# An Analysis of Patterns of Debugging Among Novice Computer Science Students

Marzieh Ahmadzadeh
School of CS & IT
University of Nottingham
NG8 1BB UK
Tel: +44 115 951 4232
mqa@cs.nott.ac.uk

Dave Elliman
School of CS & IT
University of Nottingham
NG8 1BB UK
Tel: +44 115 951 4208
dge@cs.nott.ac.uk

Colin Higgins
School of CS & IT
University of Nottingham
NG8 1BB UK
Tel: +44 115 951 4213
cah@cs.nott.ac.uk

## ABSTRACT
The process by which students learn to program is a major issue in computer science educational research. Programming is a fundamental part of the computer science curriculum, but one which is often problematic. It seems to be difficult to find an effective method of teaching that is suitable for all students. In this research we tried to gain insights into ways of improving our teaching by a careful examination of students' mistakes. The compiler errors that were generated by their programs together with the pattern that was observed in their debugging activities formed the basis of this research. We discovered that many students with a good understanding of programming do not acquire the skills to debug programs effectively, and this is a major impediment to their producing working code of any complexity. Skill at debugging seems to increase a programmer's confidence and we suggest that more emphasis be placed on debugging skills in the teaching of programming.

## Categories and Subject Descriptors
K.3.2 [**Computers and Education**]: Computer and Information Science Education – *computer science education*

## General Terms
Performance, Experimentation, Languages

## Keywords
Debugging, Programming

## 1. INTRODUCTION
Research into the teaching of introductory programming has spanned nearly two decades, and work has often focussed on students' logical misconceptions in programming and debugging. There are some dated, but still valuable, studies on students' logical misconceptions [4, 16, 19, 20, 21]. However, no comprehensive research has been done looking at students' compiler errors directly [2, 11, 23].
Much research has been carried out comparing the differences between novices and experts in terms of debugging ability [6, 7, 8, 9, 25]. Examples include "the end-user or direct-manipulation programming system" [1, 17], "the impact of the debugging environment features on program debugging" [18], "automated fault localization" [24] and "the program debugging tool" [12]. Little, if any, attention has been paid to discovering the pattern of debugging in novices or to draw conclusions, which might aid computer programming education. The skills and abilities that distinguish experts from novices in program debugging and the affect of those competencies on the ability to progress in programming seem to have been largely ignored in recent years. The goal of this research has been to refine our teaching methods by learning from students' mistakes, considering both compiler and logical errors. To make progress we needed to understand novices' behaviour when programming. Therefore this study, in its first phase, investigated the pattern of compiler errors among novice students while in the second phase the pattern of debugging logical errors was studied.
This paper is organized as follows. First the research methodology is explained. We then discuss the results from our first experiment when we focused on students' compiler errors. Then we describe the second phase of our study, when our aim was to look at students' logical debugging patterns. Finally we discuss the implications of these results for the next stage of our research, which is to produce an improved, innovative, approach to teaching.

## 2. THE RESEARCH METHOD
The Java Language is taught as a first programming language in our department. The material that is covered in the first semester of this module comprises the fundamental language constructs, including expressions, conditionals, loops, methods, arrays, classes and strings.
In order to carry out the first phase of the study, students' compiler messages were collected as they 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. The source code and time-stamp were sent to another table, if the program was free of compiler errors, for the second phase of the study.
Collecting the data on a large scale was achieved using ***Jikes*** [14], an open source Java compiler for which the source is freely available. 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. The overhead of saving source files and error messages at each compilation was less that the improvement in compilation time obtained by using *jikes* rather than *javac*. An editor called JCreator [13] was introduced to the students. Therefore they all had the same environment in which to work.

In total, fifteen exercises were given to the students covering the material mentioned above. Two programming exams were also set online, using a locally developed system known as *CourseMarker* [10]. This enabled us to ensure that the marking system was consistent for each student and a reliable basis for our research.

One hundred and ninety two students took the module, however, the number of the students may vary from one exercise to another, as some of the students failed to complete all the exercises on time. The data related to each exercise was retrieved from the database as soon as the assignment was due.

## 3. THE FIRST PHASE OF THE STUDY

The compiler that was used for this study (Jikes), produce three kinds of errors. The first class is termed a "syntax errors" which concerns the grammar, or order of the tokens in the program. Missing some symbol of other, often a semicolon produces this type of the error. The second class of errors is termed "semantic errors" that are generated when the meaning of the code is not consistent with the language, for instance using a non-static global variable inside the static method. "Lexical errors" constitute the third class of error, and this occurs when a token is unrecognised. An example for this situation is where the separator of the two literal strings in *print* statement is missing which generates an unknown token.

The results are based on an analysis of 108,652 records of student errors that were collected during one semester. Thirty-six percent of these errors were syntax errors, 63% were semantic errors, while just 1% of them were lexical errors.

### 3.1 Result 1: A Pattern in Compiler Errors

To cover the material that was taught in the programming course, one or two supportive exercises were given to students each week. For every concept the number and type of semantic errors were retrieved from our database. Also the percentage of the students who were responsible for making such errors was calculated to make sure that the archived data was significant.

The result shows that out of 226 distinct semantic messages produced by Jikes, 6 of the errors constitute more than 50% of the errors in each concept. Table 1 shows the three most common occurrences of this type of the error in each concept separately.

It can be seen from this table that the error that is most common amongst all the subjects is failing to define a variable before it is used. This was almost always the highest frequency error when teaching a range of different concepts.

To evaluate the correctness of this data, the students' errors in two on-line programming examinations was also analysed. The first exam was set after students had practiced "methods" and the second exam was set after all the material in the programming course has been taught. The most common semantic error again was failing to define a variable and this occurred in 46% and 36% of the of the students' efforts in the two exams respectively.

### 3.2 Result 2: Performance VS. Debugging Time

The number of errors per compilation, the time taken to debug a program, and the mark that was finally achieved in the examination were compared for each student to investigate whether or not there was any relation between these parameters.

The correlation coefficient describes the strength of an association between variables, and can vary over the range –1.0 to +1.0. A correlation of 1.0 or -1.0 means perfect correlation or inverse correlation where one variable can predict the other absolutely.

The results show there is no correlation between the number of errors and the time that was spent in debugging the program before it was submitted. This could be because the compiler can become confused about the program syntax in the event of an error and may produce several messages. The correlation was zero for both of the exams.

However the correlation between the time spent debugging in the exam and the mark achieved is also rather weak, but negative (-0.4) in the first exam. In second exam, where students have more experience, this correlation becomes stronger. As the correlation coefficient is now –0.5, it is clear that higher marks correlate with less time spent in debugging the program. In the second exam 25% of the marks variation could be explained by the time spent in debugging. Those who are skilled at this task are faster and more effective at finding the errors and able to achieve higher marks.

### 3.3 Discussion

It was found that there is a pattern in the way students produce compiler errors. This pattern could explain how students misunderstand a concept. The teaching could take advantage of this finding to change in such a way that increases the depth of understanding. It would be helpful if any pattern in the way that students make logical errors can be found. If such a pattern exists, the teaching approach could be changed to benefit the students by using the differences between high and low performing students.

Investigating the compiler errors was a straightforward approach as the numbers of compiler errors is limited. However this approach does not help us find the pattern of logical errors. There are an unlimited number of possible logical errors that students could make. Therefore, in order to discover the pattern of logical errors made by students it is more practical to investigate their debugging technique. By finding the differences between the high performing students' pattern of debugging and that of low performing students an insight can be obtained into their widely difference performance. If significant systematic differences can be found these could change the approach of the teaching to benefit the students' learning. The second phase of this research investigated the pattern of debugging amongst different groups of students. In addition the type of compiler errors that we found could be a topic for further research to investigate the cognitive source of the various misconceptions and mistakes.

## 4. THE SECOND PHASE OF THE STUDY

To gain an understanding of the common patterns of debugging used by novice programmers, if any, an examination environment was arranged such that all the participants were given the same time to complete their task. A program was designed in such a way that it intentionally contained some errors, including both compiler errors and logical mistakes. Students were asked to correct the program in 120 minutes.

The chosen compiler errors were similar to those most commonly made by students during the previous semester (phase 1 of the study).

The selection of logical errors was informed by the work of other researchers [3, 4, 8, 11, 16, 20, 21, 22], who previously discovered the types of logical errors most often made by novice programmers. It is obvious that not all possible logical errors could be injected into one single program, as the time for the

**Table 1: The Percentage of the Most Common Semantic Errors in Different Exercises**

| Compiler Error | Conditional | Loop | Method | Array | Class | File | String |
|---|---|---|---|---|---|---|---|
| Field Not Found | 35 | 31 | 32 | 49 | 30 | 39 | 32 |
| Use of Non-Static Variable Inside the Static Method | 12 | | 33 | 6 | 21 | 5 | 13 |
| Type Mismatch | 9 | 12 | | 8 | | | |
| Using a non-Initialised Variable | | 15 | 6 | | | | |
| Method Call with Wrong Arguments | | | | | 6 | | 10 |
| Method Name Not Found | | | | | | 8 | |

exam was limited. We chose the *bugs* that occurred most often and which were most suitable for exam conditions. Three logical errors were added to the program for the experiment. The first error removed one necessary condition from a multiple choice statement [15].

The second logical error was related to what Du Boulay [4] calls a *mis-applied analogy*. We intentionally omitted the use of a temporary variable to interchange the value of two variables in the bubble sort algorithm. Students who see variables as a *pigeonhole* are often not clear that a variable can only hold a single value. The third error was chosen from the taxonomy of the logical errors listed by Pea [16] and is termed a *parallelism bug*. In this error we changed the place of two statements of the program. The first statement called a bubble sort and the second printed the sorted list. Obviously this results in the unsorted list being displayed. However, some students do not fully appreciate the sequential nature of steps in a program but rather see it as a list of events that will have happened when the code is executed.

Once students compiled their program, their compiler errors along with the actual program and time stamp were recorded in a table. If there were no compiler errors, the actual program, which contained the recent changes together with the time stamp, was sent to another table. Students were given 120 minutes to debug the program and submit it.

After the exam was finished, all versions of the programs produced by 155 students that had compiled successfully, were studied and compared to each other in order to get an idea of how students debugged the logical errors in the program. On average each student had compiled their program 20 times by the time all the compiler errors were corrected. Therefore 3100 versions of the program had to be investigated, a substantial task!

## 4.1 The Statistical Result of the Study

The statistical results of this experiment show that the majority (66%) of the competent debuggers, that is those who were able to correct all three logical *bugs*, are also competent programmers. In contrast only 39% of the competent programmers were also competent debuggers. This is a rather surprising result, and is best seen in the Venn diagram of figure 1.

This unexpected result led us to categorise the students in various ways such that differences between the resulting groups could be studied. The first study was to investigate the differences between the good and the weak debuggers. The second was to compare the good programmers, who are not necessarily good debuggers, with the weak programmers who were able to debug 1 or 2 errors. None of the weak programmers were able to debug all the three of the errors.

We based this part of the research on the theory of Ducasse & Emde [5], who devised a classification scheme of the knowledge needed for successful debugging. This knowledge includes the knowledge of the intended program function, knowledge of the actual program, knowledge of the programming language, programming expertise, and knowledge of the application domain, the debugging method and the errors.

It should be noted that in this experiment students did not have any high level debugging system available to them. The results of this study are now described.

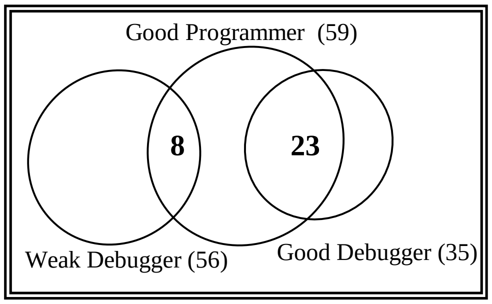

**Figure 1: The relationship between Students' Programming and Debugging Ability**

## 4.2 The Good Debugger vs the Weak Debugger

Comparing these groups of students shows that the able debuggers, regardless of the class of error, posses all seven of the types of knowledge that are required for debugging. However, their understanding of the actual program implementation varied amongst the group. Getting help by using a debugging method and isolating the *bug* was enough for those of them who had less understanding of the actual program implementation to write their own code and not to bother comprehending the actual program implementation.

Investigating the weak debuggers revealed some interesting results. Their knowledge seemed inconsistent, For example they would demonstrate knowledge of a debugging technique in correcting one error, and then seem to lack it when attempting to correct a further *bug*. It is unlikely that they completely forgot how to use a technique that they had just practiced! It seems that they do not have the ability to use the debugging knowledge in all situations. This could imply a lack of experience and confidence in using such knowledge in facing different kinds of programming errors. They have a limited understanding of the scope and applicability of debugging practices. This hypothesis was confirmed when their understanding of the programming language was investigated. In addition, their knowledge of the actual program behaviour seems to vary from error to error. In other words, their knowledge is dependent on the structure of the source code, and whether this is familiar to them.

Finally, the knowledge that this group of students mostly lacked, regardless of the class of error, was an understanding of the intended and actual program implementation.

It should be noted that some of the weak debuggers could isolate the bug but were unable to correct it while the rest could not even localise the error

## 4.3 The Good Programmer vs the Weak Programmer

In this section good programmers, defined as those who could achieve marks of more than 70%, are compared with the weak programmers. Weak programmers are the novices who did not perform well during the first semester of programming. All their marks were below 40%, which constitutes a *fail* in our regulations.

Thirty-nine percent of the good programmers were able to correct all three bugs in the exam program. The majority of the weak programmers, however (70%), were not able to correct any of the bugs, although 30% of them did manage to correct 1 or 2 errors.

This section compares the good programmers who were not able to debug completely against the weak programmers who were able to correct 1 or 2 bugs. That is 14% of the good programmers are studied versus 30% of the weak programmers.

**Good Programmers:** We believe that this group of good programmers has reasonable knowledge of the programming language, expertise and application domain as they could get an overall mark of more than 70% during the semester. Therefore we are trying to discover which factor from the following list is usually the cause of their inability to correct the errors:

Knowledge of:

1. the intended program,
2. the actual program,
3. the use of debugging methods, and
4. the error itself

Our investigation of the process of debugging of this group reveals that one third of the students who could not correct all of the errors were unable to localize the bug (12 out of 36). On the other hand two thirds of them were able to isolate the bug but could not correct it (24 out of 36).

The first group seemed to lack the knowledge of bug itself. None of them were successful in finding the errors. Half of them did not use any debugging method at all, while the other half used some strategy for debugging. Those who applied some strategy for debugging used methods of debugging that were completely irrelevant to the actual errors. Their lack of knowledge was a result of a lack of understanding of the intended program or of the error itself. It might be assumed that all of these students understood the intended program behaviour and as all of them received the program specification and as they were able to run the program they could gain the knowledge of the actual behaviour. Therefore we believe that they lacked understanding of the intended and actual program implementation. The irrelevant changes that they made during the debugging process are strong evidence of this.

On the other hand, second group of good programmers understood the error as they were able to isolate it. Most of them used *print* statement to find the error and the rest used the filtering approach. That is they commented out certain lines. In any case all of them understood basic debugging methods. In common with the first group they understood the actual and intended program behaviour. Investigating the process of debugging used by this group suggests that they were aware of the intended program implementation, as the changes they made were sensible in the context of the intended program behaviour. Although this group understood the intended program implementation and programming language and had expertise, but they did not have the necessary understanding of the actual program implementation. An examination of their attempts provides further evidence for this conclusion.

**Weak Programmers:** The weak programmers were not expected to possess a good knowledge of the programming language or to have much programming expertise yet a third of them were able to correct 1 or 2 errors out of 3. The students who managed to isolate the *bug(s)*, clearly possessed knowledge of the actual program behaviour and gained an understanding of one or two of the *bugs*. They did not use any desktop debugging strategy such as using *print* statements or filtering some code statements. We hypothesise that the novices in this exam had an understanding the intended program in terms of program behaviour as this was given in the program specification. They gained some understanding of the actual program behaviour by running the code. No programming knowledge was needed to get this far. Furthermore this group seems to have a quite reasonable level of understanding of the actual program implementation. While they lacked knowledge of the programming language and programming expertise, they were still able to simulate the right code by understanding the initial incorrect condition.

In summary it seems that the most important knowledge lacked by good programmers, who are unable to debug, is an understanding of the actual program implementation. Perhaps understanding code written by other people is a higher-level task than writing one's own. Certainly program maintenance is widely disliked by practicing programmers. Their knowledge of the error and their debugging strategies also seems to be in need of improvement because a group of the good programmers did not possess this knowledge and skill. On the other hand weak programmers who were able to correct the mistakes had a fair understanding of the actual program implementation.

## 4.4 Discussion

We discovered the rather surprising result that the majority of good debuggers are good programmers while less than half of the good programmers are good debuggers. An attempt to explain this was made by comparing the various kinds of knowledge necessary for debugging amongst differently performing student groups. In particular, the good programmers were compared with those who were weak at this skill. The knowledge of the actual program implementation seems to be the key factor that prevents many of the good programmers becoming good debuggers as well. It is clear that the ability to read and understand other people's code is an important skill, which improves the overall competence of novice programmers.

## 5. FUTURE WORK

In first phase of the study we observed that "failing to define a variable" is almost always the most common mistake. This should be an issue for a qualitative research to investigate whether this error is caused by a misconception or is simply a mistake in the same sense as a typographical error.

In the second phase of the study, we found that the understanding of the program is a major issue even for competent programmers. Therefore we will design our next programming course in such a way that students not only practice ‘programming’ but also get more experience in reading and understanding existing programs. By enhancing the level of the comprehending a program, we will aim to find out whether or not this increases their overall competence at programming.

## 6. REFERENCES

[1] Cook, C., Burnett M., and Boom D. A bug's eye view of immediate visual feedback in direct-manipulation programming systems. Papers presented at the seventh workshop on Empirical studies of programmers. Alexandria, Virginia, United States: ACM Press New York, NY, USA, 1997. 20 - 41.

[2] Coull, N., Duncan I., Archibald J., and Lund G. Helping Novice Programmers Interpret Compiler Error Messages. 4th Annual LTSN-ICS Conference. National University of Ireland, Galway. (August, 2003). 26 - 28.

[3] Cunniff, N., Taylor R. P., and Black J. B. Does programming language affect the type of conceptual bugs in beginners' programs? A comparison of FPL and Pascal. Proceedings of the SIGCHI conference on Human factors in computing systems table of contents, 1986. 175 - 182.

[4] Du-Boulay, B. Some Difficulties of Learning To Program. In: (Ed.). Studying the Novice Programmer, 1989. 283-299.

[5] Ducasse, M., and Emde A. M. A review of automated debugging systems: knowledge, strategies and techniques. Proceedings of the 10th international conference on Software engineering, 1988. 162-171.

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

[7] Green, T., and Petre M. Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework. Journal of Visual Languages and Computing, 7, 2, (June 1996). 131-174.

[8] Gugerty, L., and Olson G. Debugging by skilled and novice programmers. ACM SIGCHI Bulletin , Proceedings of the SIGCHI conference on Human factors in computing systems. (April, 1986).

[9] Gugerty, L., and Olson G. M. Comprehension differences in debugging by skilled and novice programmers. Papers presented at the 1st workshop on empirical studies of programmers. Washington, D.C., US. (June 1986). 13-27.

[10] Higgins, C., Symeonidis P., and Tsintsifas A. The Marking System for CourseMaster.Proceedings of the 7th annual conference ITiCSE. Aarhus, Denmark: ACM Press New York, NY, USA, 2002. 46-50.

[11] Hristova, M., Misra A., Rutter M., and Mercuri R. Identifying and correcting Java programming errors for introductory computer science students. Proceedings of the 34th SIGCSE technical symposium on Computer science education. Reno, Navada, USA: ACM Press New York, NY, USA, 2003. 153 - 156.

[12] Ishio, T., Kusumoto S., and Inoue K. Program slicing tool for effective software evolution using aspect-oriented technique. Proceedings of the Sixth International Workshop on Principles of Software Evolution, 2003. 3 - 12.

[13] Jcreator. Java IDE. http://www.jcreator.com/. 2003

[14] Jikes. IBM- developerworks - open source software.(http://oss.software.ibm.com/developerworks/opensource/jikes/index.shtml). Last Accessed on October,2003

[15] Johnson, L., Soloway E., Cutler B., and Draper S. Bug Catalogue: I, Technical Report No. 286. Yale University, Department of Computer Science. New Haven. 1983

[16] Pea, R. D. Language-independent conceptual bugs in novice programming. Journal of Educational Computing Research, 2, 1, 1986. 25-36.

[17] Robertson, T. J., Prabhakararao S., Burnett M., Cook C., Ruthruff J. R., Beckwith L., and Phalgune A. Impact of interruption style on end-user debugging. Proceedings of the 2004 conference on Human factors in computing systems. Vienna, Austria: ACM Press New York, NY, USA, 2004. 287 - 294.

[18] Romero, P., Lutz R., Cox R., and Du Boulay B. Co-ordination of multiple external representations during Java program debugging. Proceedings of IEEE Symposia on Human Centric Computing Languages and Environments. (september, 2002). 207 - 214.

[19] Spohrer, J., Soloway E., and Pope E. A Goal/Plan analysis of Buggy Pascal Programs. Human Computer Interaction, 1, 1985. 163-207.

[20] Spohrer, J. C., and Soloway E. Analyzing the high frequency bugs in novice programs. Empirical Studies of Programmers: First workshop: Ablex Publishing Corp, 1986.

[21] Spohrer, J. C., and Soloway E. Novice Mistakes: Are The Folk Wisdom Correct? Communications of the ACM, 29, 7, (july, 1986). 624-632.

[22] Spohrer, J. C., Soloway E., and Pope E. Where the bugs are. Proceedings of the SIGCHI conference on Human factors in computing systems, (April, 1985). 47-53.

[23] Truong, N., Roe P., and Bancroft P. Static analysis of students' Java programs. Proceedings of the sixth conference on Australian computing education, 30, (January, 2004). 317-325.

[24] Tubaishat, A. A knowledge base for program debugging. International Conference on Computer Systems and Applications, ACS/IEEE. Beirut. (June, 2001). 321 - 327.

[25] Uchida, S., Monden A., Iida H., Matsumoto K., and Kudo H. A multiple-view analysis model of debugging processes. Proceedingsof the International Symposium on Empirical Software Engineering. (Oct, 2002). 139 - 147.