Computer Science Education
Vol. 18, No. 2, June 2008, 93–116

Routledge
Taylor & Francis Group

Debugging: finding, fixing and flailing, a multi-institutional study of novice debuggers

Sue Fitzgerald[a]*, Gary Lewandowski[b], Renée McCauley[c], Laurie Murphy[d], Beth Simon[e], Lynda Thomas[f] and Carol Zander[g]

[a]*Department of Information and Computer Sciences, Metropolitan State University, St. Paul, USA;* [b]*Department of Mathematics and Computer Science, Xavier University, Cincinnati, USA;* [c]*Department of Computer Science, College of Charleston, Charleston, USA;* [d]*Department of Computer Science and Computer Engineering, Pacific Lutheran University, Tacoma, USA;* [e]*Department of Computer Science and Engineering, University of California, San Diego, USA;* [f]*Department of Computer Science, Aberystwyth University, Aberystwyth, UK;* [g]*Department of Computing and Software Systems, University of Washington, Bothell, USA*

Debugging is often difficult and frustrating for novices. Yet because students typically debug outside the classroom and often in isolation, instructors rarely have the opportunity to closely observe students while they debug. This paper describes the details of an exploratory study of the debugging skills and behaviors of contemporary novice Java programmers. Based on a modified replication of Katz and Anderson's study of novices, we sought to broadly survey the modern landscape of novice debugging abilities. As such, this study reports general quantitative results and fills in the picture with qualitative detail from a relatively small, but varied sample. Comprehensive interviews involving both a programming and a debugging task, followed by a semi-structured interview and a questionnaire, were conducted with 21 CS2 students at seven colleges and universities. While many subjects successfully debugged a representative set of typical CS1 bugs, there was a great deal of variation in their success at the programming and debugging tasks. Most of the students who were good debuggers were good novice programmers, although not all of the good programmers were successful at debugging. Students employed a variety of strategies to find 70% of all bugs and of the bugs they found they were able to fix 97% of them. They had the most difficulty with malformed statements, such as arithmetic errors and incorrect loop conditions. Our results confirm many findings from previous studies (some quite old) – most notably that once students find bugs, they can fix them. However, the results also suggest that some changes have occurred in the student population, particularly an increased use of debugging tools and online resources, as well as the use of pattern matching, which has not previously been reported.

Keywords: debugging; novice programmers; debugging strategies; bugs; programming errors; CS1; pedagogy

1. Introduction

This study was motivated by our experiences teaching introductory programming courses, where we observed many beginners having difficulty learning to debug. We began our research with an extensive review of the literature investigating programmers' debugging skills (McCauley et al., 2008). Both our literature review and this study concentrated on

*Corresponding author. Email: sue.fitzgerald@metrostate.edu

ISSN 0899-3408 print/ISSN 1744-5175 online

DOI: 10.1080/08993400802114508
http://www.informaworld.com

problems experienced by novices as they learn to program, since that was most relevant to our perspective as computer science educators.

Much of the previous research on debugging dates back to the 1980s, and while some of those findings likely hold true today, the languages, environments, and even the students have certainly changed in over 20 years. To focus future research efforts and gain insight into the overall landscape of contemporary novice debugging we performed a multi-institutional, exploratory study of beginning programmers as they debugged typical novice errors. The study focused on the finding and fixing of bugs. We defined a bug as ''found'' when the student correctly indicated the line in which an error exists. ''Fixing'' a bug involved making the correct change, possibly after a series of incorrect changes, to modify that line to behave correctly in the context of the program.

While we gathered quantitative data, such as numbers of bugs found and fixed, we focused on the rich, qualitative data obtained from closely observing students as they debugged. Our goal was not to enable comparisons between institutions, but to describe the debugging abilities of novices as generally as possible – from a range of backgrounds and preparation, in various types of introductory courses employing multiple approaches, taught by different instructors at diverse institutions. The numbers reported here are intended for general descriptive purposes, but we feel their overall strength is augmented by the diverse learning contexts of the sample they represent.

Regardless of the context in which they learn to do it, debugging is often difficult for novices because it requires the use of many new skills simultaneously. Students must understand the operation of the intended program and the execution of the actual (buggy) program; have general programming expertise and an understanding of the programming language; comprehend the application domain; and have knowledge of bugs and debugging methods (Ducassé & Emde, 1988). Unfortunately, most introductory students' knowledge of debugging skills is fragile at best (Perkins & Martin, 1986), causing many to find debugging not only difficult but also frustrating. This suggests that investigations of novice debugging could lead to important insights for improving computer science instruction.

The difficulty of debugging is a common thread throughout the literature, much of which was reported in the mid to late 1980s. The wealth of early studies led us to wonder if those findings hold true today, when new paradigms and sophisticated development environments provide students with far different learning experiences than those of students a quarter of a century ago. Similarly, we wondered what impact contemporary students' extensive use of computers and technology might have on their abilities to program and debug.

Our goal was to examine the debugging skills and behaviors of introductory Java programmers, with an eye to helping the struggling students of the future become more effective debuggers. We revisited ground covered by a number of previous investigations building on and extending that previous work. Our general research questions included the following.

- How well can novices debug?
- What types of bugs are most difficult for them to find and to fix?
- What is the relationship between novices' programming and debugging abilities?
- Are novices' self-perceptions of ability in line with their performance on programming and debugging tasks?
- What strategies and behaviors do novices exhibit as they debug?

The details of this study are presented in the sections that follow. In section 2 we discuss the literature that informed our experimental design. In section 3 we detail the design of

our experiment. In section 4 we present the initial findings from our quantitative and qualitative analyses of the data. In section 5 we relate our results back to those in the literature and discuss the implications of our findings. In section 6 we suggest avenues for continued research.

2. Background

In this section we define "debugging" and discuss the specific investigations of novice debugging on which we based this study. We describe how the current study is situated within the context of recent debugging research.

Ko and Myers (2005) defined a bug as a combination of one or more errors in the code (software errors), which may produce errors in execution (runtime faults), which in turn may produce failures in program behavior (runtime failures). They defined debugging as "determining what runtime faults led to a runtime failure, determining what software errors were responsible for those runtime faults, and modifying the code to prevent the runtime faults from occurring" (p. 44). Although novice programmers experience difficulty with syntax errors as well, our study focuses primarily on this sort of runtime fault.

The current study was influenced by the work of Katz and Anderson (1987). In four interrelated studies of students' LISP debugging skills Katz and Anderson studied the debugging process as a particular case of troubleshooting. Troubleshooting consists of four stages: understanding the system, testing the system, locating the error, and repairing the error (and retesting). Katz and Anderson's studies provided strong evidence that for debugging these stages are separate and, in particular, that the skills required to understand the system are not necessarily connected to the skills required to locate the error. For their subjects locating a bug was more difficult than actually fixing it. Subjects took less time and were more successful when debugging their own code than when debugging code written by others. They also found that subjects applied different bug location strategies depending on whether they were debugging code written by themselves or by someone else. Subjects debugging their own code used a backward reasoning strategy by beginning from the symptom or output to find the source of the error. Those debugging someone else's code used a forward reasoning strategy, mentally executing the program as if they were the computer.

In the early to mid 1980s the Cognition and Programming Group at Yale University, under the direction of Elliot Soloway, studied introductory student programmers and programming – specifically they studied the programming processes used and the types of and the reasons for the bugs that the students generated. Johnson, Soloway, Cutler, and Draper (1983) categorized bugs produced by novice programmers that were non-syntactic in nature. Their work labeled bugs as one of the following: missing (a necessary operation is not present), spurious (an unnecessary operation is present), misplaced (a necessary operation is in the wrong place), or malformed (a necessary operation is incorrectly implemented). Spohrer and Soloway (1986) analyzed high frequency bugs in an attempt to understand what the student was (or was not) thinking at the time the bug was produced. They found that only a few types of bugs accounted for the majority of mistakes in students' programs. A common belief at the time was that most bugs were due to student misconceptions about language constructs. Spohrer and Soloway's findings, however, contradicted this belief. In fact, they found that the most commonly occurring bugs fell into several non-language-construct-related categories.

Perkins and Martin (1986) investigated the thinking of high school BASIC programmers and sought to understand why errors occurred. They observed that students' difficulty in programming, including finding and removing bugs, was related to

their fragile knowledge. Four types of fragile knowledge were observed: missing knowledge has not been acquired; inert knowledge is knowledge that the subject is unable to retrieve when needed; misplaced knowledge is knowledge that is used in the wrong context; conglomerated knowledge is a misuse of knowledge in which a programmer combines two or more known structures incorrectly.

While our current experiment looks specifically at novice debugging, some previous studies have investigated novice and expert differences. For example, Gugerty and Olson (1986), when comparing novice and expert programmers' ability to debug, observed that experts were able to debug more quickly and accurately, primarily because they generated higher quality hypotheses based on less study of the code. They also found that novices tended to introduce new bugs into the program while searching for the original one and that adding bugs was correlated with unsuccessful debugging. Vessey (1985) also investigated novice and expert differences. She observed her subjects in the process of debugging and identified the hierarchy of steps/goals they followed in debugging. She classified her subjects, all practicing COBOL programmers, as experts or novices based on their ability to chunk programs and debug quickly. She found that experts were able to chunk programs into meaningful pieces and take a "systems view" of programs, while novices could not do either of these. Nanja and Cook (1987) found that most novices did not correct all of the errors. Some novices made extensive modifications, including recoding and replacing sections of code, rather than fixing bugs. Similarly to Gugerty and Olson (1986), Nanja and Cook found that beginners introduced many new errors.

A more recent study of novice Java programmers by Ahmadzadeh, Elliman, and Higgins (2005) analyzed the source code and error messages of over 100,000 records of introductory programming student errors. This yielded insights into the patterns of compiler errors generated by those novice programmers. Using this data, a second phase of the study asked students to debug a program seeded with compilation and logic errors. This two phase design – programming followed by debugging – facilitated a comparison of good versus weak programmers and good versus weak debuggers, showing that a majority of the good debuggers (66%) were also good programmers, but that only 39% of good programmers were also good debuggers. None of the weak programmers were good debuggers. Weak debuggers applied debugging techniques inconsistently, depending on how familiar they were with the underlying program structure. They also appeared to lack understanding of the actual and intended program implementation. Ahmadzadeh, Elliman, and Higgins observed that after isolating a bug some students did not bother to understand and fix the error, but rather simply rewrote the code. Lastly, they found structural familiarity and understanding of the intended program to be particular advantages for good debuggers as they debugged unfamiliar code.

While few early studies focused on the difficulty or ease of fixing specific types of errors, in a study of expert FORTRAN programmers Gould (1975) found that errors in assignment statements were the most difficult to find, since locating them required a deeper understanding of the program than other types of seeded errors. Because he was working with experts, Gould focused on only three types of errors that he believed to be most common or most difficult to find. He seeded his programs with assignment errors (occurring in an assignment statement), iteration errors (in the control of a loop), and array errors (causing an array index range to be exceeded).

Other recent novice debugging work includes that of Hristova, Misra, Rutter, and Mercuri (2003), which identified instructor- and teaching assistant-reported novice Java syntax errors in an attempt to construct a tool to generate more meaningful error messages than those displayed by the Java compiler.

The intention of our study is to reflect and build upon this earlier literature, asking if these findings hold true in the current era.

3. Methodology

The current study examined the debugging skills of introductory programming students during their second Java programming course. Students from seven universities completed a programming exercise designed to facilitate problem contextualization followed by a debugging task on a buggy version of the same problem. A follow-up interview and survey were then used to assess students' impressions of the exercises and thoughts on debugging.

Subjects

Subjects were 21 undergraduate student volunteers from seven colleges and universities, six in the USA and one in the UK. Six subjects (29%) were female and 15 (71%) were male; ages ranged from 18 to 51, with a median age of 19. All had completed 15–20 weeks of university Java programming instruction. Typically subjects were in the early to middle phase of their second Java course at the time of the interview. Although these students were still novice programmers, they had completed at least one programming course and, therefore, were likely to have experienced debugging a variety of errors. It also seemed likely that they had been programming long enough to have established at least a modest repertoire of debugging techniques and strategies.

Students at five of the seven institutions were compensated for their participation in the study, the others were not. As discussed in more depth in Section 6, some of the researchers had directly instructed their subjects in either the first or second programming course.

For the purposes of anonymity each subject was assigned a reference number (e.g., D02) which uniquely identified the institution (letters A–G) and subject (01–05) at that university.

Protocol

One-on-one sessions of approximately 90 minutes in length were conducted with each subject. Interview sessions involved: (1) a background questionnaire, (2) a programming exercise, (3) a debugging exercise, (4) a semi-structured interview, and (5) a debugging survey. The design of each of these components is described below.

Background questionnaire

A background questionnaire was administered in order to determine whether our multi-institutional subjects were, in fact, novices. Subjects rated their experience with various programming languages on a scale of 1, indicating "never used," to 5 indicating "have used a lot." Subjects reported an average experience with Java of 3.48 (median 4, mode 4); all were in the 15–20th week of Java instruction.

None of the subjects had ever used C, Ada, or Scheme. Six claimed experience with Visual Basic or BASIC (with average experience ratings of 2), three had experience with C++ (with average experience ratings of 2.67), three had experience with PHP (rating 2), two with Python (experience 2.5), one had used Pascal and one FORTRAN (both with ratings of 2). See Table 1 for details. On average students reported familiarity with 1.9 languages.

Table 1. Self-reported skill level (1, never used; 5, used a lot).

Programming language	Number of subjects	Average self-rating
Java	21	3.48
VisualBasic or BASIC	6	2
C++	3	2.67
PHP	3	2
Python	2	2.5
Pascal	1	2
Perl	1	2
HTML	1	5
JCL	1	3
FORTRAN	1	2
C, Ada, Scheme	0	

On average the subjects rated themselves as more familiar with Java than any other language.

Programming exercise

The goal of this study was to investigate the debugging skills and habits of introductory programmers in a typical programming context. A reasonable comparison of the debugging abilities of a group of programmers requires them to debug the same or similar errors. Since encountering precisely the same errors is unlikely when subjects are simply asked to code the same programming problem, we wanted our subjects to debug identical buggy code written by someone else. This, too, was problematic, since the debugging process is intrinsically altered when it excludes the understanding and contextualization gained from writing a program for oneself. Adopting the troubleshooting stages outlined in Katz and Anderson (1987), we looked for ways to separate problem understanding from finding, fixing, and testing bugs.

We decided to address these concerns by giving students a programming exercise followed by a debugging task (described in detail below) where they were asked to debug an erroneous solution to the problem they had just attempted. This approach enabled subjects to become well acquainted with and form a mental model of the problem during the programming phase and also allowed us to compare subjects' abilities to debug identical errors seeded within our solution to the problem. Furthermore, this design more closely models the typical novice debugging experience, which is quite different from that of most professionals, who frequently debug code written by others. Katz and Anderson (1987) took a similar approach in one of their experiments, although their subjects debugged other student programs after attempting their own solutions.

Subjects with a range of abilities participated in the study, thus another design concern was the difficulty of the programming/debugging task. Assigning all subjects an identical task was likely to result in some being overwhelmed because the problem was too difficult or unfamiliar, while others were likely to be insufficiently challenged by tasks below their ability or because they had written nearly identical code in the past. We addressed this problem by allowing students to pick from six typical "first course" programming problems. Subjects were asked to choose a problem they thought

would not be too difficult or too easy for them to solve in approximately 20 minutes. This approach is similar to that used by Perkins and Martin (1986).

We recognize that offering many problem choices limited our ability to compare students programming and debugging precisely the same code. However, we also believe it improved the likelihood that students could solve the problems, later allowing them to concentrate on finding and fixing bugs rather than focusing on problem understanding during the debugging exercise. Furthermore, this added variety also served the descriptive goal of this study – especially when bugs of the same type were found in multiple contexts across programs. Nonetheless, in future work we would perhaps limit the options to three problems, for simplicity of analysis.

The problem descriptions were arranged from least to most difficult (in our opinion) and included the following.

- Rectangles – given two integers, n and m, output three $n \times m$ rectangles of asterisks, one solid, one hollow and one checkered.
- Game of craps – given basic craps rules and a simple die class, simulate and calculate statistics for 10,000 rounds of craps.
- Bike raffle – read bike, ticket and overhead costs and students' names and tickets sold and display basic statistics for an elementary school bike raffle.
- Triangle type – read three side lengths from the keyboard and determine the type of triangle they form.
- Binary search – given an integer value to search for (entered from the keyboard), implement a method to perform a binary search for the value in a sorted array of integers.
- Simple calculator – implement a very simple infix calculator that multiplies, adds and subtracts.

The problem descriptions provided to subjects were more detailed and included example input and output. Subjects were given a minimal skeleton program from which to begin coding and allowed to use the programming environment of their choice. In general these simple problems lent themselves more readily to procedural approaches than object-oriented approaches.

The subjects were permitted to access the Java online API documentation and to refer to their textbooks (cf. Gugerty & Olson, 1986), although they could not look up the specific solutions for any of the above problems, such as binary search. Subjects were allowed to use a debugger if it was available in their programming environment (cf. Nanja & Cook, 1987).

Each subject was given 30 minutes to code the exercise and was asked to think aloud while their comments were audio-taped and their actions recorded in a programming behavior log (see Figure 1).

A minute-by-minute record of each subject's behavior was made during the programming session. Researchers noted times at which subjects wrote on scratch paper, designating whether pseudocode, diagrams, or other artifacts were used for design, whether the subject traced code, or whether a calculation was performed. A note was made whenever a subject consulted Java documentation or the textbook. The time of each compilation or program execution was noted. Each time an error was introduced the nature of the bug (syntax versus logic) was recorded. The time was logged as each error was corrected. Testing behaviors were also observed; the number of test cases was recorded and whether the testing was random or methodical in the researcher's opinion. In addition, researchers made minute-by-minute notes of any interesting behaviors.

start time: _____ environment:_____ lines at first compile: __

time	written artifacts **D**esign (**p**seudocode/ **g**raphical/ **o**ther), **T**race, **C**alculate	Jdocs/text	compile/run	error **S**yntax/ **L**ogic (assign #)	correction (indicate # from error column when corrected)	testing # of cases **R**andomly/ **M**ethodically	notes
0:01							
0:02							

Figure 1. Programming behavior log.

The number of lines of code written before the subject performed his or her first compile was also recorded.

Researchers did not offer programming or debugging assistance, limiting their responses to a few standard questions: What are you doing now? What are you thinking now? Can you explain what you just did? What have you been doing for the past few minutes? What would you do if I weren't here? What else have you tried in the past? Are you completely stuck?

Program quality analysis

During the analysis phase of the project student programs were rated on a scale from 5 (on track) to 1 (no chance of successful completion), where 'on track' meant the student had either solved the problem or was nearly there in terms of a solution. As part of the rating process a rubric was developed for evaluating the quality of the subjects' programs (see Table 2). The rubric is not based on the sort of standards a teacher would normally use to grade a program.

A notable issue affecting the rubric was the limit of 30 minutes to finish the programs. Only three subjects were able to successfully complete the programming assignment within the allocated time. We were left to determine "quality" based on an incomplete program. In addition, the difficulty of the programs varied. As a consequence, our rubric had as its primary metric of success an assessment of the degree to which the student was on track to producing a successful solution. We gave a high quality score of 5 to programs that appeared to be on track based on their use of programming logic and features, even if significant errors existed. If errors could be identified via compiler messages or if errors were due to syntactic substitutions such as accidentally using "while" instead of "if", then we deemed the program to be on track. The subject was likely to succeed eventually.

The rest of the rankings were used to differentiate the degree to which the student was off track and hence less likely to complete a working program without some significant outside help, further development of programming concepts, or understanding of the problem. Programs scored with a quality of 4 showed reasonable progress, but evidenced at least one serious semantic or logic issue making it less likely for students to eventually complete the program correctly. Level 3 programs were more seriously off track; progress was demonstrated but there was insufficient evidence to determine what misconceptions were preventing successful program development. These results were far from what one

Table 2. Programming assessment rubric.

Score	Progress	Description
5	On track	The code produced had some of the expected components (loops, ifs, computations), though this could range from an almost complete or complete program to one which was only partially complete. However, the code as developed to this point looked as if it was on track to produce a solution that would work eventually, recognizing that some compiler/syntactic errors still needed to be fixed. In some cases the student had not tried to compile the code before time ran out. The work produced so far in the code showed comprehension of what was required to complete the program.
4	Off track	The code was off track in a small way. There was at least one aspect of the code that was off track, leading the assessor to believe that the student would have to accomplish some semantic/logical-type debugging before completing the program. Current expression showed some misconception or lack of understanding.
3	Off track	The code was notably off track in a significant way. The code produced was not what an instructor would expect and it appeared from the work so far that the student was unlikely to reach a solution. The work shown indicated serious misconceptions and lack of understanding.
2	Off track	The program had been started in some small way, but so very little had been done that there seemed to be little hope of a programming solution being reached. About one or two lines were completed in the 30 minute programming session.
1	Off track	Minimal or no progress. The student made minimal or no progress toward a programming solution.

would expect a successful student to produce. Levels 2 and 1 were used when the subjects made little or no progress on the problem in 30 minutes. Given the minimal one or two lines of code produced in the 30 minute session the assessor concluded that the students had insurmountable misconceptions and were not likely to arrive at a working solution.

Based on the rubric two researchers independently evaluated the programs and compared and agreed on a rating. Then the researcher who interviewed the subject was also asked to independently rate the subject's programming performance, resulting in a final overall numeric rating. This defined a programming quality score for each subject. Later a final experienced teacher/researcher independently reassessed the programs using the rubric as a check for inter-rater reliability, achieving 95% agreement.

Recall that the primary motivator for the programming experience was to familiarize the student with the problem before undertaking the debugging task. As such, no true measure of programming ability can be inferred. However, program quality was analyzed in order to provide an evidence-based ranking of student programming performance, enabling us to compare programming ability with student debugging performance in a limited fashion. The limitations of the methodology have an impact on these rankings. The fact that over 50% (11/21) of our subjects were ranked as on track (5) should not be considered evidence that CS1 subjects (these or others) are excellent or even successful programmers on the types of programming tasks reported here.

This rating of program quality allowed us to further investigate the relationships between debugging and programming skills observed by Ahmadzadeh et al. (2005). A more detailed analysis of the programs is worthwhile and will be reported on at a later date.

Debugging exercise

Following the programming exercise the subjects were provided with a syntactically correct version of the program they had just coded. Each "buggy" program contained from three to five logical errors. The same problem description and sample input with correct output were provided. Subjects were not told ahead of time exactly how many bugs were seeded in their programs. This technique was adapted from Gugerty and Olson (1986) and Nanja and Cook (1987), who also seeded bugs into syntactically correct programs (Gugerty and Olson seeded only one error, Nanja and Cook seeded six). Errors included common novice mistakes such as incorrect assignment statements, malformed Boolean expressions, missing or incorrect initializations, out of order or missing statements, using "= =" to compare objects, forgetting curly braces, and off-by-one, infinite and incorrectly terminated loops. Errors were chosen based on our experiences with novice programmers and influenced by previous novice studies, including Bug Catalogue I (Johnson et al., 1983). Table 3 summarizes the errors seeded in each program.

Each subject was given 20 minutes to locate and fix as many errors as possible. Subjects were asked to think aloud while their comments were audiotaped. Researchers were only allowed to interact with subjects by asking one of a small set of prescribed questions: for example, What are you doing now? What are you thinking now? Can you explain what you just did? What would you do if I weren't here? What else have you tried in the past?

Each researcher again used a log sheet to record subjects' behavior on a minute-by-minute basis (see Figure 2). Noted behaviors are similar to those recorded in Gugerty and Olson (1986) and included: the subjects' focus of attention (on the problem statement, on the code, on documentation, on the output); how many times they compiled and ran the code and when; if they wrote anything on paper (designs, calculations, drawings, tracing tables); which of the intentionally inserted bugs they were able to locate, diagnose and correct, and when; whether they identified non-existent bugs; if and how they tested their corrections. Unstructured observational notes were also taken; some were think aloud comments from the students.

Debugging success and quality analysis

During the analysis phase of the project the debugged programs were examined, then compiled and tested. We noted if the program did not compile and, if that was the case, recorded the first error reported by the compiler. If the program did compile, we described the cases in which the program did not perform identically to our correct program. Each bug was tagged as fixed, not fixed, or partially fixed. Additional comments about partially fixed bugs were logged.

The number of bugs fixed offers only one measure of the subjects' debugging success. We observed that some subjects who fixed relatively few bugs actually employed quite sophisticated strategies, while others managed to fix all of the bugs but seemed to stumble upon them rather haphazardly and did little if anything to verify that their fixes were correct. To gauge the quality of students' debugging the minute-by-minute logs of the debugging exercises and the debugged programs were reviewed to determine the subjects' debugging processes (i.e., behaviors exhibited and strategies used). The quality of each subject's debugging process was rated on a scale of 1 (poor) to 5 (good) by three experienced instructors. The three researchers arrived at scores

Table 3. Seeded errors.

Problem	No. of bugs	Bug type				
		1	2	3	4	5
1 Rectangles	3	Loop vars swapped	Iteration condition	Incorrect if logic		
2 Craps	3	Boolean logic	Missing stmt	Data casting		
3 Raffle	5	Incorrect initialization	Language knowledge	If body not compound	Misplaced stmt	Arithmetic error
4 Triangle type	3	Boolean logic	Incorrect if logic	Incorrect if logic		
5 Binary search	3	Iteration condition	Arithmetic error	Incorrect if logic		
6 Calculator	5	Misplaced stmt	Missing stmt	Arithmetic error	Missing stmt	Stmt outside loop

time	look at: description	look at: code	look at: Jdocs/text	look at: output	compile/run	written artifacts (**T**race/ **C**alculate/ **D**raw)	locate give # (0 if not a bug)	diagnosed cause **C**orrectly/ **I**ncorrectly	alter	del	add	test # cases **R**andomly/ **M**ethodically	notes
									lines of code **C**orrectly/ **I**ncorrectly				
0:01													
0:02													

Figure 2. Debugging behavior log.

independently for each subject and discrepancies were resolved by discussion. These assessments considered:

- effective strategies, such as examining variable values using print statements or the debugger;
- noteworthy observations, for instance, verbalizing that loop errors can stem from either the loop controlling expression or the updating statement;
- unproductive or inferior processes, for example only testing the exact case given in the problem statement or saying "it looks right" and stopping at that.

These factors were used to form a subjective, and admittedly imperfect, composite rating of the quality of each subjects' debugging process. The ratings in combination with a count of bugs fixed formed a two-part assessment of each subject's debugging achievement.

Semi-structured interviews

After completing the debugging task the subjects were asked to reflect on the exercise and respond to a set of semi-structured interview questions. These were designed to elicit subjects' impressions of the debugging exercise, their thoughts on debugging processes and strategies, and the motivation for their problem choice. We also specifically asked if their experience during the debugging portion was similar to the kinds of debugging they generally do – which it was.

Debugging survey – post-exercises self-assessment

In the final component of the interview subjects were asked to complete a survey in which they self-ranked their debugging skill and their programming skill on a scale of one to five.

4. Results

Our results come from analyzing the students' debugging success. We consider the quality of their debugging skills (ability to find and fix bugs) as compared with their self-reported abilities and with the degree to which their programming solutions were on track for successful completion. "Finding" a bug is defined as identifying a buggy line. "Fixing" a bug is defined as making a correct change to a line to cause it to execute properly in the

context of the program. We examine the bugs per problem and then discuss the debugging strategies students used.

Debugging success

On average the subjects found 70% and fixed 68% of the bugs that were seeded into their programs (only two bugs found were not fixed). This corroborates Katz and Anderson's (1987) finding that locating bugs is more difficult than fixing them, since those successful at finding bugs managed to fix almost all of them. Table 4 summarizes the subjects' debugging success broken down by percentage of bugs fixed. Nearly half the students (9 of 21) fixed all of the bugs, while one-third (7 of 21 students) were able to fix one or none of the errors.

Debugging time

Subjects were allowed 20 minutes to correct a buggy implementation of the problem they had just programmed. The average time spent on the debugging task was 17.1 minutes, with a standard deviation of 3.53 minutes. Two subjects, both working on the binary search problem, stopped debugging at 10 minutes, one having fixed only one and the other having fixed none of the three errors. It should be noted that the subject who fixed no errors replaced the buggy code she/he was given with her/his own working implementation of linear search from the programming exercise. Nearly half the subjects used the full 20 minutes allowed, with varied success.

Overall, the average time to fix a bug once it was found (find-to-fix time) was 3.04 minutes, with a standard deviation of 3.27 minutes. Clearly there was a great deal of variation in the find-to-fix time among subjects. To investigate this we report in Table 4 the average time spent fixing a bug based on how many bugs the subject managed to fix. One anomalous subject who fixed only one bug did spend a long time on it (15 minutes) – an indication of flailing. In general subjects who fixed fewer bugs ($1/3^1$ or 3/5) did not spend as long fixing each bug once it was found, but spent significant time getting started and spent longer finding bugs to fix. It appeared from the debugging logs that these students fixed easy bugs quickly but had difficulty finding the less obvious bugs. Subjects who fixed more of the available bugs (2/3, 4/5, and 5/5) spent, on average, longer fixing each bug – possibly because they solved more difficult bugs. This assumes that the bugs students find first are ''easier'' than bugs they find later.

Debugging success versus quality of debugging strategies and debugging self-assessment

Here we consider student debugging success. We take as a baseline the students' debugging success as defined by the percent of bugs they fixed during the debugging task and compare it to the subjective analysis of the quality of their debugging strategies as gathered

Table 4. Percentage of bugs fixed and time taken to fix them.

Bugs fixed (%)	0	20	33	60	67	80	100
No. of subjects	3	1	3	2	2	9	
Mean time on task (min)	14.33	20.00	16.67	20.00	20.00	16.50	17.00
Mean find-to-fix time (min)		15.00	1.33	1.33	2.50	2.25	2.76

from the debugging logs. We then look at the quality of their debugging strategies in relation to a post-exercise self-assessment of debugging ability. Both of these values are ranked on a scale of 1 to 5 (1 indicates poor strategy quality while 5 indicates good quality). Self-assessments are based on the subjects' Likert responses to "Do you consider yourself a good debugger?" (with 1 meaning "a little" and 5 meaning "a lot"). Figure 3 shows the percentage of bugs fixed in relation to the researcher assessments of debugging strategy quality per subject.

The quality of subjects' debugging strategies (as assessed by experienced teachers) relative to their debugging success was mixed. Some of the best debuggers used effective strategies, but over half were assigned a quality score of 3 out of 5 or less, with one subject (B01) fixing all of the bugs yet receiving a debugging quality score of 1. We believe this phenomenon was due to the relative ease with which some better debuggers were able to find and fix our straightforward bugs; many did not need to use sophisticated strategies. Debugging strategy quality was more in line with debugging success for the less successful (60% of bugs or less) debuggers; all of them scored 3/5 or less for debugging strategy quality.

Figure 3 also compares our assessment of each subject's debugging strategy quality with his or her self-assessment of debugging ability. We see that these two ratings are generally aligned for the middle and upper ability students, although none of the students responded to the question "Do you consider yourself a good debugger?" with a rating of 5 (indicating "a lot"). In contrast, for those whose strategy assessments were poor, all but one responded to the question with a self-rating of 3 or 4.

One surprising aspect of these results is the relationship of students' self-reported assessment of debugging skill to the actual number of bugs fixed. Specifically, students who had just found and fixed all bugs in a program in a 20 minute window frequently (5/9) scored their debugging ability as average. None of the students rated themselves at the highest debugging skill ranking.

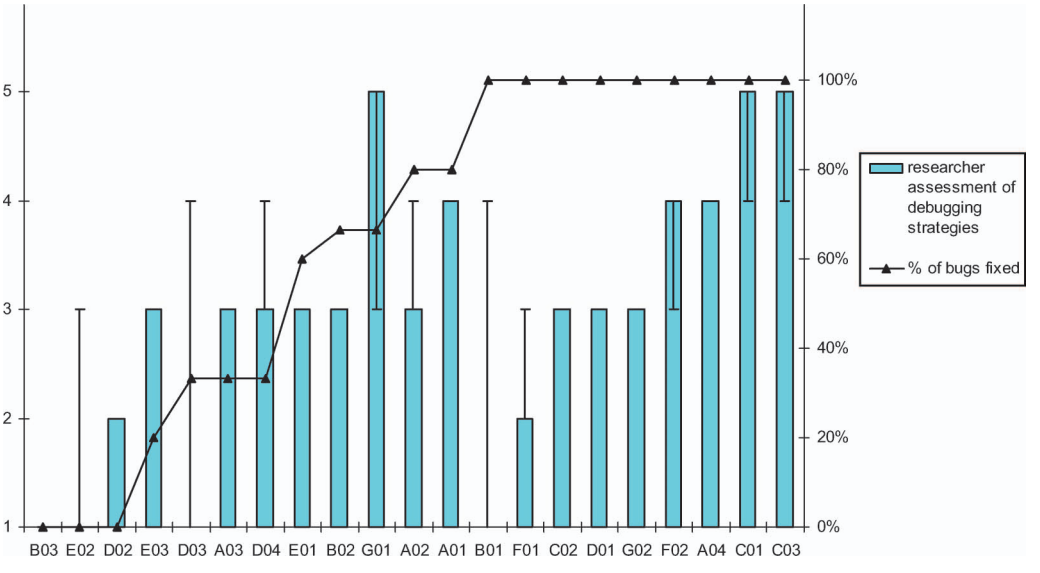

Figure 3. Quality of debugging strategies versus bugs fixed. Student self-assessment of debugging ability shown in error bars.

Debugging success vesus quality of programming ability and programming self-assessment

Debugging does not exist in a vacuum – in many scenarios (and especially for novice students) it is an integral part of the program creation process. We used a warm-up programming task not only to acclimatize our students to a particular problem, but to obtain a snapshot of the quality of their programs. To investigate the relationship between students' debugging and programming abilities we compared program quality ratings with the percentage of bugs fixed per student. This is shown for each subject in Figure 4 (subjects are ordered to match the *x*-axis of Figure 3). Program quality ratings were subjectively determined as described in section 3. Students also self-assessed their programming abilities in the post-exercise survey.

Figure 4 shows that most effective debuggers – those who fixed 80% or more of the bugs – were also on track to successfully complete the programming assignment, with all but one scoring at least 4 out of 5 on their program quality evaluations. However, program quality ratings for the less skilled debuggers – those fixing 33% or fewer of the bugs (0 or 1 bug) – were mixed, with four scoring at least 4 out of 5 and two receiving a score of 1 for programming quality. These results confirm the findings of Ahmadzadeh et al. (2005), who reported that good programmers were not necessarily good debuggers. At the same time, two of the three subjects who corrected none of the bugs also scored only 1 on their program quality evaluations, indicating that weak programmers were not good debuggers, also as noted by Ahmadzadeh et al. (2005). (Recall that D02 scored 0% on debugging success because she/he replaced the binary search method with her/his own linear search solution – making that score somewhat anomalous.)

Figure 4 also shows that the majority of students (18/21) ranked their general programming ability at or below our subjectively ranked program quality, with the

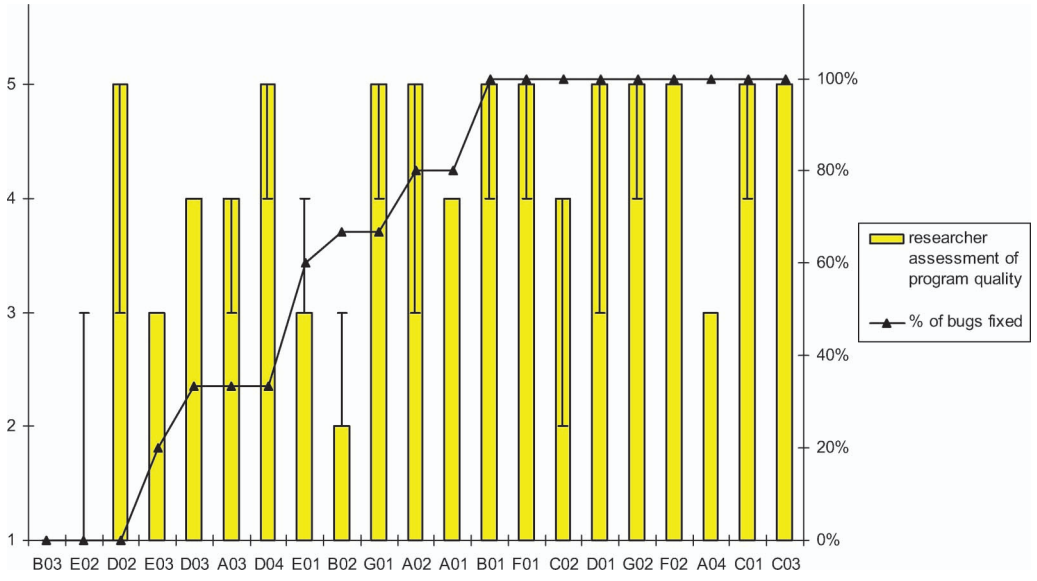

Figure 4. Programming ability versus bugs fixed. Student self-assessment of programming ability shown in error bars.

majority of them ranking themselves lower than we ranked their programs. We caution against reading too much into this comparison, as the students were ranking overall ability, which is likely influenced by their daily experiences in learning new materials, and program quality ranking was based on a single program assessment. It is also worth noting that although we rated 11 of the 21 subjects as on track, only three subjects actually produced correctly working programs in 30 minutes (one rectangle, one craps, and one calculator). This indicates novice inaccuracy in estimating how long it will take to solve a problem. Subjects were asked to pick a problem they thought they could solve in 20 minutes; most did not manage to finish a correctly working program in the allotted 30 minutes.

Debugging success by problem and by classification

In Figure 5 we analyze overall debugging success, problem by problem, as viewed through the lens of the bug classification system of Johnson et al. (1983). For each of the seeded bugs in each of the programs we describe the bug type and further categorize those types as a language problem (synonymous with Johnson's construct-related errors) or as one of his non-construct-related classes: missing, misplaced, malformed and spurious statements. Johnson's classification of the bug and a more specific descriptor (bug type) are shown in the leftmost columns. We did not observe any of Johnson's spurious statement bugs in this exercise and have not included them in the table. The lack of spurious bugs likely stems from having provided subjects

Bugs fixed by subjects

Values are the ratio of bugs fixed to bugs seeded; S is the number of subjects, B is the number of bugs in that problem.

Johnson et al Classification	Bug Type	rectangles (S=4, B=3)	craps (S=3, B=3)	raffle (S=7, B=5)	triangles (S=1, B=3)	binary search (S=2, B=3)	calculator (S=4, B=5)	ratio of bugs fixed to bugs seeded in the program
Language problems	language knowledge error			6/7				6/7 (86%)
Missing Statement	loop condition update		1/3				4/4	5/7 (71%)
	assignment						4/4	4/4 (100%)
	if body not compound			5/7				5/7 (71%)
Misplaced Statement	should not be in loop but is or should be but is not			4/7			8/8 *	12/15 (80%)
Malformed Statement	conditional: incorrect if logic	2/4			1/2 *	1/2		4/8 (50%)
	loop: boolean logic		3/3		0/1			3/4 (75%)
	loop: iteration condition error	3/4				0/2		3/6 (50%)
	loop: nested loop variable swap	3/4						3/4 (75%)
	arithmetic error			3/7		0/2	4/4	7/13 (54%)
	data casting error		2/3					2/3 (67%)
	data initialization/ update error			4/7				4/7 (57%)
per problem ratio fixed		8/12 (67%)	6/9 (67%)	22/35 (63%)	1/3 (33%)	1/6 (17%)	20/20 (100%)	58/85 (68%)

Figure 5. Types of bugs fixed and ratios of bugs fixed to bugs seeded.

with buggy code rather than asking them to generate their own bugs through programming.

Each column in Figure 5 is dedicated to each program. The number of subjects selecting a problem along with the number of bugs seeded in each problem are shown in the column headers (S = *n*, B = *m*). Within each program the seeded bugs are characterized by cells in the table. The cells show a ratio of bugs fixed to bugs seeded where bugs seeded equals the number of that class of bug times the number of subjects who chose that program. Bugs fixed equals the number of times the bug was successfully fixed.

On the right side of the table a sum of total bugs fixed divided by bugs seeded across all programs is given for each bug type. The bottom row shows the ratio of bugs fixed over bugs seeded on a per program basis.

Overall, the 21 subjects fixed 68% (58/85) of the possible bugs. Most of the individual seeded bugs (16/22) were found and fixed by at least half of the subjects who attempted the problem. Said another way, the subjects were generally able to find the majority of our bugs, probably because these bugs were not overly difficult for subjects with this level of Java programming experience.

There were only two errors which were found but not fixed. These were both errors which did not involve programming language constructs such as loops or if statements. One was a missing statement in the craps problem and the other an incorrect initialization error in the raffle problem.

Bug fixing success did vary notably by problem solved. The three easiest problems (rectangles, craps, and raffle) had approximately the same bug fix rate (see bottom row of table). The hardest program, calculator, had a 100% bug fix rate, which we believe (from responses in the semi-structured interview) was due to the best students choosing the hardest problem available. It is also notable that the two subjects who chose binary search had an unusual debugging experience which had an impact on their success rate. One subject (D02) chose to replace all the buggy code with his linear search code from the programming exercise – resulting in not fixing any of our seeded bugs. D04 essentially didn't get ''warmed up'' by the programming experience, since she/he initially programmed linear search. During the debugging phase she/he attempted to debug binary search and only fixed one bug.

Finally, the arithmetic error in raffle (fixed by fewer than half the subjects) was probably infrequently fixed because the subjects ran out of time in solving raffle's five bugs.

Using the bug categorization structure of Johnson et al. (1983), language knowledge (or construct-related) errors were the most likely to be fixed (86% of all construct-related errors were fixed). The raffle program contained our only example of a language knowledge error: it contained an erroneous comparison of String objects: (studentName ! = ''stop'') using ! = when what was needed was (!studentName. equals(''stop'')).

Of the non-construct-related errors the most difficult class for students to fix was that of malformed statements (26 of 45, or only 58%, fixed) (see rightmost column). Our malformed statements included:

- off-by-one loop conditionals;
- Boolean loop conditional logic errors (using || instead of && or vice versa);
- integer division errors;
- initializing a variable to 1 instead of 0;
- swapping conditional block bodies in an if statement;
- incorrect calculations to support logical algorithm correctness.

Missing statement errors (77%) and misplaced statement errors (80%) were fixed at approximately the same rate. As examples of missing statements, we removed:

- statements that were needed to redefine a ''roll'' value in the craps game (inside a loop);
- curly braces that were needed to block a two-statement if block (although proper indentation was shown);
- a statement to parse the first character of a call to next() used to determine an operator in calculator;
- a statement to read whether the user wants to calculate another expression.

Our misplaced statements included:

- printing the results of the raffle inside the loop where student names were read (code that should go outside the loop);
- printing the calculator answer inside the ''Do you want to play again?'' loop (code that should go inside the loop);
- initializing the answer to the calculator outside the loop.

Debugging strategies

We observed that most students used at least one debugging strategy during the debugging exercise, although some applied those strategies ineffectively. A full discussion and analysis of the strategies students used is beyond the scope of this paper, but an overview is discussed here.

Strategies exhibited by students included mental tracing with and without println statements, hand tracing, and tracing using the debugger. Students also tested their code, many using only the example input from the problem description, although some made up their own test cases. Others isolated the problem by assigning program variables to specific constants or commenting out statements. Students also applied pattern matching techniques to correct code that just didn't look quite right. Examples included misplaced initialization statements and curly braces. Some students thought that correct code contained errors simply because it didn't 'look right'. Resources such as the JavaDocs, the BlueJ debugger, and automatically compiling IDEs such as Eclipse were used by a number of students to find and fix bugs.

Less effective strategies included tinkering or making small changes in code to get it to work (cf. Perkins, 1986) and rewriting entire sections of code, thereby solving the problem by avoiding truly understanding and fixing a bug.

Our subjects often used a forward reasoning (tracing) strategy while debugging. This was previously observed by Katz and Anderson (1987), who reported that students debugging others' code tend to use forward reasoning. On the other hand, causal reasoning (reasoning backward from output or symptoms) was observed infrequently. One student noted that the probability was 0.0 at the end of the Craps game and used that observation to make inferences about how integer division was working. Another recognized that the results in the raffle were off by one and worked back from there.

We note that these problems were not overly difficult and the students were not intimately familiar with them, so it is perhaps not surprising that the students did not use backward reasoning or more 'expert' strategies.

As previously mentioned, the quality of subjects' debugging strategies relative to their debugging success was mixed (see Figure 3). Some of the best debuggers used high quality strategies, but over half scored 3 out of 5 or less for quality of strategies, as did the less successful debuggers. Students who were quantitatively and qualitatively judged to be poor debuggers did not appear to use fewer strategies, but rather used strategies less effectively. For instance many students inserted printlns, but some inserted them in key places to print useful information and others less effectively printed the same fixed string in two different places. This finding is consistent with a previous study of code tracing strategies (Fitzgerald, Simon, & Thomas, 2005).

5. Discussion

In this section we discuss our findings in the context of our general research questions, presented in section 1, and in the context of earlier research studies. We also discuss the implications for teaching.

How well can novices debug?

On average subjects found 70% and fixed 68% of the implanted bugs during the debugging exercise. A fair number of the students (9 of 21 or 43%) fixed all of the bugs, while one-third (7 of 21 students) were able to fix 1 or none of the errors. It took subjects longer to find than to fix the bugs, suggesting that finding bugs is more difficult. In general, students at this level took only a minute or two to fix a bug once they found it, thus fixing bugs is probably not a serious impediment for students when completing regular programming assignments. Our results confirm the findings of Katz and Anderson (1987) that bug location continues to be the difficult task for novices; once they find the bugs they can usually fix them. This implies that guidance and practice with bug location strategies should be emphasized during debugging instruction.

Unlike the novices in Nanja and Cook (1987), the subjects in this study were quite successful at debugging overall, although we believe the bugs were of comparable difficulty in the two studies. Several causes for this phenomenon seem plausible. The program used by Nanja and Cook was longer than the programs we provided to our students. On the other hand, additional debugging studies have used programs similar in size to our exercises and generally reported poorer debugging results as well (Gugerty & Olson, 1986). It could be that the subjects in the current study had had more instruction than novices in previous studies, making the bugs easier for them to find and fix. However, Nanja and Cook's novices were finishing their second term of programming instruction, which is at least as much instruction as the most experienced subject in this study. The most likely cause is that these students' had increased problem domain knowledge gained from completing the warm-up programming exercise, which enabled them to perform more like experts during the debugging task. The poor results of the students attempting binary search make sense in this light – these students did not actually know binary search and both implemented a linear search instead. Lacking the domain knowledge they should have gained during programming, their debugging performance was dismal, with only one subject finding even one of the bugs. This suggests that teachers should emphasize problem comprehension – making sure students understand the problem well, perhaps through opportunities for comprehension self-assessment, may improve students' overall debugging performance.

In contrast to the subjects in Nanja and Cook (1987) and Gugerty and Olson (1986), the subjects in this study did not introduce many new bugs into their programs.

Students used online resources and debugging tools with some frequency, also contrary to Nanja and Cook's findings. Internet searching, a technology not available 20 years ago, was commonly performed by our subjects. Additionally, the effective use of debuggers (when readily available), suggests students can benefit from these and IDEs that detect obvious semantic errors (e.g., the missing '}' bug was highlighted by the Eclipse IDE on opening the file).

Similarly to the novices in Perkins and Martin (1986), the novices in the current study also demonstrated fragile knowledge – some were confused when code did not appear as they were accustomed to seeing or writing it. Subjects using unfamiliar environments, whether it was the programming editor or the style of coding, struggled more than other students. For example, one subject did not complete a program because an unfamiliar method was used to read data in the skeleton program provided, a few subjects ran out of time because they chose to write classes although they were unnecessary, and several subjects were slowed by unfamiliar editors and environments. This implies students could benefit from exposure to code written in different styles. Without such exposure fragile knowledge could hinder their success, especially if test or exam questions are written in an unfamiliar style.

What types of bugs are most difficult for novices to find and fix?

Our subjects found arithmetic bugs (53% were fixed), in particular, and malformed statement bugs (58% were fixed), in general, more difficult to find and fix. It could be that finding these bugs requires a deeper understanding of the program; it is worth noting that our own code for the raffle problem had an inadvertent bug that was arithmetic. In a study of FORTRAN programmers Gould (1975) found that errors in assignment statements were the most difficult to find, since finding them required a deeper understanding of the program than other types of seeded errors. If we consider all of our malformed statement errors using Gould's categories we see that four of them would be classified as assignment errors: the off-by-one error in raffle, an arithmetic error in binary search, an arithmetic error in calculator, and a data casting error in craps. Recalculating our results, in order to compare them with those of Gould, we see that subjects were able to find and fix 63% (10 of 16) of assignment errors.

In general our subjects found non-construct-related bugs (language-independent bugs) harder to find and fix than construct-related bugs (bugs related to a misunderstanding or confusion about the language). This supports the results of Soloway and Spohrer (1986), who found that non-construct-related bugs were much more common problems for novices than construct-related bugs. In the post-session interview the subjects indicated that the easiest bugs to fix were those found by the compiler or some other tool. This suggests that students find locating bugs more difficult than fixing them, or that the types of bugs found by the compiler and other tools, which are most often construct-related, are easier to fix than other bugs.

As noted above, malformed statements were the most difficult bugs to fix. Loop conditions, conditional logic, arithmetic, and data initialization and updating were difficult errors to find. Laboratory or homework exercises to find malformed statements may help students become more adept at locating these bugs.

What is the relationship between novices' programming and debugging abilities?

Assuming our on track programmers are good programmers in general, our results confirm those of Ahmadzadeh et al. (2005): good programmers are not necessarily good debuggers. Good debuggers are usually good programmers. Confirmation of these results suggests that instructors may need to shift from thinking of designing/writing/debugging collectively as programming and begin to consider each as a distinct skill. The following illustrates three cases of the on track programmer–poor debugger phenomenon.

D03 had a high program quality ranking, but only fixed one of three bugs. Finding a bug early on (minute 5 – a missing statement), she/he then spent a long time reading and mentally tracing the code, searching for a second bug without ever seeming to gain traction. Only after finally compiling for the first time at minute 11 did the student begin to make forward progress again. She/he then took a new look at the code and found (minute 15) and fixed (minute 17) a different bug quickly. A stubborn desire to understand and debug the code through reading alone caused this student to become mired down in the complexity mental tracing requires. Furthermore, the student lacked the meta-cognitive awareness to recognize that this technique was ineffective and that a different approach was required. Clearly this inability to recognize and cope with cognitive overload causes difficulty during debugging, although it may not surface or may have a distinctly different character with programming.

E03 appeared to suffer from incomplete or inert language knowledge. After finding and fixing a bug within 2 minutes, the student hit a wall because s/he did not recognize that the equals() method was required instead of = = to compare Strings. The subject flailed for 10 minutes without ever discovering the problem or making use of available resources such as the online JavaDocs or a textbook.

A03 spent 11 minutes reading and mentally tracing the code – only running it twice before finding and fixing one bug, in a single minute. It appeared that A03 could have made quicker progress in understanding the code with a few well-placed println statements, rather than continuing only to read the code.

These scenarios suggest that meta-cognitive awareness, in particular recognizing when one is over-taxed mentally or has become stuck, and guidelines for what to do in such situations would be useful facets of debugging instruction.

Are novices' self-perceptions of ability in line with their performance on programming and debugging tasks?

None of the subjects rated themselves in the highest category for debugging skills, even after successfully completing the debugging task. This is particularly interesting because students are rarely graded on debugging as a separate skill and therefore have not been exposed to metrics for making such assessments. Designing debugging metrics and assessing debugging skills may be crucial in improving debugging instruction.

What strategies and behaviors do novices exhibit as they debug?

Tracing, a primary strategy used by the subjects in this study, took several forms: students traced in their heads, on paper, using print statements, and via the debugger. It may be helpful to systematically teach these approaches, with suggestions for when each technique should be used. Giving students progressively harder debugging problems would serve to

illustrate when different strategies are needed. Along these lines, it may be possible to develop some heuristics, such as:

- if you have to keep up with more than one or two variables or there's a loop involved then you need to trace on paper, not just in your head;
- if the bug can't be determined by looking at only at the input and output values then you need to add some print statements in the middle;
- make sure that your print statements are well placed and print useful information;
- if you have to put in too many print statements, your program 'hangs' or you think it has an infinite loop, use the debugger.

Several students used the strategy of forcing the execution of a specific case, although they were not always systematic about it. This was accomplished by assigning a variable to a specific value or commenting out a loop header, causing a specific path to be taken in the code. This suggests another important strategy for instructors to emphasize. Exercises could involve students working through a debugging example with questions such as "To find the bug what case should you force?" and "What will it tell you?"

Unlike previous debugging studies, we noticed students using pattern matching as a key element of debugging. Many noticed errors because code didn't "look right" (e.g., the curly braces around the compound if, an initialization in the wrong place) or because they had a mental pattern of how the code should be structured (e.g., needing a read at the bottom of the loop to update the loop variable). This suggests incorporating patterns into novice programming instruction might also have a positive influence on their ability to debug. Further research is needed in this area.

In contrast, students also used patterns unproductively. Some commented out perfectly correct but unfamiliar code without bothering to look up how a method worked. They also "corrected" things that weren't wrong (e.g., removing a space between String and [] in the main() method header and adding curly braces to ifs with single statement bodies). This suggests debugging productivity may increase if students are shown examples of programs that do exactly the same thing but look completely different due to formatting or alternative implementation choices. It also implies exposing students to unfamiliar code and discussing how to cope with it could be beneficial.

Our data revealed evidence of both forward (tracing) and backward reasoning (deducing from output or symptoms) approaches to debugging, although forward reasoning was more prevalent. Katz and Anderson (1987) also found that novices tended to use forward reasoning, particularly when they were unfamiliar with the program. Both forward and backward reasoning strategies can be explicitly taught.

6. Threats to validity

There are a number of concerns and methodological issues that should be considered in evaluating these results. Most notably, the debugging we asked students to complete differs from that students must master for a typical homework assignment. We asked students to debug code that they did not write – a process common in industry but not typical for novice programmers. In contrast to prior debugging studies however, we engaged students with a warm-up programming exercise. We believe this served to more closely model the novice programming experience and also had a significant impact on their debugging success. This is supported by the two subjects who essentially coded a different program for binary search; they had unique debugging experiences and were not very successful at the debugging task. One subject simply removed all the "weird looking"

code provided for him in the buggy code and re-wrote his own version of linear search. The other subject spent a long time reading and tracing the buggy code provided and ''learning'' the process of binary search.

At five of the institutions involved the researcher recruiting the participants and conducting the interviews was also the subjects' instructor in either their first (prior) or second (current) programming class. This may have influenced the subjects' decision to participate in the study.

Students who participated were all at similar places in their study, but instruction at each institution moves at a different pace. It was also impossible to control selection of subjects with respect to previous programming background. Although we attempted to compensate for this by allowing students to select a program to write and debug, overall these results can only be said to apply to ''beginning'' programmers generally.

Although every attempt was made to provide the subjects with familiar programming and debugging environments, some subjects were more comfortable than others with their interview settings. In particular, some students were accustomed to using the Internet to find examples or solutions; others had been given or collected a set of sample programs they used extensively as templates. Researchers were inconsistent in allowing this access.

Also of importance is the variation in detail and accuracy recorded in the programming and debugging logs. We chose not to make video recordings of the exercises due to logistical challenges in our multi-institutional environment. Researchers made notes of varying detail.

Finally, our processes for determining debugging and programming quality were meaningful to us, as experienced programming instructors, but may be subject to different interpretations.

7. Conclusion and future directions

This study revisits ground covered by a number of previous investigations, many of which were conducted 20 years or more ago. This study builds on that work and provides some evidence as to whether those results are still valid for today's novice programmers. Our findings both validate and contradict some past results. With respect to cognitive style, particularly the problems of fragile knowledge and the forward reasoning approach to finding bugs, novice debuggers now are similar to those in the past. This work supports previous findings that domain knowledge and program comprehension are key, as the warm-up programming exercise led to high rates of debugging success. Additionally, we find it important to note that, contrary to our anecdotal expectations as computing instructors, students are able to fix bugs once they find them. This confirms the finding of Katz and Anderson (1987), but in a less restrictive and more natural setting for novice Java programmers.

Similar to Soloway and Spohrer's 1986 findings, our subjects had more difficulty with language-independent bugs rather than bugs related to misunderstanding or confusion about the language. Arithmetic errors were particularly troublesome debugging problems. Malformed statements were the most difficult bugs to fix. Loop conditions, conditional logic, arithmetic errors, and data initialization and updating were difficult errors to find.

This study finds changes from previous studies, particularly the use of electronic tools, including IDEs, debuggers, and the Internet. Subjects in this study used pattern matching during programming and debugging, an approach not noted in previous studies.

Both the findings and, to some extent, the limitations of this study suggest avenues for future work. While students were asked to think aloud while working, many found it distracting and chose not to speak very often, which limits the conclusions that can be drawn from some of their actions. Developing a protocol to encourage more speaking and

possibly recording the actions of the subjects may provide further insight. A study using pairs of students may help increase the think-aloud aspects of the protocol while also providing insight into differences between pair debuggers and solo debuggers. This study did not emphasize an object-oriented approach and a study focusing on the use of classes and objects may reveal more information for those teaching objects-first courses. The discussion section mentions a variety of implications for teaching; building and testing materials to modify debugging instruction is important in improving debugging skills in novices. More in-depth analyses of the rich data (e.g., debugging strategies and programming behaviors) generated by this study and their implications for teaching and learning are planned for the future.

Note

1. 1/3 denotes one of three bugs.

References

Ahmadzadeh, M., Elliman, D., & Higgins, C. (2005). Novice programmers: An analysis of patterns of debugging among novice computer science students. In *Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE)* (pp. 84–88). New York: Association for Computing Machinery.

Ducassé, M., & Emde, A.-M. (1988). A review of automated debugging systems: Knowledge, strategies and techniques. In *Proceedings of the 10th International Conference on Software Engineering* (pp. 162–171). Piscataway, NJ: Institute of Electrical and Electronics Engineers.

Fitzgerald, S., Simon, B., & Thomas, L. (2005). Strategies that students use to trace code: An analysis based in grounded theory. In *Proceedings of the 2005 International Workshop on Computing Education Research (ICER '05)* (pp. 69–80). New York: Association for Computing Machinery.

Gould, J. (1975). Some psychological evidence on how people debug computer programs. *International Journal of Man-Machine Studies*, *7*(1), 151–182.

Gugerty, L., & Olson, G. (1986). Debugging by skilled and novice programmers. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (pp. 171–174). New York: Association for Computing Machinery.

Hristova, M., Misra, A., Rutter, M., & Mercuri, R. (2003). Identifying and correcting Java programming errors for introductory computer science students. In *Proceedings of the 2003 SIGCSE Technical Symposium on Computer Science Education* (pp. 153–156). New York: Association for Computing Machinery.

Johnson, L., Soloway, E., Cutler, B., & Draper, S. (1983). *Bug catalogue: I* (Technical Report No. 286). New Haven, CT: Yale University, Department of Computer Science.

Katz, I., & Anderson, J. (1987). Debugging: An analysis of bug location strategies. *Human–Computer Interaction*, *3*(4), 351–399.

Ko, A., & Myers, B. (2005). A framework and methodology for studying the causes of software errors in programming systems. *Journal of Visual Languages and Computing*, *16*, 41–84.

McCauley, R., Fitzgerald, S., Lewandowski, G., Murphy, L., Simon, B., Thomas, L., et al. (2008). Debugging: A review of the literature from an educational perspective. *Computer Science Education*, this issue.

Nanja, M., & Cook, C.R. (1987). An analysis of the on-line debugging process. In G. Olson, S. Sheppard & E. Soloway (Eds.), *Empirical studies of programmers, Second workshop* (pp. 172–184). Norwood, NJ: Ablex.

Perkins, D., & Martin, F. (1986). Fragile knowledge and neglected strategies in novice programmers. In E. Soloway & S. Iyengar (Eds.), *Empirical studies of programmers* (pp. 213–229). Norwood, NJ: Ablex.

Spohrer, J., & Soloway, E. (1986). Novice mistakes: Are the folk wisdoms correct? *Communications of the ACM*, *29*(7), 624–632.

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