Dynamic rich-data capture and analysis of debugging processes

Pablo Romero, Benedict du Boulay, Richard Cox,
Rudi Lutz and Sallyann Bryant
Human-Centred Technology Group
Sussex University, U.K.
email: pablor@sussex.ac.uk

Abstract

This paper proposes a methodology for the study of program comprehension and debugging through the capture and analysis of rich process data. A software debugging environment with enhanced functionality is used to capture these data and a mixture of qualitative and quantitative approaches is employed to analyse them.

The functionality added to the software debugging environment allows it to record the programmers' verbalisations, their focus of visual attention and their keyboard and mouse actions. These synchronous data are analysed to build a model that relates debugging expertise to strategy in terms of representation coordination and individual differences in representation use.

Keywords: program debugging, rich data analysis, external representations.

1 Introduction

Program debugging, the ability to detect errors in computer programs, is a skill that is central to programming. This is a complex skill that comprises comprehension of the program, troubleshooting ability and knowledge of the computerised working environment in which this activity takes place, among other skills. Program debugging is a particularly important skill for programming students to acquire, as they typically spend long periods of time trying to detect errors in their programs. Despite this, debugging is rarely taught explicitly as part of computer science courses.

Debugging is typically performed with the help of a computerised environment, a Software Debugging Environment (SDE). Modern SDEs provide the user with a highly interactive, multi-representational interface. Also, it is not uncommon for programmers to verbalise their cognitive processes while debugging as they usually do so when seeking help.

When working in a SDE, the range of data that can, in principle, be captured

includes i) use of the environment (navigation, ease of locating information, patterns of display-use, etc); ii) debugging performance (number of errors found and spotting times); and iii) verbalisations of programmers.

These data can be analysed at various levels of granularity, from keystroke and mouse clicks to the construction of debugging episodes and strategies. We are particularly interested in relating these data to debugging expertise, coordination of multiple representations and individual differences in representation use with a view to understanding the nature of the debugging process and helping students to acquire this skill in a more systematic way (Romero et al., 2002b; Romero et al., 2003b; du Boulay et al., 2003).

Rich process data capture and its analysis has proved very useful for evaluating virtual museum applications (Cox et al., 1999), investigating individual differences in logic proof development (Stenning et al., 1995) and for studying health science students' diagnostic reasoning skill acquisition (Cox and Lum, 2004) among other areas. Because of the characteristics mentioned above, we believe that rich data capture and analysis is a suitable methodology for an in depth study of the program debugging activity.

In this paper we propose a methodology for the study of program debugging through the use of a SDE equipped with the appropriate functionality to record the participants' verbalisations, focus of visual attention and keyboard and mouse actions. In addition to the capture of these data, this methodology includes a protocol for analysing synchronic events and relating them to debugging experience, representation coordination and individual differences in representation use.

2 The components of the debugging task

Program debugging comprises several subskills. These include program comprehension, troubleshooting, representation decoding and coordination, as well as knowledge about the SDE with which the programmer is working.

Program comprehension involves the understanding of several domains and the acknowledgment and understanding of various types of information implicit in a program. During domain understanding, mappings are established between the problem domain (the real world problem to be solved) and the program domain (the code that implements the solution to the problem) possibly via a number of intermediate domains. These mappings are produced via the generation and refining of hypotheses about the program's execution and its relation to the other domains (Brooks, 1983).

The programming information types that need to be understood during program comprehension indicate the different ways in which a program can be interpreted. Examples include function, data structure, data-flow and control-flow. Function refers to what the program does, data structure to the programming language objects used, data-flow to the transformations which data elements undergo and control-flow to the sequence of actions that will occur when the program is executed (Pennington, 1987b).

The result of the program comprehension process is a *mental representation* of the program being studied. The qualities and level of detail of this mental representation may vary according to a number of factors including the programmer's skill level and background knowledge, the size of the program and the task at hand.

Troubleshooting is the generic cognitive process by which people diagnose and correct faults in computer systems (Rouse et al., 1980). The majority of studies in the area have considered debugging as a troubleshooting activity and debugging strategies as specific cases of general troubleshooting strategies. Particular debugging strategies include information gathering (Jeffries, 1982; Eisenstadt et al., 1993), symptomatic search (Eisenstadt et al., 1993) and controlled experiments (Eisenstadt et al., 1993). Information gathering is reported to comprise a variety of techniques aimed at better understanding the error through the program behaviour. Symptomatic search (also called *expert recognised cliche*) is the search for 'typical' errors, which experts can easily detect on reading the program. Controlled experiments are performed once the programmer has a number of hypotheses about the error and wants to test their validity.

Despite the fact that programmers normally work with computerised multi-representational environments, there is little research about representation decoding and coordination in programming. SDEs frequently offer several other views of the program in addition to program code listing. Such environments might include data-flow and control-flow visualisations and output displays and typically permit the user to switch rapidly between these multiple, linked, concurrently displayed views (Romero et al., 2003a).

Issues to consider about representation decoding and coordination are the format of the representations (graphical or textual), information types highlighted and the programmers' level of familiarity with relevant representation formalisms (Romero et al., 2002b). Important questions to address in this area include:

- Are particular patterns of representational use associated with superior debugging performance?
- Do representation characteristics such as the information type highlighted or its format (graphical or textual) affect representation use and debugging strategy employed?
- Is there a relationship between programmers' characteristics such as their level of familiarity with representation formalisms, format preference and programming experience and their debugging behaviour?

In order to answer these kinds of question it is necessary to look at both the outcome and the process of the debugging activity and analyse the experimental data in both a quantitative and a qualitative way.

3 Analysis of process data in programming

Program comprehension and debugging studies that have looked at programmers' verbalisations have normally asked participants to *think aloud*, either while performing the task on their own (Vessey, 1985; Pennington, 1987a; Bergantz and Hassell, 1991) or collaboratively (Mulholland, 1997). Such studies have then performed protocol analysis to explore strategy and to investigate how it relates to programming experience and proficiency (Vessey, 1985); to explore the relationship between notational properties of the language and the SDE to the information types programmers consider as important (Bergantz and Hassell, 1991; Mulholland, 1997); and to study program comprehension strategy in terms of the mappings programmers establish between the program and problem domains (Pennington, 1987a).

Studies that have looked at focus of visual attention have been rare. These studies have either restricted the focus of visual attention by limiting the size of the editing buffer (Robertson et al., 1990), or employed eye-tracking devices (Crosby and Stelovsky, 1989). Such studies have analysed code reading patterns to investigate whether they are more similar to prose reading or to tasks related to problem solving (Robertson et al., 1990) and also to investigate the relationship between focusing on critical areas of the code and the participant's characteristics (programming experience and cognitive style) (Crosby and Stelovsky, 1989).

4 The Restricted Focus Viewer technology

In this paper we describe a methodology for the capture and analysis of rich process data in program debugging. The technology employed for the capture of this data comprises a SDE with additional functionality that allows the recording of programmers' interactions, verbalisations and focus of visual attention in real time. The SDE employed allows focus of visual attention to be tracked by blurring all but one of the windows in the application and allowing the section of that window around the mouse pointer to be focused. In this way, the SDE restricts how much of a stimulus can be seen clearly and it allows visual attention to be tracked as the user moves an unblurred 'foveal' area around the screen (see Figure 1). Additionally, the SDE records all mouse and keyboard actions as well as programmers' verbalisations. The mouse and keyboard events are recorded in a log file and the programmers' verbalisations are digitally recorded onto the computer's hard disk. In this way, the programmer's working sessions with this environment can be replayed for later analysis and the data recorded can be analysed in a synchronous way [1].

Besides these characteristics, the SDE employed provides some of the usual functionality present in debugging environments. The SDE enables programmers to view the execution of a Java program and presents, in addition to the code, its output and two visualisations of its execution.

[1] A Quicktime movie file containing a fraction of a debugging session can be found at *http://www.cogs.susx.ac.uk/projects/crusade/clips/subj26.mov*

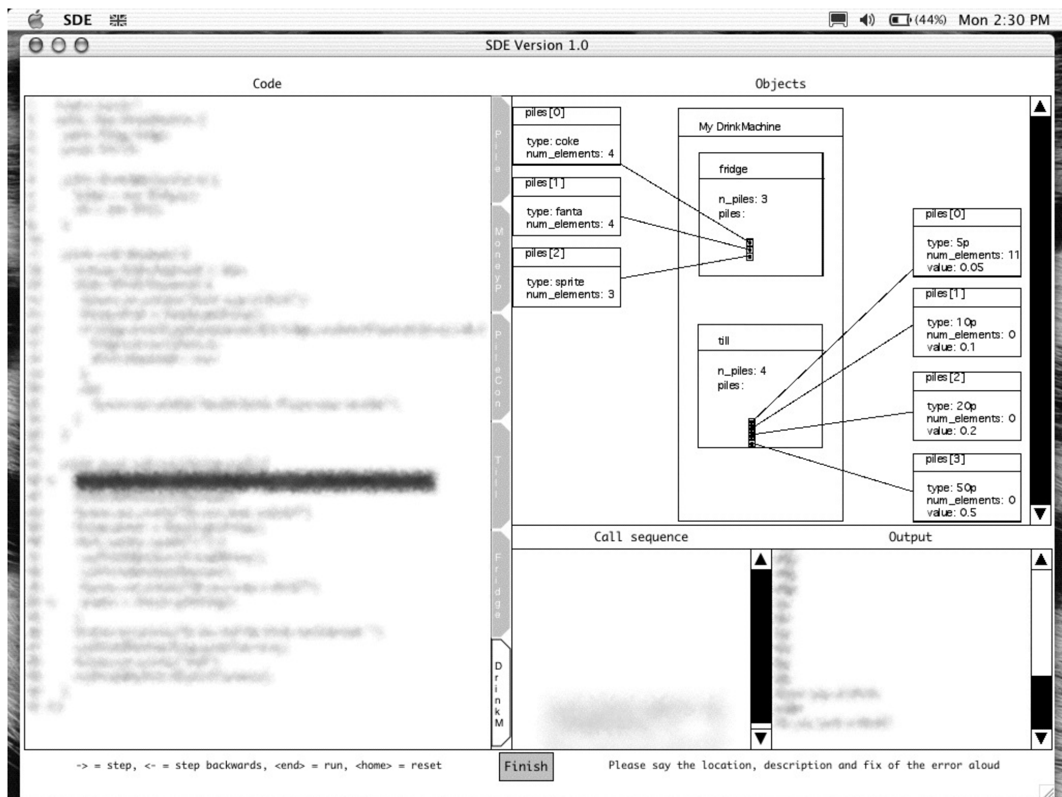

Figure 1: The SDE employed for rich data capturing

Participants are able to view stages of the execution of the program stepping between predefined *breakpoints* for a specific sample input. The SDE employed does not provide editing capabilities nor compilation and interpretation of programs. This is mainly because we are interested in error identification rather than error correction.

4.1 The Software Debugging Environment

The SDE was implemented as a modified version of the Restricted Focus Viewer (RFV) (Blackwell et al., 2000). The RFV implements the original functionality for the blurring and focusing of stimuli. This functionality was modified in several ways in the SDE. First, the stimulus images can be presented in a scroll or a tab pane. This enables us to present big images or more than one image in a specified display area. Second, in the RFV original implementation, the focused ('foveal') spot follows the movement of the mouse but in the SDE implementation users have to 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 windows 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 enabled 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.

A third functionality modification was that the stimuli images in the original RFV are static while in the SDE they are dynamic in the sense that the images in the SDE windows change as the execution of the program unfolds (the SDE allows participants to view the execution of programs in steps).

The RFV technology has been validated in the context of reasoning about simple mechanical systems via the inspection of static diagrams (Blackwell et al., 2000) and for Java program debugging (Romero et al., 2002a). In the first case, no significant differences in inspection strategies were found when working with this technology compared to eye-tracker data. When validated for Java debugging, although the RFV technology was found to slow down performance, there was no evidence to suggest any decrease in accuracy or in the qualitative nature of the debugging behaviour.

The SDE records programmers' verbalisations digitally. In our empirical studies (Romero et al., 2002a; Romero et al., 2002b; Romero et al., 2003c), participants were asked to *think aloud* while they performed a debugging task on their own. This instruction normally prompts participants to verbalise information that they are attending to in short memory. It has been found that verbalisations of this nature, although compact and incorporating many idiosyncratic referents, do not change the course and structure of participants' cognitive processes (Ericsson and Simon, 1980).

5 Rich data analysis

Capturing rich data is a technical achievement but this effort can only pay off if there is a sensible framework for the analysis of such data. The framework that we propose analyses these data both quantitatively and qualitatively. The quantitative analysis can relate programmers performance to SDE use while the qualitative analysis can explore programmers' behaviour and strategy by taking into account three sources of data synchronically: focus of attention trace, low level events (mouse and keyboard actions) and verbalisation data.

5.1 Quantitative analysis

The quantitative analysis requires programmer's performance to be extracted from the verbalisation data and the log file of keyboard and mouse actions to be transformed into a description of target program execution and window fixations and switches. The latter task can be automated (through a computer program that performs this transformation) but the former needs to be performed by a human rater.

The description of target program execution is related to how participants decide to view the execution of the program to be debugged. They can view the execution in steps, moving between predefined points (*breakpoints*) in the program. As the execution of programs is sequential, programmers can move forward to the next breakpoint, backwards to the previous breakpoint, forward to the end of the execution, or backwards to the beginning of it. One question regarding how execution is viewed is whether there is a relationship between this pattern of use and debugging performance.

Window fixation refers to the total time participants spent focusing on each window of the SDE, while window switches between the available representations refers to the number of changes of focus between these representations. Relating these two measurements to debugging performance enables us to investigate whether there are patterns of representation use associated with superior debugging accuracy.

The content and format of the visualisations presented by the environment can be manipulated. For example, the current version of the SDE presents either graphical or textual visualisations that highlight data structure or control-flow information. These experimental conditions, together with the variables mentioned before (patterns of program execution, representation use and debugging accuracy) can be analysed looking for significant main and interaction effects.

Finding out about these relationships offers important information about patterns of environment usage and programming expertise. However, this information can be complemented by investigating the debugging strategies that shaped this environment usage.

5.2 Qualitative analysis

The qualitative analysis takes into account three sources of synchronous data: focal attention trace, patterns of movement between breakpoints and programmers' verbalisations. As the SDE supports the replay of programmers' debugging sessions, a rater can execute these replays to extract the desired information. Table 1 presents part of a sample coding sheet. In this section of coding sheet there are six columns, the first one is for the event number, the second one is for the programmer's utterances, and the third, fourth, fifth and sixth are for the different windows of the SDE (the code, objects, call sequence and output windows as shown in Figure 1). Each row of this table codes one debugging event. In general, debugging events are programmers' verbalisations (utterances), inter-window switches of visual attention focus or breakpoint switches. The unit of verbalisation that we considered as an utterance is that verbalisation which is limited by a considerable pause and/or by a change of topic. Inter-window switches occur when the user moves the unblurred area from one window to another and breakpoint switches take place when the programmer moves the state of the program execution from one breakpoint to another. In the example in Table 1, most debugging events are programmer's utterances except for the last one which is a window switch.

	Verbalisation	Focus of attention			
		Code	Objects	Call Sequence	Output
8	Enter type of drink. Fanta	DrinkMachine.main (line 41)			
9	Let's have a look at it again	DrinkMachine.main (line 34)			
10	Ah! Interesting		piles[0] to piles[3]		
11	In the object window, it's interesting to see				Enter type of drink. Coke - Now enter the number of fantas. 4
12			piles[0] to piles[3]		

Table 1: Section of coding sheet for a specific debugging session

An interesting question that can be addressed taking into account these sources of data is whether there are any relationships between programming experience and debugging behaviour. In other words, are there specific debugging strategies that categorise successful and unsuccessful debugging performance?

5.3 Program comprehension and debugging behaviour coding categories

The literature on programming already suggests a classification of successful and unsuccessful program comprehension and debugging strategies (Vessey, 1985; Pennington, 1987a; Davies, 1994; Pennington et al., 1995; Détienne, 1997). We took this suggested classification as the basis for developing a list of program comprehension and debugging behaviour coding categories. This list of behaviour coding categories is presented in Table 2. This table is divided into program comprehension and program debugging coding categories, and within each of these, there are instances of successful and unsuccessful coding categories. For example, coding category *C8* is an instance of a successful program comprehension coding category because it is intended to register the occurrence of a strategy that tries to link the program domain and the problem domain. Such cross-referencing strategy has been associated to good debugging performance (Pennington, 1987a). On the other hand program debugging coding category *D13* is intended to register the occurrence of a strategy which, for example, tries to understand specific details of the program without having a clear idea of what the effects of the error are in terms of the

Comprehension coding categories	
C1	Utterances reflect the stage of program execution
C2	Use of breakpoints
C3	Comments relating the information types
C4	Switching between information types a minimum of twice (A to B then back to A)
C5	Utterances regarding hypothesis followed by switching from code to other type of representation
C6	Utterances regarding hypothesis followed by switching to code from other type of representation
C7	(Single) stepping through the code carefully watching the objects view
C8	Whilst looking at code or object view, utterances reflect *real world* objects in the problem domain
C9	Looking at the output or object view whilst talking about the code
C10	Utterances relating to higher level entities (e.g. method, sub-routine, section of code)
C11	Returning to the same line of code from another type of representation several times to understand all its implications
C12	Syntax verbalization
C13	Explaining the code to themselves
C14	Reading the code out loud from top to bottom
C15	Lack of switching between views (especially the code view)
C16	Relating only to *real world* objects and only looking at the output
C17	Erratic jumping around within the code
C18	Erratic jumping across information types
C19	Repeatedly examining stereotypical lines of code
C20	Finding a piece of code to account for the output
C21	Searching for a line of code
C22	Paraphrasing as a re-representation.
Debugging coding categories	
D1	(Single) stepping through the code carefully watching the objects view
D2	Considering negative evidence in their reasoning
D3	Focusing in on an area of code after an uttered hypothesis
D4	Re-running the code with fix in place
D5	Temporarily considering regions of the program as free of errors
D6	Attempting an a priory classification of errors and acting accordingly
D7	Higher level code browsing to build up a complete picture before testing hypothesis
D8	Being clear that something is an hypothesis
D9	Comparison of actual with expected outcome. Early comments suggesting potential causes
D10	Running the whole program again (including the previously commented out parts) or browsing previously discounted code
D11	Talk in terms of breakpoints (dynamic view) but not stepping through
D12	Utterances of code cliches
D13	Early delving into the details

Table 2: Comprehension and debugging behaviour coding categories

output. Such early commitment to analysing details is unlikely to produce good performance results.

These coding categories were developed iteratively by the authors on the basis of a subsample of protocols. The main aspects that required tuning were the granularity of the debugging event, comprehensiveness and disambiguation. As mentioned in Section 5.2, a debugging event can be an utterance, an inter-window switch of visual attention focus or a breakpoint switch. The coding categories are comprehensive, in that, for example, there are some coding categories that will never be observed when analysing the data but that were nevertheless included for the sake of completeness (see Section 5.4 for an example of this). Finally the meaning of several categories had to be carefully disambiguated. For example, while both coding categories *C1* and C2 refer to looking at the program in a dynamic way, *C1* focuses on whether programmers conceptualise the program as a dynamic entity, regardless of their use of breakpoint switching facilities. Coding category *C2*, on the other hand, is directly associated to breakpoint switching.

5.4 Data encoding

To obtain a detailed characterisation of programmers' program comprehension and debugging strategies, their coding sheets are analysed to look for occurrences of the behaviour coding categories in Table 2. This is a procedure that has to be performed by a human rater and requires the rater to consider information about focus of attention, program execution and programmers' verbalisations synchronously. For example, clue *C7* requires verification of whether simple stepping (a program execution behaviour) happened while the focus of visual attention was located in a specific window (the objects view). Verbalisations referring to executing the program in steps and/or to the contents of the objects window can strengthen the case for acknowledging the occurrence of this behaviour.

Due to the characteristics of our SDE, some of the coding categories will never be observed when analysing these data but they were nevertheless included for the sake of completeness. For example, the behaviour associated to coding category *D4*, *Re-running the code with fix in place*, cannot occur as it was mentioned in Section 4 that the SDE employed does not allow the code to be edited. This coding category, however, will be useful for future research.

These low level behaviour coding categories can be integrated into program comprehension and debugging episodes (Vessey, 1985). These episodes are groups of behaviours which have a specific goal in common and which can be used to identify programmers' strategies.

A cluster analysis can allow us to categorise groups of programmers according to their displayed strategies and to compare this categorisation with their performance data. This categorisation can also be complemented with the findings of the quantitative analysis. In this way, a model of program comprehension and debugging expertise in terms of behaviour and strategy can be empirically derived.

This way of deriving a program comprehension and debugging model, taking into account several sources of data synchronously, has advantages over models that have considered only one source of information. First, the range of behaviours, and therefore strategies, taken into account in the model can be wider. For example, behaviours *C4*, *C11* and *C19* would be difficult to take into account in a model that only considers verbal data. Also, having several sources of data enables the encoding process to have a higher level of certainty (as in the example about clue *C7* above).

Considering a wide range of strategies in a program comprehension and debugging model might increase the usefulness of the model. For example, if the model is going to be applied to the design of learning environments for programming (du Boulay et al., 2003), taking into account strategies that have to do with focus of visual attention can enable the environment, in principle, to give advice on these matters. The learning environment could, for example, 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.

6 Conclusions and further work

This paper proposes a methodology for the study of program comprehension and debugging through the capture and analysis of rich data. The data is captured by a software debugging environment equipped with the appropriate functionality to record the participants' verbalisations, visual attention focus and keyboard and mouse actions.

Both qualitative and quantitative data analyses are undertaken. The qualitative analysis takes into account the three sources of data mentioned above in a synchronous way.

We believe the models of programming derived in this way can be more complete and have a higher level of accuracy. One possible avenue of further work is to extend the proposed methodology to cover further programming tasks such as code generation and program re-use, or even activities in other areas like, for example, learning through the use of interactive learning environments.

Acknowledgments

This work is supported by the EPSRC grant GR/N64199. The support for Richard Cox of the Leverhulme Foundation (Leverhulme Trust Fellowship G/2/RFG/2001/0117) and the British Academy is gratefully acknowledged.

References

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

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

Cox, R. and Lum, C. (2004). Case-based teaching and clinical reasoning: Seeing how students think with patsy. In Brumfitt, S., editor, *Innovations in professional education for Speech and Language Therapists*. Whurr.

Cox, R., O'Donnell, M., and Oberlander, J. (1999). Dynamic versus static hypermedia in museum education: An evaluation of ilex, the intelligent labelling explorer. In Lajoie, S. P. and Vivet, M., editors, *Proceedings of the 9th Artificial Intelligence in Education (AI-ED99) Conference, Le Mans, France, July, 1999*, pages 181–188. IOS Press, Amsterdam, The Netherlands.

Crosby, M. and Stelovsky, J. (1989). Subject differences in the reading of computer algorithms. In Salvendy, G. and Smith, M. J., editors, *Designing and Using Human-Computer Interfaces and Knowledge Based Systems*, pages 137–144. Elsevier science publishers B.V., Amsterdam, The Netherlands.

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

Détienne, F. (1997). Assessing the cognitive consequences of the object-oriented approach: A survey of empirical research on object-oriented design by individuals and teams. *Interacting with Computers*, 9:47–72.

du Boulay, B., Romero, P., Cox, R., and Lutz, R. (2003). Towards a debugging tutor for object-oriented environments. In Aleven, V., Hoppe, U., Kay, J., Mizoguchi, R., Pain, H., Verdejo, F., and Yacef, K., editors, *Supplementary Proceedings of Artificial Intelligence in Education Conference (AIED2003), Sydney, Australia*, pages 399–407. University of Sydney.

Eisenstadt, M., Price, B. A., and Domingue, J. (1993). Software visualization as a pedagogical tool. *Instructional Science*, 21:335–365.

Ericsson, K. A. and Simon, H. A. (1980). Verbal reports as data. *Psychological Review*, 87(3):215–251.

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

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

Pennington, N. (1987a). Comprehension strategies in programming. In Olson, G. M., Sheppard, S., and Soloway, E., editors, *Empirical Studies of Programmers, second workshop*, pages 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.

Pennington, N., Lee, A. Y., and Rehder, B. (1995). Cognitive activities and levels of abstraction in procedural and object-oriented design. *Human-Computer Interaction*, 10:171–226.

Robertson, S. P., Davis, E. F., Okabe, K., and Fitz-Randolf, D. (1990). Program comprehension beyond the line. In Diaper, D., More, D., Cockton, G., and Shackel, B., editors, *Human-Computer Interaction - INTERACT '90*, pages 959–963.

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

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

Romero, P., du Boulay, B., Cox, R., and Lutz, R. (2003b). Java debugging strategies in multi-representational environments. In Petre, M., editor, *Psychology of Programming Interest Group 15th Workshop*, pages 421–434.

Romero, P., du Boulay, B., Lutz, R., and Cox, R. (2003c). The effects of graphical and textual visualisations in multi-representational debugging environments. In Hosking, J. and Cox, P., editors, *2003 IEEE Symposium on Human Centric Computing Languages and Environments*, pages 236–238. IEEE Computer Society, Auckland, New Zealand.

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

Rouse, W., Rouse, S., and Pellegrino, S. (1980). A rule based model of human problem solving performance in fault diagnosis tasks. *IEEE Transactions on Systems, Man and Cybernetics*, 7:366–376.

Stenning, K., Cox, R., and Oberlander, J. (1995). Contrasting the cognitive effects of graphical and sentential logic teaching: Reasoning, representation and individual differences. *Language and Cognitive Processes*, 10(3/4):333–354.

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