

The Impact of Improving Debugging Skill on Programming Ability

Marzieh Ahmadzadeh
ahmadzadeh@sutech.ac.ir¹

Dave Elliman
dge@cs.nott.ac.uk

Colin Higgins
cah@cs.nott.ac.uk

Abstract: This paper reports on a continuing study into teaching programming to adult novice students. As part of the study we aim to find students' pattern of behavior when they are programming in Java. In a broader perspective, we are interested in improving the students' ability to write programming. By patterns of behavior, we mean finding the frequently made compiler errors and the pattern of debugging. We claim that incorporating students' most common errors in a form of debugging exercise will improve their ability to program.

Keywords: teaching programming; Java; debugging exercise

¹ The author is currently teaching in Shiraz University of Technology.

1 Introduction

Introductory programming courses are known to be daunting for novice programming students in which students are expected to learn a new way of thinking as they learn a new language with an unfamiliar grammar. From the students' point of view, programming can be such an unintelligible module that students imagine the instructor is speaking a foreign language when he or she is teaching programming. Understanding the terminology is difficult, and presenting an analogy is not easy as there are often no similar scenarios in students' experience.

On the other hand the industry is still growing and the demand for competent programmers remains healthy. Programming is vocational and attracts students looking for a remunerative career. Often university courses teach to large number of students, which eliminates the traditional interaction between students and instructors. (Canup & Shackelford, 1998; Preston & Shackelford, 1998). This requires students to be more resolute, to be purposeful in learning and to rely on their own effort. One way to encourage students to do this is to give them the opportunity to learn from their mistakes. Nonetheless, this kind of learning should be supervised in such a way that each *error* plays a significant role in the process of learning.

These issues together require careful design of the teaching approach, which is efficient and effective. However there will not be one absolutely correct method but we hope to establish good practice. The main purpose of this research is to capture the pattern of students' programming behaviour. These patterns can be adapted to teaching to form a careful design of the *errors* in order to provide an opportunity for students to learn from the most commonly made errors.

For this purpose, three phases were established in this research. The first phase of the study investigates the pattern of compiler errors that is commonly generated by students. The second phase of this study examines the pattern of debugging amongst novice students. It aims to find out what understanding makes some students more able than others in terms of debugging. The third phase of the study applies the found patterns to the teaching of

computer programming and measures improvement. The improvement is measured by students' marks in different type of tests and also by the type and number of compiler errors generated by them.

2 First Experiment

One way to investigate students' problems in programming is by investigating 'what goes wrong'. Research into students' programming errors has spanned nearly two decades (BOULAY, 1989; Pea, 1986; J. C. Spohrer & Soloway, 1986; James C. Spohrer et al., 1985). However, little research (Coull et al., 2003; Hristova et al., 2003) has been done looking at the pattern that is created by students' compiler errors. Nevertheless the importance of the compiler errors was shown by Ebrahimi (1994), in which he shows that students who frequently made errors in language construct were more likely to make errors in plan compositions.

Amongst the research concerning compiler errors, Hristova et.al (2003) is the only work carried out on a large scale. However, even in Hristova's research which attempted to create a comprehensive list of errors, it is not clear how frequent the errors were; which errors happened most often; if the errors related to specific concepts; if there were any changes in the frequency of the errors; or when the course was proceeding. Also interestingly, in all that research, it is implied that the most common errors are those for which students most of time seek help, but it is not clear that seeking help is a valid index of frequency. In the present research, we collected the students' errors as they attempted the programming exercises (Ahmadzadeh et al., 2005).

In order to carry out the monitoring process for the current research, students' compiler messages were collected as students attempted the programming exercises that were set during the term. All error, warning and cautionary messages were collected, together with a time-stamp and the source code to which they applied, and these were stored in a database. This process continued for the whole semester and every message (error, warning and caution) from every student's compilation was recorded. This was achieved using *Jikes* (Jikes, 2003), an open source Java compiler. We were able to modify the source code of this

compiler so as to store the required information in a manner that was unobtrusive to the students, although they were informed that this data was being collected as a matter of ethical issues. In addition, no manual processing, which required student's identification, was carried out through the course of this research. The process of collecting data from the database and analysing them was done by a program written in Java. As a result, statistical calculations and students' programs, if it was needed, were provided in a file without any students' identification included. It should be mentioned that the overhead of saving source files and error messages at each compilation was less than the improvement in compilation time obtained by using *jikes* rather than *javac*.

In total fifteen exercises were given to the students covering the basic programming material in Java. Two programming exams were also set online, using a locally developed system known as *CourseMarker* (Higgins et al., 2002). This enabled us to ensure that the marking system was consistent for each student and reliable and accurate as a basis for the research.

One hundred and ninety two students took the module. However, the number of students may vary from one exercise to another, as some of the students did not complete every exercise. The data related to each exercise was retrieved from the database as soon as the assignment was due.

The patterns of behaviour found amongst novice students, when generating compiler errors learning different concepts (Ahmadzadeh et al., 2005), was not consistent with the existing research (Hristova et al., 2003). It was revealed that seeking help is not a valid index of frequency of making compiler errors. This difference can be interpreted in two possible ways. Firstly, these errors originate from a simple mistake rather than a misconception therefore students are able to solve the problem themselves rather than seeking help. The second assumption can be that students do not know what they think they know, which leads them to make similar errors frequently. For this case more research needs to be carried out.

An increase in the negative correlation between time spent in debugging a program and students' marks as a result of increased expertise was found that suggests that debugging

skills can be a useful predictor for student performance. However, in this experiment students were required to correct their own erroneous program. To investigate the correctness of this theory when students correct an unfamiliar erroneous program the second experiment was designed.

3 Second Experiment

In this experiment, the relation between synthesis skills and analytical skills in programming was investigated (i.e. programming performance vs. debugging skills). It was found that the more the students are able in debugging skills the more they perform better in programming, but the reverse is not true. This happens since some of the students comprehend the program implementation better than others (Ahmadzadeh et al., 2005). This led us to provide an environment for our students to improve their debugging skills along with the programming. This formed our third experiment.

4 Third Experiment

We proposed an approach to teaching in which students would be exposed to more debugging situations. With this approach students have similar opportunities for debugging and programming. Thus it is hoped that by enhancing the level of program comprehension, programming ability will be improved. This approach was implemented for a new intake of students and their performance was compared with the previous years students who had been taught by our traditional approach.

4.1 Research methodology

The method of data collection remained the same as for the previous experiments in which automated data collection was carried out (Ahmadzadeh et al., 2005). In average 110 students participated in different experiments of this phase of research. To make a fair comparison, the students' background in both cohorts was studied. Our school uses the same criteria for admitting the students in each of the years. Therefore it is believed that students, who register for this course, have a similar background. It should be noted that the criteria for selecting the students is based on their marks in 'A'-level examinations. This method may well

distinguish diligent students but probably does not distinguish the students according to their talent for programming very well.

Nevertheless students' background was investigated and the three most common subjects that students had chosen were studied. In both years, *mathematics*, *computing* and *physics* were the most common subjects. Mathematics had been taken by most of last year's students in comparison with those in the current year. However, the current year's students had achieved exactly the same average mark for mathematics as for the year before. If it is believed that mathematics aptitude has positive correlation with programming aptitude (Wilson & Shrock, 2001), it might therefore be said that last year's students were slightly better than the current year's, as more students entered university having a mathematics background. A comparison of the computing background reveals that last year 75% of the students had a computing background while in current year just 55% of them had this advantage. In terms of *physics*, again last years students seem to be better than the current year as not only have more people passed the course, but they also achieved a higher average mark. Overall it seems that the current years intake comprises weaker students. This provided a challenging environment to test the effectiveness of the modified approach.

To measure the performance, several issues were considered. First of all, the type and number of errors that students produced while doing the weekly exercises and online programming exams in comparison with last year. The second is their performance in terms of the marks they achieved for their assignments, on-line programming test, multiple choice question tests, in comparison with last year's students. In order to do this, the changes that were made to the teaching needed to be minimal. Students had two exercises for every concept (i.e. conditionals, loops, etc.) to be submitted weekly. Therefore the only change that was considered to the teaching approach, was to design an erroneous program to be corrected by students to replace the first exercise that last years students needed to program in a week. The second programming exercise remained the same as last year in terms of the structure, however the specification needed to differ from the previous year. This programming exercise was used to compare two groups of the students.

To design an erroneous program for each concept, the type of logical and semantic errors that were applicable to both the concept and program specification were chosen. Semantic errors were selected from the finding of our first experiment and logical errors were chosen from the existing research (BOULAY, 1989; Pea, 1986; J. C. Spohrer & Soloway, 1986; James C. Spohrer et al., 1985). Examples of given errors can be seen in the appendix. The on-line programming exam and multiple choice question exams were also designed in the same way as the last year. However where it was possible, last year's exam questions were used again as these had been kept secret.

4.2 Results

The performance of students was studied in two ways; first the number of the errors that was produced by them and second the mark that they gained during the course. This data was compared with last year's students' data to investigate if any improvement had occurred.

4.2.1 The Errors' Numbers

The results showed that regardless of the method of teaching, the type of the errors that are made most often were the same in both years. However, in this experiment, the new approach is seen to help decrease the number of errors made per person. In almost every concept this reduction is significant (Table 1). This indicates that this new method of teaching helped students to better understand the syntax than last year's students.

	2003-2004		2004-2005	
	% of Students	No. of Error per Students	% of Students	No. of Error per Students
Conditionals	97	39	92	14
Loops	93	30	91	26
Methods	97	89	97	79
Arrays	96	31	96	31
Classes & Objects	97	75	96	67

Table 1: Comparing Two Different Teaching Approach

4.2.2 Marks

To analyse the result statistically, we have used two-tail t-test, which addresses whether the means (of the marks of two years) are statically different. By calculating t-test, we are looking at the differences between scores for two groups by considering the difference between their

means relative to the spread or variability of their scores. Also we need to compare two unpaired (independent) group that means the observations in one sample are not in any way related to the observations in the other. To decide whether homoscedastic t-test or heteroscedastic should be used, we used Levene's test to examine the equality of the variances. If the variances are the same homoscedastic t-test will be used (p-value > 0.1) otherwise heteroscedastic two-tail t-test is used (p-value <0.1). In the following table p-value is shown by Sig.

To analyse this table, the following hypothesis were set (we set alpha level to 0.05):

H0: Means of marks in two years are equal.

H1: Means of marks in two years are not equal.

Therefore, if the t-test is greater than the corresponding number in the significance table, H0 would be rejected. Otherwise the averages of marks are considered not to have significant differences.

According to the significance table, the difference between means in concepts such as Conditionals, Loops, Methods, Arrays and Classes are not significant. Therefore we cannot reject H0. This significant difference shows an improvement using the innovative method when Strings are practiced. In the on-line programming exam, MCQ2 (multiple choice question number 2) and MCQ3 showed no significant improvement however MCQ1 showed significant difference (p-value= 0.009 <0.05) in favour of the traditional approach. It should be noted that MCQ1 was not used as a mean of testing our new approach, as students had not used our method prior to this exam. This exam was used as an index to see which group showed more talent in programming (Table 2).

		Levene's Test for Equality of Variances		t-test for Equality of Means						
		F	Sig.	t	df	Sig. (2-tailed)	Mean Difference	Std. Error Difference	95% Confidence Interval of the Difference	
									Lower	Upper
conditional	Not Equal variances	7.366	.007	.770	309.247	.442	.81963	1.06452	-1.27499	2.91426
loop	Equal Variances	1.032	.310	-.071	307	.944	-.11214	1.58422	-3.22944	3.00517
method	Equal variances	1.504	.221	-.372	317	.710	-.72754	1.95756	-4.57899	3.12391
array	Equal variances	2.153	.143	-.049	269	.961	-.10507	2.12727	-4.29330	4.08315
class	Equal variances	.045	.833	.268	250	.789	.56506	2.10762	-3.58589	4.71601
string	Equal variances	34.280	.000	3.215	107.456	.002	4.47368	1.39163	1.71507	7.23228
MCQ1	Equal variances	.101	.750	2.634	322	.009	4.21583	1.60070	1.06668	7.36499
MCQ2	Not Equal Variances	3.020	.083	.412	265.023	.681	.69278	1.68237	-2.61972	4.00529
MCQ3	Not Equal variances	4.448	.036	-.227	219.179	.821	-.46186	2.03726	-4.47698	3.55325
Exam_Mark	Equal variances	4.879	.028	1.873	285	.062	5.61102	2.99570	-.28549	11.50752

Table 2: Independent Samples Test Comparing Students' Marks

4.2.3 Feedback From Students

A questionnaire was designed in order to find out students' opinions about the new method of teaching where the debugging exercises were added. Eighty-three students participated in the experiment, which account for 69% of all students. 61% of the students had not programmed before, while 38% had done some programming.

It can be seen from Table 3 that the majority of the students believed that the debugging exercises had helped their learning in term of programming concepts and the Java language. Also, the majority of the students agree that carrying out the debugging exercise has helped them in recognising the compiler and logical errors, when they come to write their own programs.

	Strongly Agree	Agree	Neutral	Disagree	Strongly Disagree
Usefulness of Debugging Exercises to Learning of Programming	42	40	16	2	0
Usefulness of Debugging Exercises to Learning of JAVA	43	45	11	1	0
Usefulness of Debugging Exercises to Recognising the Compiler Errors Later	29	55	12	1	2
Usefulness of Debugging Exercises to Recognising the Logical Errors Later	20	54	18	2	2
Debugging Exercises were enjoyable	8	39	28	22	4
Debugging Exercises were boring	6	2	31	45	16
Program implementation was difficult to understand	1	12	22	46	19
Debugging exercises were frustrating.	1	14	16	49	19

Table 3: Analysis of the questionnaire: Percentage agreement

Students were asked to rate each concept from one to five with regard to how the debugging exercises helped their learning. A grading of one meant the debugging exercises were of no help, whereas five meant they were a great help. The average mark for each concept that a debugging exercise provided was three, except for the *methods* and *loops*, where the grade was 4. This means that students believed that the debugging exercises did help them learn these concepts.

Interestingly, the investigation of the students' grading regarding the usefulness of programming exercises shows exactly the same result for the different concepts. Therefore it

means that debugging exercises have been as useful as programming exercises. Having the same result means that students are comfortable carrying out such an exercises.

In an open-ended question students were asked to think about which part of the teaching strategy they thought was most helpful to their learning. 47% of the students said that programming exercises were the most useful, 35% believed that both the exercises (programming & debugging) helped them a lot, while 18% believed that the debugging exercises had been the most useful.

The majority of the students, who chose the programming exercises as being the most efficient way of learning, commented on the same issue. They all preferred to do programming exercises as this was a practical approach [*hands on experience, learning by doing, enforcing to understand the lectures, etc.*] in learning programming.

On the other hand, students who chose debugging as the most efficient way indicated that they could see a sample of programming implementation, which helped their learning. In all the cases, the emphasis is on seeing real working code to help them write the program later. For example one of the students says:

Debugging exercises [were the most useful] because they allowed me to see the code for each topic... and in making the program work I better understood the topic.

Another one indicates:

I found the debugging [exercises] very good. They took longer [to finish compared with the writing ones], but show how to do things, which previously I couldn't do. I learnt from them and then used the knowledge [later].

Although the greater number of students chose the programming exercise as most useful, this does not mean that they dislike the debugging exercises, although one did. He/she says:

Debugging exercises require guessing what the programmer was attempting to do. I tended to leave them [until] last. It just confused me.

This comment however demonstrates the novices' strategy at debugging, which was based on guessing and not producing a hypothesis. This is evidence for the result that was achieved in previous experiment, where the major problem in debugging is in understanding the program implementation.

Overall it can be concluded that students are happy about the debugging exercise as a way of learning, and that it complements programming exercises.

5 Discussion & Further Work

In our third experiment it was explained that the new comers had weaker backgrounds than the previous year's students. This background was measured by their mathematics and MCQ1 exam mark. Nevertheless they showed an improved performance in producing errors and the same performance in their marks, which we attribute to our revised approach.

Also, *learning from mistakes* is a well-known method of learning and errors are believed to have a positive impact on learning and can therefore play an important role in training process. Normally *learning from mistakes* is construed as learning from the errors generated by students when writing a program, which is the synthesis part of the programming. In this experiment, we applied *learning from mistakes* in the analytical part of the programming (i.e. debugging) by carefully incorporating some errors into the exercises which we taught, and are useful for learning to program.

Due to the nature of this work there were several constraints in designing the experiments. One big limitation was the design of the exercises. It was necessary to make the exercises as similar as possible to those used the last year in order for a fair comparison to be made. Therefore, it was not possible to add certain errors covering some concepts because of the nature of the specification. If some specific errors were to be added to the program the specification would also have to change, which was not possible. For example, when the *class* concept was taught, it was necessary to add some errors that were directly related to these concepts. Last year however, the initial structure of the class was given to the students and so we needed to do the same this year, otherwise the consistency of the research would

have been compromised. As a result of this situation a less effective error related to this subject was all that could be added to the program. That is why when students' performance is investigated it seems that for some concepts students have improved better than others. We believe that the errors that we used for some concepts were more appropriate than for others and that is why the performance on those concepts was more convincing. Therefore, if the right errors are added to the exercises learning will improve significantly.

Pair programming has been acknowledged by researchers (Williams & Upchurch, 2001) as a way of increasing retention and minimising errors. The impact of pair programming along with our suggested method for teaching could be an issue for future research. Also it will be useful if the error messages of the Jikes compiler (Jikes, 2003) are changed to something that is more understandable by students, and for the experiments to be repeated to see whether or not our results are repeated.

References

- Ahmadzadeh, M., Elliman, D., & Higgins, C. (2005, June 27-29). *An Analysis of Pattern of Debugging Among Novice Computer Science Students*. Paper presented at the Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education (ITiCSE), Monte de Caparica, Portugal.
- BOULAY, B. D. (1989). Some Difficulties of Learning To Program. In *Studying the Novice Programmer* (pp. 283-299).
- Canup, M. J., & Shackelford, R. L. (1998). *Using software to solve problems in large computing courses*. Paper presented at the Proceedings of the twenty-ninth SIGCSE technical symposium on Computer science education, Atlanta, Georgia, United States.
- Coull, N., Duncan, I., Archibald, J., & Lund, G. (2003, 26 - 28 August). *Helping Novice Programmers Interpret Compiler Error Messages*. Paper presented at the 4th Annual LTSN-ICS Conference, National University of Ireland, Galway.
- Ebrahimi, A. (1994). *Novice Programmer Error: Language Constructs and plan composition*. *International Journal of Human-computer Studies*, 41, 457-480.
- Higgins, C., Symeonidis, P., & Tsintsifas, A. (2002). *The Marking System for Coursemaster*. Paper presented at the Proceedings of the 7th Annual Conference on Innovation and Technology in Computer Science Education, Aarhus, Denmark, 46-50.
- Hristova, M., Misra, A., Rutter, M., & Mercuri, R. (2003). *Identifying and correcting Java programming errors for introductory computer science students*. Paper presented at the Proceedings of the 34th SIGCSE technical symposium on Computer science education, Reno, Nevada, USA.
- Jikes. (2003). *IBM- developerworks - open source software*. Retrieved Last Accessed on October, from <http://oss.software.ibm.com/developerworks/opensource/jikes/index.shtml>
- Pea, R. D. (1986). Language-independent conceptual bugs in novice programming. *Journal of Educational Computing Research*, 2(1), 25-36.
- Preston, J. A., & Shackelford, R. (1998). *A system for improving distance and large-scale classes*. Paper presented at the Proceedings of the 6th annual conference on the teaching of computing and the 3rd annual conference on Integrating technology into computer science education: Changing the delivery of computer science education, Dublin City Univ., Ireland.

- Spohrer, J. C., & Soloway, E. (1986). Novice Mistakes: Are The Folk Wisdom Correct? *Communications of the ACM*, 29(7), 624-632.
- Spohrer, J. C., Soloway, E., & Pope, E. (1985). Where the bugs are. *Proceedings of the SIGCHI conference on Human factors in computing systems*, 47-53.
- Williams, L., & Upchurch, R. L. (2001). *In support of student pair-programming*. Paper presented at the Proceedings of the thirty-second SIGCSE technical symposium on Computer Science Education, Charlotte, North Carolina, United States.
- Wilson, B. C., & Shrock, S. (2001). *Contributing to Success in an Introductory Computer Science Course: A study of Twelve Factors*. Paper presented at the SIGCSE.

Appendix : Examples of the Given Errors.

A multiple condition is required here.

```
do{  
  
    System.out.print("\nEnter your option;(":  
    option = readString;()  
    option = option.trim;()  
  
}while (!(option.equals("1")));
```

A correct *swapping* must be replaced.

```
if ( mark1 > mark2} (  
    marks[j][0]= marks[j+1][0];  
    marks[j][1]= marks[j+1][1];  
  
    marks[j+1][0]= marks[j][0];  
    marks[j+1][1]= marks[j][1];  
}
```