost all the debugging session. Also, this qualitative analysis detected two different strategies to locate bugs: by spotting something odd in the program code and by comparing information from the different external representations available.

The first strategy to spot errors is deployed within the initial program comprehension episode. Occasionally participants would discover a suspicious piece of code while at this initial comprehension episode. Sometimes this discovery would prompt participants to report this piece of code as containing an error.

After these initial code browsing episodes, participants would sometimes engage in a coordination of representations episode. These were characterised by frequent switches between the code and one of the other two representations. It seems that participants were trying to build a more robust understanding of the program by integrating information from different external representations, reasoning backwards and forwards from the code to the visualisations and output, combining the hand simulation and causal reasoning debugging strategies [10]. This was the second strategy to detect errors. Sometimes these episodes involving coordination of the representations would help participants to discover and report an error.

These strategies may be linked to cognitive characteristics of the programmer such as

level of programming skill and display modality preference. It seems, for example, that programmers with a high level of verbal ability might prefer to work in a uni-modal, textual environment and therefore to concentrate on the code for most of the debugging session. However, it also seems that this error finding strategy is not as effective as using and trying to coordinate the several representations available.

2.3 Using a dynamic set of representations

In the second experiment the SDE enabled participants to view the 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.

This time forty two computer science undergraduate students from the School of Cognitive and Computing Sciences at Sussex University, U.K. participated in this experiment. They debugged several Java programs presented to them in the SDE which recorded their computer interaction as well as their verbalisations. For a full description of the method of this experiment see [18].

The experimental settings for this experiment were similar to those of the static SDE experiment; however, one important difference is that in one part of this experiment participants performed a debugging session with the visualisation windows empty. This experimental condition was included to explore whether these visualisations were indeed helpful to programmers and therefore influenced their debugging performance.

The full analysis of this experiment is not yet complete but a preliminary analysis for a subgroup of the participants has shown that participants who were successful at spotting bugs rely more than less successful ones on the information provided by the visualisation representations to perform the debugging task. The empty visualisations control condition decreased the debugging accuracy of successful participants while it increased the accuracy of less successful ones.

These results also show that there are different patterns of representation and execution control use for programmers with different levels of debugging accuracy. In general, successful participants made use of the two supporting visualisations while less successful ones concentrated too much on the code. Also, successful participants executed the program in steps while less successful ones executed it in one go. By executing the program in steps, successful participants were able to detect the moment in the execution were the bug manifested itself by causing inconsistent values to appear in one of the supporting visualisations. Participants who were able to detect these critical moments and visualisations were also able to decode and make use of the information in these visualisations to spot the program errors.

In summarising the results of these two experiments it seems that good debugging performance is associated with taking advantage of the resources available in the debugging environment. For modern, multi-representational debugging environments, two important resources are the supporting visualisations and the facility to execute the program in steps. A crucial skill in taking advantage of these resources is the ability to decode the information comprised in the supporting visualisations. This skill might be fostered by providing representations with a higher degree of dynamic linking as well as by explicit instruction about the representation formalism employed in the program visualisations.

Although individual differences and cognitive preferences might be behind the decision

to adopt one debugging strategy instead of another, it seems that there are some strategies which are more effective than others. It is an open issue whether enriching the problem solving resources of students by teaching specific strategies might improve their debugging performance.

Many further questions suggest themselves. For example: is it possible to get any measure of the distance between symptom and bug and how this affects the success of various strategies? Are there further dimensions (e.g. learning style) that affect a learner's preference for individual or particular mixes of representation?

3. Towards a Debugging Tutor for Java

In teaching debugging we can distinguish broadly between general debugging heuristics and program-specific interventions. The general heuristics include sensible use of debugger step facility, choice of appropriate representations and sensible search space reduction heuristics. While program-specific tactics include noticing particular discrepant values in output or the exact nature of an error message. There is an area of overlap between the general and the program-specific, in that each program-specific intervention can be generalised.

In thinking about the design of a debugging tutor, we concentrate on general heuristics, and on the best use of representations. Just in the same way that our experimental SDE could monitor which window the user has in focus, so could a potential debugging tutor. It could embody a number of monitoring rules that kept dynamic track of both focus of attention and switching behaviour to guide the student to pay attention in more sensible places. In this way one could imagine a tutor for debugging that was trying to teach about the process of debugging in general through guidance on how to debug a particular program.

Just as it is difficult from an experimental point of view to determine exactly why a particular student is paying more attention to a particular representation rather than another, so it is also difficult from the point of view of a tutoring system. It may well need to employ specific diagnostic tests or indeed ask the student, having detected an 'unbalanced' pattern of representation, whether this use reflects:

1. A lack of familiarity with the representational forms on offer, in which case offering exercise in the interpretation of the representations might be the best strategy, or

2. The student is attempting to minimise cognitive load just because s/he is a novice, in which case gentle encouragement and patience might be the best strategy, or

3. The student is familiar with the representations and is experienced, but is a 'unimodal' person who prefers code and output only rather than a 'transmodal' person who likes to switch around a lot. In this case the best strategy depends on higher educational objectives to do with whether the system should attempt to move the student towards a more versatile transmodal interaction, or simply support them in their unimodal interaction.

Depending on the system's decision, windows could be offered/withdrawn and the SDE could be dynamically re-configured. Moreover familiarity with the visualisations could also be improved by enhancing the dynalinks between the available representations. With more analysis, our existing dynamic experimental data (pretest and process data) may allow us to identify people as falling into the three categories above to some extent and, with luck, there

may be detectable differences in SDE usage pattern associated with these underlying factors which might be exploitable in a tutor.

In terms of program-specific interventions, the tutor could also pay attention to stepping behaviour. If the student is spending too long in the first breakpoint or/and if she is executing the program in one go she is probably missing the critical breakpoint and visualisation. If the tutor knows what is the error it could even, if this is necessary, lead the student to discover this critical breakpoint and visualisation.

4. Conclusion

This paper discusses a possible design for a tutor for debugging. This design is based on the experimental results of two debugging empirical studies which are briefly reported here.

These studies suggest that good debugging performance is associated with taking advantage of the resources available in the debugging environment. For modern, multi-representational debugging environments, two important resources are the supporting visualisations and the facility to execute the program in steps. A crucial skill in taking advantage of these resources is the ability to decode the information comprised in the supporting visualisations.

These experimental results have been used as desiderata towards the design of a tutor for general debugging skills in Java that pays particular attention to representation use.

Acknowledgements

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

References

[1] J. R. Anderson and B. J. Reiser. The LISP tutor. *Byte*, 10:159–175, 1985.

[2] P. Brna, E. R. Hernandez, and H. Pain. Learning prolog debugging skills. In P. Brna, B. du Boulay, and H. Pain, editors, *Learning to build and comprehend complex information structures: Prolog as a case study*, pages 381–395. Ablex, Stamford, Connecticut, 1999.

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

[4] S. P. Davies. Knowledge restructuring and the acquisition of programming expertise. *International Journal of Human Computer Studies*, 40:703–726, 1994.

[5] M. Eisenstadt, B. A. Price, and J. Domingue. Software visualization as a pedagogical tool. *Instructional Science*, 21:335–365, 1994.

[6] J. Gould. Some psychological evidence on how people debug computer programs. *International Journal of Man-Machine Studies*, 8:151–182, 1975.

[7] L. Gugerty and G. Olson. Comprehension differences in debugging by skilled and novice programmers. In E. Soloway and S. Iyengar, editors, *Empirical Studies of Programmers, first workshop*, pages 13–27, Norwood, New Jersey, 1986. Ablex.

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

[9] W. L. Johnson and E. Soloway. Intention-based diagnosis of programming errors. In *Proceedings of the National Conference on Artificial Intelligence*, pages 162–168. AAAI Press, 1984.

[10] I. Katz and J. R. Anderson. Debugging: an analysis of bug location strategies. *Human-Computer Interaction*, 3:359–399, 1988.

[11] M. Lee. An expert system for debugging Prolog programs. *Journal of Artificial Intelligence in Education*, 2(4):91–101, 1991.

[12] C.-K. Looi. Analysing novices' programs in a prolog intelligent teaching system. In B. Radig and Y. Kodratoff, editors, *Proceedings of the 8th European Conference on Artificial Intelligence (ECAI88)*, pages 314–319. Munich, Germany, 1988.

[13] 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.

[14] 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.

[15] N. Pennington. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology*, 19:295–341, 1987.

[16] P. Romero. Focal structures and information types in Prolog. *International Journal of Human Computer Studies*, 54:211–236, 2001.

[17] P. Romero, B. du Boulay, R. Cox, and R. Lutz. Java debugging strategies in multi-representational environments. In M. Petre, editor, *Psychology of Programming Interest Group 15th Workshop*, 2003.

[18] P. Romero, B. du Boulay, R. Lutz, and R. Cox. Strategy deployment in java debugging environments. In M. Burnett and J. Grundy, editors, *Submitted to 2003 IEEE Symposia on Human Centric Computing Languages and Environments*. IEEE press, Auckland, New Zealand, 2003.

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

[20] E. Y. Shapiro. *Algorithmic Program Debugging*. The MIT Press, Cambridge, Massachusetts, 1980.

[21] I. Vessey. Expertise in debugging computer programs: a process analysis. *International Journal of Man-Machine Studies*, 23:459–494, 1985.

[22] M. Weiser. Program slicing. In *Proceedings of the Fifth International Conference on Software Engineering*, pages 439–449, 1981.

[23] S. Yan. Declarative testing-debugging as tutoring. *Journal of Artificial Intelligence in Education*, 2(4):71–79, 1991.