Enhancing Syntax Error Messages Appears Ineffectual

Paul Denny, Andrew Luxton-Reilly, Dave Carpenter
Dept. of Computer Science
The University of Auckland
Auckland, New Zealand
{paul, andrew}@cs.auckland.ac.nz, dcar111@aucklanduni.ac.nz

ABSTRACT

Debugging is an important skill for novice programmers to acquire. Error messages help novices to locate and correct errors, but compiler messages are frequently inadequate. We have developed a system that provides enhanced error messages, including concrete examples that illustrate the kind of error that has occurred and how that kind of error could be corrected. We evaluate the effectiveness of the enhanced error messages with a controlled empirical study and find no significant effect.

Categories and Subject Descriptors

K.3.2 [**Computers and Education**]: Computer and Information Science Education—*computer science education*

General Terms

Design, Human Factors

Keywords

debugging; errors; syntax error; error messages; feedback; novice; programming

1. INTRODUCTION

Despite numerous studies on debugging, it remains a difficult skill for novices to acquire [12]. Code written by students can be filled with errors, and poor debugging skills can lead to frustration and the introduction of new errors [13]. Furthermore, very few students who have trouble debugging code are able to perform well in a course [1].

Although Fitzgerald et. al [7] found that construct-related bugs (i.e. those that are due to misunderstanding or confusion about the language) are easier to fix than those which are language independent, syntax errors can still be major obstacles that slow the progress of novice programmers. As Kummerfeld and Kay [11] note, "Syntax error correction is the first step in the debugging process. It is not possible to continue program development until the code compiles. This means it is a crucial part of the error correction process."

ITiCSE'14, June 21–25, 2014, Uppsala, Sweden.

http://dx.doi.org/10.1145/2591708.2591748.

Denny et. al [5] explored the frequency with which students encountered compilation errors when writing relatively short fragments of code. In their study, although successful solutions consisted of a median of only 8 lines of code, approximately 70% of students experienced four or more consecutive syntax errors even when the compiler output was being shown to them. Students in the lowest performance quartile of the course encountered particular difficulty with syntax. In some cases, even after a significant amount of time and effort, students were observed abandoning an exercise when they were unable to write a compiling submission and receive feedback on the logic of their code.

Our experience is that syntax errors can be a significant barrier to student success. One example typifying the extent of this problem involves a student who spent close to 2 hours attempting to test whether the sum of two numbers was even or odd. The student was trying to solve the following exercise using an online tool in which all compilation attempts were captured:

> *Complete the* `isOddSum()` *method in Java. This method should calculate the sum of the two input values and return true if the sum is an odd number, and false if the sum is an even number. For example, the sum of 10 and 20 is 30, an even number, so the method call* `isOddSum(10, 20)` *should return false.*

To begin the exercise, the student was provided with the method signature as follows:

```
public boolean isOddSum(int a, int b)
```

The student made 31 distinct code submissions in their attempt to solve this exercise. As shown in Figure 1, which plots the time at which each submission was made, the student appeared to work fairly conscientiously on this problem.

Their first submission, made at approximately 8:35pm, is shown in Figure 2. There are many syntax errors in this code: the cast is performed incorrectly, the assignment operator is being used where an equality test is needed, semicolons are missing, the second keyword "if" has a capital letter, the return variable "sum" is declared with the wrong scope and is not the correct type as required by the method.

Over the next hour and a half the student made progress towards correcting these errors, and the second to last submission they made, at approximately 10:15pm is shown in Figure 3.

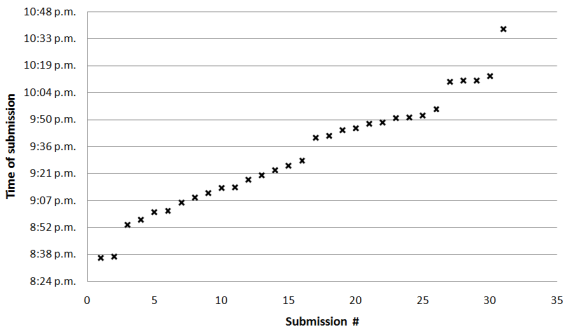

Figure 1: Time at which successive attempts were made

```
if((a=2*int( a/2))&&(b=2*(int b/2))) {
    int sum=a+b
}
If(sum=2*int(sum/2)) {
    sum=0
}
return sum
```

Figure 2: Student X's first attempt at the isOddSum() exercise

```
int sum = 0;

if(a==2*(a/2)) {
    if(b!=2*(b/2)) {
        sum = a+b;
    }
} else {
    if(b==2*(b/2)) {
        sum = a+b;
    }
}

return sum;
```

Figure 3: Student X's 30th, and penultimate, attempt at the isOddSum() exercise

```
if (n%2==0) {
    n = false;
} else {
    n = true;

return n;
```

Figure 4: Student X's final, and unsuccessful, attempt at the isOddSum() exercise

Although logically this code is not correct (for example, both branches of the condition assign the same value to the return variable), only one syntax error remains (the type of the return value does not match the return type of the method) and as a result the compiler would generate a "Type mismatch" error message. However, at this point the student appears to give up on this approach and approximately 20 minutes later made their final submission to this exercise, which was again both syntactically and logically incorrect, as shown in Figure 4.

It is unfortunate that this student was not able to resolve their syntax errors, as this meant they never received feedback on the logical correctness of their code. Providing targeted help to get students over this problematic syntax hurdle is the primary motivation behind our current work.

In this paper we report on an attempt to improve novice debugging performance by providing enhanced syntax error messages, directly within the development environment, that include a more verbose description of the error than the standard Java compiler. Inspired by the work of others in this area, our enhanced feedback also displays an example of code that contains the same kind of error, and a corresponding corrected version, along with details of how the example code could be corrected. We evaluate the effectiveness of the enhanced error messages with a controlled empirical study involving first year students. Our main research question is: "Does the enhanced feedback have any measurable impact on how effectively students resolve syntax errors in their code?".

2. RELATED WORK

A number of other attempts to provide enhanced feedback have been reported in the literature.

Kummerfeld and Kay [11] observe that students struggle to correct syntax errors, and speculate that perhaps the incomprehensibility of the compiler error messages contribute to the difficulties. They provided students with a reference guide that catalogues common compiler errors. Each error message is explained and illustrated with an example of code that contains the error, and how the error could be corrected. A qualitative study of debugging behaviour involving students with access to the reference guide showed that some students seemed to not understand the syntax error until they looked at the example code. This suggests concrete examples are important for teaching syntax errors.

Code Analyser for Pascal (CAP) was developed to provide automated feedback on program syntax, logic and style errors [15]. The CAP tool performed static analysis of the code and provided feedback about what was wrong, why it was wrong and how to fix the problem. The feedback messages often included correct exemplar code for students to model. Students who used the tool reported that they valued CAP as a learning tool, and instructors reported that they spent less time grading because CAP improved the quality of student programs. However, the author observed that some students did not read the error messages, and others became dependent on the error messages for debugging.

Gauntlet [8] provides enhanced feedback by searching student's Java code for a number of commonly encountered syntax and semantic errors and presenting them to students in a more colloquial manner. After using the Gauntlet system in classes at the United States Military Academy for 18 months, teaching staff noted that the work produced by students was of a higher quality and that their workload was reduced, as fewer students were visiting them in their office hours looking for assistance. However, it became apparent that Gauntlet was not addressing the most common errors encountered by students because the errors supported by Gauntlet were chosen by instructors based on personal experience rather than empirical data [9].

Dy & Rodrigo [6] investigated automating the detection of non-literal Java errors where following the compiler's suggested correction will create a different error. They analysed data collected from student compilations to determine the most common errors, and developed an error detector that could automatically detect the kind of error based on the compiler message and the source code. This detector could, in principle, be used to provide enhanced feedback which may help students to correct common errors, but the system has not yet been deployed in a classroom context.

Toomey [16] modified the BlueJ environment to provide more detailed error messages, along with examples of incorrect and correct code related to the error message. However, this work has not (to our knowledge) been published or evaluated.

Carter & Blank [2] report on the design of an intelligent tutoring system that provides enhanced error messages similar to those we present here. The intelligent tutoring system is currently being developed and as yet no evaluation of the enhanced error messages has been performed.

It should be noted that few of the previous reports have attempted to rigorously evaluate the effectiveness of enhanced error messages in classroom contexts, and none of them provide empirical data on the impact that the error messages have on student debugging performance.

3. ENHANCING ERROR MESSAGES

We decided to implement an enhanced feedback system to users of CodeWrite [4]. CodeWrite is a web-based tool in which students complete a series of exercises that require them to write the body of a method in Java. The header of the method is always provided. Code is entered directly in the browser and is compiled and run against a set of test cases automatically. When code fails to compile, CodeWrite passes the first two compiler-generated error messages back to students. Limiting the number of errors shown to students is consistent with other well-known Java teaching environments such as BlueJ [10], and is intended to help students focus on a single error at a time.

The first step in building the enhanced feedback module was to create a recognizer that parsed both the submitted source code and the raw compiler messages, to categorize the messages according to error type. We began by examining previous student submissions (from the Second Semester 2012) that failed to compile in order to find common patterns in the code. The data set that we analysed to create the recognizer had 12369 submissions containing syntax errors.

The compiler error messages could be used to correctly identify some of the errors (approximately 78%), but the message alone was not sufficient to distinguish between some other kinds of errors. For example, consider the following two code fragments:

```
int x = 10;
int z = int x;
```

and

```
int x = 10;
int y = 20;
int z = Math.min(int x, int y);
```

In both cases, the compiler generates exactly the same error message:

```
Syntax error on token "int", delete this token
```

Although both problems are similar, if we can determine that the errant type appears prior to a parameter being passed to a method, we are able to provide more contextual feedback that includes a discussion of how to correctly pass inputs to methods via parameters.

To categorize the errors that had ambiguous compiler messages, we performed a static analysis of the code using regular expressions to match commonly occurring patterns of code that caused errors. This approach successfully identified another 14% of the total errors. We stopped adding new error types to the recognizer after the recognizer was capable of categorizing the errors present in 92% of the submissions from the 2012 data set, and each remaining error type was present in only a few submissions.

Once the error had been detected by the recognizer, we extracted the line containing the error from the code so that it could be highlighted in the feedback to aid the student in locating the error. Our recognizer identifies 53 different types of syntax errors, which we have classified into 9 different categories. Table 1 summarises the different categories of error and the number of errors included in each category.

Category	N
Incorrect return statements	3
Misused or unmatched braces or parenthesis	10
Variable or type cannot be resolved	2
Type mismatches	2
Incorrect if statements	9
Incorrect method calls	8
Missing or unexpected character e.g. semicolon	6
Incorrect assignment/creation of variable/object	9
Other	4

Table 1: Number of errors in each category identified by the error recognition module

The feedback provided by the enhanced feedback module typically contains the line that the error occurs on (as extracted by the recognizer) and a detailed explanation of what is most likely causing the syntax error. The enhanced feedback also includes a table showing two code fragments side by side, the first of which includes a simple syntax error of the type that has been recognised and the other showing the corrected version of the code with the syntax differences highlighted. The final column of this table provides an explanation of the error in the first code fragment and describes how it has been corrected in the second.

Consider the method definition below, in which the compound conditional statement is missing surrounding parentheses.

```
public boolean validScore(int score)
{
    if (score < 0 ) || (score > 100)
        return false;
    return true;
}
```

The raw error message produced by the compiler for this code is shown in Figure 5. In contrast, the enhanced feedback is shown in Figure 6 and represents an example of an error in the "*Incorrect if statements*" category listed in Table 1.

```
if (score < 0 ) || (score > 100)
Syntax error on token "||", if expected

1 problem (1 error)
```

Figure 5: An example of an original error message

4. EVALUATION

To evaluate the effectiveness of the enhanced feedback, we trialled the modified CodeWrite tool in an introductory programming course (CS1) in the Summer Semester 2013 at The University of Auckland. The Summer Semester course covers the standard CS1 curriculum at an accelerated pace with 6 lectures and 2 laboratories per week over a 6 week period compared with the standard 3 lectures and 1 laboratory per week over a 12 week period in other semesters.

Students were required to successfully answer 10 exercises in CodeWrite for 1% of their final grade. This activity spanned the second and third weeks of the six week course. As the students were still in the first half of their course, the exercises covered expressions, conditionals and methods from the java.lang.String and java.lang.Math classes, but did not cover arrays and loops.

In total, there were 90 students in the class, but only 83 contributed at least one submission. Students were randomly allocated to a control group (N=42) that received raw compiler feedback, or an intervention group (N = 41) that received the enhanced feedback. Both groups completed the same exercises.

We classify every student submission into one of the three types shown in Table 2.

Type	Explanation
P	the submission compiles and all tests pass
F	the submission compiles but fails at least one of the tests
X	the submission does not compile

Table 2: Types of student submissions

Once a student submits their code they receive instant feedback on its correctness. In the case of an "F" submission, the student is shown the passing and, importantly, failing test cases. In the case of an "X" submission, they are shown either the raw compiler error message or the enhanced feedback produced by our module, depending on the group to which they have been assigned.

Our main goal in this project was to improve the effectiveness of student debugging. In particular, we wanted to help those students who tended to repeatedly submit non-compiling code, apparently unable to resolve the syntax errors they encountered. To evaluate the enhanced feedback, we investigated whether it had any impact on:

- the number of consecutive non-compiling submissions made while attempting a given exercise,
- the total number of non-compiling submissions across all exercises, and
- the number of attempts needed to resolve the most common kinds of errors.

5. RESULTS

5.1 Did the enhanced feedback reduce the number of consecutive non-compiling submissions?

As mentioned earlier, a previous study by Denny et. al [5] revealed that the majority of students experienced a "syntax issue", in which code was unsuccessfully compiled at least 4 consecutive times before syntax errors were resolved. We were particularly concerned about students (such as Student X mentioned in the introduction) who were unable to correct the syntax errors in their code using compiler messages.

We looked at the submissions made to each exercise separately. For each student, we classified their sequence of submissions to a given exercise using the categories described in Table 2. For example, a sequence such as "XXXXFXXP" indicates that a student submitted code that failed to compile 4 times in a row, followed by code that compiled but failed one or more test cases, followed by two submissions of code that failed to compile before finally submitting code that compiled and passed all the test cases.

For each exercise, we captured the degree to which a given student was stuck with a "syntax issue" by recording the longest sequence of consecutive "X" submissions the student made to that exercise. We then compared these values for all students in the control group with all students in the intervention group using a two-sample t-test. Although data items varied considerably, Shapiro-Wilk tests indicated normality of the data in most cases. For two of the exercises, in which the data was particularly skewed with a few students encountering a large number of consecutive syntax errors, the data was log transformed prior to testing (results shown in the log p column). Table 3 summarizes the results, giving both the mean and standard deviation (in parentheses) for each exercise. There were no significant differences between groups.

Ex	$\mu_{control}(\sigma)$	$\mu_{enhanced}(\sigma)$	p	log p
1	9.72 (12.5)	6.74 (9.82)	0.25	
2	2.05 (2.20)	3.68 (3.83)		0.08
3	9.65 (11.8)	8.79 (10.1)		0.89
4	4.46 (5.01)	6.24 (8.38)	0.25	
5	5.73 (8.08)	6.35 (7.99)	0.73	
6	3.83 (5.77)	3.44 (5.61)	0.77	
7	4.63 (5.43)	5.16 (7.44)	0.72	
8	2.14 (6.69)	1.69 (2.89)	0.70	
9	3.07 (7.26)	2.34 (3.79)	0.57	
10	8.80 (20.7)	4.56 (7.92)	0.23	

Table 3: The longest consecutive sequence of submissions that failed to compile for the control group ($\mu_{control}$) compared with the intervention group ($\mu_{enhanced}$) that received enhanced feedback.

5.2 Did the enhanced feedback reduce the total number of non-compiling submissions?

To determine if the enhanced error messages made a difference to the total number of non-compiling submissions, we compared the mean number of submissions of each type made by students in each group. Table 4 gives the total number of submissions of each type for the control and in-

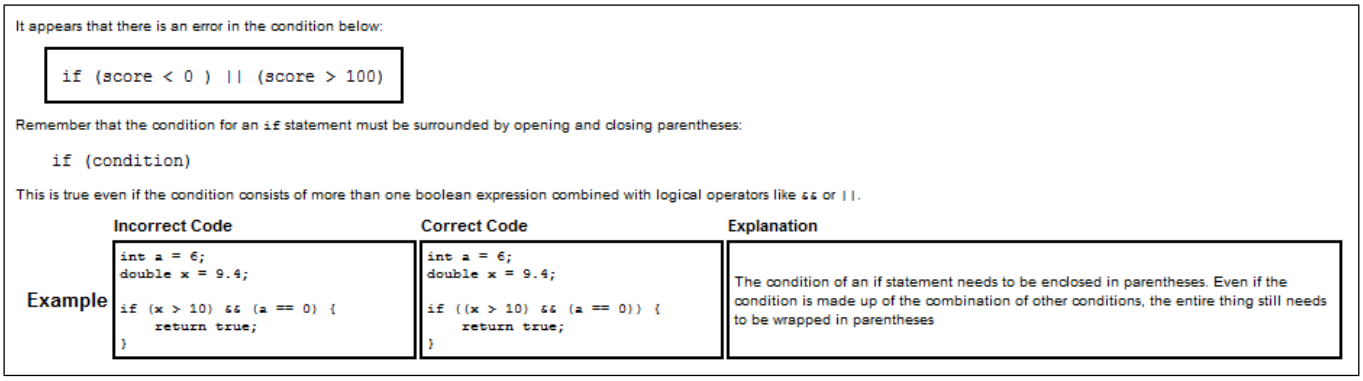

Figure 6: An example of an enhanced error message

tervention groups (means and standard deviations are in parentheses).

Type	Control	Intervention
P	**450** ($\mu, \sigma = 10.7, 3.36$)	**434** ($\mu, \sigma = 10.6, 4.00$)
F	**1892** ($\mu, \sigma = 45.0, 49.4$)	**1656** ($\mu, \sigma = 40.4, 52.8$)
X	**3343** ($\mu, \sigma = 79.6, 86.0$)	**2760** ($\mu, \sigma = 67.3, 68.8$)

Table 4: Summary of submissions of each type for the control group (raw error messages) and the intervention group (enhanced error messages). Totals are in bold, means and standard deviations are parenthesized

Although students viewing the enhanced error messages made fewer non-compiling submissions overall, the variance of both groups was high, and the difference between the means was not significant (p = 0.9471).

Note that the number of submissions with logic errors (i.e. submissions that compiled, but failed one of the tests) was also lower among the intervention group, but the difference between means was not significant (p = 0.9941)

5.3 Did the enhanced feedback reduce the number of attempts needed to resolve the most common kinds of errors?

For this question we have looked more in depth at the three most common syntax errors as reported by Denny et al. [3]. Each of these syntax errors are investigated separately:

1. Cannot resolve identifier
2. Type mismatch
3. Missing semicolon

A submission is said to have a syntax error of a particular type when the error is first reported in response to compilation. The error is said to have been resolved when the syntax error is no longer reported to students in the feedback for that submission. We measured the average number of compiles that it takes a student to resolve each type of error. Having calculated the average number of compiles for each student, we compare the mean of these averages for the control group and the intervention group.

For each syntax error investigated, we found that the average number of compiles for students to resolve the error varied greatly. In order to perform a two-sample t-test on the data, we first performed a log transformation to make the variances approximately equal. Table 5 summarizes the results of the t-tests. In each case, a t-test for a difference between the means of the logged data did not result in a significant p-value, so we have no evidence that the enhanced feedback affects the average number of compiles needed to resolve any of these common syntax errors.

Syntax error	$\mu_{con} - \mu_{enh}$	p
Cannot resolve identifier	0.15	0.2369
Type mismatch	-0.15	0.2783
Missing semicolon	-0.10	0.4449

Table 5: The difference between the means of the control group (μ_{con}) and the intervention group receiving enhanced feedback (μ_{enh}) for the average number of compiles required to resolve each type of error

6. DISCUSSION

Although we anticipated that the enhanced error messages would help students to identify and correct errors, analysis of the data shows no significant (or practical) effect. Each student experienced approximately 70 submissions that failed to compile, but the enhanced error messages did not appear to reduce the number of "syntax issues" experienced, the total number of compilations required to solve all problems, or to help resolve the most common errors encountered. In essence, we found no empirical evidence to support the use of enhanced error messages.

There are a number of reasons why the enhanced error messages may not have helped students. It is possible that the majority of errors may have been simple enough to solve without the enhanced messages. For example, missing semicolons represent one of the more common student errors and the corresponding error message generated by the compiler, "*Syntax error, insert ; to complete statement*", may provide adequate information to most students without the need for additional explanation. The cases for which the enhanced messages are particularly useful may be too infrequent in our data to yield significant results.

Another explanation for our findings is that students in the intervention group did not pay much attention to the additional information in the enhanced error messages. This is consistent with findings by Kummerfeld and Kay [11], who

note that some students did not use their reference guide that helped explain the likely cause of the errors encountered, and with the observation of Schorch who states "Some students do not read the CAP error messages fully and thus are probably not learning as much as we would like" [15].

The enhanced error messages we provided were more verbose than the raw error messages and although they may have provided an opportunity to learn about the likely cause of the error, students may have been resistant to reading additional detail beyond the simple compiler output, especially when they encountered the same error multiple times.

A third possible explanation was that the enhanced feedback did not provide examples and explanations that students could relate to their own code. The examples were intended to illustrate the kind of situation that might cause an error of the type encountered by a student, but it relies on students understanding the idea and transferring the knowledge to their own situation.

One possible threat to the validity of our findings is that the raw compiler feedback shows up to two compiler errors, while the enhanced feedback module displays only one error message to reduce the complexity for students. This may allow some students to correct two errors at once while using the raw compiler messages, or may possibly confuse other students by presenting more than one error to correct. However, it should be noted that previous research by Denny et al. [3] found that most (approximately 70%) submissions that failed to compile had only one syntax error.

In future, an observational study of how students are using the enhanced feedback in practice may shed some light on the reasons why our implementation was unsuccessful. We note that further study of enhanced error feedback would be valuable for systems (such as our own, and that of CloudCoder [14]) intended to support short exercises and practice in environments outside of typical supervised laboratories where personal debugging support is limited.

7. CONCLUSIONS

Syntax errors are one of the biggest problems for students learning to program. They slow students' progress and prevent them from getting feedback on the logic of their code. As educators, we have a limited amount of time to spend with each student so providing students with automated and useful feedback about why they are getting syntax errors is very important.

Although a number of researchers have previously investigated the use of enhanced compiler feedback, few provide any quantifiable results. We built an enhanced feedback module that recognises syntax errors and generates more descriptive feedback on what caused the error. The enhanced feedback module was evaluated using a controlled empirical study in which the debugging behaviour of students receiving enhanced feedback was compared with that of students receiving standard compiler feedback.

Despite our initial prediction that the enhanced feedback would improve students' performance, our results show that there was no significant benefit for students that received the enhanced feedback. This is a somewhat surprising result that warrants further investigation. It does, however, signify that the development of teaching resources need to take account of student behaviour and that innovations intended to support student learning should be evaluated in the context of real classroom situations.

8. REFERENCES

[1] M. Ahmadzadeh, D. Elliman, and C. Higgins. An analysis of patterns of debugging among novice computer science students. In *Proc. ITiCSE '05*, pages 84–88, 2005. ACM.

[2] E. Carter and G. D. Blank. A tutoring system for debugging: status report. *J. Comput. Sci. Coll.*, 28(3):46–52, Jan. 2013.

[3] P. Denny, A. Luxton-Reilly, and E. Tempero. All syntax errors are not equal. In *Proc. ITiCSE '12*, pages 75–80, 2012. ACM.

[4] P. Denny, A. Luxton-Reilly, E. Tempero, and J. Hendrickx. Codewrite: Supporting student-driven practice of java. In *Proc. SIGCSE '11*, pages 471–476, 2011. ACM.

[5] P. Denny, A. Luxton-Reilly, E. Tempero, and J. Hendrickx. Understanding the syntax barrier for novices. In *Proc. ITiCSE '11*, pages 208–212, 2011. ACM.

[6] T. Dy and M. M. Rodrigo. A detector for non-literal java errors. In *Koli Calling '10*, Koli, Finland, October 28–31, 2010. ACM.

[7] S. Fitzgerald, G. Lewandowski, R. McCauley, L. Murphy, B. Simon, L. Thomas, and C. Zander. Debugging: finding, fixing and flailing, a multi-institutional study of novice debuggers. *Computer Science Education*, 18(2):93–116, 2008.

[8] T. Flowers, C. Carver, and J. Jackson. Empowering students and building confidence in novice programmers through gauntlet. In *Frontiers in Education, 2004. FIE 2004. 34th Annual*, pages T3H/10–T3H/13 Vol. 1, 2004.

[9] J. Jackson, M. Cobb, and C. Carver. Identifying top java errors for novice programmers. In *Frontiers in Education, 2005. FIE '05. Proceedings 35th Annual Conference*, pages T4C–T4C, 2005.

[10] M. Kölling, B. Quig, A. Patterson, and J. Rosenberg. The bluej system and its pedagogy. *Computer Science Education*, 13(4):249–268, 2003.

[11] S. K. Kummerfeld and J. Kay. The neglected battle fields of syntax errors. In *Proc. ACE '03*, vol 20, pages 105–111, Australia, 2003. ACS.

[12] R. McCauley, S. Fitzgerald, G. Lewandowski, L. Murphy, B. Simon, L. Thomas, and C. Zander. Debugging: a review of the literature from an educational perspective. *Computer Science Education*, 18(2):67–92, 2008.

[13] L. Murphy, G. Lewandowski, R. McCauley, B. Simon, L. Thomas, and C. Zander. Debugging: the good, the bad, and the quirky – a qualitative analysis of novices' strategies. In *Proc. SIGCSE '08*, pages 163–167, 2008. ACM.

[14] A. Papancea, J. Spacco, and D. Hovemeyer. An open platform for managing short programming exercises. In *Proc. ICER '13*, pages 47–52, 2013. ACM.

[15] T. Schorsch. Cap: an automated self-assessment tool to check pascal programs for syntax, logic and style errors. In *Proc. SIGCSE '95*, pages 168–172, 1995. ACM.

[16] W. Toomey. Bluej with modified error subsystem. *http://minnie.tuhs.org/Programs/BlueJErrors*, 2011.