AN INTEGRATED APPROACH TO INSTRUCTION
IN DEBUGGING COMPUTER PROGRAMS

BY

Ryan Chmiel and Michael C. Loui
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign
1406 W. Green St.
Urbana, IL 61801
{rchmiel,loui}@uiuc.edu

Abstract: This study demonstrates that formal training in debugging helps students develop skills in diagnosing and removing defects from computer programs. To enhance debugging skills in an assembly language course, students completed debugging exercises, debugging logs, development logs, reflective memos, and collaborative assignments. The debugging exercises were optional, but the other activities were mandatory. Students who completed the debugging exercises spent 37% of their time on debugging programming assignments, whereas students who did not complete the debugging exercises spent 47% of their time debugging. Students also provided qualitative data for each activity, and they responded to summative evaluation surveys. Students agreed that formal debugging training enhanced their debugging skills. This paper also proposes a model of debugging abilities and habits based on students' comments in their debugging logs, development logs, reflective memos, and evaluation surveys. Students and instructors can use the model to diagnose students' current debugging skills and take actions to enhance their skills.

Key words: computer science education, curriculum development, debugging, teaching, troubleshooting

I. INTRODUCTION

A. Debugging and the Software Industry

Debugging is a continual process of hypothesis generation and verification [1] in which programmers remove defects from computer programs. If a computer program does not work according to its specification, programmers must debug the code and correct the defects. Three types of defects can occur in a computer program: syntax, semantic, and logic. Syntax errors are identified by a compiler or interpreter; however, semantic and logic defects must be identified by the programmer.

Debugging is an arduous task on which programmers spend a considerable amount of time [2]. Some defects, such as off-by-one errors in a loop, are easy to identify; more complex defects, such as race conditions in multithreaded programs, may take a substantial amount of time to correct. Furthermore, defects have varying degrees of severity. For example, a defect that formats output incorrectly is much less severe than a defect that corrupts data.

Programmers should strive to identify and correct every defect that they inject into their programs. Unfortunately, no commercial software product is defect-free. For example, a case study of the Apache web server found that 2.64 defects per thousand lines of code existed in a post-release version of the software [3].

To handle defects, many software companies release product updates. Companies occasionally release updated versions of their products to correct defects and add new features. Sometimes, companies also provide a detailed list of the changes from one version to the next so consumers can learn which defects were corrected and which features were added. However, is this solution acceptable? Why should consumers purchase a product with known defects?

While perfect debugging is theoretically possible, achieving it in industry is impossible because of the associated costs [4]. These costs include the salaries of the software testers and developers, the

loss in profit due to the delay of the software release, and so on. To find a balance between cost and perfect debugging, companies could use cost models to determine the level of perfection that they want to obtain.

A cost-effective method of improving the quality of software could be to train programmers to become better debuggers. In doing so, programmers could enhance their debugging skills to identify and correct defects more effectively. The task, then, becomes determining the best methods of teaching debugging skills to programmers.

B. Overview of Findings

In this study, students enhanced their debugging skills by completing carefully planned debugging activities. Students who completed optional debugging exercises spent significantly less time on debugging the programming assignments than students who did not complete the exercises. Specifically, students who completed the exercises spent 37% of their time on debugging, whereas students who did not complete the exercises spent 47% of their time on debugging. In addition, students provided feedback on the activities by completing a summative evaluation survey. Overall, the qualitative data supported the results of the quantitative data. Students agreed that the debugging activities enhanced their debugging skills.

The students' comments led to the development a model of debugging habits and abilities. This model, which is outlined in section V, characterizes a student's development of debugging skills from novice to expert. Students and instructors could use the model to diagnose students' debugging skills and take actions to enhance their skills.

II. REVIEW OF LITERATURE ON DEBUGGING

A. Software Testing

Before debugging a program, a programmer must first determine that a defect exists. According to Whittaker [5], software testers face several difficulties through testing. One difficulty is that the input domain to any program is extremely large; therefore, testers cannot possibly test all input values. Instead, testers should strive to cover the input domain as best as possible by selecting inputs with the same statistical properties as the entire domain. Similarly, a program may contain an extremely large number of execution paths, so testers may not be able to ensure that they test each line of code at least once. Again, testers should choose a set of execution paths that represents the input domain.

Developing test cases is difficult and tedious. When developing test cases, testers struggle to balance quality and cost. A large set of test cases takes a considerable amount of time to execute. A large test set may be necessary because of the complexity of the program; however, if the test set contains many similar test cases, the tester adds unnecessary cost to the testing process. Testers must also balance between quality and cost in regression testing, in which a new version of a program is tested with the same test cases used on previous versions. From the tester's perspective, the best approach is to run all the previous tests before running the new tests on the new program. This approach would ensure that programmers who updated the code for the new program did not introduce or bring back defects that were non-existent or corrected in the old program. Unfortunately, this approach may not be cost-effective because of the considerable time and resources that these tests could require [6].

Jones [7] integrated software testing into a computer science curriculum. He created classroom activities that introduced students to the different phases of the software testing life cycle. These activities include testing and grading another student's program and writing test cases for a program based on its specification before coding it. Jones also designed a Software TestLab facility for students

to enhance their skills further. In the TestLab, students performed supervised testing activities, and their results were stored in the TestLab test bed for use by other TestLab users. Although Jones did not provide any evidence from students regarding the effectiveness of the curriculum enhancements or the Software TestLab, his approaches seem that they could enhance debugging skills.

It is important for software testers, who may or may not be programmers, to have software testing skills in order to be effective. If a tester cannot determine that a program is defective, a programmer will be unable to debug it altogether.

B. Demonstrated Need for Debugging Training

Although debugging is an integral and time-consuming aspect of software development, computing curricula rarely give formal debugging training any attention. The computing curricula proposed by the Association of Computing Machinery and the IEEE Computer Society [8] make little reference to the importance of debugging. These curricula mention that computing courses should discuss debugging strategies but do not include guidelines or methods of formal debugging training.

Because computing curricula rarely include training in debugging, students are left to develop debugging skills on their own. Students begin debugging, and thus developing debugging skills, as novices, with limited abilities in formulating hypotheses about the possible defects in their code. Therefore, it seems logical that students who receive formal debugging training at an early stage would become better debuggers more quickly. During this training, students would gain debugging experience, which could improve their debugging skills. Students with more debugging experience could become better debuggers because they could rely on experiences to assist them. Thus, students should be given carefully planned debugging exercises as early as possible.

Spohrer and Soloway [9] analyzed the defects in novice student programs in an introductory programming course. They obtained the first versions of programs submitted for compilation for each of

the ten programming assignments in the course. Spohrer and Soloway selected three of the ten assignments for their study: an assignment early in the course, a later assignment, and the first assignment that required students to use loops. Spohrer and Soloway counted the total number of defects and the number of distinct defect types in the programs.

Spohrer and Soloway observed that many novice programmers inject high-frequency defects into their programs. In an even distribution, a certain percentage of defect types should yield the same percentage of total defects for a particular program. Instead, Spohrer and Soloway found that the most common 50% of defect types, such as off-by-one and incorrect logic expression defects, accounted for a range of 77% to 84% of total defects for the three programs. Similarly, the most common 20% of defect types accounted for a range of 46% to 64% of total defects. Debugging training that emphasizes high frequency defects might help students reduce the occurrences of these defects in their programs.

Benander et al. [10] showed that programmers debug recursive code better than iterative code in approximately the same amount of time. In their experiment, more than 250 subjects were split into two groups of equal size. One group received an iterative piece of code to debug, while the other group received the recursive version of the code, which performed the same task as the iterative version. Benander et al. found that 63.2% of subjects who debugged the recursive version located the defect, but only 41.5% of subjects who debugged the iterative version located the defect.

Programmers can benefit from these findings because they can write code using programming constructs that they can more easily debug. Furthermore, programmers can rely on their individual experiences to determine which programming constructs they should use. Programmers who tailor their code to their personal strengths may improve the debugging process because they eliminate opportunities for defect injection.

C. Methods to Improve Debugging Skills

How can students become better debuggers? Instructors can use a combination of methods in order to teach students how to debug their code effectively and enhance their debugging skills. Some of these methods are discussed below.

1) Code Review and Code Inspection: Code review is an effective method for debugging [11]. In a code review, a programmer analyzes the code individually after the initial coding phase and before the compilation phase in order to locate defects. A programmer often reviews the code by printing out the program source code and analyzing every line. A programmer may read a printout more easily than a listing on a monitor, and the printout allows the programmer to make notes while analyzing the code. By reviewing the code before compilation, a programmer can increase the effectiveness of the review because code review encourages programmers to identify logical defects, whereas compilation identifies syntactic defects.

Code review is time-consuming. Statistics show that a programmer typically spends 30 minutes analyzing 100 lines of code. Thus, a code review for a longer program would require a considerable amount of time. According to Humphrey [11], however, the benefits of code review outweigh the costs. Programmers who perform code reviews identify a majority of total program defects, and they are able to correct the defects more quickly than those programmers who do not perform code reviews. Thus, a programmer should invest time in performing code reviews in order to become a more effective debugger.

Fagan [12] introduced the code inspection method. In a code inspection, a team of programmers collectively identifies defects in one piece of code. Fagan suggested that each team member take a unique role. For example, one team member could be responsible for testing the code to determine whether a defect exists. Fagan also suggested that the optimal team size is four members.

Teams can benefit from code inspections because the team can share ideas about possible defects. Code inspections lead to the generation of more hypotheses and a more detailed analysis of these hypotheses than an individual code review. In addition, the other team members may identify defects overlooked by the author. Thus, the total number of man-hours spent by a team on debugging a piece of code may be less than the number of hours spent by an individual. In all, code inspections can reduce the number of defects in code.

Gilb and Graham [13] presented a code inspection case study conducted by a senior software engineer at Applicon, Inc., which showed positive results. The engineer instituted code inspections at Applicon where software engineers previously performed no code inspections or other collaborations. Employees interested in becoming inspection team leaders received training in code inspection methodologies. Then the team leaders conducted inspections on software currently in development.

McDowell et al. [14] showed that code inspections are also effective in the classroom. They conducted a study using two sections of the same course taught by the same instructor. In one section, students completed the programming assignments in pairs. McDowell et al. encouraged student teams to work together on all assignments in the course. In addition, students in each team alternated "driver" (student at the computer) and "nondriver" roles every hour while completing the assignments. In the other section, students completed the assignments individually.

McDowell et al. compared average assignment scores and final exam scores in both sections. Students who programmed in pairs had an average assignment score of 86%, while students who worked alone averaged 67%. To show that the scores of students who programmed in pairs were not solely the result of the stronger programmer in the team, McDowell et al. calculated the average assignment grade for the top half of students in the nonpaired section. The average score of this subset was 77%. Similarly, McDowell et al. compared average final exam scores in both sections. In the paired section,

students averaged 72.9% on the final exam, while in the non-paired section students averaged 74.6%. An ANOVA showed that this small difference was not statistically significant, however. They concluded that pair programming results in improved programs but does not affect a student's mastery of course material.

2) Program Comprehension: Gugerty and Olson [15] hypothesized that program comprehension determines a programmer's ability to debug code. They conducted an experiment in which a group of novice programmers and a group of expert programmers first received a Pascal program and read it for comprehension. Gugerty and Olson classified programmers in their first or second Pascal course as novices and advanced graduate students in computer science as experts. Gugerty and Olson administered a posttest that measured each group's ability to comprehend the program. They found that experts answered correctly 11.3 of 13 questions on average, while novices answered only 8.5. In the second part of the experiment, the programmers received another program containing one defect, which they were to identify. Gugerty and Olson found that the expert programmers were more successful in identifying the defect and could do so more quickly. In addition, Gugerty and Olson found that experts tested half as many hypotheses about the cause of the defect as novices did, even though both groups spent the same percentage of time reading the program.

From these results, Gugerty and Olson concluded that an expert programmer's superior program comprehension ability results in greater success in debugging code. Since experts can better understand a program, they make higher quality hypotheses than novices, and as a result, experts have fewer hypotheses to test. It is possible, however, that Gugerty and Olson could have mistaken correlation for causation: debugging skill may be correlated with program comprehension rather than caused by it.

Rifkin and Deimel [16] conducted a case study at a large-scale manufacturing company which, in part, develops software for engineering computations. Two groups of employees regularly performed

code inspections. A third group, whose members at one time belonged to one of the two previous groups, attended a workshop on code inspections that specifically targeted program comprehension techniques. Workshop attendees spent time understanding programs and analyzed the benefits of program comprehension. Rifkin and Deimel counted the number of customer-reported defects for software maintained by each of the three groups. Rifkin and Deimel noticed that the group of workshop attendees showed no noticeable improvement until ten days after the workshop. Thus, they selected this ten-day period as the baseline period for their results, and they presented defect rates as percentages of the defect rates during the baseline period.

Rifkin and Deimel reported a dramatic decrease in the number of defects over time for the group of workshop attendees. A noticeable decline in the number of defects began on day 11, and by day 41, the number of defects reached a constant level of 10% of the baseline defect rate. On the other hand, the other two groups showed no improvement in the number of defects over time. Rifkin and Deimel concluded that program comprehension skills significantly improve code inspections in debugging.

3) Learning through reflection: Self-reports of programming experiences could help programmers enhance their debugging skills. By recording defects, the programmer could notice trends or multiplicities in the types of defects that he or she injects. This analysis could prompt the programmer to pay special attention to avoid injecting these defects while developing code. Unfortunately, documented self-reports are rare. One notable exception is the logbook [17] that Knuth kept over a ten-year period while developing TeX. In his logbook, Knuth recorded every defect he encountered. While Knuth did not include comments on the benefits of keeping the logbook, some entries referred to defects previously made.

Korgel [18] incorporated journal writing into a core, junior-level chemical engineering course. Approximately once a week, students wrote a journal entry in the form of an analogy to describe

10

previously covered material or to explore new concepts. For example, one student compared Raoult's Law to a dinner party. Korgel encouraged creativity in the student journals and sought to promote self-confidence in the students by awarding extra credit to the best journal entry voted on by the class. In addition, since students shared their journals with each other, they could learn course material through many different interpretations.

Korgel found that student journals promoted a deeper understanding of course material and encouraged students to create conceptual links between course material and life experiences. One student stated that the reflection aspect of journal writing forced the student to apply course material to the student's everyday life. Thus, students gained self-knowledge through writing their journal entries. Student feedback confirmed this claim. Overall, the inclusion of journal writing into this course allowed students to obtain a deeper self-knowledge.

D. Debugging Tools

Programmers can rely on debugging tools for assistance. While the complexity of these tools varies considerably, all can be helpful to a programmer.

Primitive tools provide programmers with a starting point in the identification of defects. One such tool is a dump of the processor registers, program stack, and sections of computer memory. Some operating systems provide this information if a running program throws an exception and terminates unexpectedly. From this information, a programmer can determine where in the machine code a program crashed, and the programmer may then use a more sophisticated tool to identify the incorrect statement in the high-level code. A similar technique is the use of **print** statements to display program variables. With **print** statements, values are displayed at run-time rather than when the program terminates. Programmers who use **print** statements can track variable changes in order to determine the point at which the program no longer behaves correctly.

Software debuggers are computer programs that run other computer programs. Examples of commercial debuggers include Turbo Debugger, Microsoft Visual Debugger, and GDB (The GNU Project Debugger). Debuggers allow programmers to execute lines of code one at a time, set breakpoints in the execution of the code, and view and modify variable values at run-time. These features enable a programmer to observe the current state of the program in order to identify defects.

Lee and Wu [19] developed DebugIt, an interactive tool to help novice student programmers improve their debugging skills. The first release of this tool, DebugIt:Loop, helped students identify and correct defects related only to loops. Using DebugIt, a student received a problem containing a faulty piece of code and solved the problem by correcting all of the defects in the code. A student who needed assistance could view optional hints. In addition, if a student did not successfully solve the problem in three attempts, the student received the solution and explanations of the defects. Students who did not solve a problem correctly still gained debugging experience because they could refer to the solution. Learning from their mistakes, the students should be better able to identify the same defects in the future.

Lee and Wu conducted an experiment in which two groups of subjects practiced debugging and then took a posttest to determine differences in the debugging and programming abilities of the groups. The treatment group used DebugIt:Loop to practice, while the control group practiced using traditional methods such as hand tracing. Subjects in the treatment group performed better on the posttest. Lee and Wu concluded that DebugIt:Loop is effective in enhancing the debugging skills of novice programmers.

Zeller [20] expanded the capabilities of automated software testing to allow for automated debugging. Zeller developed the Delta Debugging algorithm, which uses results from automated tests to reduce the set of possible defective code statements. Delta Debugging analyzes the differences in the

results and can identify failure-inducing errors in program input, user interactions, and recently modified code statements. Delta Debugging leaves the task of correcting defects up to the programmer, however.

Zeller and his colleagues used Delta Debugging to debug the Mozilla open-source web browser. They obtained and reproduced a defect from Bugzilla, the Mozilla bug database. Delta Debugging identified the incorrect HTML statement out of an 896-line HTML file in approximately 20 minutes. In addition, Delta Debugging simplified the set of user interactions from 95 to 3. After these simplifications, Zeller et al. successfully identified the cause of the defect.

Delta Debugging and other automated debugging tools could become alternative debugging methods for programmers. These tools could relieve programmers of their debugging duties, but they may have some negative side effects. For example, programmers may become too dependent on automated debuggers. As a result, programmers could become careless with their code because they know the debugger would identify any defects. In addition, programmers could notice declines in their program comprehension abilities for the same reason.

Although helpful, debugging tools may not assist programmers in all situations. Eisenstadt [21] collected stories of other programmers' defects by posting a request for them on an Internet newsgroup. After analyzing the stories, he found that approximately 25% of the defects rendered debugging tools inapplicable. For example, programmers could encounter these defects while debugging race conditions in multithreaded programs. Thus, the programmers needed to correct the defects without the assistance of debugging tools.

III. ACTIVITIES TO ENHANCE DEBUGGING SKILLS

After considering the various ways in which programmers become better debuggers, the following activities were developed to help students improve their debugging skills. These activities were introduced in ECE 291, a course on assembly language and real-time computing that is required

for juniors in computer engineering and an elective for electrical engineering students at the University of Illinois at Urbana-Champaign [22]. Core components of the course include five regular programming assignments, which range from 200 to 1000 lines of code as the course progresses, and a team-based final project, which ranges from 3000 to 4000 lines of code. Although few students in ECE 291 have previously programmed in assembly language, all have experience in at least one programming language.

Activities related to program design with emphasis on minimizing defects were not included to avoid inundating students with too many activities or overburden them in one course. These actions could cause them to become discouraged with the activities. The chosen activities acquainted students with a variety of debugging techniques whose effectiveness could be evaluated. It would be difficult, say, to measure a student's improvement in debugging while the student enhanced his or her program design skills simultaneously.

A. Debugging Exercises

Two sets of debugging exercises were created for each of four of the five regular programming assignments before the final project, starting with the second assignment. Without completing the exercises for the first assignment, students could become motivated to undertake the debugging exercises. A novice assembly language programmer could especially benefit from the exercises.

The exercises were an optional part of the course because of federal regulations on experimentation with human subjects; students could start or discontinue participation at any time. Students were advised, however, that spending approximately two hours on each set of exercises could save dozens of hours in debugging time. Students were encouraged to work on each set of exercises before beginning the corresponding programming assignment. As compensation for their efforts, students could complete the debugging exercises as a method of obtaining a small amount of extra

credit. Alternatively, students could earn the same amount of extra credit by completing the programming assignment before its respective due date. Thus, participation in the exercises was not coercive: students decided to spend extra time on either completing the exercises or finishing the assignment early. In addition, students who completed all sets of debugging exercises received gift certificates to the university bookstore.

Each set of debugging exercises contained two types of problems, and either one or two problems of each type were present in each set of exercises. For the first type of debugging problem, students received short subroutines (about 20 instructions long) with a stated number of defects, ranging from three to five defects per problem. The students were not told the types of defects present in the subroutines, however. The students solved the problems by hand, with no outside resources, by performing code reviews. These problems were designed to enhance the student's code review and program comprehension skills, as analyzing code would help a student understand the behavior of the code. As a result, students could develop a process for identifying defects in code. For the second type of debugging problem, students received a file containing the source code to faulty subroutines. In addition to identifying the defects in these subroutines, the students corrected the defects. Thus, the students continually modified and tested the code until each subroutine worked as specified. Students could also use the Turbo Debugger tool, which allows students to single step through lines of code, set breakpoints in code, and display the contents of memory and processor registers.

Originally, the debugging exercises did not contain the second type of problem described above. Instead, they contained another type of problem, which described a situation and then asked students to identify the defects. An example situation was, "John edited the source code to his program in an attempt to correct a defect and then reran the program; however, he did not notice any change in the program output." One acceptable solution to this problem was that John forgot to recompile his code.

15

While these problems were initially appealing because students could identify the defect in a programmer's thought process, the code modification problems were selected for many reasons. First, it was difficult to create effective situation problems, as they focused on high-level concepts rather than types of defects. Second, one could argue that students would benefit more from the code modification problems because they better prepared students for correcting defects directly in the source code, a task they would perform during the programming assignments. Third, the code modification problems provided immediate feedback because students would know when the subroutine worked correctly on a test input. Last, students could better transfer the skills learned from the code review problems to the code modification problems.

The defects in each set of debugging exercises were related to the major topics covered by the corresponding programming assignment and the common defects that a student might make while completing the assignment, such as off-by-one iteration in a loop, wrong jump condition, and addressing mode error. However, more obscure defects, such as rarely occurring algorithm boundary cases, were also included to give students practice in locating them as well. The number of defects per problem also varied. Earlier sets of exercises contained problems with fewer defects to give students confidence in their debugging skills. The number of defects per problem gradually increased with later exercises. If students became better debuggers throughout the course, they would be able to identify more defects in subroutines of the same length.

There are many benefits to developing the exercises in this manner. First, students were encouraged to use code review as an initial method of defect identification. Students who first completed the "solve by hand" problems (which require code review) could develop a habit of performing code reviews when solving the computer-assisted problems in which they could otherwise immediately begin modifying code. Second, students received exposure to many types of defects with various levels of

difficulty. Some defects are much more difficult to identify than others, and the problem solving skills that students developed to identify these defects could be useful in identifying future defects of any difficulty level. Third, relating the exercises to the programming assignments improves the training process for students. It would be foolish, say, to test a student's ability to debug graphics code when the current assignment has nothing to do with graphics programming.

B. Debugging Logs

Students kept a debugging log as they worked on the programming assignments in the course. The logs allowed a student to document his or her defects to aid in future debugging, and the student would spend little time in order to maintain his or her log. A defect could be entered into the log in one or two minutes.

The log was modeled after a log developed by Humphrey [11]. For each defect, the student recorded the following information in the log: the name of the subroutine containing the defect, the time taken to correct the defect, the incorrect program output or behavior (if not discovered during code review), the faulty code, and most important, the solution to the defect. Humphrey's log does not ask for the solution, but it is a useful addition to his log. When a student repeated a defect, the log reminded the student how to correct the defect, thus saving time.

After analyzing his or her logs, the student could create a personal checklist to use when he or she encounters a new defect. Creating personal checklists is beneficial because a student can gear the checklist according to the types of defects that the student injects into his or her programs. A standard checklist may not be as effective because the student may only inject a subset of the defects mentioned in the standard checklist. Furthermore, personal checklists could be modified throughout the course. A student could also use the log to document improvement in debugging skills throughout the course.

C. Development Logs and Reflective Memos

The course instructor required all students in the course to document their programming experience in a development log with each assignment. In the development logs, students documented their design decisions, development plans, and overall debugging experiences. The students also included the time spent on different phases of each assignment (design, coding, testing, etc.).

Based on the development log, each student submitted a reflective memo of 200 to 400 words along with the code for each assignment. In the memos, students answered the following questions:

- How much time did you spend on the design, coding, and testing of each part or subroutine?

- What kinds of defects did you find during the development of the program? When did you discover these defects (during code review or during testing)? How did you find them?

- What you would do differently for the next programming assignment?

D. Collaborative Assignments

The course instructor converted the last regular programming assignment to a team assignment. Students were assigned to teams of four and worked collaboratively on the assignment for two weeks. Team members were encouraged to code different subroutines but to assist other team members when necessary. When a team member finished coding his or her designated subroutines, the entire team would perform a code inspection to identify defects before executing the program. Then the team would work together to correct the defects discovered after executing the program. Through this approach, each team member would still have a working knowledge of the entire program because of these reviews even though he or she coded only part of it.

These teams remained together to complete the final project for the course, which since 1994 has been a team project of substantial size, usually a video game. Final projects average 3000 to 4000 lines of code and typically contain graphics, game play logic, sound, and networking. Again, teams were

encouraged to use code inspections for the final project. With an opportunity to work with their teams before the final project, students could organize themselves better for the final project.

IV. RESULTS AND ANALYSIS

In the fall of 2002, these activities were piloted in ECE 291 to obtain preliminary qualitative results. Overall, students responded well to the activities. Students felt that the exercises helped improve their debugging skills. Furthermore, students suggested improvements in the exercises. For example, they suggested that the number of defects per code review problem be decreased. In the fall, each code review problem contained five defects, and students became frustrated when they could not identify all five defects in a subroutine of only 20 instructions. After considering their suggestions, the exercises were modified accordingly for the spring of 2003. The following results were obtained from a study conducted in the spring 2003 offering of ECE 291. Of the 116 students enrolled in ECE 291 during the spring 2003 offering, 27 students participated in the study.

A. Data Collection

In order to determine the effectiveness of the activities, students completed summative evaluation surveys. The survey included statements for participants to evaluate different aspects of each activity. The participants rated each statement on a five-point scale, with 1 corresponding to "strongly disagree" and 5 corresponding to "strongly agree." One exception is a statement that asked participants to state the number of hours they typically spent on one set of debugging exercises. The survey also included a narrative response question in which participants commented on the strengths, weaknesses, and areas for improvement for the activities. Students' comments were analyzed for similarities.

The quantitative data collected from students' debugging logs and development logs includes:

- The number of defects per assignment
- The time needed to correct each defect

- The total time spent on debugging each assignment

- The time spent on the design, coding, and debugging phases of development

- The total time spent on each assignment

B. Debugging Exercises

Students provided qualitative data regarding the effectiveness of the debugging exercises through summative evaluation surveys. Table 4.1 contains the survey results.

Table 4.1. Survey Results for Debugging Exercises

Survey Statement	Result
Amount of time spent to complete one set of exercises, in approximate hours.	2.1
Each set of exercises contains defects relevant to the corresponding assignment.	4.0
I watch out for the defects I discover while completing the exercises when I code the corresponding assignment.	4.0
The level of difficulty of the exercises is reasonable.	4.2
I perform code reviews before compilation to identify and correct defects.	3.9
When I encounter a defect while coding the assignments, I first analyze the code carefully in order to locate any defects instead of immediately modifying the code.	3.3
As a result of completing the exercises, I have developed a process for identifying defects in the assignments.	3.8
I feel that the exercises are enhancing my debugging skills.	4.3

Overall, students felt that the exercises enhanced their debugging skills, and they spent the intended amount of time on each set of exercises, two hours on average. In addition, many participants developed processes, such as performing code reviews, for identifying defects in their code.

Students also provided many good comments and suggestions on the exercises, some of which are included below.

- "There are certain things in [the code review exercises] which I think are correct/incorrect. Those same things appear in [the code modification exercises], and then I realize whether I was right/wrong. This helps me a lot in rectifying my concepts."

- "It gives me a chance to face some defects that could occur in my program."

- "They provided a way to develop an analytical approach to debugging."

- "The exercises don't enhance my debugging skills while programming as much as on paper. The programming ones should be designed better and should cover some more interesting things."

- "Include more [code modification exercises], as it is rather easy to pick errors out when you know there is a set amount to look for."

- "At times it was difficult since you did not write the code in the first place."

The constructive comments indicate how the exercises could be improved. The summative evaluation will help create even more effective exercises for future students.

C. Debugging Logs

Students submitted debugging logs after completing each programming assignment. These logs contained the number of defects they injected into the program and the total time they spent on debugging the program. Since the programming assignments in the course became longer and more difficult as the course progressed, the number of defects per program and time spent debugging each program could not be used as reasonable performance metrics. Instead, students' performance was evaluated through the time spent per defect and the number of defects per lines of code. Ideally, students would notice declines in both of these statistics as the course progressed.

Unfortunately, analysis of the quantitative data from the debugging logs was inconclusive. Many students did not submit detailed debugging logs with their assignments; instead, they submitted responses to the general reflective memo question about defects. The students who submitted detailed logs did not do so consistently, so no improvement in skills could be observed. One student even stated, "Please stop making us submit debugging logs. They are really very irritating, and I don't find them even a bit helpful." Students often fail to realize that finding the correct answer is not the only important

aspect of solving problems; they also need to become aware of how they solved the problems and how they can adapt their problem solving skills to deal with more difficult problems [23].

In the future, course instructors will explain the benefits of debugging logs more clearly to students. Many students did not keep the logs as instructed. Perhaps students were unreceptive to the paper format; it may be easier for students to keep the log electronically. In addition, there was no evidence that students who kept debugging logs used them to create personal debugging checklists. While this information was not specifically requested, students did not mention the creation of personal checklists in their reflective memos.

D. Development Logs and Reflective Memos

Students reported the number of hours they spent on each phase of the development process in their development logs. After analyzing students' development logs from the first four assignments, the percentage of time students spent on debugging was calculated for each assignment; the fifth assignment was a collaborative assignment, and it was difficult to collect reliable data for individual students. The data was separated into two groups: the treatment group contained students who had completed the debugging exercises, and the control group contained students who had not completed them. Table 4.2 contains the percentage of time each group spent on debugging their assignments.

Table 4.2. Percentage of Time Spent on Debugging by Treatment and Control Groups

Assignment	Treatment	Control	Significance
1	42%	43%	$p < .890$
2	38%	52%	$p < .001$
3	35%	43%	$p < .002$
4	36%	49%	$p < .002$

T-tests were used to determine the significance of each difference. For the first assignment, the null hypothesis was accepted, which stated that there was no difference in debugging skills between the two groups. This result was expected because neither group had received any formal debugging training;

thus, the percentage of time that each group spent on debugging should have been almost identical, and this observation was confirmed by the t-test. On the later assignments, for which the treatment group completed optional debugging exercises, there were statistically significant differences in the percentage of time spent debugging. These results led to the rejection of the null hypothesis for these assignments. These differences could be the result of the treatment group's enhanced debugging skills, which resulted from completing the debugging exercises.

Although the treatment group spent a smaller percentage of time on debugging than the control group, this statistic does not show whether the treatment group also spent less actual time on debugging the programming assignments. Thus, the total time students spent on each assignment was also calculated. The treatment group spent approximately one hour less on each assignment than the control group, with average completion times for each assignment ranging from 10 to 25 hours. None of these differences, however, was statistically significant. This lack of significance could result from the large variances of the data sets. Students in each group were not consistently spending the same amount of time on each assignment.

Students' exam scores were analyzed to determine whether there was a difference in aptitude between the groups. Scores from the first midterm, which was administered between the due dates of the first and second programming assignments, were used as a pretest, and the final exam scores were used as a posttest. On the first midterm, the treatment group averaged 70.7% while the control group averaged 72.0%. On the final exam, the treatment group averaged 78.6% while the control group averaged 74.0%. T-tests showed neither of these differences to be statistically significant. Thus, there was no noticeable difference in the aptitude of the two groups.

To some extent, differences in program design abilities between the two groups could be disregarded as well. For each assignment, the course instructor or a teaching assistant specified the

major subroutines. Each assignment included a description of each subroutine, including its purpose and input/output values. In addition, the assignment contained pseudocode to any complex algorithms. Thus, a portion of the program design was already completed for students.

From these observations, it appears that the improvement in debugging shown by the treatment group over the control group is directly related to their completion of the debugging exercises rather than differences in aptitude or in program design skills. Students who completed the debugging exercises spent significantly less time debugging their programs than those who did not complete the exercises.

Students provided qualitative data regarding the effectiveness of the reflective memos through summative evaluation surveys. Table 4.3 contains the survey results.

Table 4.3. Survey Results for Reflective Memos

Survey Statement	Result
I kept a development log while coding the programming assignments.	3.3
In writing the reflective memo, I observed strengths and weaknesses in my approaches to designing, coding, and debugging the programming assignments.	3.5
I feel that the development logs and reflective memos helped refine my approach to programming assignments.	3.3
I adapted my approach to the programming assignments based on the observations made in my reflective memos.	3.7

Students' responses were both positive and unexpected. Many students complained about having to write the memos, as they felt it was a superfluous task after completing the assignments. Ironically, the same students also stated that they benefited from the memos. By documenting their experiences, many students noticed areas for self-improvement in their approach to the assignments.

One common theme among many memos was the lack of time spent on the design phase of the program. Many students noticed that they rushed into the coding phase too quickly, and as a result, they spent more time on the coding and testing phases than necessary. Memos for later assignments revealed that students spent more time proportionately on the design phase; thus, they adapted their strategies for

program development accordingly. The survey results confirm this finding because a majority of students agreed that the memos helped adapt their approach to the assignments. Overall, students felt that the memos were effective in documenting their approaches not only to debugging but also to program development as a whole.

E. Collaborative Assignments

Students provided qualitative data regarding the effectiveness of the collaborative assignments through summative evaluation surveys. Table 4.4 contains the survey results.

Table 4.4. Survey Results for Collaborative Assignments

Survey Statement	Result
Our team for the last programming assignment worked collaboratively and performed code inspections.	4.0
The code inspections allowed us to locate defects more quickly.	4.1
It was easier to debug code as a team rather than individually.	4.3
I feel that debugging the last programming assignment and the final project as a team enhanced my debugging skills.	3.9
I feel that debugging the last programming assignment and the final project as a team enhanced my teamwork skills.	4.2

Students benefited from the collaborative assignment. Many student teams collaborated on the assignment as encouraged. Students agreed that debugging in teams is more effective than debugging alone. Many students stated that because their team members located defects that the author had overlooked, the team debugged the assignment faster than an individual could. In addition, students felt that they enhanced their debugging and teamwork skills by completing the assignment collaboratively.

Students also stated that it was easier to complete the final project by first gaining experience with their teams on the collaborative assignment. Students observed the group dynamics and each other's strengths and weaknesses in order to adapt their roles in the group. This process enabled the groups to work more effectively on the final project.

V. A MODEL OF DEBUGGING ABILITIES AND HABITS

Students' comments from debugging logs, development logs, reflective memos, and evaluation surveys were used to construct a model that describes the progression of students' debugging abilities and habits. The model is based on the Dreyfus model of skill development [24]. For each stage of the Dreyfus model, the typical actions and emotions of students observed from the students' comments are listed.

A. The Dreyfus Model

The Dreyfus model of skill development contains five stages: novice, advanced beginner, competent, proficient, and expert.

Novices make observations in a context-free manner, i.e., with no attention to the overall situation. They follow the rules or specifications exactly without challenging them. Novices have difficulty in dealing with mistakes, and they become frustrated and confused quickly. Instead of giving up because of frustration, ***advanced beginners*** remember their experiences for future reference. These experiences lead to situational learning. When advanced beginners encounter a similar situation in the future, their experiences influence their course of action. ***Competent*** individuals organize these experiences into formal decision-making processes. They also construct conceptual models that aid them in their decision-making. ***Proficient*** individuals, who take the conceptualization one step further, seek to understand the overall situation. In addition, they strengthen their decision-making abilities by optimizing their processes. Since ***experts*** have a plethora of experiences from which to draw, they are able to act on intuition; they are not limited by the rules. Experts handle complex situations successfully. Furthermore, they are able to teach other individuals and offer them guidance.

Students entering ECE 291 could be considered novice debuggers because ECE 291 is typically the second or third programming course taken by electrical and computer engineering students. Thus,

they have limited debugging experience. Furthermore, few students have had prior assembly language experience, so debugging assembly language is a new experience for nearly all students.

B. Model of Debugging Abilities and Habits

Table 5.1 presents the model of debugging abilities and habits. The table shows each stage of the Dreyfus model in terms of the abilities and habits of debuggers at that stage.

Table 5.1. A Model of Debugging Abilities and Habits

Stage	Debugging Abilities and Habits
Novice	• Lacks debugging skills • Repeats the same types of defects frequently throughout a program • Debugs programs haphazardly instead of following a plan • Spends considerable amounts of time on debugging • Gives up easily and depends on others for assistance
Advanced Beginner	• Develops debugging skills through experience; recognizes symptoms of previously experienced defects • Repeats the same types of defects occasionally throughout a program • Begins adapting approaches to debugging based on prior successes and failures • Attempts less familiar debugging techniques only as a last resort • Relies to some extent on others for assistance
Competent	• Knows a variety of debugging techniques • Approaches debugging systematically; evaluates the situation to determine which techniques to use • Alternates between techniques when certain techniques are not effective • Identifies most defects independently
Proficient	• Sees debugging as part of program development as a whole • Develops skills in other areas of program development to facilitate debugging • Optimizes decision-making abilities and approaches to debugging • Asks others for assistance rarely and assists others
Expert	• Approaches debugging intuitively due to extensive experience; debugging is second-nature • Identifies complex and/or unfamiliar defects successfully • Asks others for assistance rarely and assists others

The model illustrates how students' comments exhibited the characteristics of the Dreyfus model. Through debugging experience, students should enhance their debugging skills, and they should

progress to the advanced stages of the model. In some situations, however, students' behaviors may be consistent with multiple stages of the model within the development of a single program. For example, a student could show competence in identifying simple defects, but the student could be classified as a novice when identifying complex defects.

Overall, students felt more confident in their debugging abilities as the course progressed, although it is unlikely that students became expert debuggers after completing the debugging activities. Instead, the students' comments suggested that most of the students showed competence in debugging by the end of the course.

C. Student Responses for Each Stage of the Dreyfus Model

This section includes some of the student comments used in creating the model. The comments are organized by the different stages of the Dreyfus model to highlight their differences. The numbers in parentheses after each comment are the programming assignment numbers from which each comment came.

1) Novice:

- "The hardest part of this [assignment] was figuring out if my code was actually doing what I thought it was doing." (1)

- "I made a lot of stupid mistakes." (1)

- "I think I just need more practice and in future [assignments]." (1)

- "One major error that I got a few times that took me hours to debug was pushing and popping registers in subroutines." (2)

- "My main debugging frustrations came out of sloppy coding." (2)

2) Advanced Beginner:

- "Next time, I would probably do more code review instead of having to debug the program which might save me a good amount of time." (1)

- "I wrote the code completely before debugging. Next time I will debug individual [subroutines] as I work. I think this will make things easier especially with larger programs." (1)

- "I used Turbo Debugger a lot this time around and I know that I definitely will use it a lot during the next [assignment]." (2)

- "In my next [assignment], I will spend more time on preparation and design of each [subroutine], as it usually resulted in fewer errors." (2)

3) Competent:

- "For the next assignment, I will make a flowchart or diagram of some sort before writing the code." (2,3)

- "I [designed the code] on paper… it aided in the general logic." (3)

- "My method was to [complete] the shorter [subroutines] first, which I could [complete] more easily, and then take on the more complicated ones." (2,3)

- "Turbo Debugger was sometimes useless in debugging certain bugs, so in those cases I [switched to manual debugging] to fix them." (3)

- "I found the CV32 debugger to be just about worthless. I usually ended up visually debugging my code, using the lab notes as a reference." (4)

4) Proficient:

- "I plan to comment more thoroughly so that I can better keep track of my thought process." (3)

- "I wrote the [main subroutine] first so I could have a better understanding of how the [individual subroutines] work together." (3,4)

- "I had fewer bugs in this [assignment] than in previous ones. The majority of time was spent learning [new concepts]." (4)

5) Expert:

- "When I finished my [subroutines], I helped my teammates with theirs, and I was able to fix their code." (5)

- "I am spending noticeably less time on debugging my code." (4,5)

D. Importance of a Debugging Model

A model of debugging abilities and habits is important because instructors can use the model to diagnose their students' debugging skills. From this diagnosis, instructors can develop the appropriate activities that would enhance their students' debugging skills. Those activities could include the methods and activities outlined in sections II and III. As a result, students could enhance their debugging skills, gain confidence in their debugging abilities, and spend less time completing their programming assignments.

VI. CONCLUSIONS AND FUTURE WORK

This study investigated the effectiveness of formal debugging training in computing curricula. This study showed that students could enhance their debugging skills by completing carefully planned debugging activities. The difference between this study and previous studies that have shown this result is that this study includes a variety of debugging activities; students could learn many debugging techniques that could make them well-rounded debuggers.

The debugging literature surveyed in section II provided an inclusive analysis of the debugging process. The literature demonstrated the need for experienced debuggers in the software industry and offers different techniques that programmers can use to enhance their skills. In addition, the literature discussed tools that programmers can use to debug programs. While none of these methods is guaranteed to work, programmers should choose a combination of methods that best suit their current skills and strengths in order to become more effective debuggers.

Based on the literature, a few, well known debugging techniques were selected, which led to the development of debugging exercises, debugging logs, development logs, reflective memos, and collaborative assignments that incorporated these techniques. Exposure to diverse activities rather than one or two detailed activities could be more beneficial to students. A student could learn several debugging techniques, and the student could determine which techniques best enhanced his or her skills. Student participation in the debugging exercises was optional; however, participation in the other activities was mandatory.

Quantitative data was collected through students' debugging logs and development logs, and qualitative data was collected through summative evaluation surveys. Students who completed the debugging exercises spent significantly less time on debugging the programming assignments. This time saved by these students was attributed to their enhanced debugging skills. Furthermore, the qualitative data supported the positive results of the quantitative data. Students agreed that formal debugging training enhanced their debugging skills. The students who participated in this study were especially receptive of the debugging exercises and collaborative assignments.

To improve students' debugging skills, debugging activities should be integrated throughout an entire curriculum, not just in one course. Like problem solving and writing, debugging is an important, complicated skill that requires repeated practice. For example, program design activities could be included in an advanced course after students gain programming experience in introductory courses and need to design programs as part of their assignments. By spreading the activities over multiple courses, students could receive the necessary training at opportune times.

These activities developed skills that are not specific to assembly language. Thus, similar activities could be designed for use in courses of any programming language. This study was conducted in an assembly language course because ECE 291 is the only programming intensive course offered by

the Department of Electrical and Computer Engineering that is not cross-listed with a computer science course.

The following questions were generated from this study for possible further investigation:

- What are the psychological effects of making the earlier sets of debugging exercises shorter and less difficult and later sets longer and more difficult? Does giving students confidence at an early stage improve their debugging skills more quickly? Would students become discouraged and stop participation if earlier sets of exercises were more difficult?

- Are the types of activities outlined in this study the best methods by which to enhance debugging skills? Should more emphasis be placed on training students to identify defects themselves, or should they be trained in the use of debugging tools instead?

- Should problems be added or subtracted from each set of exercises, or is there a good balance between skills learned and time spent with the current number of problems per set? Would students be willing to spend more time on exercises if it meant a further improvement in their debugging skills?

- How can professors convince students that writing reflective journals is beneficial? How should journal entries be structured to maximize their effectiveness? Are personal checklists of potential defects actually helpful for students?

- Would skills developed in ECE 291 transfer to other activities such as debugging hardware or debugging programs written in high-level languages?

- The course instructor or a teaching assistant designs each programming assignment in ECE 291, and students complete the predetermined subroutines; however, the final project is entirely student designed. What is an effective method of teaching program design in order to minimize defects? How can these program design skills be integrated into the programming assignments?

SAMPLE DEBUGGING EXCERCISE AND ITS SOLUTION

Factorial()

Factorial() calculates the factorial of an integer. The input integer is given in AX, and the factorial of that integer is returned in AX. Recall that 0! = 1. Assume that the function will be called with only non-negative inputs.

```
 1 Factorial
 2         pusha
 3         mov     cx, ax
 4         mov     ax, 0
 5         cmp     cx, 0
 6         je      .FactorialDone
 7
 8 .FactorialLoop
 9         mul     cx
10         loop    Factorial
11
12 .FactorialDone
13         popa
14         ret
```

Solutions:

- The `pusha` and `popa` instructions cannot be used because `ax` is a return value and will never be returned to the calling function.

- `ax` should be initialized to 1, not 0, on line 4.

- The loop on line 10 should loop back to the `.FactorialLoop` label, not the `Factorial` label. As it is now, the function will remain in an infinite loop since it keeps jumping back to the beginning of the function.

REFERENCES

[1] K. Araki, Z. Furukawa, and J. Cheng, "A general framework for debugging," *IEEE Software*, vol. 8, no. 3, pp. 14-20, 1991.

[2] M. Nanja and C. R. Cook, "An analysis of the on-line debugging process," in *Empirical Studies of Programmers: Second Workshop*. Norwood, NJ: Ablex Publishing, 1987, pp. 172-184.

[3] A. Mockus, R. Fielding, and J. Herbsleb, "Two case studies of open source software development: Apache and Mozilla," *ACM Transactions on Software Engineering and Methodology*, vol. 11, no. 3, pp. 309-346, 2002.

[4] M. Xie and B. Yang, "A study of the effect of imperfect debugging on software development cost," *IEEE Transactions on Software Engineering*, vol. 29, no. 5, pp. 471-473, 2003.

[5] J. A. Whittaker, "What is software testing? And why is it so hard?" *IEEE Software*, vol. 17, no. 1, pp. 70-79, 2000.

[6] G. Rothermel and M. J. Harrold, "A safe, efficient algorithm for regression test selection," in *Proceedings of the IEEE Software Maintenance Conference*, 1993, pp. 358-367.

[7] E. L. Jones, "An experiential approach to incorporating software testing into the computer science curriculum," in *Proceedings of the 31st ASEE/IEEE Frontiers in Education Conference*, vol. 2, 2001, pp. F3D 7-11.

[8] "Computing Curricula," Dec. 2001, http://www.computer.org/education/cc2001/final/.

[9] J. G. Spohrer and E. Soloway, "Analyzing the high frequency bugs in novice programs," in *Empirical Studies of Programmers: First Workshop*. Norwood, NJ: Ablex Publishing, 1986, pp. 230-251.

[10] A. C. Benander, B. A. Benander, and J. Sang, "An empirical analysis of debugging performance – differences between iterative and recursive constructs," *The Journal of Systems and Software*, vol. 54, pp. 17-28, 2000.

[11] W. S. Humphrey, *Introduction to the Personal Software Process*. Reading, MA: Addison-Wesley, 1997, pp. 143-149, 159-172.

[12] M. E. Fagan, "Design and code inspections to reduce errors in program development," *IBM Systems Journal*, vol. 15, no. 3, pp. 182-211, 1976.

[13] T. Gilb and D. Graham, *Software Inspection*. Reading, MA: Addison-Wesley, 1993, pp. 264-279.

[14] C. McDowell, L. Werner, H. Bullock, and J. Fernald, "The effects of pair-programming on performance in an introductory programming course," in *Proceedings of the 33rd SIGCSE Technical Symposium on Computer Science Education*, 2002, pp. 38-42.

[15] L. Gugerty and G. M. Olson, "Comprehension differences in debugging by skilled and novice programmers," in *Empirical Studies of Programmers: First Workshop*. Norwood, NJ: Ablex Publishing, 1986, pp. 13-27.

[16] S. Rifkin and L. Deimel, "Program comprehension techniques improve software inspections: A case study," in *Proceedings of the 8th International Workshop on Program Comprehension*, 2000, pp. 131-138.

[17] D. E. Knuth, "The errors of TeX," *Software – Practice and Experience*, vol. 19, no. 7, pp. 607-681, 1989.

[18] B. Korgel, "Nurturing faculty-student dialogue, deep learning and creativity through journal writing exercises," *Journal of Engineering Education*, vol. 91, no. 1, pp. 143-146, 2002.

[19] G. C. Lee and J. C. Wu, "Debug It: A debugging practicing system," *Computers and Education*, vol. 32, no. 2, pp. 165-197, 1999.

[20] A. Zeller, "Automated debugging: are we close?" *Computer*, vol. 34, no. 11, pp. 26-31, 2001.

[21] M. Eisenstadt, "My hairiest bug war stories," *Communications of the ACM*, vol. 40, no. 4, pp. 30-37, 1997.

[22] M. C. Loui, "The case for assembly language programming," *IEEE Transactions on Education,* vol. 31, no. 3, pp. 160-164, 1988.

[23] T. A. Angelo and K. P. Cross, *Classroom Assessment Techniques*. San Francisco, CA: Jossey-Bass, Inc., 1993, pp. 222-225.

[24] H. L. Dreyfus and S. E. Dreyfus, *Mind over Machine: The Power of Human Intuitive Expertise in the Era of the Computer*. New York: Free Press, 1986.