

AN INTEGRATED APPROACH TO INSTRUCTION  
IN DEBUGGING COMPUTER PROGRAMS

BY

RYAN JOSEPH CHMIEL

B.S., University of Illinois at Urbana-Champaign, 2001

THESIS

Submitted in partial fulfillment of the requirements  
for the degree of Masters of Science in Electrical Engineering  
in the Graduate College of the  
University of Illinois at Urbana-Champaign, 2004

Urbana, Illinois

## **ABSTRACT**

I conducted a study to demonstrate that formal training in debugging helps students develop skills in diagnosing and removing defects from computer programs. To accomplish this goal in an assembly language course, I designed multiple activities to enhance students' debugging skills. These activities included debugging exercises, debugging logs, development logs, reflective memos, and collaborative assignments. The debugging exercises were optional, but the other activities were mandatory. Students who also 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. I also collected qualitative data for each activity through summative evaluation surveys. Students agreed that formal debugging training enhanced their debugging skills. I also developed 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 could use the model to diagnose students' current debugging skills and take actions to enhance their skills.

*To Nicole, the only woman in this world for me.*

## ACKNOWLEDGMENTS

I am grateful to the College of Engineering at the University of Illinois at Urbana-Champaign and to the National Science Foundation for providing the financial assistance that supported my research.

I would first like to extend many thanks to my graduate advisor, Professor Michael C. Loui. Not only did I find an advisor willing to support an educational research project in a department where such projects are uncommon, but I also found an advisor whose enthusiasm and passion for the subject provided an excellent source of motivation while conducting my research. His never-ending advice and constant pursuit of excellence greatly improved the quality of this work.

Thanks to ECE 291 professors Michael C. Loui and Zbigniew Kalbarczyk for allowing me to conduct this study in the fall 2002 and spring 2003 semesters, respectively.

Thanks to the students in the fall 2002 and spring 2003 semesters of ECE 291 who participated in this study, especially those who completed the optional debugging exercises.

Finally, thanks to my family and friends for keeping me sane during my graduate studies.

## TABLE OF CONTENTS

CHAPTER 1 INTRODUCTION.....	1
1.1 Debugging and the Software Industry.....	1
1.2 The Scholarship of Teaching and Learning.....	2
1.3 Overview of Findings.....	2
CHAPTER 2 REVIEW OF LITERATURE ON DEBUGGING.....	4
2.1 Software Testing.....	4
2.2 Demonstrated Need for Debugging Training.....	5
2.3 Methods to Improve Debugging Skills.....	7
2.3.1 Code review and code inspection.....	7
2.3.2 Program comprehension.....	9
2.3.3 Software design.....	11
2.3.4 Learning through reflection.....	13
2.4 Debugging Tools.....	14
CHAPTER 3 ACTIVITIES TO ENHANCE DEBUGGING SKILLS.....	17
3.1 Debugging Exercises.....	17
3.2 Debugging Logs.....	19
3.3 Development Logs and Reflective Memos.....	20
3.4 Collaborative Assignments.....	21
CHAPTER 4 RESULTS AND ANALYSIS.....	22
4.1 Data Collection.....	22
4.2 Debugging Exercises.....	23
4.3 Debugging Logs.....	24
4.4 Development Logs and Reflective Memos.....	25
4.5 Collaborative Assignments.....	27
CHAPTER 5 A MODEL OF DEBUGGING ABILITIES AND HABITS.....	29
5.1 The Dreyfus Model.....	29
5.2 Model of Debugging Abilities and Habits.....	30
5.3 Student Responses for Each Stage of the Dreyfus Model.....	31
5.3.1 Novice.....	31
5.3.2 Advanced beginner.....	31
5.3.3 Competent.....	32
5.3.4 Proficient.....	32
5.3.5 Expert.....	32
5.4 Importance of a Debugging Model.....	33
CHAPTER 6 CONCLUSIONS AND FUTURE WORK.....	34
APPENDIX A DEBUGGING EXERCISES AND SOLUTIONS.....	37
A.1 Code Review Debugging Exercises and Solutions.....	37
A.2 Code Modification Debugging Exercises and Solutions.....	47

APPENDIX B FORMS AND SURVEYS .....	57
B.1 ECE 291 Debugging Log.....	58
B.2 Debugging Exercises Consent Form .....	59
B.3 Summative Evaluation Survey .....	60
REFERENCES .....	62

## CHAPTER 1 INTRODUCTION

### 1.1 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.

## **1.2 The Scholarship of Teaching and Learning**

The scholarship of teaching and learning (SoTL) could be defined as scholarship undertaken in the name of change, with the desire to create stronger curricula and more powerful pedagogies [5]. It inspires intensive studies of teaching and learning methodologies. SoTL shares the established criteria of scholarship, in that it is made public, can be reviewed critically by members of the community, and can be built upon by others to advance the field [5]. The goal of a SoTL study is to improve the quality of teaching in a discipline.

Studies focused on SoTL begin with one or more questions about teaching and learning. For example, the questions that fueled my research were, “Can programmers become better debuggers through formal training? What methods should be included in the training?” From these questions, I investigated techniques in the debugging literature and developed training activities for students to complete.

SoTL is essential for the continued success of all academic institutions. Teaching without learning is just talking [6]. Instructors should be aware of their teaching effectiveness. By reviewing previous SoTL studies, instructors could adapt their teaching methods based on the successes and failures of others. If an instructor feels that he or she could improve upon the methods of a previous study, the instructor could conduct a SoTL study and investigate new methods. In the end, students would benefit greatly from SoTL studies. When instructors analyze and investigate methods of teaching and learning, students enjoy a higher quality experience in the classroom.

## **1.3 Overview of Findings**

In this study, I found that students enhanced their debugging skills by completing carefully planned debugging activities. Students who completed optional debugging exercises, the primary activity I developed, 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, I distributed a summative evaluation survey to obtain students' comments on the activities. Overall, the qualitative data supported the results of the quantitative data. Students agreed that the debugging activities enhanced their debugging skills.

From the students' comments, I developed a model of debugging habits and abilities. This model, which is outlined in Chapter 5, 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.

## CHAPTER 2 REVIEW OF LITERATURE ON DEBUGGING

### 2.1 Software Testing

Before debugging a program, a programmer must first determine that a defect exists. According to Whittaker [7], 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 [8].

Jones [9] 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.

## **2.2 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 [10] 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 [11] 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.

Xie and Yang [4] studied the effect of imperfect debugging on software development cost. Imperfect debugging occurs when an attempt to correct a defect fails or when new defects are injected while another defect is corrected.

Xie and Yang developed a cost model for imperfect debugging and analyzed the model. They modeled imperfect debugging using a nonhomogeneous Poisson process. The model included parameters for the expected cost of removing a defect during testing, the expected cost of removing a defect after release of the software, the expected cost per unit testing time, and the level of perfection desired. One important parameter excluded from the model, though, was the cost to the user of running defective software. Xie and Yang tested their model with three different data sets. The graph of their model from each data set was U-shaped; that is, there was a point at which an increase in the level of perfection added to the cost. Xie and Yang concluded that the level of debugging perfection greatly affects the cost of software development.

Benander et al. [12] 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.

## **2.3 Methods to Improve Debugging Skills**

How can students become better debuggers? I believe that instructors can use a combination of methods to teach students how to debug their code effectively. I present the following methods as means of enhancing students' debugging skills.

### **2.3.1 Code review and code inspection**

Code review is an effective method for debugging [13]. 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 [13], 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 [14] 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 [15] 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.

Employees at Applicon were skeptical about the value of code inspections at first. Some programmers were reluctant to change their development processes, and some managers did not want to change processes midway through a product's development. The development groups who conducted code inspections quickly noticed their benefits. These groups noticed a decrease in the time necessary to identify and correct a defect. In addition, the teamwork, cooperation, and open discussion among these groups also increased. At the time of the writing of the case study, more than half of the design groups at Applicon regularly conducted code inspections. Last, the code inspections successfully reduced software development costs at Applicon. Overall, Applicon clearly improved the quality of its software products through the integration of code inspections into its development process in a cost-effective way.

McDowell et al. [16] 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.

### **2.3.2 Program comprehension**

Gugerty and Olson [17] 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.

Nanja and Cook [2] analyzed the on-line debugging processes of novice, intermediate, and expert programmers. Nanja and Cook classified students finishing their second term of an introductory course as novices, students finishing their third term of a data structures course as intermediates, and graduate students as experts. In their study, subjects in each group received a

short Pascal program containing three logic errors and three semantic errors. The subjects debugged the program individually and corrected all the defects they discovered. Nanja and Cook observed the subjects and recorded data such as a subject's initial program reading time, number of statements modified, and new errors introduced.

Nanja and Cook found that expert programmers were much more successful than the other two groups in debugging the program. Experts used a comprehension approach in which they first read and understood the program before attempting to correct the suspected defects. Nanja and Cook observed that experts also spent more time on the initial reading of the code and read the code in the order in which it would be executed. Conversely, novices and intermediates attempted to isolate the defects without fully understanding the program. In addition, they spent very little time reading the program, and they read it top to bottom. Furthermore, experts corrected all of the defects, modified fewer statements when correcting the defects, and rarely introduced new defects when correcting the original defects.

Nanja and Cook, like Gugerty and Olson [17], concluded that experts are better able to debug programs because of their program comprehension abilities. Again, it is possible that Nanja and Cook mistook correlation for causation.

Uchida et al. [18] also agreed that programmers should have program reading and comprehension skills in order to debug programs effectively. They proposed a multiple-view analysis of the debugging process by observing programmers in a controlled environment. The programmers received a small program of approximately 300 lines that contained one defect. For the decision view model, which represents the internal actions of the programmers, programmers recorded their internal decision processes by classifying the program subroutines into three sets: the set of subroutines that appear to contain the defect, the set of subroutines that do not appear to contain the defect, and the set of subroutines of which the programmer is unsure as to whether they contain the defect. The programmers updated the sets every five minutes until they discovered the defect.

All programmers in the study correctly identified the defect, although some programmers took longer than others. The programmers who took less time to identify the defect reported smaller average sizes for the set of suspicious subroutines. Furthermore, the best debuggers read the subroutine that was key to identifying the defect. From these results, Uchida et al. suggested that more experienced programmers reduce the set of suspicious subroutines more quickly.

Rifkin and Deimel [19] 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.

### **2.3.3 Software design**

Hunt and Thomas [20] advocated better program design as a means of improving the debugging process. They suggested that programmers should write “shy code.” Shy code has well-defined, isolated responsibilities and does not communicate with other code more than is necessary.

Hunt and Thomas recommended that programmers follow the Law of Demeter, which states that a module should communicate only with other modules that are immediately related to the module. Defects in a module can result in the transmission of erroneous data to another module, which would in turn produce erroneous results. By limiting the communication of a code unit, programmers could avoid these situations.

One drawback of designing shy code is creating many smaller subroutines that delegate traversal among objects. Doing so is inefficient because shy code requires more subroutine calls,

and call overhead adds to the running time of the program. Hunt and Thomas justified this inefficiency by arguing that, in the long run, a minor run-time delay is much less costly than the cost to the consumer of using defective software and the cost of maintaining highly coupled code. However, they did not provide direct evidence that the creation of shy code definitively leads to a reduction in the number of defects. Nonetheless, it appears that programmers could improve the debugging process by better designing their code in order to eliminate opportunities for defect injection.

Williams [21] acknowledged that the most efficient method of producing quality software is to prevent defect injection. Williams described an attempt to accomplish this goal at the University of Utah in which undergraduate students could take an elective course in Cleanroom Software Engineering. Cleanroom [22] is a software development process that uses engineering-based practices to develop high quality, reliable software.

Williams reported that although not all students in the course accepted the Cleanroom principles, many students applied them in their other courses. Students may have not accepted Cleanroom because it is a labor-intensive process; incorporating Cleanroom into a university setting is difficult when students have other responsibilities as well. Williams stated that time did not allow for a thorough exposure to Cleanroom. Overall, the course successfully taught students an approach to software development that limits the number of defects injected into programs.

Grove [23] integrated a subset of the Personal Software Process [13] into introductory programming courses to teach students a proper programming methodology. For example, Grove required students to keep a time log for each programming assignment. In the logs, students recorded the time spent on each of the software development stages. After completing each assignment, students summarized their time logs and determined the percentage of time spent on each stage. Students could use their logs to identify strengths and weaknesses in their software development skills and then adapt their approaches accordingly. Grove collected the students' logs for quantitative analysis.

Students who spent more time on the planning and design stages spent considerably less time on the testing stage. In addition, the majority of these students spent less time overall on the assignments. Last, these students received higher grades on the assignments on average. Grove shared these results with the students, who were impressed with the findings. Grove concluded

that the Personal Software Process teaches students the benefits of careful program planning and design.

#### **2.3.4 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 [24] 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.

Card [25] discussed the benefits of Defect Causal Analysis as an effective method of reflection. Defect Causal Analysis (DCA) [26] is a process that studies software defect reports in order to prevent defects. Periodically, a DCA team analyzes a sample of the developers' defect reports. The team classifies the reports into categories such as defect insertion times and defect types. From the classifications, the team identifies systematic errors, or errors that are repeated on different occasions. Last, the team determines the causes of the systematic errors and develops strategies to prevent them in the future. Thus, the DCA team positively adapts the developers' approach to software development.

Card presented the results of a DCA study at IBM. The IBM study classified the defects by the stages of software development and presented results as the number of defects per thousand lines of code for each stage. After implementing DCA, IBM noticed an average decrease of 50% in the number of defects for each stage. In addition, IBM's software budget rose only 0.5% after implementing DCA. Card concluded that DCA is a successful, cost-effective method of defect prevention.

Korgel [27] 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 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.

## 2.4 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.

Satratzemi et al. [28] developed AnimPascal, an educational programming environment to help programmers develop, test, and debug their code. AnimPascal is a program animator that incorporates the ability to record the problem-solving paths followed by students. This feature allows teachers to observe the students' problem-solving behaviors. As a result, teachers may correct misconceptions about programming and help students modify their problem-solving strategies. Satratzemi et al. used AnimPascal to record the development of students' programs as they implemented the binary search algorithm in an introductory laboratory course. Satratzemi et al. analyzed the student input recorded by AnimPascal. They categorized student defects by defect type and observed some trends among the student programs. Satratzemi et al. concluded that AnimPascal appears to be of great assistance to novice programmers, although they present little evidence to support their claim. For example, Satratzemi et al. mention no comments or survey data from the students in the study. Thus, it is impossible to determine whether AnimPascal enhanced students' program development and debugging skills.

Lee and Wu [29] 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 [30] 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 [31] 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.

## CHAPTER 3 ACTIVITIES TO ENHANCE DEBUGGING SKILLS

After considering the various ways in which programmers become better debuggers, I developed the following activities to help students improve their debugging skills. I introduced these activities 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 [32]. Although few students in ECE 291 have previously programmed in assembly language, all have experience in at least one programming language.

I did not include activities related to program design with emphasis on minimizing defects. I did not want to inundate students with too many activities or overburden them in one course. These actions could cause them to become discouraged with the activities. I chose activities that would acquaint students with a variety of debugging techniques whose effectiveness I could evaluate. It would be difficult, say, to measure a student's improvement in debugging while the student enhanced his or her program design skills simultaneously.

### 3.1 Debugging Exercises

I created two sets of debugging exercises for each of four of the five regular programming assignments before the final project, starting with the second assignment. I felt students should develop one program on their own to become motivated to undertake the debugging exercises. A novice assembly language programmer could especially benefit from the exercises. Appendix A contains the debugging exercises and their solutions.

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. While these problems were initially appealing because students could identify the defect in a programmer's thought process, I instead opted for the code modification problems for many reasons. First, I found it difficult to create effective situation problems, as they focused on high-level concepts rather than types of defects. Second, I felt 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. Examples of these defects can be found in Appendix A. 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.

### **3.2 Debugging Logs**

I developed a debugging log for students to keep 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 I developed, which can be found in Appendix B, was modeled after a log developed by Humphrey [13]. 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 I feel that 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.

### **3.3 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?

### **3.4 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.

## CHAPTER 4 RESULTS AND ANALYSIS

In the fall of 2002, I piloted the activities from Chapter 3 in ECE 291 to obtain preliminary qualitative results. Overall, I observed positive outcomes. 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 taking their suggestions into account, I modified the exercises accordingly for the spring of 2003. I now present the results of my study from 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.

### 4.1 Data Collection

In order to determine the effectiveness of the activities, I distributed a summative evaluation survey to participants in the study. 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. I analyzed students' comments for similarities.

I collected the following quantitative data from students' debugging logs and development logs:

- 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

## 4.2 Debugging Exercises

I obtained 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.

I obtained many good comments and suggestions on the exercises, some of which I share 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. I believe the summative evaluation will help create even more effective exercises for future students.

### **4.3 Debugging Logs**

After the completion of each programming assignment, I reviewed students' debugging logs and determined 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, I could not use the number of defects per program and time spent debugging each program as reasonable performance metrics. Instead, I used 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 [6].

In the future, I will need to 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 I did not specifically request this information, students did not mention the creation of personal checklists in their reflective memos.

#### 4.4 Development Logs and Reflective Memos

Students reported the number of hours they spent on each phase of the development process in their development logs. Using the development logs, I calculated the percentage of time that they spent on debugging the programming assignments. I considered development logs only from the first four assignments because they were individual assignments; the fifth assignment was a collaborative assignment, and it was difficult to collect reliable data for individual students. I separated the data 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%	$p < .002$

I conducted t-tests on the data for each assignment to determine the significance of each difference. For the first assignment, I accepted the null hypothesis, which stated that there was no difference in debugging skills between the two groups. I expected this result 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 me to reject the null hypothesis for these assignments. I believe that these differences are due to 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, I recorded the total time students spent on

each assignment. 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.

I analyzed the exam scores of students in each group to determine whether there was a difference in aptitude between the groups. I compared scores from the first midterm, which was administered between the due dates of the first and second programming assignments, and the final exam. 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.

I obtained 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

I observed both positive and unexpected results from the reflective memo surveys. 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.

#### 4.5 Collaborative Assignments

I obtained 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.

## CHAPTER 5 A MODEL OF DEBUGGING ABILITIES AND HABITS

Based on comments in the students' debugging logs, development logs, reflective memos, and evaluation surveys, I constructed a model that describes the progression of students' debugging abilities and habits. My model is based on the Dreyfus model of skill development [33]. For each stage of the Dreyfus model, I listed the typical actions and emotions of students that I observed from the students' comments.

### 5.1 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.

## 5.2 Model of Debugging Abilities and Habits

Table 5.1 presents my 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	<ul style="list-style-type: none"> <li>• Lacks debugging skills</li> <li>• Repeats the same types of defects frequently throughout a program</li> <li>• Debugs programs haphazardly instead of following a plan</li> <li>• Spends considerable amounts of time on debugging</li> <li>• Gives up easily and depends on others for assistance</li> </ul>
Advanced Beginner	<ul style="list-style-type: none"> <li>• Develops debugging skills through experience; recognizes symptoms of previously experienced defects</li> <li>• Repeats the same types of defects occasionally throughout a program</li> <li>• Begins adapting approaches to debugging based on prior successes and failures</li> <li>• Attempts less familiar debugging techniques only as a last resort</li> <li>• Relies to some extent on others for assistance</li> </ul>
Competent	<ul style="list-style-type: none"> <li>• Knows a variety of debugging techniques</li> <li>• Approaches debugging systematically; evaluates the situation to determine which techniques to use</li> <li>• Alternates between techniques when certain techniques are not effective</li> <li>• Identifies most defects independently</li> </ul>
Proficient	<ul style="list-style-type: none"> <li>• Sees debugging as part of program development as a whole</li> <li>• Develops skills in other areas of program development to facilitate debugging</li> <li>• Optimizes decision-making abilities and approaches to debugging</li> <li>• Asks others for assistance rarely and assists others</li> </ul>
Expert	<ul style="list-style-type: none"> <li>• Approaches debugging intuitively due to extensive experience; debugging is second-nature</li> <li>• Identifies complex and/or unfamiliar defects successfully</li> <li>• Asks others for assistance rarely and assists others</li> </ul>

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. I do not suggest 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.

### **5.3 Student Responses for Each Stage of the Dreyfus Model**

In this section, I include some of the student comments that I used to create 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.

#### **5.3.1 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)

#### **5.3.2 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)

### 5.3.3 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)

### 5.3.4 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)

### 5.3.5 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)

## **5.4 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 I outlined in Chapters 2 and 3. As a result, students could enhance their debugging skills, gain confidence in their debugging abilities, and spend less time completing their programming assignments.

## CHAPTER 6 CONCLUSIONS AND FUTURE WORK

My study investigated the effectiveness of formal debugging training in computing curricula. I showed that students could enhance their debugging skills by completing carefully planned debugging activities. The difference between my study and previous studies that have shown this result is that my 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 Chapter 2 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.

I selected a few, well known debugging techniques based on this literature, and I designed debugging exercises, debugging logs, development logs, reflective memos, and collaborative assignments that incorporated these techniques. I felt it wisest to create diverse activities rather than focusing on one or two detailed activities. As a result, a student learned 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.

I collected quantitative data through students' debugging logs and development logs and qualitative data through summative evaluation surveys. Students who completed the debugging exercises spent significantly less time on debugging the programming assignments. I attribute the time saved by these students 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 my 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.

My activities developed skills that are not specific to assembly language. Thus, similar activities could be designed for use in courses of any programming language. I conducted my study 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.

Last, I share some questions that were generated from my 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 my 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?

## APPENDIX A DEBUGGING EXERCISES AND SOLUTIONS

### 1) Code Review Debugging Exercises and Solutions

Section A.1 contains the debugging exercises and solutions for the code review exercises that students completed by hand.

### 2) Code Modification Debugging Exercises and Solutions

Section A.2 contains the debugging exercises and solutions for the code modification exercises that students completed while at a computer.

#### A.1 Code Review Debugging Exercises and Solutions

##### 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.

## Power()

`Power()` calculates  $\text{base}^{\text{exponent}}$  using recursion. The base, given in `BX`, is a 16-bit two's complement integer, and the exponent, given in `CX`, is a 16-bit nonnegative integer. The value is returned in `AX`.

```
1 Power
2     push    bx
3     push    cx
4
5     cmp     cx, 1                ; check for recursion's
6     ja     .Recurse            ; base case
7     mov     ax, bx
8
9 .Recurse
10    call    Power                ; recursive call
11    push    dx
12    mul     cx
13    pop     dx
14
15 .Done
16    pop     cx
17    pop     bx
18    ret
```

Solutions:

- The base case should check for the exponent to reach zero and then return one.
- The `dec cx` instruction must go before the recursive call.
- The instruction `jmp .Done` should be added to line 8 to skip the recursive case after handling the base case.
- `mul cx` should be changed to `imul bx`.

## CountTallMountains()

CountTallMountains() counts the number of mountains that are greater than twice the height of Mount Washington, which is 6,288 feet tall. The *Mountains* array contains 1000 mountain heights, and each mountain can be up to 29,035 feet tall, the height of Mount Everest. The count should be returned in AX.

```
1 CountTallMountains
2     push    bx
3     push    cx
4
5     mov     ax, 0
6     mov     cx, 1000
7
8 .CountLoop
9     mov     bx, Mountains
10    cmp     bx, 6288*2
11    jle    .LoopContinue
12    inc     ax
13
14 .LoopContinue
15    inc     bx
16    loop   .CountLoop
17
18    pop     cx
19    pop     bx
20    ret
```

### Solutions:

- Move line 9 to line 7: **bx** is being reset to the beginning of the array every time, but it should traverse the array.
- Brackets should be placed around **bx** on line 10 to read the value of the current array location pointed to by **bx**.
- Since mountains can be up to 29,035 feet high, the array must be an array of words, not bytes. Thus, the **inc bx** on line 15 should be changed to **add bx, 2** to correctly traverse the array.

## CalculateMinGrade()

CalculateMinGrade() calculates the minimum grade in the *Grades* array. There are 20 grades in this array, and each grade is between 0 and 100 inclusive. The minimum grade should be written to the *MinGrade* variable.

```
1 CalculateMinGrade
2     push    ax
3     push    bx
4     push    cx
5
6     mov     bx, [Grades]
7     mov     al, [bx]           ; First grade is initial minimum
8     mov     cx, 19           ; Only need 19 comparisons
9
10 .GradeLoop
11     inc     bx
12     cmp     [bx], al
13     jmp     .NotMin
14     mov     al, [bx]
15
16 .NotMin
17     loop    .GradeLoop
18
19     pop     cx
20     pop     bx
21     pop     ax
22     ret
```

### Solutions:

- The *MinGrade* variable never receives the result of the function.
- Remove the brackets from *Grades* on line 6 because **bx** should be a pointer to the array. Instead, **bx** is set to the value of the first 2 locations in the array since it is an array of bytes.
- The **jmp** on line 13 should be changed to a conditional jump, either **jge** or **jbe**. **jmp** is an unconditional jump so the program will jump every time, regardless of the result of the comparison.

## AvgRandomNums()

AvgRandomNums() continually sums up random numbers as long as they are even. When an odd number is generated, the loop terminates without including it in the sum. Then, the sum is divided by the number of random numbers generated and the average is returned in AX (ignore any remainder). Assume that Random() returns a random number between -10,000 and 10,000 inclusive in AX.

```
1 AvgRandomNums
2     push    cx
3     push    dx
4
5 .SumLoop
6     call    Random
7     add     dx, ax
8     test    ax, 01h    ; Check LSB for even/odd
9     jz     .SumDone
10    inc     cx
11    jmp     .SumLoop
12
13 .SumDone
14    mov     ax, dx
15    cwd
16    idiv   cx
17
18    pop     dx
19    pop     cx
20    ret
```

### Solutions:

- Initialize **cx** and **dx** to 0 before line 5 so the average is computed correctly.
- **jz** on line 9 should be changed to **jnz** because the loop should terminate when odd number is encountered.
- Line 7 should be placed between lines 9 and 11. Otherwise, the sum will also contain the odd number that terminates the loop.

## CountOverUnder()

CountOverUnder() keeps a plus/minus count of the number of word-sized array elements that are greater than, less than, or equal to a certain number. Every time an array element is greater than the number, the count is incremented. Similarly, the count is decremented when the number is larger, and the count does not change when the two are equal. The array is pointed to by SI and has length CX, and the number to be used in the comparisons is given in AX. The plus/minus count should be returned in AX.

```
1 CountOverUnder
2     push    cx
3     push    dx
4     push    si
5
6     mov     dx, 0           ; reset to 0 to use as +/- count
7
8 .CountLoop
9     cmp     ax, [si]
10    je     .Continue
11    jl     .Under
12
13 .Over
14     inc     dx
15 .Under
16     dec     dx
17 .Continue
18     add     si, 2
19     loop   .CountLoop
20     mov     ax, dx         ; write count to AX
21
22     pop     cx
23     pop     dx
24     pop     si
25     ret
```

### Solutions:

- The registers should be popped in reverse order – first `pop si`, then `pop dx`, then `pop cx`.
- `jmp .Continue` should be placed between lines 14 and 15, otherwise the `.Over` case falls through to the `.Under` case.
- The `j1` on line 11 should be changed to `jb` in order to meet the function specifications.

## IsPalindrome()

IsPalindrome() determines whether a string is a palindrome (read the same forward and backward, such as madam, noon, etc.). The pointer to the beginning of the string is given in SI, and the string has length CX. If the string is a palindrome, set AX = 1 on return, 0 otherwise.

```
1 IsPalindrome
2     push    cx
3     push    si
4     push    di
5
6     mov     di, si
7     add     di, cx           ; DI points to the end of the string
8     shl     cx, 1           ; loop over half the array
9
10 .PalindromeLoop
11     mov     al, [si]
12     cmp     al, [di]
13     jne     .NotPalindrome
14     inc     di
15     dec     si
16     loop   .PalindromeLoop
17     mov     ax, 1
18     jmp     .Done
19
20 .NotPalindrome
21     mov     ax, 0
22
23 .Done
24     pop     di
25     pop     si
26     pop     cx
27     ret
```

### Solutions:

- The instruction on line 8 should be changed to **shr** (shift right multiplies by two, not divides by two).
- Place the **dec di** instruction between lines 7 and 8 because otherwise the function begins checking values one byte past the end of the string. The string starts at memory location **[si]** and ends at **[si+cx-1]**.
- **di** should be decremented and **si** should be incremented. The pointers should move towards each other, not away from each other.

## StupidSort()

StupidSort() is a terribly inefficient sorting algorithm. It randomly swaps two values in the array and then checks to see if the array is sorted in ascending order. This process continues until the array is sorted. The array is pointed to by SI and has length CX. Assume that Random() takes as input in CX an upper bound N on the random number to be generated (i.e., it will be in the range 0 to N), and the number is returned in AX. Each array element is an 8-bit two's complement integer.

```
1 StupidSort
2     pusha
3
4 .SortLoop
5     call    Random           ; Generate random array locations
6     mov     bx, ax
7     call    Random
8     mov     bp, ax
9
10    mov     ax, [ds:bp]      ; Swap array values
11    mov     dx, [bx]
12    mov     [bx], ax
13    mov     [ds:bp], dx
14
15    mov     bx, 0
16
17 .CheckSorted                ; Check to see if array is sorted
18    mov     dx, [bx]        ; Read in adjacent values at once
19    cmp     dl, dh
20    jg      .SortLoop
21    inc     bx
22    jmp     .CheckSorted
23    popa
24    ret
```

Solutions:

- `[bx]` and `[ds:bp]` should be changed to `[bx+si]` and `[ds:bp+si]`, respectively.
- Swap bytes, not words, since the array is an array of bytes.
- The instruction `dec cx` should be inserted on line 3 to get the correct upper bound for the calls to Random() and to loop the correct number of times for the `.CheckSorted` loop.
- On line 22, the instruction `jmp .CheckSorted` should be changed to `cmp bx, cx` and `j1 .CheckSorted`.

## KeyboardISR()

KeyboardISR() processes keyboard input from the user. The input is a keyboard scancode waiting at port 60h. The scancode for all key presses should be written to the *Scancode* variable. Furthermore, when ESC is pressed, the *Exit* variable should be set to non-zero. Scancodes for key releases should be ignored.

```
1 KeyboardISR
2     pusha
3     push    cs
4     pop     ds                ; make sure ds = cs
5     mov     byte [Scancode], 0
6     mov     al, [60h]
7     cmp     al, 1             ; ESC scancode
8     je      .Exit
9     cmp     al, 80h          ; Ignore key releases
10    jl      .Done
11    jmp     .Keypress
12
13 .Exit
14     mov     byte [Exit], 1
15
16 .Keypress
17     mov     [Scancode], al
18     mov     al, 20h          ; Send EOI signal to PIC
19     out     20h, al
20
21 .Done
22     popa
23     ret
```

### Solutions:

- **ret** should be changed to **iret**.
- On line 10, **jl** should be changed to **jb**.
- On line 6, **mov al, [60h]** should be changed to **in al, 60h**.
- **jmp .CheckLoop** should be replaced with **cmp bx, cx** and **jl .CheckLoop**.
- Even though key releases are being ignored, the end-of-interrupt signal must still be sent to the PIC, as key releases still generate interrupts. Thus, move the **.Done** label in between lines 17 and 18 to ensure that this happens.

## DrawDiagLine()

DrawDiagLine() draws a diagonal line on a text mode video screen. The upper left-hand coordinate (x,y) = (DL,DH), the length of the line is given in CX, and the color of the line is given in BL.

```
1 DrawDiagLine
2     pusha
3
4     shl     bx, 8           ; move color to high byte, clear out character
5     mov     dx, ax         ; calculate initial offset
6     mov     al, 80
7     mul     dh
8     add     al, dl
9     shl     ax, 1
10    mov     di, ax
11
12    .LineLoop
13        cmp     di, 80*25   ; check for off-screen values
14        jge     .Continue
15        mov     [es:di], bx
16
17    .Continue
18        add     di, 82       ; go down one and right one
19        loop    .LineLoop
20
21        popa
22        ret
```

### Solutions:

- On line 18, **82** should be changed to **162** (80\*2+2) because there are two bytes per screen location.
- The carry of the addition on line 8 should be added to **ah**. Insert the instruction **adc ah, 0** between lines 8 and 9.
- **es** should be initialized to **B800h** before the loop.

## A.2 Code Modification Debugging Exercises and Solutions

### DisplayArray()

DisplayArray() displays an array of 16-bit two's complement integers, one per line. SI contains a pointer to the beginning of the array, and AX contains the length of the array.

```
1 DisplayArray
2     pusha
3
4 .DisplayLoop
5     mov     ax, [si]
6     mov     bx, Binascbuf
7     call   binasc
8     mov     dx, bx
9     call   dspmsg
10    mov     dx, Newline
11    call   dspmsg
12    inc     si
13    loop   .DisplayLoop
14
15    popa
16    ret
```

Solutions:

- Must set **cx** = **ax** on line 3 to use as a counter for the **loop** instruction.
- **inc si** on line 12 should be changed to **add si, 2** because the array is an array of words, not bytes.
- **cx** must be preserved around the call to **binasc**. One way to accomplish this is to **push cx** before the call to **binasc** and **pop cx** after the call to **binasc**.

## CalculateGrade()

CalculateGrade() calculates the grade of a student in a course with the following grading scale: A  $\geq$  900, B  $\geq$  800, C  $\geq$  700, D  $\geq$  600, F < 600. SI contains a pointer to the beginning of an 11-element array of bytes. The first 10 elements in the array are the student's grades, and the last element in the array is where the grade should be stored.

```
1 CalculateGrade
2     pusha
3
4     mov     ax, 0
5     mov     cx, 10
6
7 .AddGradesLoop
8     add     ax, [si]
9     inc     si
10    loop    .AddGradesLoop
11
12    cmp     ax, 900
13    jg     .A
14    cmp     ax, 800
15    jg     .B
16    cmp     ax, 700
17    jg     .C
18    cmp     ax, 600
19    jg     .D
20    jmp     .F
21
22 .A
23    mov     al, 'A'
24    jmp     .StoreGrade
25 .B
26    mov     al, 'B'
27    jmp     .StoreGrade
28 .C
29    mov     al, 'C'
30    jmp     .StoreGrade
31 .D
32    mov     al, 'D'
33    jmp     .StoreGrade
34 .F
35    mov     al, 'F'
36
37 .StoreGrade
38    mov     [si], al
39
40    popa
41    ret
```

Solutions:

- The `add ax, [si]` instruction on line 8 reads a word from the array instead of a byte and adds it to the total, which gives an incorrect sum. To correct this, the instruction could be replaced with the instructions `movzx dx, byte [si]` and `add ax, dx`.
- All `jb` instructions should be replaced with `jge`.

## CToF()

CToF() converts a temperature from Celsius to Fahrenheit using the formula  $F = (9/5)C + 32$ . The input integer, a Celsius temperature, is given in AX, and the corresponding Fahrenheit temperature should be returned in AX.

```
1 CToF
2     push    ax
3     push    bx
4     push    cx
5     push    dx
6
7     mov     bx, ax
8     mov     ax, 9
9     cwd
10    mov     cx, 5
11    div     cx
12    mul     bx
13    add     ax, 32
14
15    pop     dx
16    pop     cx
17    pop     bx
18    pop     ax
19    ret
```

Solutions:

- Do not `push ax` or `pop ax` because `ax` is a return value. Remove these instructions from the code.
- Since Celsius and Fahrenheit temperatures can be negative, `div` and `mul` should be replaced with `idiv` and `imul`, respectively.
- `imul` should occur before `idiv`, otherwise the calculated Fahrenheit temperatures will be incorrect.

## DivisibleBy6()

DivisibleBy6() determines whether or not an integer is divisible by six. Recall that a number is divisible by six if it is divisible by both two and three. The input integer is given in AX. If the integer is divisible by six, return 1 in AX, otherwise return 0.

```
1 DivisibleBy6
2     push    cx
3     push    dx
4
5     mov     dx, ax
6     shr     dx, 2
7     jnc     .DivisibleBy6      ; LSB=1, # is odd; LSB=0, # is even
8
9     mov     cx, 3
10    cwd
11    div     cx                 ; Divide by 3 and then
12    test   dx, dx             ; Check if remainder is 0
13    jz     .DivisibleBy6
14
15 .NotDivisibleBy6
16     mov     ax, 0
17
18 .DivisibleBy6
19     mov     ax, 1
20
21 .Done
22     pop     dx
23     pop     cx
24     ret
```

Solutions:

- Shifting right once divides by two, so change the **2** on line 6 to **1**.
- Change line 7 to `jc .NotDivisibleBy6` so that if the number is odd, it is not divisible by 6. As it is now, the code considers all even numbers to be divisible by six, but that is not necessarily true.
- Add the instruction `jmp .Done` on line 17 so that the `.NotDivisibleBy6` case does not fall through to the `.DivisibleBy6` case. As it is now, the code considers all numbers to be divisible by 6.

## BubbleSort()

BubbleSort() sorts an array of 16-bit two's complement integers in ascending order according to the BubbleSort algorithm. SI contains a pointer to the beginning of the array, and CX contains the length of the array.

```
1 BubbleSort
2     pusha
3     mov     bx, cx           ; save length of array in BX
4
5 .OuterLoop
6     mov     di, si         ; set DI to beginning of array
7     mov     cx, bx         ; reset inner loop counter
8
9 .InnerLoop
10    mov     ax, [di]
11    mov     dx, [di+1]
12    cmp     ax, dx
13    jg     .NoSwap
14    mov     [di], dx
15    mov     [di+1], ax
16
17 .NoSwap
18    inc     di
19    loop   .InnerLoop
20    loop   .OuterLoop
21
22    popa
23    ret
```

Solutions:

- Insert the instruction `dec bx` on line 4. The loop should execute *length-1* times, not *length* times.
- The contents of `cx` should be preserved around the inner loop. One way to accomplish this is to `push cx` on line 8 and `pop cx` between lines 19 and 20.
- On lines 11 and 15, `[di+1]` should be changed to `[di+2]` since the array contains words, not bytes.
- On line 18, `inc di` should be changed to `add di, 2` since the array contains words, not bytes.
- On line 13, `jg` should be changed to `j1` in order to sort the numbers in ascending order.

## Fibonacci()

Fibonacci() calculates the  $n^{\text{th}}$  Fibonacci number. The input integer is given in AX, and the  $n^{\text{th}}$  Fibonacci number should be returned in AX.

```
1 Fibonacci
2     cmp     ax, 2
3     jl     .Done           ; Covers both base cases
4
5     dec     ax
6     call   Fibonacci
7     mov     dx, ax         ; F(N-1)
8     dec     ax
9     call   Fibonacci     ; F(N-2)
10    add     ax, dx         ; F(N-1) + F(N-2)
11
12 .Done
13     ret
```

### Solutions:

- **dx** needs to be preserved in this function. To accomplish this, insert **push dx** between lines 1 and 2 and **pop dx** between lines 12 and 13.
- **ax** needs to be preserved around the F(N-1) call. To accomplish this, insert **push ax** between lines 6 and 7 and **pop ax** between lines 8 and 9.

## Delay()

Delay() halts program execution for a specified number of seconds, which is given in AX.

Assume that the function is called only with nonnegative values.

```
1 Delay
2     push    eax
3     push    edx
4
5     mov     edx, 18           ; Determine number of ticks to delay
6     mul     edx             ; NumTicks = 18 * NumSeconds
7
8 .DelayLoop
9     mov     edx, [TimerTicks]
10    add     edx, eax
11    cmp     [TimerTicks], edx
12    jg     .DelayLoop
13
14 .Done
15    pop     edx
16    pop     eax
17    ret
```

### Solutions:

- On line 12, **jb** should be changed to **jb**.
- Lines 9 and 10 should be moved above line 8, otherwise an infinite loop occurs.

## TimerISR()

TimerISR() processes hardware interrupts generated by the system timer. This function should keep track of the number of seconds that the program runs. Each second is approximately 18.2 timer ticks. The TimerISR() function should account for (i.e., ignore) the extra timer tick every fifth second.

```
1 TimerISR
2     cmp     byte [Fifths], 5           ; Check for fifth second
3     jl     .Increment
4     dec    word [TimerTick]
5     jmp    .Done
6
7 .Increment
8     inc    word [TimerTick]
9     cmp    word [TimerTick], 18
10    jl     .Done
11
12    inc    byte [Fifths]               ; Count number of .2 ticks
13    inc    word [NumSeconds]
14
15 .Done
16    mov    al, 20h
17    out   20h, al
18    ret
```

Solutions:

- Preserve the value of register **ax**.
- **ret** should be changed to **iret**.
- *TimerTick* must be reset to zero when incrementing *NumSeconds*. Add the instruction **mov dword [TimerTick], 0** to line 11.
- *Fifths* must be reset to zero every five seconds. Change the instruction on line 4 to **mov byte [Fifths], 0**.

## DrawFilledRect()

DrawFilledRect() draws a filled green rectangle to the 80x25 text mode video screen. The x and y coordinates of the upper left corner of the rectangle are given in DL and DH, respectively, and the width and height of the rectangle are given in CL and CH, respectively. Assume that the entire rectangle will fit on the screen.

```
1 DrawFilledRect
2     pusha
3
4     mov     ax, 0B800h
5     mov     es, ax
6
7     mov     al, dh           ; calculate initial offset
8     mov     ah, 80
9     mul     ah
10    xor     dh, dh
11    add     ax, dx
12    shl     ax, 1
13
14    mov     ax, 4000h
15
16 .Loop
17    xor     ch, ch
18    rep     stosw           ; draw to video screen
19
20    add     di, 160        ; move to next row
21    movzx   ax, cl
22    shl     ax, 1
23    sub     di, ax
24    dec     ch
25    jnz     .Loop
26
27    popa
28    ret
```

### Solutions:

- **4000h** should be changed to **2000h**.
- The **cx** register must be preserved around lines 17 and 18.
- The use of **ax** on lines 21-23 overwrites the value used for **rep stosw**. Either preserve **ax** around these lines or use an available 16-bit register, such as **bx**, to assist in the calculation.
- Add the instruction **mov di, ax** on line 13.

## CapitalizeScreen()

CapitalizeScreen() capitalizes all the lowercase letters on the text mode video screen. There are no inputs to the function. Note that the difference in ASCII values between an uppercase and lowercase letter, i.e., 'A' – 'a', is 20h.

```
1 CapitalizeScreen
2     pusha
3
4     mov     ax, 0B800h
5     mov     es, ax
6     mov     cx, 80*25
7     mov     di, 0
8
9     .CapLoop
10    mov     ax, [di]
11    cmp     ah, 'a'
12    jge     .Lowercase
13    cmp     ah, 'z'
14    jle     .Lowercase
15    jmp     .NotLowerCase
16
17    .Lowercase
18    sub     ah, 20
19
20    .NotLowerCase
21    mov     [di], ax
22    inc     di
23    loop   .CapLoop
24
25    popa
26    ret
```

Solutions:

- Change all occurrences of **ah** to **al**.
- Line 22 should be change to **add di, 2** because there are two bytes per screen location.
- On line 21, **[di]** should be changed to **[es:di]**.
- The lowercase letter determination is incorrect. The current code does not correctly check characters within the range 'a' to 'z' inclusive. One solution is to change line 12 to **j1 .NotLowercase**, line 14 to **jg .NotLowercase**, and to remove line 15.

## **APPENDIX B FORMS AND SURVEYS**

### 1) ECE 291 Debugging Log

Section B.1 contains the debugging log that students used to record the defects they identified while completing the programming assignments.

### 2) Debugging Exercises Consent Form

Section B.2 contains the consent form that was required for each student who completed the optional debugging exercises.

### 3) Summative Evaluation Survey

Section B.3 contains the summative evaluation survey distributed to all students in the course after completing the debugging activities of the semester.

**B.1 ECE 291 Debugging Log**

ECE 291 Debugging Log

Assignment #: \_\_\_\_\_

Function: \_\_\_\_\_ Date: \_\_\_\_\_ Time Found: \_\_\_\_\_ Time Fixed: \_\_\_\_\_  
Description: \_\_\_\_\_

Solution: \_\_\_\_\_

Function: \_\_\_\_\_ Date: \_\_\_\_\_ Time Found: \_\_\_\_\_ Time Fixed: \_\_\_\_\_  
Description: \_\_\_\_\_

Solution: \_\_\_\_\_

Function: \_\_\_\_\_ Date: \_\_\_\_\_ Time Found: \_\_\_\_\_ Time Fixed: \_\_\_\_\_  
Description: \_\_\_\_\_

Solution: \_\_\_\_\_

Function: \_\_\_\_\_ Date: \_\_\_\_\_ Time Found: \_\_\_\_\_ Time Fixed: \_\_\_\_\_  
Description: \_\_\_\_\_

Solution: \_\_\_\_\_

Function: \_\_\_\_\_ Date: \_\_\_\_\_ Time Found: \_\_\_\_\_ Time Fixed: \_\_\_\_\_  
Description: \_\_\_\_\_

Solution: \_\_\_\_\_

## B.2 Debugging Exercises Consent Form

### ECE 291 Debugging Exercises Consent Form

#### Purpose and Procedures:

The purpose of this research is to determine the effectiveness of formal debugging training. Students should improve their skills in locating and correcting defects (bugs) in code through completing sets of debugging exercises. This result should lead to faster completion times of programming assignments.

Students will complete multiple sets of debugging exercises. These exercises appear in two types. For the first type, students will debug program functions by hand without the aid of a computer. For the second type, students will have access to an assembler and debugger to correct the defects in program functions. Students will continually modify, assemble, and test the functions until each works as expected. Each set should take students approximately two hours to complete.

Students will also keep debugging logs as they work on the programming assignments for the course. In these logs, students will record the source of each defect, how the defect was corrected, and the time needed to correct the defect. Students should submit these logs to course staff when submitting the corresponding programming assignment. The logs will be returned to the students a few days later.

#### Voluntariness:

Participation in this research is voluntary. Students may refuse to participate or may discontinue participation at any time without penalty or loss of benefits to which they are otherwise entitled.

#### Benefits and Risks:

As a result of completing these exercises, students may become more proficient in debugging, and this skill will aid them not only in this course but also in their future programming. Students may experience some mild, temporary discomfort, for example stress, related to working on a homework assignment and their performance on the assignment. Since these exercises are only for personal gain and do not affect the students' grades in the course, the level of discomfort will most likely be minimal.

#### Compensation:

Students may choose to complete the exercises for a particular programming assignment as an alternative to early submission in order to receive extra credit. By completing the exercises for one assignment, students can receive up to five extra credit points. Furthermore, students who complete all sets of exercises will be given \$20 gift certificates to the Illini Union Bookstore.

#### Confidentiality:

The data to be used in this research is limited to that requested in the debugging logs and cover memos for programming assignments. Only course staff will have access to this data, and only Ryan Chmiel will analyze the data. In the event of publication of this research, no personally identifying information will be disclosed.

#### Whom to Contact with Questions:

Questions about this research should be directed to Ryan Chmiel (rchmiel@uiuc.edu). Questions about your rights as a research participant should be directed to the UIUC Institutional Review Board at (217) 333-2670.

I certify that I have read this form and volunteer to participate in this research study.

Please print name: \_\_\_\_\_

Signature: \_\_\_\_\_ Date: \_\_\_\_\_

### B.3 Summative Evaluation Survey

#### ECE 291 Debugging Activities Evaluation Survey

This survey relates to the debugging activities in ECE 291 this semester. Please complete this survey only for the activities in which you participated; if you did not participate in the optional debugging exercises, please leave that section blank. Your feedback on these activities via this survey is important, and we thank you for taking the time to complete it.

For each of the statements below, with the exception of the first statement regarding the debugging exercises, use the following rating system:

1 = Strongly Disagree, 2 = Disagree, 3 = No Opinion, 4 = Agree, 5 = Strongly Agree

#### Debugging Exercises

Amount of time spent to complete one set of exercises, in approximate hours.	1	2	3	4	5
Each set of exercises contains defects relevant to the corresponding assignment.	1	2	3	4	5
I watch out for the defects I discover while completing the exercises when I code the corresponding assignment.	1	2	3	4	5
The level of difficulty of the exercises is reasonable.	1	2	3	4	5
I perform code reviews before compilation to identify and correct defects.	1	2	3	4	5
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.	1	2	3	4	5
As a result of completing the exercises, I have developed a process for identifying defects in the assignments.	1	2	3	4	5
I feel that the exercises are enhancing my debugging skills.	1	2	3	4	5

#### Development Logs and Reflective Memos

I kept a development log while coding the programming assignments.	1	2	3	4	5
In writing the reflective memo, I observed strengths and weaknesses in my approaches to designing, coding, and debugging the programming assignments.	1	2	3	4	5
I feel that the development logs and reflective memos helped refine my approach to programming assignments.	1	2	3	4	5
I adapted my approach to the programming assignments based on the observations made in my reflective memos.	1	2	3	4	5

### **Collaborative Assignments**

Our team for the last programming assignment worked collaboratively and performed code inspections.	1	2	3	4	5
The code inspections allowed us to locate defects more quickly.	1	2	3	4	5
It was easier to debug code as a team rather than individually.	1	2	3	4	5
I feel that debugging the last programming assignment and the final project as a team enhanced my debugging skills.	1	2	3	4	5
I feel that debugging the last programming assignment and the final project as a team enhanced my teamwork skills.	1	2	3	4	5

### **Additional Comments**

In the space below, please provide additional comments on the three activities. Your comments can address their strengths, weaknesses, opportunities for improvement, etc.

## 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] P. Hutchings, *Opening Lines: Approaches to the Scholarship of Teaching and Learning*. Menlo Park, CA: The Carnegie Foundation for the Advancement of Teaching, 2000, pp. 1-10.
- [6] T. A. Angelo and K. P. Cross, *Classroom Assessment Techniques*. San Francisco, CA: Jossey-Bass, Inc., 1993, pp. 3, 222-225.
- [7] J. A. Whittaker, "What is software testing? And why is it so hard?" *IEEE Software*, vol. 17, no. 1, pp. 70-79, 2000.
- [8] 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.
- [9] E. L. Jones, "An experiential approach to incorporating software testing into the computer science curriculum," in *Proceedings of the 31<sup>st</sup> ASEE/IEEE Frontiers in Education Conference*, vol. 2, 2001, pp. F3D 7-11.
- [10] "Computing Curricula," Dec. 2001, <http://www.computer.org/education/cc2001/final/>.
- [11] 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.

- [12] 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.
- [13] W. S. Humphrey, *Introduction to the Personal Software Process*. Reading, MA: Addison-Wesley, 1997, pp. 143-149, 159-172.
- [14] M. E. Fagan, "Design and code inspections to reduce errors in program development," *IBM Systems Journal*, vol. 15, no. 3, pp. 182-211, 1976.
- [15] T. Gilb and D. Graham, *Software Inspection*. Reading, MA: Addison-Wesley, 1993, pp. 264-279.
- [16] 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 33<sup>rd</sup> SIGCSE Technical Symposium on Computer Science Education*, 2002, pp. 38-42.
- [17] 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.
- [18] S. Uchida, A. Monden, H. Iida, K. Matsumoto, and H. Kudo, "A multiple-view analysis model of debugging processes," in *Proceedings of the 2002 International Symposium on Empirical Software Engineering*, 2002, pp. 139-147.
- [19] S. Rifkin and L. Deimel, "Program comprehension techniques improve software inspections: A case study," in *Proceedings of the 8<sup>th</sup> International Workshop on Program Comprehension*, 2000, pp. 131-138.
- [20] A. Hunt and D. Thomas, "The art of enbugging," *IEEE Software*, vol. 20, no. 1, pp.10-11, 2003.
- [21] L. Williams, "Instilling a defect prevention philosophy," in *Proceedings of the 28<sup>th</sup> ASEE/IEEE Frontiers in Education Conference*, 1998, pp. 1308-1311.
- [22] M. Deck, "Cleanroom and object-oriented software engineering: A unique synergy," presented at the 8<sup>th</sup> Annual Software Technology Conference, Salt Lake City, UT, 1996.

- [23] R. F. Grove, "Using the personal software process to motivate good programming practices," in *Proceedings of the 3<sup>rd</sup> Annual Conference on Innovation and Technology in Computer Science Education*, 1998, pp. 98-101.
- [24] D. E. Knuth, "The errors of TeX," *Software – Practice and Experience*, vol. 19, no. 7, pp. 607-681, 1989.
- [25] D. Card, "Learning from our mistakes with defect causal analysis," *IEEE Software*, vol. 15, no. 1, pp. 56-63, 1998.
- [26] R. G. Mays, C. L. Jones, G. J. Holloway, and D. P. Studinski, "Experiences with defect prevention," *IBM Systems Journal*, vol. 29, no. 1, pp. 4-32, 1990.
- [27] 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.
- [28] M. Satratzemi, V. Dagdilelis, and G. Evagelidis, "A system for program visualization and problem-solving path assessment of novice programmers," in *Proceedings of the 6<sup>th</sup> Annual Conference on Innovation and Technology in Computer Science Education*, 2001, pp. 137-140.
- [29] G. C. Lee and J. C. Wu, "Debug It: A debugging practicing system," *Computers and Education*, vol. 32, no. 2, pp. 165-197, 1999.
- [30] A. Zeller, "Automated debugging: are we close?" *Computer*, vol. 34, no. 11, pp. 26-31, 2001.
- [31] M. Eisenstadt, "My hairiest bug war stories," *Communications of the ACM*, vol. 40, no. 4, pp. 30-37, 1997.
- [32] M. C. Loui, "The case for assembly language programming," *IEEE Transactions on Education*, vol. 31, no. 3, pp. 160-164, 1988.
- [33] 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.