Debugging: The Good, the Bad, and the Quirky – a Qualitative Analysis of Novices' Strategies

Laurie Murphy
Department of Computer Science and Computer Engineering
Pacific Lutheran University
Tacoma, WA, USA
lmurphy@plu.edu

Gary Lewandowski
Department of Mathematics and Computer Science
Xavier University
Cincinnati, OH, USA
lewandow@cs.xu.edu

Renée McCauley
Department of Computer Science
College of Charleston
Charleston, SC, USA
mccauleyr@cofc.edu

Beth Simon
Dept. of Computer Science and Engr.
University of California, San Diego
La Jolla, CA, USA
esimon@cs.ucsd.edu

Lynda Thomas
Dept. of Computer Science
University of Aberystwyth
Aberystwyth, Wales, UK
ltt@aber.ac.uk

Carol Zander
Computing & Software Systems
University of Washington, Bothell
Bothell, WA, USA
zander@u.washington.edu

ABSTRACT
A qualitative analysis of debugging strategies of novice Java programmers is presented. The study involved 21 CS2 students from seven universities in the U.S. and U.K. Subjects "warmed up" by coding a solution to a typical introductory problem. This was followed by an exercise debugging a syntactically correct version with logic errors. Many novices found and fixed bugs using strategies such as tracing, commenting out code, diagnostic print statements and methodical testing. Some competently used online resources and debuggers. Students also used pattern matching to detect errors in code that "just didn't look right". However, some used few strategies, applied them ineffectively, or engaged in other unproductive behaviors. This led to poor performance, frustration for some, and occasionally the introduction of new bugs. Pedagogical implications and suggestions for future research are discussed.

Categories and Subject Descriptors
K.3.2 [**Computers & Education**]: Computer & Information Science Education – *Computer Science Education.*

General Terms
Human Factors

Keywords
debugging, novice programming, pedagogy, strategies

SIGCSE '08, March 12–15, 2008, Portland, Oregon, USA.

1. INTRODUCTION
Debugging is difficult for novice programmers. Similar to new drivers who must learn to steer, accelerate, brake, etc. all at once, novice debuggers must apply many new skills simultaneously. They 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 [2]. Unfortunately, most novices' knowledge of these skills is fragile at best [12], causing many to find debugging difficult and frustrating.

What is the best way to help novices overcome these difficulties? Despite numerous investigations of novice debugging, many dating back to the 1980s, there are no established best practices to guide instructors. Furthermore, the attention introductory programming texts give to debugging varies greatly. Often when instructors teach debugging skills (e.g., hand tracing, diagnostic print statements, how to use a debugger), students seem to practice them haphazardly, debugging whatever bugs they happen to encounter as they code using whatever method occurs to them at the time.

Both an earlier investigation of novices [1] and the current study (see [4] for a discussion) suggest skill at debugging may be somewhat distinct from general programming ability, and as such deserves individual attention pedagogically. This view was echoed in a 1986 paper by Kessler and Anderson [8] who observed that "… debugging is a skill that does not immediately follow from the ability to write code. Rather, it consists of several sub-skills ..." [8, page 208].

While interactions with students (e.g., during lab sessions or via questions) often give instructors useful anecdotal evidence of the sorts of bugs novices encounter, they rarely include opportunities to observe students' independent debugging processes. An understanding of the skills novices employ, as well as the misconcep-

tions and difficulties they confront, can offer useful insights for debugging instruction.

This paper reports on a multi-institutional study of novice Java programmers at seven universities in the U.S. and U.K. Its purpose is to illuminate novice strategies with the goal of improving debugging instruction. Effective strategies suggest topics for instruction; while poor application of those strategies, or unproductive behavior, highlight pitfalls to be avoided. An overview of these findings and their pedagogical implications follow.

2. BACKGROUND

In a 1975 study, Gould [5] was one of the first to study strategies subjects used in debugging FORTRAN programs. Gould found that subjects used a variety of tactics and their debugging could be described as a process of repeatedly choosing a strategy to find the suspicious area of the code, then generating a hypothesis and choosing a tactic, until the subject had found the bug. The most popular strategies were: examining the output early; looking for grammatical errors in the language that cannot be caught by the compiler; and reading the code to understand the program.

Vessey [13] found that both novices and experts employed breadth-first approaches to debugging, but that experts developed a whole-system point of view that novices did not. The novices also used a depth-first debugging strategy. She found experts were more proficient due to a greater ability to chunk information.

Katz and Anderson [7] found that three, general, bug-location strategies were used by students: mapping program behavior to a bug, hand tracing of the code, and causal reasoning. They found that few subjects traced code. Subjects took longer to debug the code of others' than their own. Moreover, when debugging others' code, subjects used forward reasoning (reading code sequentially or in execution order) to determine the bug; however when debugging their own code subjects were more likely to use causal reasoning (work backward from the output) to determine the bug.

Gugerty and Olson [6] found that while experts and novices used the same strategies in debugging; experts were faster and more accurate in their comprehension of the code, leading to better hypotheses about the bugs. Experts were also less likely to introduce new errors as they debugged.

Nanja and Cook [11] also compared expert and novice debuggers. Like this study, their experiment used a realistic programming environment and programs with multiple bugs. While all subjects began by examining the code, experts spent more time than novices examining the code. They also discovered that experts read for program comprehension; that is, they read the program in the order it would be executed. In contrast, novices tended to read programs sequentially, from top to bottom. Additionally, novices appeared to seek understanding of isolated sections of the code that they believed, through examination of the output, contained the error. Both experts and novices used output statements in debugging, but novices used many more. Experts more often used a debugging tool and corrected multiple errors at a time. The strategy of running the program was much more frequent for novices by a 3:1 margin.

We recognize that these studies are all at least 20 years old and only Nanja and Cook provided a relatively unconstrained programming environment (while there have been other debugging studies since the 1980's[1], none focused primarily on strategies of debugging). Our study examines students' strategies using contemporary environments and the Java language.

3. THE EXPERIMENT

One-on-one interviews of approximately 1.5 hours were conducted with each subject. Interviews involved: (1) a background questionnaire, (2) a programming exercise, (3) a debugging exercise, (4) a semi-structured interview and (5) a debugging survey.

3.1 Subjects

Subjects were 21 undergraduate volunteers from seven universities, six in the United States and one in the United Kingdom. Six were women (29%) and 15 were men (71%) and their ages ranged from 18 to 51, with a median age of 19. All had completed 15-20 weeks of Java instruction at the college level and were typically enrolled in the early to mid phase of their second Java course at the time of the interview. Our rationale for studying students at this level was that, although they were still novice programmers, they had completed at least one programming course and therefore were likely to have debugged 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 strategies.

3.2 Study Protocol

All subjects completed the following components during a single interview session.

Background form: gender, age, and programming experience.

P*rogramming exercise*: The purpose of the programming exercise was to acclimate students to the problem they would later debug. Subjects had 30 minutes to code their choice of six typical CS1 exercises. The problem descriptions were arranged from least to most difficult:

- **Rectangles** - given two integers, n and m, output three n x m rectangles of asterisks (solid, hollow, and checkered).
- **Game of Craps** – given basic craps rules and a Die class, simulate and calculate statistics for 10,000 rounds .
- **Bike Raffle** – input bike, ticket and overhead costs, ticket sellers' names, and numbers of tickets sold; display basic statistics for and profit from bike raffle.
- **Triangle Type** – input lengths of three sides and determine if they form a triangle, and if so, the type of triangle.
- **Binary Search** - input an integer key, implement a method to perform a binary search for the key in a sorted array.
- **Simple Calculator** – implement a simple infix calculator that adds, subtracts, and multiplies.

Students were allowed to work in the environment of their choice and to use typical resources such as textbooks or the online Java-Docs. Investigators maintained a minute-by-minute log of activities (time/number of compiles/runs, what subject is looking at, errors introduced/found, etc.) during the exercise. Final programs were collected and analyzed for correctness and completeness (details of the programming exercise are forthcoming).

Debugging exercise: Subjects had 20 minutes to debug a syntactically correct solution to the problem they had just programmed

[1] See [10] for a more thorough review of the debugging literature.

Time (min)	look at:				compile/run	written Artifacts (**T**race/ **C**alculate/ **D**raw)	located bug give # (0 if not a bug)	diagnose cause **C**orrectly/ **I**ncorrectly	alter	delete	add	test # cases **R**andomly/ **M**ethodically	notes
	description	code	Jdocs/text	output					lines of code **C**orrectly/ **I**ncorrectly				

Figure 1: Debugging Log sheet

that contained 3 to 5 logic errors. Bugs included typical novice errors such as malformed loop conditions, incorrect initializations, arithmetic errors, using == instead of .equals() to compare Strings, etc. Observers maintained a minute-by-minute log of subjects' activities, entering information under the headings in Figure 1.

Final programs were collected and analyzed for correctness and completeness (a quantitative analysis is forthcoming in [4]).

Semi-structured interview: Subjects responded verbally to a set of questions designed to elicit their impressions of the debugging exercise, strategies used, and motivation for their problem choice.

Debugging survey: A one-page, paper survey was used to illicit subjects' self-assessments of programming and debugging ability, perspectives on debugging approaches and evaluation of how they use their time as they debug.

3.3 Data Analysis

Qualitative, and where appropriate quantitative, analyses were conducted on the rich data from this experiment. The strategies reported herein emerged from a qualitative analysis of the debug logs and final debug solutions. Two researchers evaluated the logs and noted any strategies or interesting behaviors exhibited. In consultation with a third researcher (and in some cases the interviewer), a complete list of strategies and behaviors was derived.

Debug solutions were similarly evaluated by two researchers for correctness and modifications or additions which suggested successful or unsuccessful strategy use or other interesting behaviors.

The resulting complete list of strategies was then sorted by consensus such that very similar strategies were placed in the same category. This resulted in 12 final categories (see Table 1) which are described in section 4.

Table 1: Summary List of Strategies

Gain domain knowledge	Isolating the problem
Tracing, including: Mental, Print & Debugger	Pattern matching
	Consider alternatives
Testing	Environmental
Understanding code	Work around problem
Using resources	Just in Case
Using Tools	Tinkering

4. STRATEGIES

A recent study [9] considered strategies that successful students used to get "unstuck" while learning computing concepts. While this study did not focus only on debugging, they did identify 35 distinct strategies, organized into 12 categories, which are similar to the strategies discovered here. For example, tracing, using resources/tools, pattern-matching, and others were found in both studies. In common with a study of effective tracing strategies [3], we noted that while some strategies, such as Tinkering, are almost always ineffective, others, such as Tracing, while usually effective, may be undermined by inappropriate or inconsistent use.

The good news, however, is that novices do apply reasonable strategies to locate and fix bugs. In the subsections that follow, we present the effective and ineffective strategies subjects used, give examples, and discuss the unproductive behaviors students demonstrated as they debugged.

4.1 The Good

Many strategies listed in Table 1 were employed effectively allowing students to successfully debug their programs.

Gain domain knowledge: Many students reread the specification or reexamined the sample output to gain insight into the problem. While students rarely used the scratch paper provided to work through solutions, one student found an obscure mathematical error in Raffle by checking input and output on paper.

Tracing: A majority of the students used tracing extensively. Students traced mentally, comparing the output with the code, both silently and out loud. Three students traced on paper. More than half put println's in their programs, mainly to follow the flow of control, but one student, for example, printed the value of variables as execution progressed. Two students used a debugger to step through the code and check the value of variables.

Testing: Almost all students tested their programs. One used a calculator to verify arithmetic and another tested boundary conditions extensively and then predicted an incorrect output and confirmed it with a test. Two students fixed all the bugs and then said "I am just going to test it a few more times to be sure." Most tested the sample data in the specification and left it at that.

Understanding the code: Students certainly read the code. Only one explicitly said they were trying to understand what the variables were used for. One actually learned Binary Search from the code (picking this problem although clearly the student had never seen Binary Search).

Using resources: Three students were observed using resources. One used JavaDocs, another used the Java Tutorial, and a third used old programs. No student used the textbook.

Using tools: Two students used a debugger (see tracing).

Isolating the problem: Nine students commented out or altered code to isolate a problem. Two put in constant values to replace a variable. Another forced a specific flow of control with a constant.

Pattern Matching: Students recognized that things did not 'look right'. For example, one student recognized an error with missing braces by inspection and explained it as 'TA experience.'

Considering alternatives: Three students noticed that essentially the same error may have multiple causes. For instance, one remarked that an infinite loop might be caused by the loop controlling expression, or from a problem at the bottom of the loop.

Environmental: Some students took advantage of functionality made available by their environment, although it was not specifically debugging related. For instance one student made frequent use of the undo command. Another used comments to delineate what they had done and what should happen.

4.2 The Bad

Many of the strategies listed in Table 1 were, unfortunately, employed less effectively. Observing students provided some interesting insights into how misconceptions, faulty mental models, and unproductive behavior can impede their debugging ability.

Tracing: While used extensively, tracing was frequently used ineffectively. Students often inserted lines like println ("hello"), which although useful for following flow of control did not yield other information. One student put the same println in both parts of an if-else statement. Only a small number of students used the debugger, although that could be because these programs were fairly short.

Testing: Although students tested, it was often only with the data from the specification, or sometimes even less rigorous data. One student tested but did not note that the answers were still wrong. Another made a change but did not then compile or run. In an interesting take on robustness, a student noted that "stop" had to be input in lowercase for Raffle to terminate correctly but did not bother to alter the code despite finishing early.

Understanding the code: Although students read the code, it was unclear how hard they tried to understand it. One student read the code but then did not compile it or run it before trying to debug.

Using resources: The student who looked at old programs was essentially flailing – although this could be a useful technique.

Using tools: One student used an unfamiliar IDE because it just happened to open rather than seeking out a familiar one. This lack of familiarity resulted in changes not being saved correctly.

Isolating the problem: Several students commented out lines that looked 'suspicious' even though they were correct. Their *Pattern Matching* was working against them. Another removed blanks between String and [] in the main method header, thinking it was a bug. Another believed the scan.nextln() after reading an integer was unnecessary.

Work around problem: Sometimes students worked around a problem rather than facing it. They replaced code they did not understand with completely new code, or they changed the type of loop rather than understand why the current loop did not work. One student fixed a bug in Raffle (incorrect initialization of a count to 1 rather than 0) by subtracting 1 from the count after the loop. Another added a special case to handle 0 0 0 as triangle sides using an if statement in the loop rather than correcting the loop condition. This inelegant fix would have worked, but it only printed "exit" rather than actually exiting.

Just in Case: Some students did unnecessary things. This was not always a problem, for example, two found and deleted an unused variable. Others added unnecessary parentheses or extra braces. One fixed a spelling error in a comment (and nothing else).

Tinkering: This refers to fairly random and usually unproductive changes. For example, the spelling mistake in the comment is an example of tinkering. Students copied in large chunks of irrelevant code from other programs, or replaced an assignment with a '==', or compiled again even though they had made no changes. In another case, a visibly agitated student nearing the end of the exercise changed the only error they had correctly fixed back to the original code.

Some students demonstrated a general lack of understanding. One weak debugger, who was unable to find any bugs in Rectangle, seemed confused that the program compiled but did not run correctly. Another mistook a logic error for a 'syntax' error. A third did not compile for 10 minutes and a fourth appeared to randomly copy in code they thought might work.

4.3 The Quirky

Some student behaviors were rather surprising, affording us a source of amusement. Some of these have already been mentioned as examples of *good* or *bad* behavior.

equals() equals ==? – Three students, from the same institution, were unfamiliar with the equals() method. They were able to diagnose that the "==" was the source of an incorrect loop but could not fix it despite being told about .equals().

"Ah ha" understanding – One student exclaimed, "Oh, that's how DecimalFormat works".

The textbook, what is that again? – Students did not use printed resources. Few used paper and pencil at all. No student used the textbook. The electronic age is definitely here.

It's never too late to learn – Sometimes students used techniques that we had not thought of. One student tried entering blank names/values to diagnose a reading problem.

Sarcasm early – A student remarked sarcastically after compiling the program for the first time, "No syntax errors—fun!"

Two wrongs make a right – Instead of fixing an incorrect initialization, a student adjusts the final counter.

Robustness/schmobustness – Noting that the program only took a lowercase sentinel, a student moves on without any changes.

Hey, it compiles! – A student wonders why the program doesn't work correctly since it does compile.

It just doesn't look right – Several students changed correct code, added unneeded parentheses, or changed identifiers ... just because.

5. IMPLICATIONS FOR TEACHING

Students' productive strategies suggest topics for instruction. Along with learning debugging techniques, such as tracing and testing, students should be taught heuristics, or patterns, for applying techniques effectively. Examples for tracing might include:

- If you have to track more than one or two variables or there's a loop involved then you should trace on paper.

- If the bug can't be determined from the input and output then you need to add print statements.
- Make sure that your print statements are well-placed and print meaningful information.
- If you'd have to use many print statements, your program 'hangs' or has an infinite loop, use a debugger.

Many unproductive activities appeared to stem from insufficient meta-cognition. Some students did not recognize when they were stuck, thus they did not know to try a different approach. Those who stubbornly traced in their heads often did not realize they were suffering from cognitive overload and might be more productive if they tracked on paper or via print statements. Others were blind to alternative bug sources. Debugging instruction should incorporate these meta-cognitive factors, perhaps taking the form of self-questions: "What else could I try?', "Is this too much to keep track of in my head?", and "What are other possible sources of the bug?"

Our observations and student comments (e.g., "All students should do this – it is really good for you.") suggest benefits of offering similar opportunities in introductory courses. Students could debug a buggy implementation of a problem they had previously coded. Having everyone in the class debug the same errors offers a common context for class discussions. Students' difficulty with and observations about unfamiliar styles or implementations (e.g., "definitely way easier than the version I wrote."), also support this. Exercises could include new methods or different styles to help students learn to cope with unfamiliar code.

Since students clearly favor the electronic world, making resources available online and teaching the debugger at an appropriate time, likely in CS2, would also support their debugging efforts.

6. CONCLUSIONS AND FUTURE WORK

The opportunity to closely observe student debugging has not left us with a sense of doom and gloom. Despite patchy coverage in textbooks and (we confess) in teaching, students seemed to be familiar with and used many common debugging techniques.

Unfortunately, many students apply the techniques ineffectively or inconsistently. Testing is patchy and incomplete. Many seem unaware they should test more input than the specification outlines. Only one appeared to methodically test boundary conditions. Likewise, the use of print statements was not systematic.

Some 'strategies' were clearly ineffective. Several students 'tinkered' more or less aimlessly. Some worked around problems rather than really debugging or fixed things that were not broken. While better than nothing these are not good long term strategies.

Future work will compare observed strategies with those reported during student interviews. This study's findings could also be applied to pedagogical interventions designed to improve debugging instruction and the impact of those interventions studied.

7. ACKNOWLEDGMENTS

Thank you to Sue Fitzgerald for her superb leadership; to Jan Erik Moström and Umeå University for the VoIP system we used for our collaborative meetings; and to Sally Fincher, Josh Tenenberg and Marian Petre for starting us down this path. This work was supported in part by NSF DUE grant #0647688. Any opinions, findings, and conclusions or recommendations are those of the authors and do not necessarily reflect the views of NSF.

8. REFERENCES

[1] M. Ahmadzadeh, D. Elliman and C. Higgins. Novice programmers: An analysis of patterns of debugging among novice computer science students. *Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education (ITiCSE),* pp. 84-88, 2005.

[2] M. Ducasse and A.-M. Emde. A review of automated debugging systems: knowledge, strategies and techniques. In *Proceedings of the 10th international conference on Software engineering*, 162-171, 1988.

[3] S. Fitzgerald, B. Simon and L. Thomas. 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)*, pp. 69 – 80, 2005.

[4] S. Fitzgerald, G. Lewandowski, R. McCauley, L. Murphy, B. Simon, L. Thomas and C. Zander. Debugging: Finding, Fixing and Flailing –A multi-institutional study of novice debuggers. Forthcoming in *Computer Science Education* -- Special Issue on Debugging, 18(2), June 2008.

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

[6] L. Gugerty and G. Olson. Debugging by skilled and novice programmers. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, Boston, MA, April 13-17, pp. 171-174, 1986.

[7] I. Katz and J. Anderson. Debugging: An analysis of bug location strategies. *Human-Computer Interaction*, 3(4):351-399, 1987.

[8] C. Kessler and R. Anderson. A model of novice debugging in LISP, in Soloway, E. & Iyengar, S. (Eds.) *Empirical Studies of Programmers: First Workshop*, (pp. 198- 212). Norwood, NJ: Ablex Publishing Corporation, 1986.

[9] R. McCartney, A. Eckerdal, J.E. Moström, K. Sanders, and C. Zander. Successful students' strategies for getting unstuck. *ACM SIGCSE Bulletin, Proceedings of the 12th annual SIGCSE conference on Innovation and technology in computer sci. education (ITiCSE)*. Vol 39, Issue 3, pp. 156-160, 2007.

[10] R. McCauley, S. Fitzgerald, G. Lewandowski, L. Murphy, B. Simon, L. Thomas and C. Zander. Debugging: A review of the literature from an educational perspective. Forthcoming in *Computer Science Education* -- Special Issue on Debugging, 18(2), June 2008.

[11] M. Nanja and C.R. Cook. An analysis of the on-line debugging process. In G. Olson, S. Sheppard and E. Soloway, (Eds.) *Empirical Studies of Programmers: Second Workshop*, pp. 172-184, 1987.

[12] D. Perkins and F. Martin. Fragile Knowledge and Neglected Strategies in Novice Programmers. In E. Soloway and S. Iyengar (Eds), *Empirical Studies of Programmers: First workshop*. Ablex, NJ, USA. pp. 213-229, 1986.

[13] I. Vessey. Expertise in debugging computer programs: A process analysis. *International J. of Man-Machine Studies*, *23*, pp. 459-494, 1985.