An Empirical Study of Debugging Patterns Among Novices Programmers

Basma S. Alqadi
Department of Computer Science
Kent State University
Imam Muhammad Ibn Saud Islamic University
Riyadh, Saudi Arabia
balqadi@kent.edu

Jonathan I. Maletic
Department of Computer Science
Kent State University
Kent, Ohio 44242
jmaletic@kent.edu

ABSTRACT
Students taking introductory computer science courses often have difficulty with the debugging process. This work investigates a number of different logical errors that novice programmers encounter and the associated debugging behaviors. Data is collected and analyzed data in two different experiments from 142 subjects. The results show some errors are more difficult than others. Different types of bugs and novices' debugging behaviors are identified. Years of experience showed a significant role in the process of debugging in terms of correctness level and time required for debugging.

CCS Concepts
• **Social and professional topics→Computing education→CS1**

Keywords
Debugging; novice programmers; logical errors; user study

1. INTRODUCTION
Debugging is a necessary process that is known to be difficult for novice programmers to learn and for instructors in the field of computer science to teach. Students need to understand the operation of the program and the execution of the buggy program; have knowledge in application domain and the programming language; and have knowledge of (specific) bugs and debugging methods [5]. Unfortunately, these skills are fragile for most novice programmers [14]. Educators note that students made the same mistakes and become frustrated trying to understand the error messages and correct their code; often wasting hours of time on a simple error.

By better understanding how novice programmers understand and deal with errors, we will be able to improve tool support to assist novices in learning to program. Educators can also adapt their curriculum to cover the specific problems/bugs that students encounter and thus improve the learning process.

There are several types of errors that student make and encounter when programming. It is useful to distinguish between syntactic errors; those caught by compilers and non-syntactic errors that compilers do not detect. A syntactic error occurs due to violation in the grammar rules of the language. The other type is logical errors that prevents the program from running as intended. A logical error occurs when the code is syntactically correct, but the program does not execute properly. This type of errors is typically quite challenging and more difficult for students to find and fix.

The goal of this paper is to experimentally study novice programmers' debugging common logical errors in order to better understand roadblocks to learning. While there have been a number of studies that examine student's debugging ability, many of those experiments are now somewhat dated [9, 16]. Additional empirical studies to support or refine these experiments in the light of current languages can offer insights into how to improve teaching of debugging.

The recent empirical studies that investigate how novices do such things as fault localization or bug fixing mainly studied language liabilities and little research is dedicated to directly investigate logical errors. Hristova et al. [7] attempted to specifically create a comprehensive list of Java errors. Ahmadzadeh et al. [1, 2] and Jackson et al. [8] only look at errors that can be found by compiler. All these errors must be faced by student before any other debugging begins. The work here focuses on run-time semantics and logical errors. Additionally, this previous work examines the frequency of errors for which students often seek help. However, it is not obvious that seeking help is a valid index of frequency. Here we investigate error difficulty by injecting faults into code, hence the bugs have an equal likelihood to be found or missed.

The remainder of this paper is structured as follows. Section 2 discusses related research on novice programmers and debugging. Section 3 and 4 give the research questions and methodology applied in two experiments. In Sections 5 and 6, we discuss the experiments results. Finally, Section 7 and 8 conclude the paper with threats to validity and future work.

2. RELATED WORK
Researchers have focused on identification of specific bugs for different reasons: some used their results to create debugging tools [7, 9] while others to gain insight into computer programming education [1, 6, 10]. Various methods have been used including surveys, interviews, talk aloud exercises, observing students while they solve problems, and hand analysis of program assignments. In a study focusing on goal/plan analysis, Spohrer et al. [16] compared the first compiled versions of programs submitted with later versions and confirm that ''breakdowns''

SIGCSE '17, March 08-11, 2017, Seattle, WA, USA
 ISBN 978-1-4503-4698-6/17/03 $15.00
DOI: http://dx.doi.org/10.1145/3017680.3017761

between goals and plans led to many of the bugs found. An extension of the work by Johnson et al. [9] described the bugs found through hand inspecting of introductory Pascal students' solutions to a typical programming problem and creates a catalog of novice programming bugs by using the goal/plan model. Some categorizations emerged as common knowledge from decade of teaching experience [6]. Danielsiek et al. identified multiple algorithm and data structure topics which are prone to error [4].

Hristova et al. [7] used surveys of faculty, teaching assistants, students, and their own teaching experience to identify common Java errors and misconceptions. [1] Studied novice bugs by categorizing compiler-detected errors. The authors in [8] conducted a study of novice Java errors during a first semester course. They used a logging program to capture the errors every time students compiled code. [3] Examined the problems that students believe they cannot solve on their own and for which they ask tutors for assistance. [15] attempted to classify student problems and mistakes in a first semester programming course as concept-based or knowledge-based mistakes.

A more recent study, Lewis and Gregg discussed the benefits of introducing certain debugging tools earlier or later in the curriculum [10]. Both [11] and [12] investigated novices debugging strategies and suggested that skills at debugging are distinct from general programming ability which deserves individual attention pedagogically.

3. RESEARCH QUESTIONS

The research questions in this study are intended to provide a better understanding of the most common errors that students are struggling with the most, as evidenced by the fix times for the errors and the correctness level. The research addresses 3 main questions: R1) What errors are most difficult for students to locate? R2) How long does it take students to find and fix these errors? R3) How well do students fix these errors?

We collect the following information to address these questions: the type of recognized/missed error, the time needed to locate such an error, the students' ability to connect the error to its type, and time to fix the error.

4. METHOD

In order to investigate the research questions, two differently experiments were conducted over two years. The first is mainly concerned with the ability of students to locate errors and their level of confident in identifying the type of the located errors. We believe that correctly specifying the type of the located error is a reasonable indicator of how well the student understands the error and how to eventually fix it. The second experiment aims to examine the debugging process in a real setting where the students are able to execute the program and determine the problem with the program by comparing correct and incorrect output.

4.1 First Experiment

In the first experiment the study group included 59 students. The participants are mainly students of CS II course. A small number of graduate students, with little programming experience (i.e., novice programmers) are also included in the study. Information about the participants for both studies is summarized in Table 1.

4.1.1 Types of Bugs Used

For the first experiment we studied eight different types of bugs as listed in Table 2. We use the types 1-10 in Table 2 to refer to the bug types in the rest of the paper. The types are selected because they occur frequently and/or are especially difficult for novice programmers to detect based on previous literature [3, 6, 7].

Table 1. Overview of participants for both studies.

Feature	Experiment 1	Experiment 2
	No. of students (59)	No. of students (83)
Age		
25 or younger	47	76
26 or older	12	7
Gender		
Female	11	7
Male	48	76
Current class standing		
Undergraduate	53	83
Graduate	6	0
Years of experience in general programming		
2 or less	48	69
3 or more	11	14

The first bug type happens when using single equal sign "=" to check equality in a loop condition instead of using double equal signs "==". Thus, the program will assign the value on the right side of the expression to the variable on the left hand side, and the result of this statement is always TRUE. Therefore, the loop will never end. The next error (2) is a loop that has no body and causes the program to function improperly. This happens when an extra semicolon is incorrectly placed after for or while statement. Another common type of bug (3) is forgetting a break in a switch statement, thus causing a program to execute improperly. C/C++ does not break out of a switch statement when a case is encountered unless it hits the break statement.

Table 2. The bugs used in the experiment.

Bug Type	Bug Description
1	Using a single '=' when ''==" is intended
2	Loop has no body- extra semicolon
3	Misusing Switch statement, missing break.
4	Misusing && and || operators
5	Division by integers, so quotient gets truncated
6	Condition variable has not been updated
7	Array index out of bounds
8	Attempt to access deleted dynamic variable
9*	Off by one
10*	Buffer overflow
*These bugs included in the second experiment only.	

Another common bug (4) is the case when an infinite loop happens because the programmer uses an OR instead of AND, thus never evaluating to FALSE. A division by integer error (5) occurs when both operands are integers and the division yields a possibly fractional result but without a cast operator. So the remainder will be discarded. The bug 6 is when a condition is specified in a loop, but a terminating condition is not specified, this causes the program to behave improperly. In this case the variable used to repeat the action does not update so the loop condition remains TRUE forever.

The array out of bounds (7) is a common bug. It occurs when an index into an array exceeded the allocated memory. Finally, properly freeing dynamic objects (8) turns out to be a rich source of bugs. Dynamically allocated objects have a lifetime that is

independent of where they are created; they exist until they are explicitly freed. A frequent error occurs when programmer attempt to access dynamic variable that already been deleted and no longer exists. The remaining two bugs are discussed with the next experiment.

We developed eight short programs, each representing one type of the listed bugs. For each program the logical error is manifested by a single line of code. That is, the bug can be fixed by modifying only one line of code. All programs are syntactically correct, but executing each produces incorrect output. More detail on the survey questions and materials for our studies can be found online.

4.1.2 Experimental Setup

The 8 programs are deployed in a web application that composed of 10 parts. The first part contains detailed instructions and general guidelines of how to proceed in the experiment. Followed by a second part that collects general information about the participant. The following 8 parts each is dedicated for one type of the bugs. The order of these 8 programs are randomly ordered for each participant to help avoid any learning bias.

Each participant is instructed to debug each of the 8 programs by identifying the one line in each program that contains the bug. Subjects are not required to correct the bug, however, they are asked to map the located error to its corresponding type in the list of bug types. They are asked to choose the one that best match the located error. In addition to the code and the list of bug types, a sample of a correct output that the program fails to produce is given. Participants are informed that all programs are syntactically correct.

Timing was recorded by calculating the difference between time of successfully loading each question in the user screen till the time of successfully submitting each part. Any incomplete experiment was discarded and not included in the result.

Correlation between time and number of correct answers with general features including age, gender, current class and years of experience of general programming are assessed using Mann Whitney U Test. All p-values are two-sided and considered statistically significant if less than 0.05.

4.2 Second Experiment

The study group included 83 students. The participants are made up of students of CS II class. The experiment was held in the following 2 semesters of the first experiment. Information of the participants is summarized in Table 1.

4.2.1 Types of Bugs Used

In addition to the eight types of bugs used in the first experiment and explained in section 4.1.2, a buffer overflow (9) and off by one error (10) are added. Thus a total of ten bugs, as listed in Table 2, were studied in the second experiment.

A buffer overflow (9) happens when a program attempt to store more data in a buffer than it is intended to hold. Buffers are created as a sequential section of memory and allocated to hold a finite amount of data. Writing extra information produces undefined behavior that can overflow into adjacent buffers, corrupt data or crash the program.

Off by one error (10) occurs when a program uses an improper maximum or minimum value that is one more or one less than the proper value. This problem arises when a programmer performs loop iteration a number of times that is greater or less than expected. Thus, the program reads or writes beyond the bounds of allocated memory, which can result in data corruption, or application crash.

We developed a program and injected these ten logical errors into a single C++ program composed of 80 lines of code. The level of difficulty of the problem is typically found in a CS I/II course. Specifically, the program solves the problem that projects the growth of a university's student population over 5 years.

4.2.2 Procedure

For the experiment the buggy program is deployed within an online C++ shell where the students are able to compile and execute the code. A sample of desirable output that the program fails to produce is also given along with sample input. General information about the subject is collected at the beginning of the experiment.

The experiment is designed to track the order that the participant fixed the bugs in addition to the time spent on each one of the bugs (completion of one until completion of another). Students are informed that the program is syntactically correct and multiple logical errors exist, however, the exact number of errors is not explicitly stated. To help uncover all the bugs, the students are encouraged to use the provided sample input that the program should properly execute.

The same statistical analysis is performed on this experiment as in experiment one.

5. RESULTS OF FIRST EXPERIMENT

Fifty-five (93.2%) subjects were able to solve half or more of the survey's problems. The correct answers by participants are as follow, 2 correct answers out of 8 by 3 (5.1%) subjects, 3 out of 8 by 1 (1.7%), 4 out of 8 by 8 (13.6%), 5 and 6 out of 8 by 12 students for each, 7 out of 8 by 14 (23.7%) and 8 out of 8 by 9 (15.3%) participants.

Based on the error type, subjects' answers are shown in Figure 1. None of the bug categories are totally solved or unsolved by all students. Bug 2 has more incorrect answers than correct answers. Whereas, Bug 4, 5, and 8 have the highest number of correct answers compared to incorrect answers.

5.1 Timings

The time spent by subjects to complete the problems is varied between students. The subjects spent an average of 20 minutes. The time is not affected by the number of correct answers as the students' completion time varied greatly regardless of the correctness level. By analyzing the time spent on each type of errors, the average time to finish any of the tasks is 2:30 m:s with small difference between the categories (i.e., range: 2:01-3:10 m:s) (see Figure 2).

5.2 Debugging Ability

To address our third research question we analyze the subjects' answers in details. For each problem students are required to locate the line in which the bug is found and to identify the type of that bug. We found three patterns of solutions:

- Correct: Students who solved both questions.
- Incorrect: Students who could not solve either question.
- Partially correct: Students who are able to solve only one of the two questions. That is, either by finding the line where the bug is located or identifying the type of the bug.

The student answers based on this analysis is shown in Table 3.

5.3 Correlational Analysis

The correlation between median time with age, gender, and current class standing did not show any statistical difference between the two groups in each variable. However, median time is significantly shorter in students with more years of experience than students with less years of experience (p=0.009). The correlation analysis results are shown in table 4.

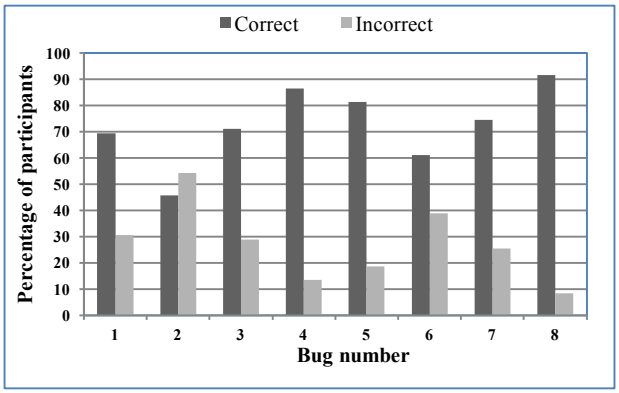

Figure 1. Bug types vs. correctness in experiment 1.

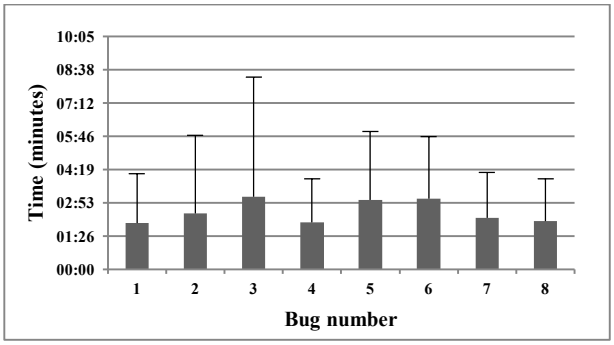

Figure 2. Bug types vs. time in experiment 1.

6. RESULTS OF SECOND EXPERIMENT

Sixty-six (79.5%) subjects are able to solve half or more of the survey's problems. The correct answers by participants are as follow, 10 correct answers out of 10 by 21 (25.30%) students, 9 out of 10 by 19 (22.89%), 8 out of 10 by 13 (15.66), 7 out of 10 by 7 (8.43%), 6 and 5 out of 10 by 3 (3.61%) students for each, 4 out of 10 by 6 (7.23%), 3 out of 10 by 3 (3.61%), 2 out of 10 by 2 (2.41%), 1 out 10 by 4 (4.82) and 0 out of 10 by 2 (2.41%).

Based on the error type, subjects' answers are given in figure 3. None of the bug categories are totally solved or unsolved by all students. Bug 4 and 8 have the highest incorrect answers among other errors. While, Bug 2 and 5 have the highest correct answers as at least 80% of the participants solved them correctly.

6.1 Timing

The time required by students to complete the debugging tasks is varied. The subjects spent an average of 65 minutes (range: 0:04:37–2:00:58 h:m:s). Similar to results of first experiment, time is not affected by the number of correct answers. However, participants who solved fewer number of bugs spent shorter time than students who solved more number of bugs (Figure 4).

The average time needed to fix each type of bug is 6:23 minutes (range of 04:12 - 10:07 m:s). Bug 1 has the shortest period of time (04:12 m:s). While bug 5 and 8 have the highest period of time needed for fixing the corresponding bugs (10:07 and 9:30 m:s, respectively) (see Figure 5).

Table 3. Students debugging ability. (+) correct, (-) incorrect, (+/-) partially correct.

Bug Type	First Experiment			Second Experiment		
	+	-	+/-	+	-	+/-
1	40	16	3	60	22	1
2	27	11	21	75	8	0
3	42	13	4	55	25	3
4	51	0	8	54	17	2
5	48	6	5	68	9	6
6	36	4	19	61	22	0
7	44	2	13	62	5	16
8	54	1	4	48	4	31
9	NA	NA	NA	60	22	1
10	NA	NA	NA	64	17	2

Table 4. Correlation analysis with time and correct answers

Variable		First Experiment		Second Experiment	
		Median	P value	Median	P value
Years of experience vs. Time	2 or less	17:32	**0.009**	1:08:30	0.199
	3 or more	12:11		54:10	
Years of experience vs. Correctness	2 or less	6	0.8	8	**0.033**
	3 or more	6		9.5	

6.2 Debugging Ability and Behaviors

We examined subject answers to understand their behaviors in debugging. First, we examined the order of bugs' appearance in the code against the order of fixing the bugs by the participants. Most of the students follow the same order of appearance. They used a top down approach by starting from the beginning towards the end of the program in locating and fixing the flaws.

Second, different kinds of characteristic behavior are certainly evident when observing novices in the process of debugging. Perkins et al. [13] distinguished between two main kinds of novices in writing program, *stoppers* and *movers*. When challenged with a problem or a lack of a clear direction to proceed, stoppers, as the name implies, simply stop. Movers are participants who keep trying, modifying their code, and use feedback about bugs. Movers are believed to have the potential to solve the current problem and progress effectively. However, extreme movers, *tinkerers*, who are not able to trace their program, make changes more or less at random, and have little effective chance of progressing. We define different type of novices:

- *Stopper*: participants who fixed 6 or less of the problems.
- *Mover*: participants who successfully fixed 7 problems or more.
- *Tinkerer*: disregard the number of fixed bugs; participants in this group made changes more or less at random. They are susceptible to generate more bugs while fixing the existed ones or making random changes that do not help solving any problem.

Participants according to this classification are 57 (68.67%) movers, 22 (26.51%) stoppers and 4 (4.82%) tinkerers.

To assess the debugging patterns, we applied same analysis as in the first experiment with slight difference due to the experiment design. Thus, three patterns of answers are identified:

- Correct: Answers that fixed the bug correctly.
- Incorrect: Answers that left the error untouched.
- Partially correct: Answers that indicate the students are aware about the presence of the bug but unable to solve it. Students provided different scenarios of frustration with debugging. They either modified the line containing the bug but their modification failed to solve the problem, isolate the error by commenting the line has the bug, or explicitly write a note as a comment indicating their awareness of the bug.

The analysis is given in Table 3.

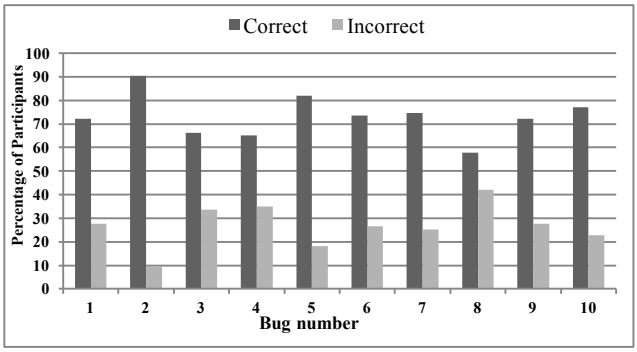

Figure 3. Bug types vs. correctness in experiment 2.

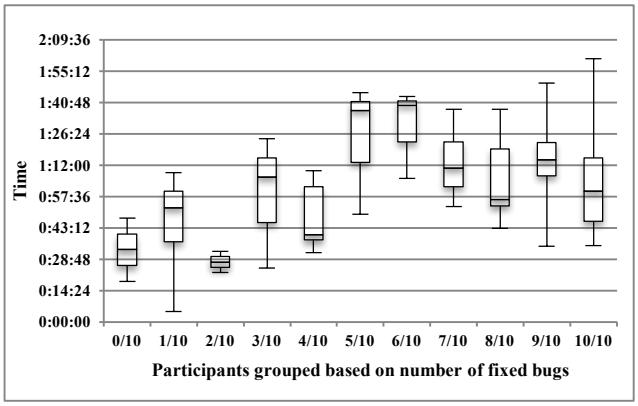

Figure 4. Time spent by participants grouped based on how many bugs they fixed.

6.3 Correlational Analysis

The correlation between median time with age, gender and current class did not show any statistical difference between the two groups in each variable. However, students with more years of experience have significantly higher number of correct answers compared to students with less years of experience (p=0.033). The correlation analysis results are given in Table 4.

7. Discussion

Concerning what errors are most difficult for student (R1), the results show that most participants are able to correctly answer half or more of the problems in both experiments, 93.2% and 79.5%, respectively.

Based on the number of correct answers, most of the bugs fell within range of 60% to 80% correct in both experiments. The results imply that students in introductory programming have the moderate difficulty with bugs 1 (using = vs ==), 3 (misusing switch), 5 (division by integer), 6 (condition variable not updated), 7(array index out of bounds), 9 (off by one) and 10 (buffer overflow).

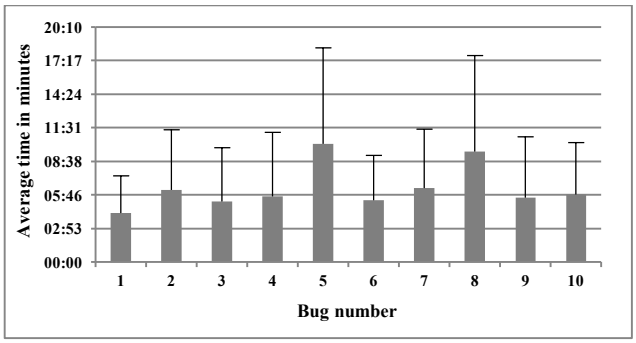

Figure 5. Bug types vs. time in experiment 2.

We found that the error 2 (loop has no body) is the most difficult for participants in the first experiment as number of incorrect answers are higher than number of correct answers. However, this error is the easiest one in the second experiment. The design of the second experiment as real setting where students are able to execute their program provides a good explanation for this discrepancy. As soon as the students compiled their code, the C++ shell shows a warning at the line of the bug stating that the loop has no body.

Error 8 (access deleted dynamic variable) and 4 (misusing && and || errors) are correctly solved by a majority of the students (91% and 86%) in the first experiment. This result is different from the finding in the second one where only 57% and 65% of the students could fix these two errors. We believe that the preciseness and small size of code in the first study helped the students to easily locate these bugs. While in the second experiment a combination of 10 errors existed in one larger program can obscure identifying these errors.

In our analysis of the timings (R2), the subjects spent an average of 20 minutes to complete the 8 tasks in the first experiment. None of the errors are found to require significantly more time than others. However, in the second experiment, when the setting is more realistic and a combination of 10 bugs existed in one program, students spent much more time with an average of 65 minutes. The time needed to debug 8 (access deleted dynamic variable) and 5 (division by integer) are much longer compared to others. The longer time required for debugging 8 concurs with the result of the first research question above. Both results indicate bug 8 has a high difficultness level for novices.

To find out how well the participants solve the problems (R3), the students' answers fall into three categories correct, incorrect, or partially correct. Based on both experiments, we conclude that the errors can be classified into two main groups.

Easy/Hard problem in which students' answers mainly fall in the the correct or incorrect category. Students are either able to fix the error correctly or they leave it untouched. The bugs of this category are, 1 (using = to check equality), 3 (misusing switch), 5 (division by integer), 9 (off by one), and 10 (buffer overflow). We believe that students who performed well have adequate previous knowledge about the specific bug. They either have faced the error in their assignments or have been taught about this error. On the other hand, the rest of this group lack the knowledge and fail in uncovering the bug. We feel that incorporating these bugs into the lesson plan (e.g., lecture, lab, or assignment) will improve student performance on these bugs.

Easy/Tricky problem in which students' answers are found in the correct or partially correct category. Students either fix the bug or just identify the existence of the bug without solving it. Students demonstrated different scenarios of solving. The bugs of this type are: 7 (array index out of bounds) and 8 (access deleted dynamic variable). In addition to adequate knowledge about the bugs, we suggest instructors add additional practical tasks in the learning process to overcome obstacles that may arise due to this type of errors. The errors 2 (loop has no body), 4 (misusing && and ||), and 6 (condition variable not updated) showed inconsistent patterns as they fall into the Easy/Tricky in the first experiment and the Easy/Hard pattern in the second. The ability for participants to be able to execute the program helped in clarifying these bugs. For bug 2, the C++ shell gives a warning at the line of the bug stating that the loop has no body. Similarly, output feedback helped the students to identify bugs 4 and 6. Accordingly, we believe these two errors are Easy/Hard.

We found years of experience in programing play a significant role in debugging activity. The first experiment shows participants with more experience required shorter debugging time. While in the second one, expert participants had more number of correct answers than participants with less experience. While this is not a surprising result, it clearly demonstrates that experience is a key factor in debugging skills.

8. THREATS TO VALIDITY

As with any empirical study of subject's programming ability, we suffer from external threats that prevent us from generalizing our results to all students at every university. We do use data about type of programming bugs that is already published and not our own list that extracted directly from students. We minimized this threat by considering bugs appearing frequently as described in literature and examined by others and found to be commonly encountered novice programmers.

Since both experiments are within-subjects and with regard to internal validity, we assure that there is no learning effect. We addressed this by asking similar questions for the first experiment and provide the same piece of code in the second one for all participants to have a fair and unbiased comparison. The questions are presented to each subject in a randomized order. The subjects are sufficiently motivated to do the experiments since this was part of a subject's grade in a course. A possible threat to validity on the debugging practices of the students may be the impact that the a list of bug categories influence the participant's searching. We tried to minimized this threat by combining the bug localization ability and bug category identification as indicator of debugging ability of participants.

9. CONCLUSIONS AND FUTURE WORK

The work studies common errors of novice programmers. The results show some errors are more difficult than others. Different types of bugs and novices' debugging behaviors are identified.

We feel that systematically exposing students to each different types of bugs, in a learning environment, has the potential to drastically improve debugging skills. Additionally, while presenting individual bugs in isolation makes it easy to find and fix the problem, a more realistic scenario (i.e., multiple bugs in the same program) must also be part of the full rubric.

The next step of this research will be to construct assessment mechanisms (entrance and exit surveys) to determine what teaching methods are most successful in teaching debugging skills.

10. REFERENCES

[1] M. Ahmadzadeh, D. Elliman and C. Higgins, "An analysis of patterns of debugging among novice," SIGCSE, pp. 84-88, 2005.

[2] M. Ahmadzadeh, D. Elliman and C. Higgins, "The impact of improving debugging skill on programming ability," ITALICS, p.72-87, 2007.

[3] R. Bryce, A. Cooley, A. Hansen, and N. Hayrapetyan, "A one year empirical study of student programming bugs," The Frontiers in Education Conference (FIE), pp. 122-127. IEEE, 2010.

[4] H. Danielsiek, W. Paul and J. Vahrenhold, "Detecting and understanding students' misconceptions related to algorithms and data structures," ACM SIGCSE, 2012, North Carolina, USA.

[5] M. Ducassé, A.-M. Emde, "A review of automated debugging systems: knowledge, strategies and techniques," The 10th international conference on Software engineering, p.162-171, April 11-15, 1988, Singapore.

[6] A. Ford and T. Teorey, "Practical debugging in C++," Upper Saddle River, NJ: Prentice-Hall, 2002.

[7] M. Hristova, A. Misra, M. Rutter, and R. Mercuri. "Identifying and correcting Java programming errors for introductory computer science students," ACM SIGCSE, pp. 153-156, 2003.

[8] J. Jackson, M. Cobb and C. Carver, "Identifying top java errors for novice programmers," The 35th Annual Conference Frontiers in Education (FIE), pp. T4C-24-T4C-27, 2005.

[9] W. Johnson, E. Soloway, B. Cutler and S. Draper, "Bug catalogue: I," Technical report, Yale University, 1983.

[10] C. M. Lewis and C. Gregg, "How do you teach debugging?: resources and strategies for better student debugging," ACM SIGCSE, pp. 706-706, 2016.

[11] R. McCartney, A. Eckerdal , J. E. Mostrom , K. Sanders , C. Zander, "Successful students' strategies for getting unstuck," The 12th annual SIGCSE conference on Innovation and technology in computer science education, 2007, Dundee, Scotland

[12] L. Murphy, G. Lewandowski , R. McCauley , B. Simon , L. Thomas , C. Zander, "Debugging: the good, the bad, and the quirky - a qualitative analysis of novices' strategies, ACM SIGCSE Bulletin, v.40 n.1, March 2008.

[13] D.W. Perkins, C. Hancock, R. Hobbs, F. Martin and R. Simmons, "Conditions of learning in novice programmers". Educational Technology Center, Office of Educational Research and Improvement, 1985.

[14] D. N. Perkins, F. Martin, "Fragile knowledge and neglected strategies in novice programmers", workshop on empirical studies of programmers on Empirical studies of programmers, p.213-229, June 1986, Washington, D.C., USA.

[15] A.H. Rosbach and A.H. Bagge, "Classifying and measuring student problems and misconceptions," Proc. of Norsk informa-tikkonferanse (NIK), pp. 110–121, Akademika Forlag, 2014.

[16] J. C. Spohrer, E. Soloway and E. Pope, "A goal-plan analysis of buggy Pascal programs", Human-Computer Interaction, vol. 1, no. 2, pp. 163-207, 1985.