PERGAMON

Computers & Education 32 (1999) 165–179

COMPUTERS & EDUCATION

Debug It: A debugging practicing system

Greg C. Lee*, Jackie C. Wu

Department of Information and Computer Education, National Taiwan Normal University, Taipei, Taiwan

Received 21 December 1998

Abstract

This study reported the research findings on improving programming skills of novice programmers by way of debugging practices. There were two objectives of the debugging training: (1) to uncover and to correct any misconceptions of the programmers; and (2) to improve the debugging abilities of the programmers. To meet these objectives, a model of debugging practices, DebugIt, was presented. The proposed model called for supervised debugging practices on short programs involving frequently committed programming errors. A system, DebugIt:Loop, was developed specifically for debugging practices on programs with loop related errors. Two sets of experiments were conducted with 26 college students and 46 senior high school students enrolled in introductory Pascal courses. For each experiment, students were randomly assigned into the experimental group (using DebugIt:loop for debugging practice) and the control group (using traditional programming practices). A posttest was administered to compare the debugging and errorless programming abilities among the students in the two groups. The statistical procedure of an ANCOVA was used to analyze the gathered data. The results showed that this model of supervised debugging practices was effective in improving novice programmers' programming skills. © 1999 Elsevier Science Ltd. All rights reserved.

1. Introduction

Ever since publication of the Computing Curricula 1991 report (Tucker, 1991), recommending that closed laboratories be an integral part of undergraduate computing curricula, closed labs and related software tools have become major topics of discussions at many Computer Education Symposiums. Many research results have reported on the design and implementation of software systems that aid learning of programming languages (Canas, Bajo & Gonzalvoet, 1994; Ross, 1991; Boroni, Eneboe, Goosey, Ross & Ross, 1996; http://

* Corresponding author. Fax: 886-2-2351-2772
E-mail address: leeg@ice.ntnu.edu.tw (G.C. Lee)

0360-1315/99/$ - see front matter © 1999 Elsevier Science Ltd. All rights reserved.
PII: S0360-1315(98)00063-3

www.ulst.ac.uk/cticomp, 1998). However, there seems to be few which concentrated on debugging activities. Debugging has been known to account for more than 50% of the time and effort spend in the development of a computer program (Myers, 1997; Ward, 1988). Although structured programming can lower the risk of faulty programming, it does not guarantee bug-free programs (Williams, 1985; Benander & Benander, 1990). Debugging training is even more important for the novice programmers. Unlike experienced programmers, who can quickly locate seemly "obvious" errors or narrow down the causes of a problem, novice programmers often resort to trial and error when debugging programs. Regardless of the importance of debugging training, computer programming classes often remain concentrated on teaching programming language, language syntax, problem analysis and program design. Seldom is time allotted for debugging practices. Novice programmers are often left to develop their own debugging skills as they struggle to find the causes for errors in their programs.

Although many studies have been reported on the subject of teaching debugging skills, there has been no consensus on what this entails. Benander (Benander & Benander, 1989) advocate that debugging teaching is to teach techniques such as placing extra output statements in the program to produce intermediate outputs, or using the single-step mode of the compiler's debugging environment to trace execution of the program. After comparing debugging behaviors among experienced and novice programmers, Gugerty and Olson (1986) concluded that program comprehension ability, not debugging strategy, is the sole factor in being able to debug programs efficiently. Among the many program debugging strategies proposed in the literature, program comprehension was often cited as the first and foremost criteria for being able to debug effectively (Kessler & Anderson, 1986; Vessey, 1985). Furthermore, the reason that structured programming can result in less faulty programs is that the inherent nature of structured programming requires programmers to better understand the problem at hand and to design a better programming plan (Stone, Eleanor & Wright, 1990). Thus, improving program comprehension ability is one of the keys to improving debugging ability. This is even more so for novice programmers because they are often trapped coping with the programming syntax rather then dealing with programming logic when debugging their own programs.

Another distinct trait separating experienced programmers from novice programmers is that experienced programmers can make use of previous debugging experiences while debugging (Gugerty & Olson, 1986; Gould, 1975). As experienced programmers (debuggers), they are more aware of the errors commonly found in programs written by novice programmers; therefore, are less likely to commit them. The novice programmers gradually learn to write programs with less bugs as they become aware of the common errors through debugging their own programs. However, in most introductory programming courses, students only have to write a few programs so the accumulation of debugging experience is slow and limited. Intensified debugging practice is needed to help novice programmers quickly gain experience and improve debugging skills.

The goal of helping novice programmers quickly becoming better programmers, in terms of both debugging programs and writing error free programs, can be achieved by improving their program comprehension abilities and accumulating debugging experiences. This paper presents a system and rational for debugging training for novice programmers. In developing systems that facilitate debugging training, several factors must be considered. First, there should be a

large, varied set of bugged programs on which to practice. In addition, the system should be able to provide helpful debugging hints whenever requested. Furthermore, all programs should be short in nature so as to encourage the students to comprehend the logic of the given bugged program rather than guessing for the correct solution. To ensure better practice results, bugs in the programs should not be randomly generated. Instead, mistakes frequently committed by novice programmers should be collected and embedded into the practice programs. Finally, an autonomous problem selector is needed to select appropriate problem for practice.

DebugIt is a debugging practicing system designed specifically with the above features in mind. It provides an environment for the novice programmers to practice debugging and to accumulate debugging experiences through system supervised debugging activities. In the next section, we will describe the architecture of the DebugIt system. An implementation, specifically for debugging programs involving loop constructs, is presented. Field testings were conducted at both the college and at the high school levels. The results are analyzed and discussed. Conclusions are reported in the last section of the paper.

2. DebugIt: system architecture

The goal of DebugIt is to help novice programmers to quickly become better in terms of both debugging programs and writing error free programs. The key to reaching this goal is to improve program comprehension ability and to accumulate debugging experiences. Our system centers around being able to provide supervised debugging activities. By supervised activities, it is meant that the system should not only provide suitable problems for practices, but should be able to provide timely hints, to analyze students' debugged programs for correctness, and to

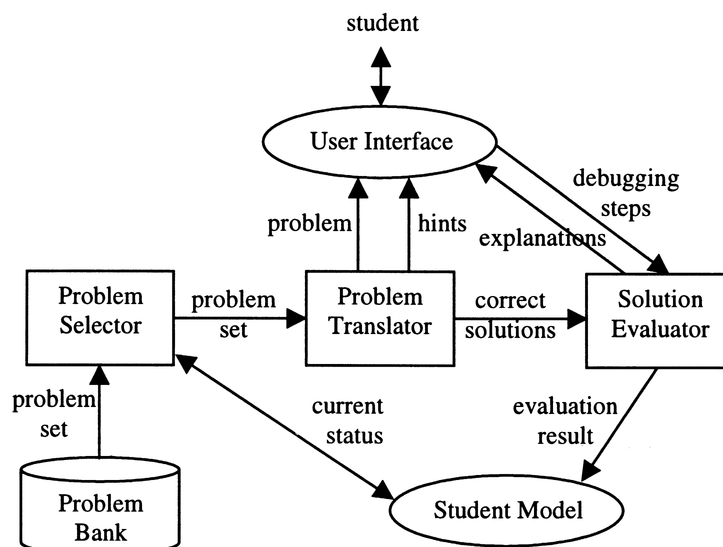

Fig. 1. The six major components of DebugIt.

explain the embedded bugs. Furthermore, to encourage program comprehension, the programs provided for practices need to be short so that students will not be frightened away by seeing the length of the program. Below are explanations of the six major components of the DebugIt system. Graphical representation of their interconnections is depicted in Fig. 1.

- *Problem Bank*: The problem bank contains problem sets of various difficulty levels. A complete problem would include the problem statement, the bugged program, hints for the students, and the correct solutions.
- *Student Model*: The student model is used to record the student's learning progress.
- *Problem Selector*: By referencing the student model for an individual's learning progress, the problem selector selects problems of appropriate difficulty for the student to practice.
- *Problem Translator*: The problem translator translates the given pseudo-coded program into the programming language specific plan. Currently, the programming language of choice is Pascal.
- *Solution Evaluator*: The solution evaluator compares changes made to the bugged program against the correct solutions.
- *User Interface*: The user interface provides a gateway between the student and the internal representation of all the data.

The working logistics of the DebugIt system is as follows: When a student registers to use the system for the first time, the *student model* is initialed. As practice progresses, the student model is updated to reflect his/her current state of debugging knowledge. Upon a debugging practice request, the *problem selector* selects, according to the current state of the student model, an appropriate problem from the *problem bank*. The *problem translator* then translates a bugged version of the solution program to a Pascal program. The problem statement and the bugged program are given to the student for debugging practices. Hints may be provided if requested at anytime during the practice session. Once debugging is completed, instead of comparing the "debugged" program to the correct program line by line, the *solution evaluator* checks the changes made to the initial bugged program against the possible "correction steps" that are provided as part of the problem set. The "correction steps" are logically represented as a finite state machine. As such, only the correct set of debugging steps and statements will terminate in a final state. This representational scheme has the advantage of being able to represent the most common correction procedures compactly. The result of the evaluation is sent back to the student for explanation and to the student model for updating purposes. Detailed discussions on the technical aspect of the system can be found in Lee and Wu, 1997.

3. DebugIt:Loop—an implementation

The current version of DebugIt contains debugging exercises for loop constructs. The 12 most often committed novice-programming errors associated with loops, as shown in Table 1, are collected from literature (Stemler, 1989; White, Collins & Gremillion, 1988; Pea, 1986; Johnson, Soloway, Cutler & Draper, 1983) and from our teaching experience. Twenty sample problems have been designed for the DebugIt:Loop system. The corresponding programs are

Table 1
The twelve most often committed loop construct related programming errors by novice programmers

1.	Initialization of loop control variable is incorrectly placed.
2.	Incorrect (including none) initialization of loop control variable.
3.	Division by 0 (within loop).
4.	Unnecessary output statement within the loop.
5.	Incorrect update of the loop control variable.
6.	Improper loop stopping condition, resulting in infinite loop.
7.	Improper loop stopping condition, resulting in wrong number of loop execution.
8.	Incorrect loop contents, resulting in useless loop.
9.	Improper handling of trailing record, resulting in reading past end of all input.
10.	Improper check of input values.
11.	Illogical output values.
12.	Missing input statement inside the loop, resulting in only one set of data read.

written in Pascal and all programs are embedded with one or more of the 12 common errors. The system has been implemented using LISP in a PC Windows environment.

When using DebugIt:Loop, the user must first register so that personal learning history can be kept in the student model. The system will present problems one at a time. When debugging, there is no limit on the number of times to move, to delete, to insert or to modify the given program. When completed, the system will analyze the ‘‘debugged’’ version of the program for correctness and give feedback. If the program is not correctly debugged after three attempts, the correct program along with the original mistakes will be shown. In addition, an

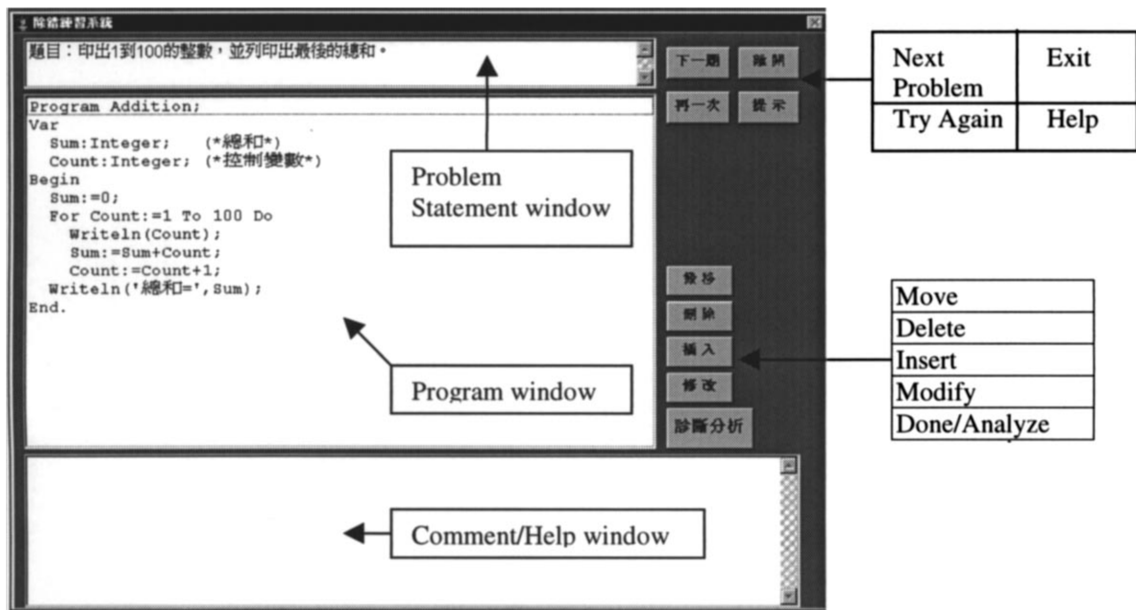

Fig. 2. Main screen of the DebugIt:Loop system.

explanation is given. An example is given in Fig. 3, where the user is being advised about the incorrect usage of "while" in place of "if" and the consequences of such mistake. The system will record the student's progress and come back to the same problem in the future to give him/her another chance at debugging the program.

3.1. The problem bank

The problem bank currently has 20 problems of various difficulty levels. Each problem is made up of several components. An example is depicted in Table 2. The "text" section contains the problem statement, which gives a word description of what the bugged program is suppose to do. The "program" section contains a bugged Pascal program for solving the given problem. This is the program to be debugged if this particular problem were selected. The "solution" section contains the most natural ways of correcting the bugs in the bugged program. The "hint" section contains debugging hints that are displayed when requested. The "explain" section contains one set of debugging procedures for correcting the bugs that are to

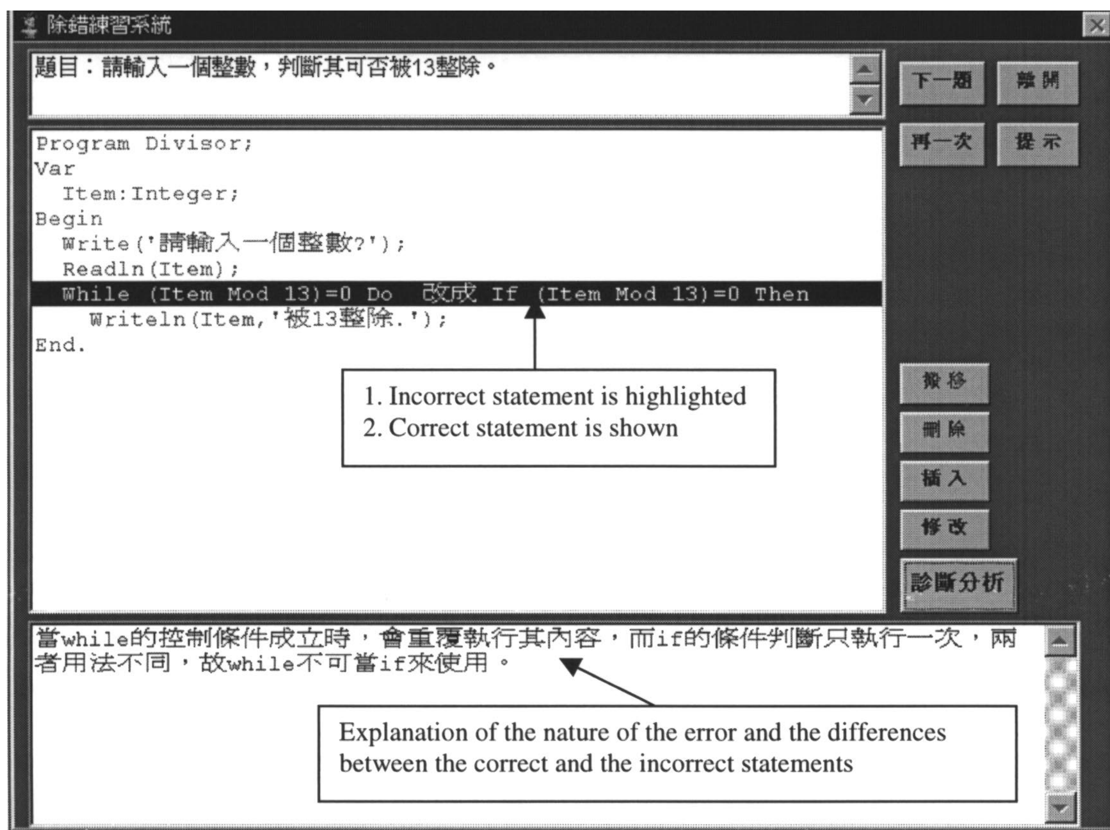

Fig. 3. Screen of DebugIt:Loop after 3 unsuccessful debugging attempts.

Table 2
List representation of a complete problem in the Problem Bank

```
( (text (“Problem: Find the largest n for which 2 to the nth power is < 1000”))
(program ((0 “program orders;”)
   (0 “var”)
   (2 “product,order:integer;”)
   (0 “begin”)
   (2 “product: = 0;”)
   (2 “order: = 0;”)
   (2 “while product < 1000 do”)
   (4 “begin”)
   (6 “product: = product*2;”)
   (6 “order: = order + 1;”)
   (4 “end;”)
   (2 “writeln(’n is’,order);”)
   (0 “end.”)))

(solution (brnchs (4 “product” “:=” “1” “;” rtn 2
   11 and “order” “–” “1” and “;” rtn 11 end-rtn)
   (4 “product” “:=” “1” “;” rtn 2
   11 “order” “:=” “order” “–” “1” “;” rtn 11 end-rtn)))

(hint (“Mistake 1: initial value of loop control variable.
Mistake 2: output value does not conform to the output specification of the problem.”
))

(explain (4 M “change 0 to 1” 11 M “change order to order – 1”))

(analysis (“1. If the initial value of ‘product’ were set to 0, multiplication results will always be 0. Therefore, the
initial value of ‘product’ should be set to 1.
2. Since the loop is exited when the ‘product’ is greater than 1000, therefore must decrement ‘order’ by 1 to yield
the correct answer."
)))
```

be shown after three unsuccessful attempts at debugging the program. Finally the “analysis” section gives an analysis of the misconceptions embedded within the bugged program. The analysis includes an explanation of the nature of the error and the differences between the correct and the incorrect statements. For example, in Table 2, the problem is to find the largest n so that 2^n is less than 1000. The sample bugged program contains 13 lines and 2 bugs. The number in front of each Pascal statement is used internally by the user-interface module to indent the statement when it is shown on the screen. There are two natural ways to correct the embedded bugs in the program. One is to modify line 4 so that “product” will have initial value of 1 and to modify the output statement on line 11 so that the printed “order” is one less than the value stored in “order.” The other is to set the initial value of “product” to 1 on line 4 but to insert a statement to decrement “order” by 1 before line 11. All the non-quoted (non-Pascal like) symbols are used for internal processing purposes.

3.2. The student model

In this research, the Student Model is used only to record the student's practice progress. Thus, there is no need for a complex model as would be required for an intelligent computer assisted instructional or learning system. Here, the Student Model simply keeps track of the problems that each student has attempted, had correctly solved, and all the correct and the incorrect solution programs that were previously given by the student. The information on the problems attempted and correctly solved is useful to the Problem Selector in selecting the next problem for practice. The incorrect solutions produced by the student can be analyzed to help detect the student's misconceptions on the loop construct. In the future, a record of each student's debugging activities in terms of the use of hints, the number of attempts, elapsed time, etc. could be included. These records would be of importance to instructors for determining individual and class progress.

3.3. The problem selector

Since there are only 20 problems to select from, the problem selection strategy is relatively straightforward: each problem is selected in turn, skipping those that have already been solved correctly. If a problem is incorrectly solved, then that same problem will be selected again after a random number of other problems have been attempted.

3.4. The problem translator

The task of the Problem Translator is one of reformatting the program codes and text associated with each problem suitable for display on the window screens. When a new problem is selected, the Problem Translator is responsible for displaying the problem statement and the bugged program on screen. When the student requests debugging help, it displays the message contained in the "help" section of that problem set. When a student fails to correctly debug a program after three attempts, it will automatically "debug" the originally bugged program, according to the debugging procedure described in the "explain" section, and will display the debugged program and the problem analysis provided in the "analysis" section.

3.5. The solution evaluator

The Solution Evaluator is responsible for determining the correctness of the "debugged" program. Instead of comparing the debugged program word for word, our approach is to compare the changes made against changes that should have been made to correctly debug the program. The modifications are treated as sentences that are parsed using the built-in solutions as grammar rules. If the modifications are successfully parsed, then the program is deemed to be correctly debugged. To do so, the line numbers and the statements of all the final changes made on each line of the original bugged program are extracted. The texts of the extracted statements are then tokenized before parsing took place. Thus, the irrelevant details, such as correction order, can be hidden from the evaluation process. For detail discussions of the technical issues, please see Lee and Wu, 1997.

3.6. The user interface

The main screen of DebugIt:Loop, as shown in Fig. 2, is divided into 3 window frames and a command buttons section. The Problem Statement window is used to display the problem statement. The Program window shows the program that is currently being debugged. The Comment/Help window is used by the system to give help or feedback to the user. There are two sets of command buttons. The top set of buttons allows the user to select a new problem, to re-debug the current problem, to ask for help on debugging the displayed program, and to leave the system. The lower set of buttons allows the user to move, to delete, to insert, and to modify any program statement in the Program window. When the debugging is completed, the "done/analyze" button can be used to check the correctness of the program.

4. Field tests

The DebugIt:Loop system was put to use in two introductory computer programming classes to assess the effectiveness of intensified debugging practices on novice programmers. There were two phases of the experiment. The practice session allowed the students to practice debugging with/without DebugIt. A post-practice achievement test was used to evaluate the effectiveness of the practices.

4.1. Subjects

Two groups of students with different programming experiences were selected for the experiment. The first group consisted of 26 college freshmen majored in computer science at the National Taiwan Normal University. At the time of the experiment, these students had completed one semester of the Introduction to Computer Science course (CS1) with emphasis on Pascal programming and had written at least four Pascal programs. The second group of students was comprised of 46 Yi-Lan Senior High School sophomores, all of whom were enrolled in a Pascal programming course. The experiment was conducted immediately after the instructor had completed lectures on looping constructs. The students had written only one Pascal program before the experiment. The students in both sections (college and high school) were randomly divided into experimental and control groups of equal size.

4.2. Procedure

The experiment was carried out in a computer room equipped with PCs. Each student had access to a PC during the entire session. After a 10-min introduction about the experiment, the experiment group and the control group practiced simultaneously and the practice time was approximately 150 min. The experimental group was instructed to practice debugging using the DebugIt:Loop. A total of 20 "bugged" programs were available in the DebugIt:Loop for practice. Students were not allowed to ask questions and had to work individually. Also, the Pascal compiler was not available to the students in this group; therefore, they had to work solely in the DebugIt:Loop environment. On the other hand, students in the control group

practiced using the traditional method of writing and debugging their own programs. A PC with a preloaded Pascal compiler was assigned to each student. The problem statements of the 20 problems in DebugIt:Loop were given to this group for practice so that both group of students could work on the same problems. Also, students were specifically told to use the loop construct when solving the problems and were allowed to help each other during the practice session.

At the end of the practice session, a 50-min post-practice achievement test was administered to all students to evaluate the effectiveness of the practices. In addition, students in the experimental group were asked to complete a questionnaire (see Table 9) providing feedback on the DebugIt:Loop debugging practicing tool.

4.3. Achievement test

The achievement test was a written exam that contained two parts: program debugging and program writing. The program debugging part of the test asked the students to trace and to debug the given "bugged" programs. Programs with single and multiple bugs were both present in the test. This part of the test was to measure the students' abilities to recognize and to correct mistakes frequently committed by novice programmers as shown in Table 1. The programming part of the test asked the students to write programs to solve particular problems. The problems were carefully designed so that loop constructs seemed to be the natural choice for solving them. Furthermore, students were instructed to use loops in their programs when solving the problems. These problems were intended to measure the students' abilities to write error free programs in terms of the errors listed in Table 1.

For the college students, the achievement test consisted of 9 debugging problems (containing 19 logical errors) and 2 programming problems. For the debugging problems, one point was awarded for each logical mistake corrected while one point was deducted out of possible 7 points for each logical mistake made in the two programming problems for a maximum possible score of 26. The *Cronbach's Coefficient Alpha* was used to determine the internal consistency of the achievement test and the value obtained was 0.76 with $n=26$. Taking into consideration that the high school students were less experienced in programming than the college students, an achievement test with only 5 debugging problems (containing 15 logical errors) and 2 programming problems was administered to the high school students. The maximum possible score was 22 points.

5. Results and discussion

5.1. Achievement results

The one-way Analysis of Covariance (ANCOVA) was performed to test the difference among the achievements of the experiment and the control group. Since no pretest was administered, the college group grades in the CS1 course, the high school students grades of the first examination in their Pascal Programming course were used as the covariate in the analysis.

Table 3
Descriptive statistics of the achievement test scores

Subjects	Group	n	Achievement test scores		Previous grades (covariate)	
			Mean	SD	Mean	SD
College	Experimental	13	16.00	3.32	76.08	9.35
	Control	13	10.85	4.06	78.08	12.99
High School	Experimental	23	8.04	4.02	4.00	2.61
	Control	23	4.17	2.06	3.70	1.79

Tables 3 and 4 present the results of the statistical analyses of the study. The descriptive statistics for the ANCOVA analysis are depicted in Table 3, whereas Table 4 presents a summary result of the ANCOVA analysis on the overall post-practice achievement test. For both the college and the high school students, the ANCOVA results (college: $F=26.94$, $P = 0.0001$; high school: $F=21.73$, $P = 0.0001$) indicate that the experimental group scored significantly higher than the control group on the overall achievement test. It can be concluded from the study that debugging practices can improve a novice programmer's programming ability.

To further analyze the above conclusions, an analysis was made on the scores of both the debugging and programming parts of the achievement test. The results are depicted in Tables 5–8. Tables 5 and 6 gives the descriptive statistics and the summary result of the ANCOVA analysis of the debugging test. The ANCOVA results (college: $F=14.40$, $P = 0.0009$; high school: $F=26.76$, $P = 0.0001$) indicate that there was a significant difference between the scores among the two groups. This result coincides with our intuitive feeling that the experiment group should perform significantly better at debugging exercises after having received debugging training.

The summary results of the ANCOVA analysis of the programming achievement test are given in Tables 7 and 8. Unlike the results on the debugging test, analysis on the scores of college students and high school students yielded two different results. The ANCOVA result ($F=18.66$, $P = 0.0003$) for the college students indicated that the experiment group committed

Table 4
Summary results of the ANCOVA analysis on the overall achievement test score

Subjects	Source	SS	d.f.	MS	F	P
College	Between groups	202.53	1	202.53	26.94*	0.0001
	Error	172.93	23	7.52		
High School	Between groups	28.20	1	28.20	18.66*	0.0003
	Error	34.75	23	1.51		

* $P<0.05$.

Table 5
Descriptive statistics of the debugging achievement sub-test scores

Subjects	Group	*n*	Achievement test scores		Previous grades (covariate)	
			Mean	SD	Mean	SD
College	Experimental	13	11.15	2.58	76.08	9.35
	Control	13	7.92	3.01	78.08	12.99
High School	Experimental	23	6.04	2.70	4.00	2.61
	Control	23	2.96	1.66	3.70	1.79

Table 6
Summary results of the ANCOVA analysis on the debugging achievement sub-test score

Subjects	Source	SS	d.f.	MS	*F*	*P*
College	Between Groups	79.58	1	79.58	19.40*	0.0009
	Error	127.08	23	5.53		
High School	Between Groups	97.72	1	97.72	26.76*	0.0001
	Error	157.05	43	3.65		

Table 7
Descriptive statistics of the programming achievement sub-test scores

Subjects	Group	*n*	Achievement test scores		Previous grades (covariate)	
			Mean	SD	Mean	SD
College	Experimental	13	4.85	1.46	76.08	9.35
	Control	13	2.92	1.61	78.08	12.99
High School	Experimental	23	2.00	1.73	4.00	2.61
	Control	23	1.22	1.13	3.70	1.79

Table 8
Summary results of the ANCOVA analysis on the programming achievement sub-test score

Subjects	Source	SS	d.f.	MS	*F*	*P*
College	Between Groups	28.20	1	28.20	18.66*	0.0003
	Error	34.75	23	1.51		
High School	Between Groups	5.53	1	5.53	3.14	0.08
	Error	75.64	43	1.76		

* $P < 0.05$.

significantly fewer loop construct related errors than the control group when writing programs from scratch. However, this conclusion was not reached for the experiment with high school students ($F = 3.14$, $P = 0.08$). In terms of the frequently made mistakes made by novice programmers, the experimental group, even after debugging practicing session, did not make significantly fewer mistakes than the control group. This can be attributed to the fact that these students were not at the "novice programmer" level. The students had been taught Pascal programming for only 14 h and had written one relatively straightforward program before the experiment. Unfamiliarity with program writing was most evident in the programming test where syntax errors were frequently found.

5.2. Questionnaire results

The questionnaire consisted of eight closed questions and three open questions. The responses to the questions are summarized in Table 9. The numbers shown are the percentage of students who responded "strongly agree" or "agree" to each of the statements on the questionnaire. The results showed that students have genuine interests in learning programming; yet they do not think debugging is an easy task (Statements 1–2). Although the majority of students agreed that debugging practices did help in improving their program debugging abilities (Statement 3), they lacked debugging practice opportunities (Statement 4). As for the practice tool, DebugIt, the students expressed favorable opinions: DebugIt is a good debugging practicing tool and this model of debugging practices helped clarified programming misconceptions (Statements 5–6). The college students and the high school students had a difference of opinions on how the study was conducted (Statements 7–8). Most of the college students were clear as to what to do after the pre-experiment explanations, although some thought the debugging practice time was not enough. On the other hand, the high school students, due to lack of programming experiences, had more difficulty in understanding what to do and consequently needed more time to complete the practice session.

As for the open questions, most students were positive about both the DebugIt tool and the overall experiment. Some students cited that after practicing with DebugIt, programming

Table 9
Percentage of students who " strongly agree " or " agree " with each of the statement on the post-experiment questionnaire

	Statement	College	High School
1.	You enjoyed the Pascal programming course	85%	61%
2.	Program debugging is easy	23%	26%
3.	Program debugging practices improved your debugging ability	100%	87%
4.	Before DebugIt, you often had opportunities for debugging practices	46%	17%
5.	DebugIt is a good tool for debugging practicing	84%	87%
6.	DebugIt helped you clarified some programming misconceptions	92%	87%
7.	The introduction before the experiment is adequate	77%	43%
8.	There is enough time for the experiment	61%	26%

misconceptions were clarified. Still others stated that they now have a better understanding of the loop constructs. Because of the interactive nature of the software system, as compared to paper and pencil debugging activities, the DebugIt system provided immediate feedback on the actions taken by the students. It provided timely hints and analyses of the problems. Furthermore, help could be requested asynchronously by students in the experimental group and be tailored to meet individual student's need. The instructors also benefited from the use of the software system. They can now concentrate on helping those students with special needs, such as students who write awkward programming codes. This would all be impossible if the debugging activities were carried out in a non-computer software aided environment. However, students also gave suggestions for future improvements. One of the most frequently raised suggestions was to improve the system's ability to recognize "correctly" debugged programs. Although the system was designed to recognize the most common ways of solving given problems, it was by no means complete. Going through all the solutions given by the students, it was found that some were correct, albeit awkward. Since such programs should not be encouraged, it is planned that in the future they will be included as correct solutions but explained as to their awkwardness in the comment/help window. Other suggestions included expanding the scope of DebugIt to include other programming constructs and better explanations of the system's operations. These and other comments provided many directions for future improvements.

6. Conclusion

The research reported in this paper was about improving the programming skills of novice programmers by means of program debugging practices. The model called for supervised debugging practice on short programs, involving frequently committed programming errors. The novice programmers were required to comprehend the bugged program before making changes. After being exposed to and having corrected frequently committed mistakes, the novice programmers were less likely to produce these erroneous codes in their own programs. DebugIt:Loop was developed to test these hypotheses. An experimental study was conducted and the results showed that this model for supervised debugging practice was effective in improving novice programmers' programming skills.

The result of the empirical study was most encouraging. However, improvement is already underway. We are currently working on expanding the scope of DebugIt to cover other programming constructs introduced in the CS1 and CS2 courses. It was suggested by some of the participants that a program execution simulator for simulating execution of the program would be a valuable addition to DebugIt. However, the addition of such a simulator might have an adverse effect in that the user, not concentrating on comprehending the program at hand, may rely on a trial-and-error way of debugging. The benefits of such a simulator need to be further investigated. Finally, we would like to add a friendlier user interface for creating and updating problems in the Problem bank so that instructors may tailor the problem sets in the Problem bank to suite their own need.

Acknowledgements

The research reported in this paper has been partially supported by the National Science Council of Taiwan, the Republic of China, under the grant number NSC 86-2511-S-003-035. The authors would like to thank the reviewers for their thoughtful comments and to acknowledge Dr. Cheng-Chih Wu for his help in reviewing this manuscript.

References

Benander, A. C., & Benander, B. A. (1990). An empirical analysis of style characteristics in COBOL programs relating to program correctness. *The Journal of Computer Information Systems*, (Spring).

Benander, A. C., & Benander, B. A. (1989). An analysis of debugging techniques. *Journal of Research on Computing in Education*, (Summer), 447–455.

Boroni, C. M., Eneboe, T. J., Goosey, R. W., Ross, J. A., & Ross, R. J. (1996). Dancing with Dynalab. *ACM SIGCSE Bulletin*, *28*(1), 135–139.

Canas, J. J., Bajo, M. T., & Gonzalvo, P. (1994). Mental Models and Computer Programming. *International Journal of Human-Computer Studies*, *40,* 795–811.

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

Gugerty, L., & Olson, G. M. (1986). Comprehension differences in debugging by skilled and novice programmers. In: E. Soloway, & S. Iyengar, *Empirical Studies of Programmers. Norwood, NJ: Ablex* (pp. 13–27).

http://www.ulst.ac.uk/cticomp, 1998, December.

Johnson, W. L., Soloway, E., & Cutler, B., Draper, S. (1983). Bug Catalogue: I. Technical Report 286. Department of Computer Science Yale, New Haven, CT, October.

Kessler, C. M., & Anderson, J. R. (1986). A model of novice debugging in LISP. In: E. Soloway, & S. Iyengar, *Empirical Studies of Programmers*. Norwood, NJ: Ablex.

Lee, G. C., & Wu, J. C. (1997). The effect of systematic debugging practices on debugging skill of novice programmers. Technical Report. Department of Information and Computer Education, National Taiwan Normal University, Taipei, Taiwan.

Myers, G. J. (1997). A controlled experiment in program testing and code walkthroughs/inspections. *Communications of the ACM*, *21,* 760–768.

Pea, R. (1986). Language independent conceptual "bugs" in novice programs. *Journal of Educational Computing Research*, *2*(1), 25–36.

Ross, R. J. (1991). Experience with the DYNAMOD Program Animator. *ACM SIGCSE Bulletin*, *23*(1), 35–42.

Stemler, L. (1989). Effects of Instruction on the misconceptions about programming in BASIC. *Journal of Research on Computing in Education*, (Fall), 26–34.

Stone, D. N., Eleanor, W. J., & Wright, M. K. (1990). The impact of Pascal education on debugging skill. *International Journal of Man-Machine Studies*, *33,* 81–95.

Tucker, A. B. (1991). A summary of the ACM/IEEE-CS joint curriculum task force report: Computing Curricula 1991. *Communications of the ACM,*, *34*(6), 69–84.

Vessey, I. (1985). Expertise in debugging computer programs: A process analysis. *International Journal of Man-Machine Studies*, *23*(5), 459–494.

Ward, R. (1998). Beyond design: The discipline of debugging. *Computer Language* (April), 37–38.

White, K. B., Collins, R. W., & Gremillion, L. (1988). An experimental investigation of software error detection by students. *Journal of Research on Computing in Education* (Summer), 392–408.

Williams, G. (1985). Debugging techniques. *Byte* (June), 279–290.