An Effective Approach to Enhancing Compiler Error Messages

Brett A. Becker
School of Computer Science
University College Dublin
Belfield, Dublin 4, Ireland
brett.becker@ucd.ie

ABSTRACT

One of the many challenges novice programmers face from the time they write their first program is inadequate compiler error messages. These messages report details on errors the programmer has made and are the only feedback the programmer gets from the compiler. For students they play a particularly essential role as students often have little experience to draw upon, leaving compiler error messages as their primary guidance on error correction. However these messages are frequently inadequate, presenting a barrier to progress and are often a source of discouragement. We have designed and implemented an editor that provides enhanced compiler error messages and conducted a controlled empirical study with CS1 students learning Java. We find a reduced frequency of overall errors and errors per student. We also identify eight frequent compiler error messages for which enhancement has a statistically significant effect. Finally we find a reduced number of repeated errors. These findings indicate fewer students struggling with compiler error messages.

Keywords

Java; debugging; errors; syntax errors; error messages; compiler errors; feedback; novice; programming; CS1

1. INTRODUCTION

Good compiler error messages provide feedback that is critical for novices [17]. Unfortunately they are frequently inadequate and pose a significant barrier to progress [21], particularly affecting novices more adversely [23]. Error messages and the difficulties they present to students have been pervasive themes in teaching beginners to program for several decades. In 1976, 'cryptic diagnostics' was named by students as one of the primary issues that made programming difficult [25]. Thirty years later Jadud noted that error messages generated by commercial compilers were often uninformative and sometimes misleading [14]. Unfortunately these characteristics are still indicative of the error messages provided by modern languages today, forty years later; cryptic and uninformative, often terse and misleading.

Although this issue spans many (if not all) languages, there is evidence that Java error messages are particularly difficult for novices, and means of learning how to deal with them more effectively are needed. Ben-Ari noted that Java error messages are terse and many novices find it difficult to achieve syntactically correct Java programs [3]. Our experience is that Java compiler error messages, like those in other languages, can be a significant barrier to student success, a view shared by others [7, 8].

It has been shown that the majority of students spend the majority of their time on a minority of errors and that students spend more time solving certain kinds of syntax errors than others [7]. In addition, repeating errors is common, and the number of repeated errors is an indication of struggling students. Jadud found that repeated errors were often the best indicator for how well (or poorly) a student was progressing [14].

In this paper we report on the effects of a Java editor called Decaf which enhances javac compiler error messages. When a student error generates an error message, the offending code and error message are analyzed in an effort to produce an enhanced error message which provides more usable information to the student, in the hope that the error can be rectified more effectively than with the standard error message alone. It is hoped that by presenting both the regular and enhanced errors together, students will learn from the enhanced error messages and avoid making the same mistakes in the future. The main research question is: *Do enhanced compiler error messages have a measurable effect which may help students who are learning to program?* In investigating this we looked at three specific questions: 1. Did enhanced compiler error messages reduce the overall number of student errors? 2. Did enhanced compiler error messages reduce the number of errors per student? 3. What effect to enhanced compiler error messages have on the number of repeated errors?

2. RELATED WORK

Several studies have researched enhanced error messages. A recent paper [8] describes six, and here we present another six. For a more in-depth discussion of these and more, see [2].

Hristova et al. [11] introduced Expresso, a pre-compiler which scans Java programs for 20 common errors. These errors were identified through a survey of their students and faculty at several institutions including their own. Expresso

ACM ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

provides users with explanatory messages provided the error in question is one of those identified. The authors describe their system as an interactive tool that would do a better job generating error messages than existing compilers and also provide suggestions on how to fix the code. However an assessment of Expresso was left to future work.

Thompson and Rigby [19] introduced an Eclipse plug-in called Gild, specifically for novice Java programmers. Gild includes a feature which provides compiler error message explanations and possible causes in plain English. Messages for which the authors felt a longer explanation was required were explained in a wiki. Several online resources were used to generate a list of 51 errors for which Gild could provide enhanced compiler error messages. At the end of the study, 10 of these errors accounted for 68% of all errors. The study had many objectives, with the effects of enhanced compiler error messages making up three of six research questions. In addition, it was an exploratory work with a small number of students – less than 10 for the quantitative results, depending on the sub-study in question. The results were not conclusive as to whether or not students became faster at fixing their errors over the course of the study (one semester), or if their errors changed over the course of the semester. 57% of 28 students found Gild's extra error help feature "useful at least some of the time". It was concluded that Gild needed more specific error messages and better coverage of errors most encountered by students.

Coull [5] introduced a framework for support tools that addresses both program and problem formulation for novices. One of the requirements of such tools is to present both standard compiler error messages and the enhanced messages concurrently. Only one of three systems categorized by Coull met this requirement and focused on syntax errors in high-level languages: CAP [20] which was for Pascal, and often provided exemplar code. Coull also developed SNOOPIE using the framework, for learning Java programming. Although the scope of SNOOPIE was well beyond enhancing compiler error messages, one of the primary objectives of the tool was just that. It was shown that this support was beneficial to a small group of students, particularly for non-trivial syntactic errors.

Hartmann et al. [10] developed HelpMeOut, a 'social recommender system' that aids the debugging of error messages by suggesting solutions that peers have applied in the past. A limited evaluation was carried out and a longer deployment deemed necessary to determine just how successful the system is.

Watson, Li and Godwin [24] developed Bluefix, an online tool integrated into BlueJ [15] which provides varying levels of elaborate feedback, incorporating crowd-sourced error fixes. An evaluation was conducted suggesting a 19% improvement over HelpMeOut and revealing that students viewed the tool positively.

Many of the studies discussed in this section in addition to those discussed in [8] focus on addressing the problem with compiler error messages, but lack empiricism in determining if they make any difference, particularly to novices. Denny, Luxton-Reilly and Carpenter [8] implemented an enhanced feedback system to users of CodeWrite [7], a web-based tool designed to help students complete Java exercises. This 2014 study was the first recent work on the effect of enhancing Java compiler error messages with control and intervention groups. The system was used with students attempting exercises which required them to complete the body of a method for which the header was provided. Students participated for a period of two weeks as part of an accelerated summer course. Students were required to complete 10 lab exercises using the tool with 83 students submitting at least one. The study found no significant effect on the number of consecutive non-compiling submissions, the total number of non-compiling submissions, and the number of attempts needed to resolve the most common kinds of errors.

3. THE TOOL AND METHODS

Decaf analyzes a student's source code and compiler message and presents an enhanced message where possible. A total of 30 error messages result in enhanced messages, which are then customized based on analyzing the offending code. The enhanced message appears alongside the original as recommended by [6]. If no enhanced error message is possible the user is presented with a statement indicating the line number of the first error, and referring the user to the standard javac output which is in the same window (see Figure 1). Decaf only presents an enhanced version of the first error message when more than one are present. This is consistent with other teaching tools such as BlueJ [15] and more recent work [8], but is not always the case, for instance with Expresso [11]. We do not categorize or group compiler error messages unlike previous studies [9, 12, 13] as this can complicate comparison [8].

Figure 1 shows the Decaf error window displaying an enhanced compiler error message resulting from incorrectly calling the length() method of the String class, generating a *cannot find symbol* error message.

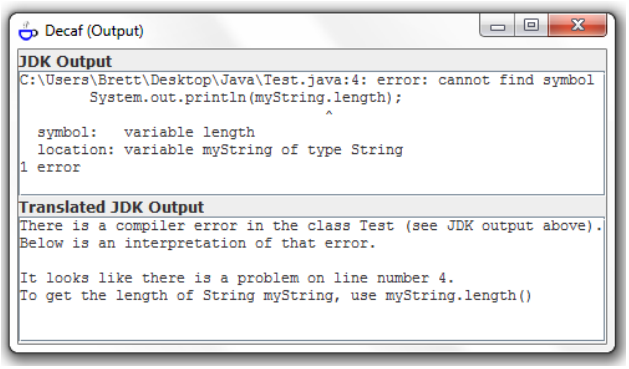

Figure 1: Decaf error window displaying an enhanced compiler error message resulting from incorrectly calling the length() method of the String class which generated a *cannot find symbol* javac error message.

The evaluation study took place at the College of Computing Technology in Dublin, Ireland, in the Semester 1 CS1 module on a BSc in Information Technology program. Anonymous data was logged for a four week period for two groups, each with just over 100 students. The control group received the raw javac compiler error messages and the intervention (enhanced) group received the raw javac messages alongside an enhanced message of the first, should Decaf be able to provide one. The groups took place in consecutive academic years and substantial effort was put into making the learning experience and environment similar for both groups. The same lecturer taught both groups and the

Table 1: Top 10 student errors per compiler error message, control group.

Compiler Error Message	*n*	%
cannot find symbol	4,614	16.0%
‘)’ expected	3,317	11.5%
‘;’ expected	3,076	10.7%
not a statement	2,142	7.4%
illegal start of expression	1,825	6.3%
reached end of file while parsing	1,406	4.9%
illegal start of type	1,316	4.6%
‘else’ without ‘if’	1,141	4.0%
bad operand types for binary operator	1,138	3.9%
<identifier> expected	1,091	3.8%

course content, lab work, schedule, and other environmental and pedagogical factors were kept as similar as possible.

Unlike some previous studies there were no constructs such as supplied method definitions to be completed by the students. Students were working largely on their own time while studying and working on weekly lab exercises based on: data types, basic mathematics, decision making, logical and relational operators, loops, arrays, user input and type casting.

4. RESULTS

4.1 Comparison of control group with other studies

Before comparing the control and enhanced groups we present the 10 most frequent compiler error messages from the control group (who experienced the standard javac error messages) to facilitate comparison with several other studies. Table 1 shows these error messages and the number of times each was generated. These 10 error messages resulted from 73% of 29,019 student errors.

Figure 2 shows that that the control group aligns well with the previous work of [4, 9, 12, 14, 22]. The fact that some studies group or categorize errors does not seem to complicate this type of comparison much, as we are looking at a frequency-based distribution, not necessarily individual errors. That being said, four of the top five errors are common to all six studies, subject to minor differences due to using different Java versions and study-specific groupings such as some grouping all *cannot find symbol* errors together and others reporting *cannot find symbol (variable/method/etc.)* separately. The top three errors in all six studies are: *cannot find symbol*, *‘)’ expected* or *‘bracket expected’*, and *‘;’ expected*. The fourth is mixed: this and two other studies share *not a statement*. The fifth is *illegal start of expression*. Overall the frequency distribution of the top 10 errors (and particularly the top five errors themselves) from this study align well with those from all five previous studies despite some study-specific methodologies and different Java versions.

4.2 Did enhanced compiler error messages reduce the overall number of student errors?

After filtering out data representing students with less than an average of 10 compilation events per week a total of 48,489 student errors remained, 19,628 for the enhanced group (104 compiler IDs) and 28,261 for control (108). These resulted in 74 unique compiler error messages including all 30 that Decaf enhances. A Wilcoxon signed-rank test (two-tail) showed a significantly lower number of student errors per compiler error message for the enhanced group ($Mdn = 18$) compared to the control group ($Mdn = 34$); $Z = -4.29$, $p < 0.001$.

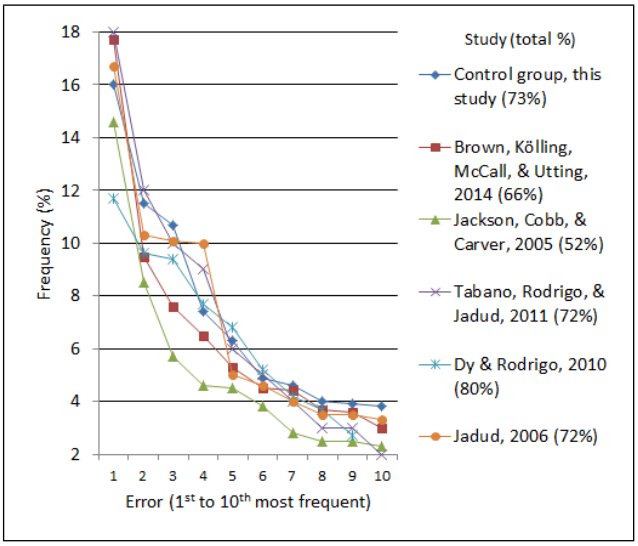

Figure 2: Frequency of the 10 most frequent compiler error messages from six studies.

The top 15 compiler error messages accounted for 86.3% of total errors, 17,144 for the enhanced group and 24,689 for control (102 compiler IDs each). A Wilcoxon signed-rank test (two-tail) showed a significantly lower number of student errors per compiler error message for the enhanced group ($Mdn = 627$) compared to the control group ($Mdn = 1,135$); $Z = -3.17$, $p = 0.002$.

Figure 3 shows a strong linear correlation between groups for all compiler error messages (left) and the 15 most frequent (right). This is somewhat expected; we do not expect enhancing compiler error messages to profoundly alter the errors students make other than possibly reducing their frequencies and the related shifts involved. The slope of both correlations is $> +1$, as the control group (with more error messages overall) is on the y-axes.

Figure 4 shows the number of student errors generating the 15 most frequent compiler error messages for the control and enhanced groups. All but three (*‘else’ without ‘if’*, *illegal start of type*, and *‘.class’ expected*) are enhanced by Decaf. There are fewer errors for the enhanced group for all errors messages except *‘else’ without ‘if’*, for which the enhanced group is marginally greater. The smallest difference occurs with *‘.class’ expected*.

4.3 Did enhanced compiler error messages reduce the number of errors per student?

Table 2 shows the average number of errors per student and average number of unique compiler error messages per student for both groups. There was not a significant difference in (student committed) errors per student between groups when taking all errors into account, however for the

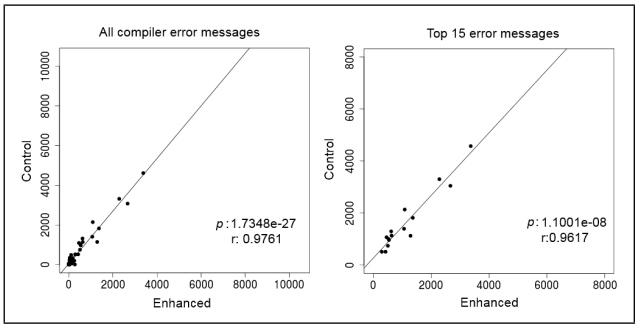

Figure 3: Scatter plots showing linear correlation of number of errors between groups. Each point represents a specific compiler error message.

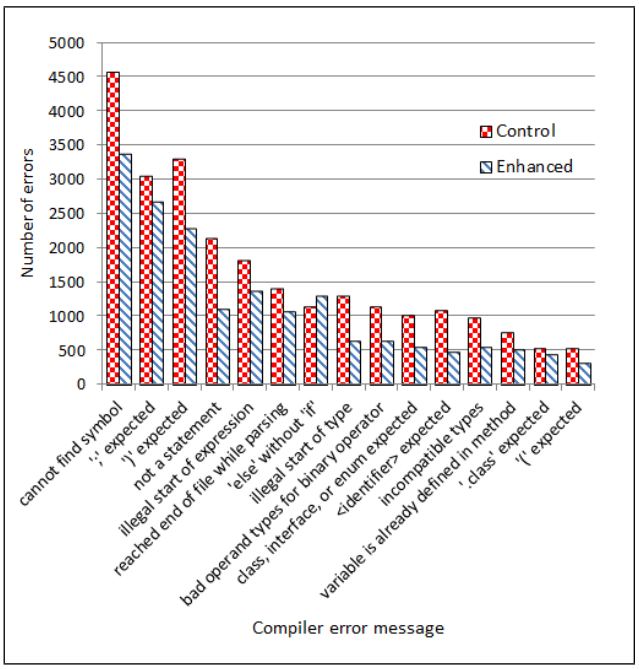

Figure 4: Number of errors for the 15 most frequent compiler error messages.

top 15 compiler error messages (accounting for 86.3% of all errors) a Mann-Whitney U test (two-tail) showed that the number of errors per student was less for the enhanced group ($Mdn = 125$) compared to the control group ($Mdn = 152$); $U = 4,254$, $p = 0.028$.

Investigating individual errors, nine of these 15 had a significant difference between groups, shown in Table 3. Eight of these nine are enhanced by Decaf. The compiler error message that is not (*'.class' expected*) had the smallest difference in number of errors between groups.

4.4 What effect do enhanced compiler error messages have on the number of repeated errors?

To investigate this question we examined the 15 most frequent error messages as they make up 86.3% of all errors. A student is said to have committed a repeated error when two consecutive compilations result in the same error message and originate from a student error on the same line of code. The control group generated 13,849 repeated errors while the enhanced group generated 9,752. Figure 5 shows the number of repeated student errors per compiler error message for the 15 most frequent error messages.

A Wilcoxon signed-rank test (two-tail) showed fewer repeated student errors per compiler error message for the enhanced group ($Mdn = 416$) than for the control group ($Mdn = 742$); $Z = -2.90$, $p = 0.004$. In addition, a Shaprio-Wilk test showed that log transformed data was normal and a paired t-test (two-tail) showed fewer repeated student errors per compiler error message for the enhanced group ($M = 2.68$, $SD = 0.35$) compared to the control group ($M = 2.87$, $SD = 0.30$); $t(14) = 4.73$, $p < 0.001$.

Table 2: Average number of errors per student and average number of unique compiler error messages per student.

Group	Average errors per student	Average unique error messages per student
Control	265	20
Enhanced	188	16

Table 3: The nine of the top 15 compiler error messages with a significant difference between groups (control, c and enhanced, e) sorted by median, *Mdn*.

Compiler Error Message	*Mdn* c, e	*p*
cannot find symbol	35, 26	0.012
not a statement	10, 6	0.003
illegal start of expression	10, 7	0.042
class, interface, or enum expected	6, 3	0.001
<identifier> expected	4, 2	0.038
incompatible types	5, 3	0.005
variable v already defined in method m	4, 3	0.043
'(' expected	2, 1	0.023
'.class' expected	2, 0	0.006

5. DISCUSSION

We have presented evidence that enhanced compiler error messages reduce the number of errors and number of errors per student, as well as the number of repeated errors per compiler error message. This is presented on the foundation that the control group of this study is comparable to findings from several previous studies, for the same language. However we must identify some threats to validity.

First, the groups were separated by a year. It is possible that the enhanced group simply generated fewer error messages. However as the number of students involved was over 200 and the total number of errors collected was nearly 50,000 we find this unlikely, particularly as attempts were made to make all environmental and pedagogical factors as similar as possible. Students learned the same topics in as similar a way as possible, experiencing the same lecturer, material, labs and other controllable factors. Nonetheless some factors could not be controlled such as scheduling differences, room availability and external pressures on students from other modules.

A more technical threat to validity is the fact that a new

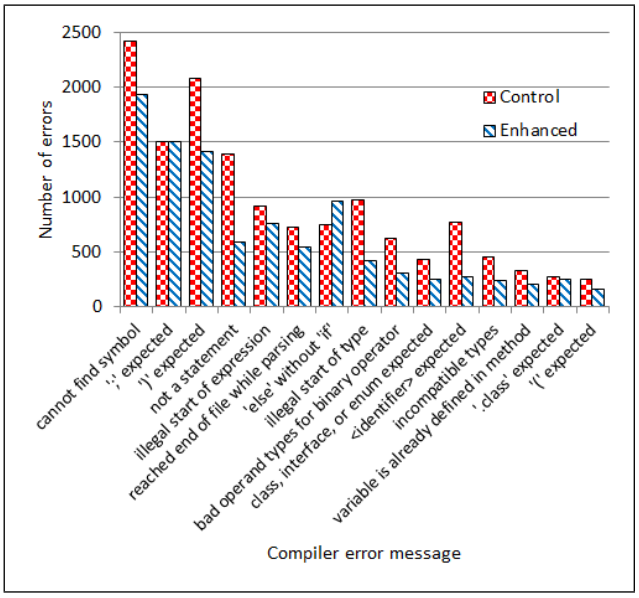

Figure 5: Number of repeated student errors per compiler error message for the 15 most frequent compiler error messages.

anonymous compiler ID is issued when Decaf is reinstalled, perhaps by the same student on the same computer, or by one student on multiple computers. This creates an issue in not having a perfect one to one mapping of compiler IDs to students. It is believed that this did not impact results to a high degree for three reasons. First, the number of compiler IDs was not much above the average attendance and the average number of students submitting lab exercises. Second, filtering data to remove inactive compiler IDs brought the number of compiler IDs even closer to the expected numbers (based on class lists and attendance). Finally and importantly, it was shown that the control group had a similar error profile to other studies. The compiler ID threat will be partially mitigated in future studies with persistent IDs. Related to this, students were encouraged to only use Decaf. A student could choose to use another environment, or use Decaf and another environment, although the lecturer noted very little evidence of this.

The control group was presented with the standard javac error messages (which often involve multiple error messages) making it possible for some students to correct more than one error at a time. Although [8] reported a similar concern, we decided not to interfere with the javac errors at all for the control group. The enhanced group however did only receive one enhanced compiler error message per compilation. It is possible that being presented with only one error reduced confusion for some students, potentially resulting in fewer errors. Future work will investigate this possibility.

We did not anticipate three of the 15 most frequent error messages and therefore Decaf does not enhance them: *'else' without 'if'*, *illegal start of type*, and *'.class' expected*. This however did provide an interesting 'self-contained' control case which spanned both groups. As both groups experienced the same raw Java error messages in these cases, it would be expected that there would be little variation in their frequencies. Indeed for two of these three error messages the frequencies were nearly equivalent. Only *illegal start of type* showed a relatively considerable difference between groups. This could be due to environmental circumstances, or it could be a knock-on effect, where enhancing some compiler error messages helps with other (perhaps related) errors that are not enhanced. See [2] for a discussion on this, which is also a direction for future work.

In [18] McCall and Kölling analyzed actual student errors instead of the diagnostic compiler error messages, noting that analyzing the compiler error messages is imprecise for two reasons. First, a single error (in code) can produce different compiler error messages, depending on context. Second, the same compiler error message can be produced by entirely different errors, again depending on context. In [1] Altadmri and Brown overcame this lack of a one-to-one mapping by parsing the code for offending errors. In the present study the source code is captured at the same time as the error which similarly can overcome this mapping problem in many cases. For instance see Figure 1, where the compiler error message *cannot find symbol* is matched with the source code to produce an accurate error message. Without the source, it would be difficult to provide an accurate message as *cannot find symbol* can occur for a variety of reasons including misspelling an identifier, using an identifier outside its scope, or calling a constructor with an incorrect parameter signature [3]. Nonetheless it should be pointed out that in some cases, matching of the source code and compiler error message does not result in information which can be used to create a usable enhanced error message.

Having presented our results and threats to validity, we do not want to overlook that student surveys revealed a positive experience with the software and enhanced error messages. The enhanced group reported that compiler error messages were less frustrating, and not as much of a barrier to progress compared to the control group. The enhanced group also reported (after spending time with other IDEs) that they found Decaf to make learning to program easier than the control group and that they would recommend Decaf to other learners more than the control group.

6. CONCLUSIONS AND FUTURE WORK

Compiler error messages are frequently inadequate and often a source of discouragement, posing a significant barrier to novices learning to program. Several studies have investigated enhancing compiler error messages but few provide substantial empirical results on their effectiveness. One recent study which did [8] found no effect on the number of non-compiling submissions, the consecutive number of non-compiling submissions, and the number of attempts needed to resolve the three most common kinds of errors.

We designed and implemented an editor providing enhanced compiler error messages and conducted a controlled empirical study with CS1 students learning Java. Comparing the 10 most frequent error messages from the control group representing 79% of 29,019 student errors, we found considerable agreement with other studies. Analyzing nearly 50,000 errors from over 200 students, we found a reduced number of overall errors, errors per student, and repeated errors per compiler error message, for students experiencing enhanced compiler error messages. We also identified eight specific error messages for which enhancement results in a statistically significant reduction in student errors.

Future work involves investigating repeated error mes-

sages in greater detail, exploring the possibilities of knock-on effects between enhanced and non-enhanced errors, and applying the rubric of [16] who provide an empirical process for analyzing the effectiveness of error messages and a rubric for understanding the difficulties students have with them.

7. ACKNOWLEDGEMENTS

The author would like to thank Dr. Catherine Mooney, Dr. Claire McDonnell, Mr. Graham Glanville and Mr. Ricardo Iwashima for their insight and support.

8. REFERENCES

[1] Amjad Altadmri and Neil CC Brown. 37 million compilations: Investigating novice programming mistakes in large-scale student data. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, pages 522–527. ACM, 2015.

[2] B. A. Becker. *An Exploration of the Effects of Enhanced Compiler Error Messages for Computer Programming Novices*. Master's thesis, Dublin Institute of Technology, 2015.

[3] M. M. Ben-Ari. Compile and runtime errors in java. www.weizmann.ac.il/sci-tea/benari/software/others/errors.pdf, 2007. Accessed: 2015-08-23.

[4] N. C. C. Brown, M. Kölling, D. McCall, and I. Utting. Blackbox: A large scale repository of novice programmers' activity. In *Proceedings of the 45th ACM technical symposium on Computer science education*, pages 223–228. ACM, 2014.

[5] N. J. Coull. *SNOOPIE: development of a learning support tool for novice programmers within a conceptual framework*. PhD thesis, University of St Andrews, 2008.

[6] N. J. Coull and I. M. M. Duncan. Emergent requirements for supporting introductory programming. *Innovation in Teaching and Learning in Information and Computer Sciences*, 10(1):78–85, 2011.

[7] P. Denny, A. Luxton-Reilly, E. Tempero, and J. Hendrickx. Codewrite: supporting student-driven practice of java. In *Proceedings of the 42nd ACM technical symposium on Computer science education*, pages 471–476. ACM, 2011.

[8] P. Denny, A.a Luxton-Reilly, and D. Carpenter. Enhancing syntax error messages appears ineffectual. In *Proceedings of the 2014 conference on Innovation & technology in computer science education*, pages 273–278. ACM, 2014.

[9] T. Dy and M. M. Rodrigo. A detector for non-literal java errors. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*, pages 118–122. ACM, 2010.

[10] B. Hartmann, D. MacDougall, J. Brandt, and S. R. Klemmer. What would other programmers do: suggesting solutions to error messages. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1019–1028. ACM, 2010.

[11] M. Hristova, A. Misra, M. Rutter, and R. Mercuri. Identifying and correcting java programming errors for introductory computer science students. *ACM SIGCSE Bulletin*, 35(1):153–156, 2003.

[12] 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. IEEE, 2005.

[13] M. C. Jadud. A first look at novice compilation behaviour using bluej. *Computer Science Education*, 15(1):25–40, 2005.

[14] M. C. Jadud. *An exploration of novice compilation behaviour in BlueJ*. PhD thesis, University of Kent, 2006.

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

[16] G. Marceau, K. Fisler, and S. Krishnamurthi. Measuring the effectiveness of error messages designed for novice programmers. In *Proceedings of the 42nd ACM technical symposium on Computer science education*, pages 499–504. ACM, 2011.

[17] G. Marceau, K. Fisler, and S. Krishnamurthi. Mind your language: on novices' interactions with error messages. In *Proceedings of the 10th SIGPLAN symposium on New ideas, new paradigms, and reflections on programming and software*, pages 3–18. ACM, 2011.

[18] Davin McCall and Michael Kolling. Meaningful categorisation of novice programmer errors. In *Frontiers in Education Conference (FIE), 2014 IEEE*, pages 1–8. IEEE, 2014.

[19] P. C. Rigby and S. Thompson. Study of novice programmers using eclipse and gild. In *Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange*, pages 105–109. ACM, 2005.

[20] T. Schorsch. Cap: an automated self-assessment tool to check pascal programs for syntax, logic and style errors. *ACM SIGCSE Bulletin*, 27(1):168–172, 1995.

[21] A. Stefik and S. Siebert. An empirical investigation into programming language syntax. *ACM Transactions on Computing Education (TOCE)*, 13(4):19, 2013.

[22] E. S. Tabanao, M. M. Rodrigo, and M. C. Jadud. Predicting at-risk novice java programmers through the analysis of online protocols. In *Proceedings of the seventh international workshop on Computing education research*, pages 85–92. ACM, 2011.

[23] V. J. Traver. On compiler error messages: what they say and what they mean. *Advances in Human-Computer Interaction*, 2010, 2010.

[24] Christopher Watson, FrederickW.B. Li, and JamieL. Godwin. Bluefix: Using crowd-sourced feedback to support programming students in error diagnosis and repair. In Elvira Popescu, Qing Li, Ralf Klamma, Howard Leung, and Marcus Specht, editors, *Advances in Web-Based Learning - ICWL 2012*, volume 7558 of *Lecture Notes in Computer Science*, pages 228–239. Springer Berlin Heidelberg, 2012.

[25] R. L. Wexelblat. Maxims for malfeasant designers, or how to design languages to make programming as difficult as possible. In *Proceedings of the 2nd international conference on Software engineering*, pages 331–336. IEEE Computer Society Press, 1976.