To appear in *Computer Science Education*
Vol. 00, No. 00, Month 20XX, 1–27

RESEARCH ARTICLE

Effective Compiler Error Message Enhancement for Novice Programming Students

Brett A. Becker[a*†], Graham Glanville[a], Ricardo Iwashima[a], Claire McDonnell[b], Kyle Goslin[a], and Catherine Mooney[c]

[a] *College of Computing Technology, 30-34 Westmoreland St, Dublin 2, Ireland*; [b] *Dublin Institute of Technology, Aungier St, Dublin 2, Ireland*; [c] *Royal College of Surgeons in Ireland, 123 Stephen's Green, Dublin 2, Ireland*

(*August 19, 2016*)

Programming is an essential skill that many computing students are expected to master. However, programming can be difficult to learn. Successfully interpreting compiler error messages is crucial for correcting errors and progressing towards success in programming. Yet these messages are often difficult to understand and pose a barrier to progress for many novices, with struggling students often exhibiting high frequencies of errors, particularly repeated errors. This paper presents a control/intervention study on the effectiveness of enhancing Java compiler error messages. Results show that the intervention group experienced reductions in the number of overall errors, errors per student, and several repeated error metrics. These results are important as the effectiveness of compiler error message enhancement has been recently debated. Further, generalizing these results should be possible at least in part, as the control group is shown to be comparable to those in several studies using Java and other languages.

Keywords: compiler errors; compiler error enhancement; syntax errors; novice programmers; Java; CS1

1. Introduction

An expected outcome of a computer science student's education is programming skill (McCracken, Almstrum, Diaz, Guzdial, Hagan, Kolikant, Laxer, Thomas, Utting, and Wilusz, 2001) which is also a core competency for employment in several IT industries (Orsini, 2013). However many students find programming difficult and struggle to master the core concepts (Bergin and Reilly, 2005). CS1, the first-year programming course in a degree program, often has high failure rates (Bennedsen and Caspersen, 2007; Porter, Guzdial, McDowell, and Simon, 2013; Yadin, 2011). Further, difficulty with computer programming has been shown to contribute to well-documented dropout rates in computer science programs (Caspersen and Bennedsen, 2007). In many countries including Germany (Schäfer, Holz, Leonhardt, Schroeder, Brauner, and Ziefle, 2013), Ireland (Liston, Frawley, and Patterson, 2016), the United Kingdom (Matthíasdóttir and Geirsson, 2011),

*Corresponding author. Email: brett.becker@ucd.ie
†Now at School of Computer Science, University College Dublin, Belfield, Dublin 4, Ireland

and the United States (Sloan and Troy, 2008), computer science has worryingly high dropout rates—often the highest of all disciplines.

Compiler error messages (CEMs) are one of the most important tools that a language offers its programmers, and for novices their feedback is especially critical (Marceau, Fisler, and Krishnamurthi, 2011a). However CEMs are often cryptic and pose a barrier to success for novice programmers who have been shown in several studies to have trouble interpreting them (Hartmann, MacDougall, Brandt, and Klemmer, 2010; Hristova, Misra, Rutter, and Mercuri, 2003; Kummerfeld and Kay, 2003; Traver, 2010).

This study investigates the effectiveness of enhanced compiler error messages (ECEMs). Although some systems which aim to help novice programmers provide ECEMs as a feature, ECEMs have not often been studied rigorously or in isolation. This is important as links can be drawn between CEMs and performance in programming (Jadud, 2006; Rodrigo, Baker, Jadud, Amarra, Dy, Espejo-Lahoz, Lim, Pascua, Sugay, and Tabanao, 2009; Tabanao, Rodrigo, and Jadud, 2011).

A recent study that is particularly significant to our analysis is (Denny, Luxton-Reilly, and Carpenter, 2014), which also investigated ECEMs in an empirical control/intervention manner but presented evidence that error message enhancement is ineffectual. In contrast our study finds that ECEMs have many positive effects.

In our own practice we have made several observations which motivate this research:

(1) Some students are confounded by compiler error messages and do not directly correlate them with errors in their code.
(2) Some students ask for help on particular CEMs multiple times. It seems that they are not *learning* from CEMs—instead they see them as hindrances, blocking them from completing the task at hand.
(3) CEMs vary in usefulness, clarity and arguably correctness—to a novice they can sometimes *seem* wrong.

This study focuses on Java, one of the most popular programming languages for teaching novices to program (Davies, Polack-Wahl, and Anewalt, 2011; Guo, 2014; Siegfried, Greco, Miceli, and Siegfried, 2012), and one of the most popular languages used in industry (Cass, 2015; TIOBE, 2016). It should be noted that the choice of Java as an introductory programming language is not without critics (Siegfried, Chays, and Herbert, 2008), and that Python has recently grown in popularity as an introductory language (Guzdial, 2011), on some counts overtaking Java (Guo, 2014).

This study utilizes a pedagogic Java editor called Decaf, specifically written for this research. The principal consideration that influenced the design of Decaf was that Java CEMs could, and should, be improved upon. This was partially inspired by the work of Michael Kölling, who said of error messages (Kölling, 1999, pp. 145-146):

> *Good error messages make a big difference in the usability of a system for beginners. Often the wording of a message alone can make all the difference between a student being unable to solve a problem without help from someone else and a student being able to quickly understand and remove a small error. The first student might be delayed for hours or days if help is not immediately available (and even in a class with a tutor it may take several minutes for the tutor to be able to provide the needed help).*

Decaf uses available information (the erroneous line of code and the CEM generated) to construct more specific and helpful ECEMs which are presented to the user, alongside the original CEMs. Where possible, Decaf attempts to utilize information such as method or variable names that are involved in the error. The aim is to help students rectify their errors more effectively, while providing a side-by-side opportunity to learn the actual meanings of the original, often cryptic CEMs.

We chose to develop our own editor because we wanted to provide our students with a simple, novice-friendly environment not based on any particular approach. This ruled out developing a plugin for more complicated IDEs such as Eclipse which has many features extraneous for novices, or for BlueJ which despite being aimed at novices, encourages an objects-first approach (Jadud, 2006).

We compared the following metrics between a control group who used Decaf in *pass-through* mode (with no ECEMs) and an intervention group who used Decaf in *enhanced* mode (with ECEMs):

- Total number of errors in each group
- Number of errors per student, including those generating specific CEMs
- Number of repeated errors, Jadud's error quotient (Jadud, 2005, 2006), and the repeated error density (Becker, 2016b)

The aim of this research is to discover if enhancing compiler error messages is effective in helping students learn to program. We seek to answer the following research questions:

(1) Do enhanced compiler error messages reduce the overall number of errors?
(2) Do enhanced compiler error messages reduce the number of errors per student?
(3) Do enhanced compiler error messages reduce the incidence of repeated errors?
(4) Do students find enhanced compiler error messages beneficial?

This paper is laid out as follows: Section 2 presents a background to, and related work on, CEMs and their enhancement. Section 3 presents our methods and Section 4 presents our results, first from a high-level 'group view', before answering the questions proposed above. We also provide a basis for generalization of our results. We then discuss threats to validity. Section 5 presents our conclusions and future work.

1.1. *Terminology*

This paper uses the term *error* to describe a student-committed error in code that generates a compiler error message (CEM). Our research questions seek to determine the impacts of enhanced CEMs (ECEMs) on student errors. Therefore *number of errors per CEM* is used to describe the number of errors that generated a particular compiler error message. This allows us to discuss errors in isolation (as errors), CEMs in isolation (as CEMs or ECEMs), or to discuss both (as errors per CEM). These distinctions are also useful as there is not necessarily a one to one mapping of errors to CEMs (and vice-versa) in Java (Altadmri and Brown, 2015). As such, it is possible for two different errors to generate the same CEM. Similarly, the same error (in different contexts) may generate different CEMs. Therefore, all that is initially known when a CEM occurs are the details of that particular CEM, and that a single error generated it. Although the CEM often indicates something about the particular nature of the error, this cannot be assumed in all

cases. In Section 3 we discuss that Decaf can at times use source code to gain more information about particular errors when they occur in an effort to provide more useful ECEMs.

2. Background and related work

2.1. *Compiler error messages*

As far back as the 1970s it became evident that in general, compiler error messages were not fit for purpose. Litecky and Davis (1976) investigated CEMs in COBOL, determining that their feedback was not optimal for users, particularly students. As computer science education became more widespread, Pascal secured its position as the first dominant programming language for teaching. Brown (1983) investigated issues with CEMs in Pascal, finding them to be inadequate, and later Chamillard and Hobart Jr (1997) addressed concerns over syntax errors in their transition from Pascal to Ada97. Kummerfeld and Kay (2003) investigated CEMs in C, and gave important insight into the growing concern over poor error messages. Bergin, Agarwal, and Agarwal (2003) pointed out numerous issues with C++ in its use as a teaching language, including CEM difficulties. C++ was a dominant teaching language of its time (taking the lead from Pascal) and eventually replaced by Java.

CEMs play at least two important roles: as a programming tool they should help the user progress towards a working program, and as a pedagogic tool they should help the user understand the problem that led to the error (Marceau *et al.*, 2011a; Marceau, Fisler, and Krishnamurthi, 2011b). However, dealing with CEMs is often a frustrating experience for students (Flowers, Carver, and Jackson, 2004; Hsia, Simpson, Smith, and Cartwright, 2005). Jadud goes as far as stating that compilers are "veritable gold mines for cryptic and confusing error messages" (Jadud, 2006, p. 1), while Traver (2010, p. 4) describes Java errors in particular as "undecipherable". Ben-Ari (2007) noted that educators resorted to writing supplementary material to help explain CEMs (doing so himself), while McCall and Kölling (2014, p. 1) stated: "Compiler error messages ... are still very obviously less helpful than they could be". Disturbingly, these statements are very similar to those made in the 1970s. Even in modern environments designed for novices such as Alice (Moskal and Lurie, 2004) and BlueJ (Kölling, Quig, Patterson, and Rosenberg, 2003), difficulty interpreting compiler error messages has been a student complaint (Hagan and Markham, 2000; Rey, 2009).

CEMs also pose problems for educators, particularly in the context of instructor-led or supported laboratory sessions. Coull (2008) identified that tutors spend large amounts of time solving trivial syntactic problems and that time spent with any individual student may be substantial, extending the time other students must wait for help. In addition, as many students tend to make mistakes similar to those of their peers at similar stages, tutors find themselves solving the same problems for several individuals independently. Denny *et al.* (2014, p. 278) noted: "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".

The frequency of errors, and particularly repeated errors, has been linked to traditional measures of academic success. Jadud (2006) investigated the link between student performance and the error quotient (EQ), a metric influenced heavily by

repeated errors. Although some correlations were found to exist they were weak, and the overall conclusion was that EQ and academic performance are related, but exactly how remained to be seen. However, Rodrigo *et al.* (2009), found that test scores could be predicted with simple measures such as the student's average number of errors, number of pairs of compilations in error, number pairs of compilations with the same error, pairs of compilations with the same edit location, and pairs of compilations with the same error location. This study clearly linked compilation behavior to performance, but the mechanisms at work, and whether this was just a special case, warrant further research. Tabanao *et al.* (2011) successfully predicted midterm exam scores with student error profiles, but was not able to accurately identify at-risk students, a goal which has remained elusive to achieve.

It should be noted that efforts to draw these and similar links are becoming more sophisticated as the amount of data available increases. Ahadi, Lister, Haapala, and Vihavainen (2015) explored machine learning techniques to analyze naturally accumulating programming process data (NAPPD) to identify students in need of assistance. Similar data is analyzed using principal component analysis by Becker and Mooney (2016) and here in Section 4. As the amount of NAPPD becoming available continues to grow due to larger studies such as Blackbox (Brown, Kölling, McCall, and Utting, 2014), such techniques will be increasingly important.

2.2. *Compiler error enhancement*

Although reports on the difficulties posed by compiler error messages have a history of over 40 years, there is not an abundance of research on enhancing them. Schorsch (1995) introduced CAP (Code Analyzer for Pascal), an automated tool to check Pascal programs for syntax, logic and style errors. CAP provided ECEMs designed to inform the student what was wrong, why, and how to fix the cause. These often included sample/example code, and did not shy away from personal touches such as humor, similar to Gauntlet (Flowers *et al.*, 2004) described later. It was reported that the quality of student programs was improved through using CAP.

Hristova *et al.* (2003) introduced Expresso, a pre-compiler which scans Java programs for 20 common errors and provides users with ECEMs where possible. A drawback of Expresso is that error messages may not appear in line-number sequence due to a multiple-pass design. Being presented with errors which are not in line-number sequence is not desirable for at least two reasons. First, novice students often think sequentially—that is line-by-line. Second, students are often taught to tackle the first error message, due to the possibility of cascading errors (Burgess, 1999). These are not true errors in as much as they are immediately resolved when the original error is. To avoid being confused by cascading errors, Ben-Ari (2007, p. 6) advises: "Do not invest any effort in trying to fix multiple error messages! Concentrate on fixing the first error and then recompile". Following this line of thought, the inclusion of the second and subsequent errors is a likely source of confusion and frustration, particularly for novices. This consideration has influenced the design of Decaf, discussed in Section 3.1.

Thompson (2004) focused on an Eclipse plug-in called Gild, specifically for novice Java programmers. Gild was updated to include a feature with "extra error support" which consisted of ECEMs for 51 of 347 possible errors. This work and the Gild editor had many objectives, with the effects of compiler error enhancement making up three of six research questions. In addition, it was an exploratory work with a small number of students—less than 10 for 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 (compilation times was the metric focused on). It was concluded that Gild needed more specific error messages and better coverage of errors most encountered by students.

Flowers *et al.* (2004) introduced a tool called Gauntlet which focused on providing Java ECEMs. After targeting the top 50 novice errors, they focused on nine which they believed to be most common. The authors used Gauntlet for 18 months in a first-year module which included programming. The authors believed that the quality of student work improved, time was saved, and instructor workload was reduced, however no empirical results were presented.

Coull (2008) 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 and enhanced support concurrently. This influenced the design of Decaf discussed in Section 3.1. Only three systems categorized by Coull met this requirement, CAP being one. The same work also presented SNOOPIE (Supporting Novices in an Object Oriented Programming Integrated Environment), using the framework, for learning Java. Although the scope of SNOOPIE was well beyond ECEMs, they were one of the primary facets. It was shown that this support was beneficial to a small group of students, particularly for non-trivial syntactic errors.

Other systems which provided some form of ECEMs, but for which no quantitative evaluations were carried out include Argen (Toomey and Gjengset, 2011), HelpMeOut (Hartmann *et al.*, 2010), a system by Lang (2002), and JJ (Motil and Epstein, nd), discussed by Kelleher and Pausch (2005) and Farragher and Dobson (2000).

All of the studies discussed so far put most focus on addressing the problem (providing ECEMs), but lack empiricism in determining if they make any difference to novices. Denny *et al.* (2014) implemented an enhanced feedback system to users of CodeWrite (Denny, Luxton-Reilly, Tempero, and Hendrickx, 2011), a web-based tool designed to help students complete Java exercises. This paper presented a control/intervention study on the effects of Java ECEMs. The system was used with students attempting exercises which required them to complete the body of a method for which the header was provided. Thus students were not writing code from scratch, and may not have been experiencing the full gamut of CEMs that novices may encounter. Students participated by completing lab exercises for a period of two weeks as part of an accelerated summer course. To evaluate their system, the authors investigated the following.

(1) The number of consecutive non-compiling submissions made while attempting a given exercise.
(2) The total number of non-compiling submissions across all exercises.
(3) The number of attempts needed to resolve the most common kinds of errors.

Their analysis concluded the following, with reference to the three lines of investigation identified above.

(1) There were no significant differences between groups.
(2) 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.
(3) There was no evidence that the enhanced feedback affected the average num-

ber of compiles needed to resolve three common syntax errors (*cannot resolve identifier*, *type mismatch*, and *missing semicolon*).

The authors state several possible reasons for their null results as well as threats to validity, including that the raw compiler feedback shows up to two CEMs, while the enhanced feedback module displays only one in an attempt 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 confuse other students by presenting more than one error to correct. This was a consideration when designing Decaf, discussed in Section 3.1.

In previous work we showed that there was a significant reduction in the number of errors encountered by an intervention group that received ECEMs compared to a control group that did not (Becker, 2016a). We also reported preliminary evidence that the number of repeated errors was significantly reduced. In Becker (2016b) we provided further evidence that the number of repeated errors was reduced in an intervention group receiving ECEMs.

Our previous work simply compared groups of students (control and intervention) without separating out raw CEMs and ECEMs—the control group experienced raw CEMs for all errors, while the intervention group received 30 enhanced CEMs and the remainder raw. This is due to the fact that Decaf does not enhance all possible CEMs. The fact that the intervention group received a mix of enhanced and unenhanced CEMs means that we were not comparing the effects of CEMs and ECEMs as directly as possible. The present study more directly measures how the control and intervention groups interact with raw CEMs and ECEMs by measuring how each group interacts with these separately. That is, we directly measure the effect of CEMs and ECEMs on each group rather than simply comparing the two groups.

3. Methodology

3.1. *Decaf and the enhanced error messages*

This research utilizes a Java editor called Decaf, specifically written for this research by the authors. Decaf uses the raw CEM generated and the erroneous line of code (which may contain relevant information such as identifier names) to construct more specific and helpful ECEMs which are presented to the user, along with the original CEMs. Decaf has two modes, *pass-through* and *enhanced*. In pass-through mode there is no enhancement of the raw javac CEMs. In enhanced mode, enhanced CEMs are presented alongside the original raw CEMs, for 30 selected CEMs. Figure 1 shows a schematic of how Decaf interacts with the system and users. Figures 2 and 3 show screenshots of Decaf in pass-through and enhanced modes respectively.

Decaf's ECEMs were designed by gathering recommendations from several sources including many previous works in Section 2.2. We also utilized the work of Traver (2010) who provided eight principles of good error message design using examples of C++ CEMs to illustrate them. As the syntax of beginner-level Java is C-like, these were translated into practical advice for writing Decaf's ECEMs. Other sources used include (Lang, 2002; Marceau *et al.*, 2011a; Nielsen, 1994; Pane and Myers, 1996). The CEMs to be enhanced were compiled from lists of frequent Java errors from 11 studies presented in Table S1. Details of some individual CEMs including likely causes from Ben-Ari (2007) were also used. Finally, we included

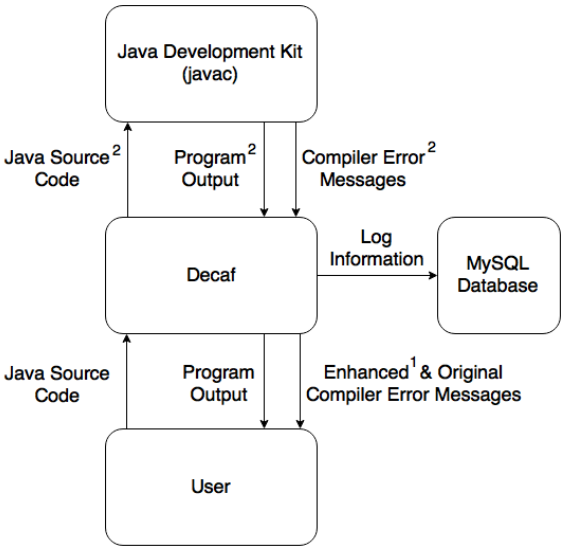

Figure 1. Schematic of Decaf and interactions with user, JDK/javac and database. [1]In pass-through mode, the enhanced error is omitted. [2]Through the runtime environment.

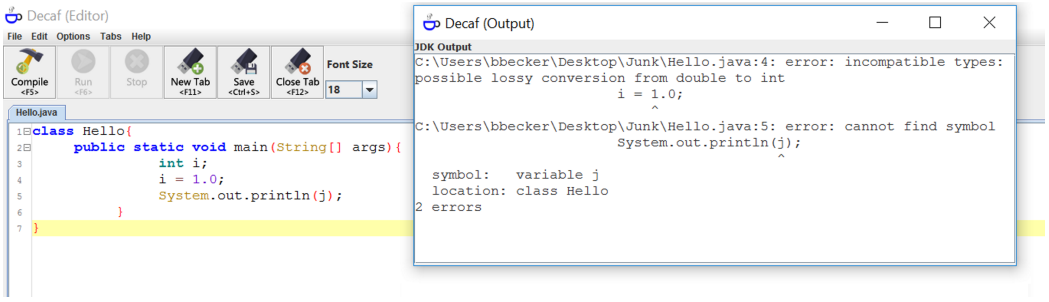

Figure 2. Decaf in pass-through mode, where CEMs are not enhanced, but passed straight on to the user. Here the CEMs *incompatible types* and *cannot find symbol* have occurred.

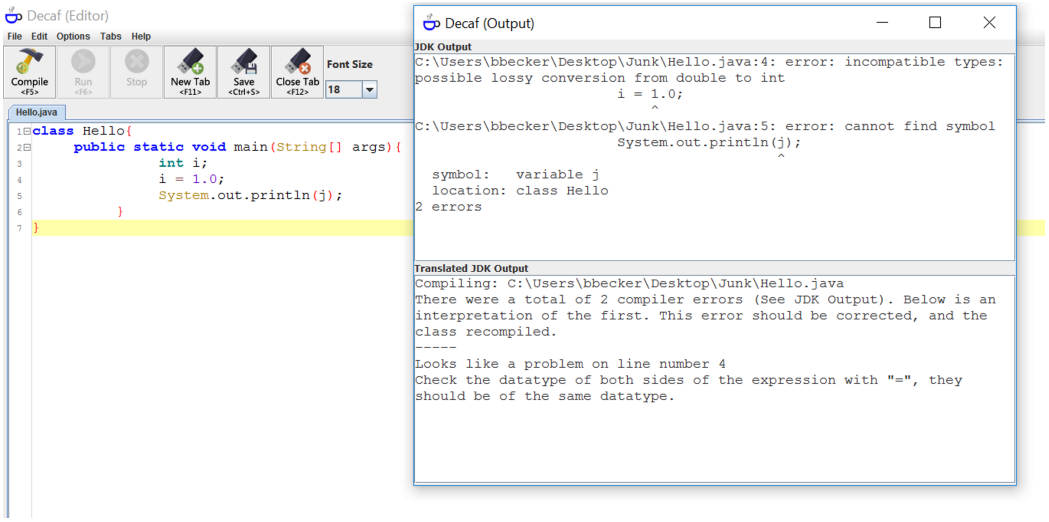

Figure 3. Decaf in enhanced mode. Here the CEMs *incompatible types* and *cannot find symbol* have occurred. The raw CEM(s) are presented at the top, while the first is enhanced and presented at the bottom.

several errors which we have seen occur frequently with beginners in our own practice that were not mentioned in the above:

- *class <class name> is public, should be declared in a file named <class name>.java*
- *'.' expected*
- *illegal character <character>*
- *array required, but <type> found*

Table 1 shows all 30 CEMs enhanced by Decaf. In cases where a particular CEM can be generated by one of several errors, program logic attempts to determine the specific error by analyzing the offending line of user code which may contain useful information such as identifier names. One such example is the CEM *cannot find symbol*. Ben-Ari (2007) notes that this error can be caused by inconsistencies between the declaration of an identifier and its use. A non-exhaustive list of syntax errors resulting in this CEM is:

(1) misspelled identifier (including capital letters used incorrectly)
(2) calling a constructor with an incorrect parameter signature
(3) using an identifier outside its scope

Table 1. CEMs enhanced by Decaf.

CEM number	CEM description
1	'(' expected
2	'(' or '[' expected
3	')' expected
4	'.' expected
7	';' expected
8	'[' expected
9	']' expected
10	'{' expected
11	'}' expected
12	<identifier> expected
16	array required, but *type* found
19	bad operand type *type_name* for unary operator '*operator*'
20	bad operand types for binary operator '*operator*'
24	cannot find symbol
29	class *class_name* is public, should be declared in a file named *class_name*.java
32	class, interface, or enum expected
47	illegal character: '*character*'
51	illegal start of expression
57	incompatible types: *type* cannot be converted to *type*
61	invalid method declaration; return type required
67	missing return statement
73	non-static variable *variable_name* cannot be referenced from a static context
74	not a statement
77	package *package_name* does not exist
78	possible loss of precision
83	'try' without 'catch', 'finally' or resource declarations
86	unclosed comment
89	unexpected type
91	unreported exception *exception_type*; must be caught or declared to be thrown
92	variable *variable_name* is already defined in method *method_name*

Program 1 shows a small Java program containing one syntax error of type (1) above, and Table 2 shows the javac (unenhanced) CEM along with the Decaf (enhanced) ECEM which are generated by this error.

Program 1. Java program with syntax error

```
1 public class Hello{
2    public static void main(string[] args){
3       System.out.println("Hello World");
4    }
5 }
```

Table 2. javac (unenhanced) CEM and Decaf (enhanced) ECEM generated by Program 1

CEM (javac)	C:\Users\Brett\Desktop\junk\Hello.java:2: error: cannot find symbol public static void main(string[ ] args){ ^ symbol: class string location: class Hello 1 error Process terminated ... there were problems.
ECEM (Decaf)	Looks like a problem on line number 2. If "string" refers to a datatype, capitalize the 's'!

This work is primarily concerned with compile-time errors (syntactic and those semantic errors which are caught by javac). However, inspired by Murphy, Kim, Kaiser, and Cannon (2008) who developed a tool which enhanced runtime errors in Java, Decaf also provides enhanced error messages for the following runtime errors, in a manner very similar to the compile-time ECEMs it provides:

- *java.lang.ArrayIndexOutOfBoundsException*
- *java.lang.NullPointerException*
- *java.lang.ArithmeticException: / by zero*
- *java.lang.StringIndexOutOfBoundsException*
- *java.util.InputMismatchException*
- *FileNotFoundException*
- *NumberFormatException*

3.2. *Study design*

Two cohorts of approximately 100 students, separated by one academic year, were included in the study. The students were enrolled in the CS1 module as part of a BSc in Information Technology. The control group had a female to male ratio of 1:2.57 and an average age of 26. The intervention group had a female to male ratio of 1:3 and an average age of 28. Attendance records and numbers of lab submissions indicate similar levels of engagement and motivation for both groups. All students in this study used Java SE 7.

The module was delivered by the same lecturer in year 1 and year 2 and the lecture schedule, content and assessment strategy was as similar as possible for both years. In year 1, control group students used Decaf in pass-through mode, where there is no enhancement of CEMs. In year 2, intervention group students used Decaf in enhanced mode, with the ECEMs presented alongside CEMs (for CEMs where ECEMs are available). We logged data for six weeks but only used weeks 2-5 for analysis. During this period each group was working at steady-state—In week 1 Decaf was being installed and in week 6 students were transitioning to

another editor. Each group had the following data logged, for each CEM generated:

- compiler ID (an anonymous integer representing a unique Decaf installation)
- line of code and class generating CEM
- CEM
- ECEM (intervention group)
- date / time

A recent study particularly relevant to ours provided evidence that enhancing compiler error messages is not effective (Denny *et al.*, 2014). The present study differs from that in the following ways:

(1) We analyze the number of errors generating all CEMs, not just three.
(2) We do not measure the number of non-compiling submissions to an assignment, but the number of errors generated.
(3) We measure errors not only while completing laboratory exercises but all student programming activities such as simply practicing programming.
(4) Our ECEMs do not provide examples of code, only enhanced versions of the raw CEMs.
(5) We do not provide any skeleton or starter code to students.
(6) Decaf only presents one ECEM at a time compared to two.
(7) Our study is over four weeks compared to two.
(8) Our study involves over 200 students compared to 83.

In this study we directly distinguish between two sets of CEMs, the 30 that are enhanced by Decaf and those that are not. We then explore if the control and intervention groups respond differently when they are presented with these. For CEMs enhanced by Decaf the control and intervention groups experience different output. The intervention group, using Decaf in enhanced mode, see the enhanced and raw javac CEMs. The control group, using Decaf in pass-through mode, only see the raw javac CEMs.

Thus for CEMs not enhanced by Decaf, both groups see the same raw CEMs. This provides us with an important subgroup within the intervention group, namely when the intervention group experiences errors generating CEMs not enhanced by Decaf. We hypothesized that there would be no significant difference between the control and intervention groups when looking at these cases for which both groups receive the same raw CEMs. On the other hand, if enhancing CEMs has an effect on student behavior, we would see a significant difference between the two groups when looking at errors generating the 30 enhanced CEMs (due to the intervention group receiving enhanced CEMs and the control group receiving raw CEMs).

4. Results and analysis

We recorded 48,800 errors generating a total of 74 distinct CEMs, including all 30 for which Decaf provides ECEMs (see Table 1). The full list of CEMs generated is shown in Appendix I. Table 3 shows the total number of errors recorded and the number of compiler IDs[1] for both groups. The intervention group logged 32% fewer errors than the control group overall. We noted that a number of compiler IDs generated very few errors, most of these occurring in the first week of the study

[1] an anonymous integer representing a unique Decaf installation

Table 3. Profiles of control and intervention groups. *Filtered for inactive Compiler IDs.

Group	Number of errors	Number of compiler IDs
Control	29,015	122
Intervention	19,785	120
Total	48,800	242
Control*	28,861	108
Intervention*	19,628	104
Total*	48,489	212

period. This is consistent with the lecturer noting that some students reinstalled Decaf early on and the fact that when Decaf is reinstalled a new compiler ID is issued. This is in Section 4.6 as a threat to validity.

We filtered the data removing compiler IDs recording less than an average of 10 errors per week. See Table 3 for the number of recorded errors and compiler IDs after filtering. This strikes a good balance between removing compiler IDs with very low activity and retaining those which are the result of a Decaf reinstall, but generating a representative and useful amount of data. Other studies such as (Jadud, 2006) filtered their data in similar ways.

For data that can be paired we test for significance with Wilcoxon signed-rank tests. For unpaired data we employ Mann-Whitney U tests. In all cases we use two-tail tests and results are considered significant if $p < 0.05$.

In a previous study (Becker and Mooney, 2016) we analyzed the control group data using principal component analysis (PCA). We sought to categorize CEMs by relating them to each other on the basis of how users encounter them. We were interested in seeing if a students who make errors generating certain CEMs frequently would also have a high likelihood of encountering other identifiable CEMs with high frequency. Here, before answering our research questions, we use PCA to gain a view of any differences between our control and intervention groups before looking at more specific measures of behavior.

PCA is a non-parametric method of reducing a complex data set to reveal hidden, simplified dynamics within it (Shlens, 2003). PCA is useful for retaining data that accounts for a high degree of variance, and removing data which does not. PCA takes as input a set of variables (which may be correlated) and converts them into a set of linearly uncorrelated principal components (PCs). These principal components may then reveal relationships between the original variables. The number of PCs is less than or equal to the number of original variables, and are ordered in terms of the fraction of variance each retains. Thus principal component 1 (PC1) contains the highest variance of all PCs, PC2 the second highest, and so on. The PCA was performed with the `ggbiplot`[2] function for the R statistical/graphical programming language.

Figure 4A shows the results of a PCA taking all errors into account. Each data point represents a student and groups are represented by different colors. The ellipses are 68% probability confidence ellipses. It can be seen that the intervention group exhibits less variance in PCs 1 and 2 (those with the most and second most variance) as it has a smaller confidence ellipse (specifically, both axes are shorter). In addition, the control group is more widely distributed with more outliers. These

[2] `http://github.com/vqv/ggbiplot`

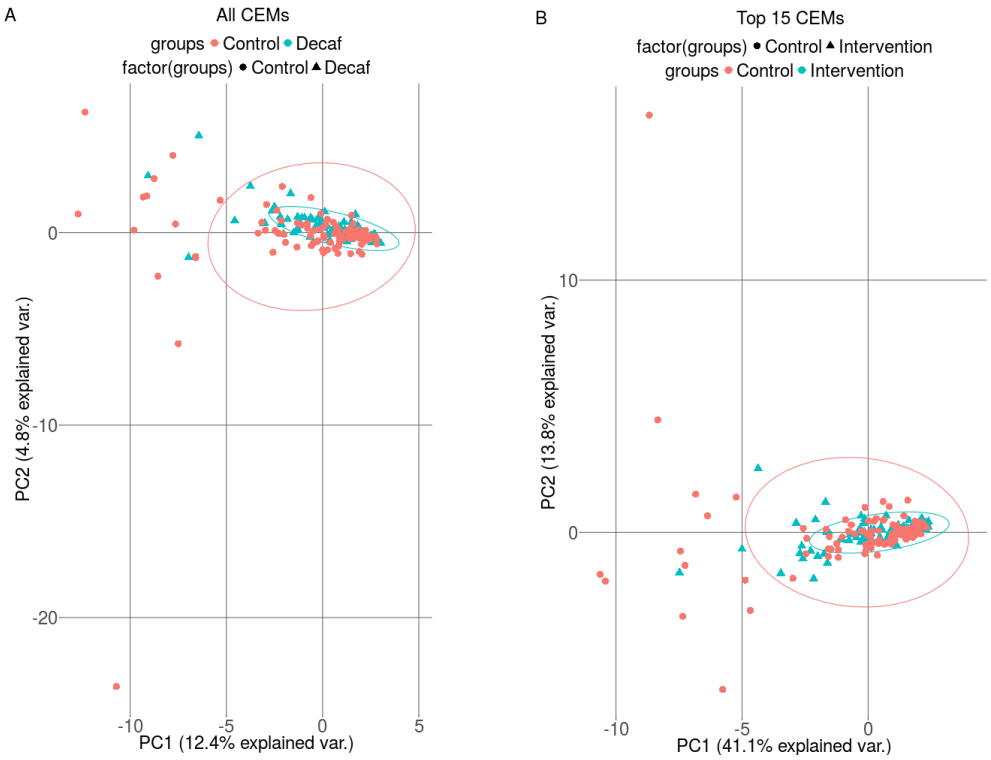

Figure 4. Principal component analysis showing clustering of the control and intervention groups based on their error profiles. (A) All CEMs. (B) Top 15 CEMs.

outliers may represent students that are struggling more with CEMs.

However, outliers in the data can influence the results of PCA, which is another reason that inactive students have been filtered from the data, and here the further step of investigating only errors generating the top 15 CEMs was taken. Again, it is believed that the data remaining is representative and useful, and that any outliers that remain are outlying for valid reasons. A PCA of the reduced (Top 15 CEM) data is shown in Figure 4B. There are three immediate observations to be made:

(1) The group profiles remain very similar.
(2) Individual students do not vary much in terms of relative position within their groups (labels have been removed from the figures for clarity, but they are identifiable with the labels turned on).
(3) The variance of the PCs increase substantially (PC1 from 12.4 to 41.1% and PC2 from 4.8 to 13.8%). Thus for the reduced (15 CEM) data, PCs 1 and 2 account for 54.9% of the variance in each group.

It is important to note when comparing Figure 5A and Figure 5B that the direction (positive/negative) of the PCs and the resulting correlation with variables (CEMs) is arbitrary, so for instance the fact that the outlying student beyond $y = -20$ in Figure 5A, is located beyond $y = 10$ in Figure 5B, does not represent anything of interest in and of itself, as all students have been shifted accordingly between the figures. It is the relative position of students (and the distinction between groups) within each figure that is of interest.

Along with having fewer, less distant outliers, it can be inferred that the intervention students are behaving as a more cohesive, homogeneous group. Since the

PC are linked to the CEMs, we can take the fact that both groups show up distinctly as evidence that on a group level the control and intervention groups are interacting with CEMs differently, and that difference is due to the intervention group experiencing ECEMs.

Having presented the data from an overall perspective demonstrating the differing profiles of the control and intervention groups, we now seek to answer the questions forming the aim of this research presented in Section 1.

4.1. *Do ECEMs reduce the overall number of errors?*

Again, the present study directly distinguishes between raw CEMs and ECEMs, providing a direct measure of the effects of CEM enhancement. This is achieved by comparing control and intervention groups for two sets of CEMs—those that are enhanced by Decaf and those that are not.

Figure 6A shows a strong linear correlation in the number of errors per CEM between the control and intervention groups both in cases where errors do and do not have CEMs enhanced. However, a relatively lower number of errors is seen for the intervention group in the case of enhanced CEMs 6B. It is important to remember that the control group does not experience enhanced CEMs for these errors. This is evidence that enhancing CEMs reduced the number of errors that students make.

The 30 CEMs enhanced by Decaf represent 78.7% of all errors. A Wilcoxon signed-rank test (two-tail) showed that the number of errors was greater for the control group ($Mdn = 229$) than for the intervention group ($Mdn = 189$), $Z =$ -3.19, $p < 0.001$, providing further evidence that ECEMs reduced the number of errors made by the intervention group.

Figure 7 shows the number of errors generating the 10 most frequent CEMs enhanced by Decaf for both groups. It can be seen that the number of errors is lower for the intervention group for all CEMs.

The CEMs not enhanced by Decaf represent 21.3% of all errors, and many of these

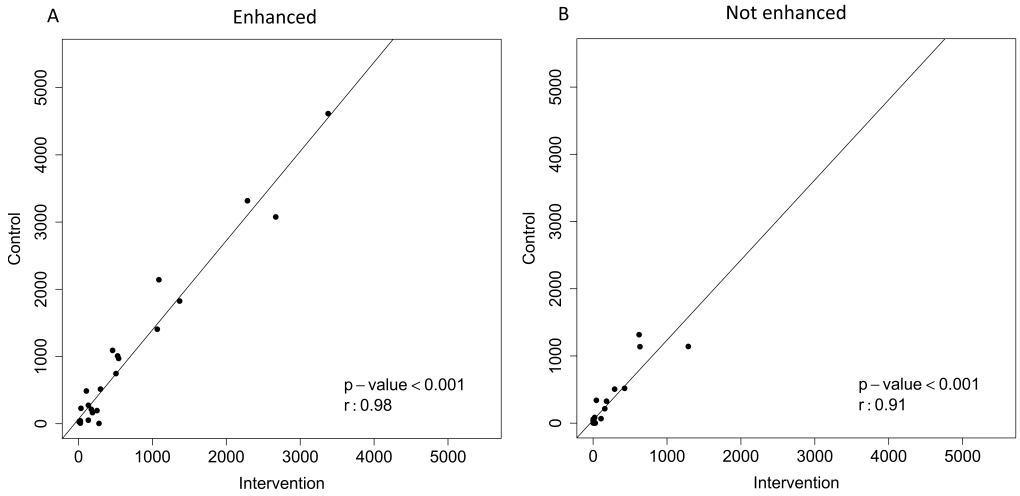

Figure 5. Correlation of errors between the control and intervention groups for errors which do not have CEMs enhanced by Decaf (A) and errors which have CEMs that are enhanced by Decaf (B). Each point represents one CEM. x- and y-axes show number of errors per CEM, for each group.

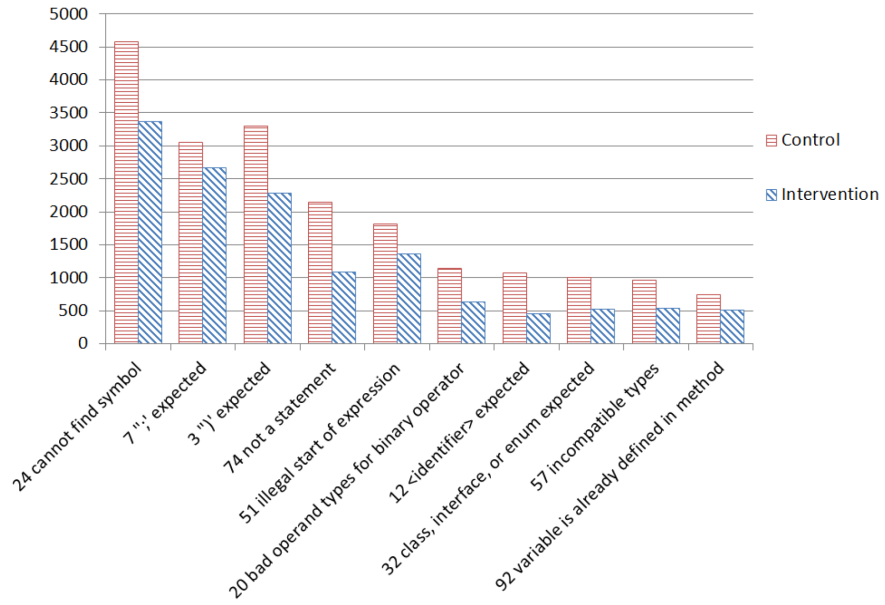

Figure 6. Number of errors per CEM (10 most frequent CEMs enhanced by Decaf).

are infrequent (< 100 errors in either group, accounting for $< 10\%$ of non-enhanced errors). Therefore we selected the 10 most frequent of these CEMs representing over 90% of all errors generating CEMs not enhanced by Decaf. A Wilcoxon signed-rank test (two-tail) did not show a significant difference between the control and intervention groups. These results are in line with our hypothesis in Section 3.2. In the next section we explore the number of errors per student and if the differences presented here are significant in that context.

4.2. *Do ECEMs reduce the number of errors per student?*

Table 4 shows the average number of errors and average number of CEMs per student for each group. For CEMs enhanced by Decaf the intervention group showed a significantly lower number of errors per student for the intervention group ($Mdn = 109$) compared to control ($Mdn = 132$), $U = 4{,}691$, $p = 0.048$. This is evidence that ECEMs reduced the number of errors per student in the intervention group.

Table 4. Average number of errors and CEMs per student.

	Average errors per student	Average CEMs per student
Control	265	20
Intervention	188	16
Overall	228	18

Similar to in Section 4.1, A Mann-Whitney U test (two-tail) showed no significant difference between control and intervention for CEMs not enhanced by Decaf. This is evidence that students in both groups were behaving similarly in the absence of ECEMs.

Becker (2016a) investigated the 15 most frequent CEMs and found that of these,

Table 5. Details of nine CEMs for which enhancement leads to a statistically significant reduction of errors per student.

CEM number	CEM description	Enhanced by Decaf?	Average, Median (Control)	Average, Median (Intervention)	Mann-Whitney U test (two-tail)
32	class, interface, enum[1]	Yes	9.9, 6.0	5.2, 3.0	$U = 3740$, $p = 0.001$
74	not a statement	Yes	21.0, 10.0	10.7, 6.0	$U = 3968$, $p = 0.003$
57	incompatible types[2]	Yes	9.5, 5.0	5.3, 2.5	$U = 4012$, $p = 0.005$
5	'.class' expected	No	5.1, 2.0	4.2, 0.0	$U = 4034$, $p = 0.006$
24	cannot find symbol	Yes	44.9, 35.0	33.0, 25.5	$U = 4148$, $p = 0.012$
1	'(' expected	Yes	5.0, 2.0	2.9, 1.0	$U = 4245$, $p = 0.023$
12	<identifier> expected	Yes	10.5, 4.0	4.5, 2.0	$U = 4330$, $p = 0.038$
51	illegal start of expression	Yes	17.9, 9.5	13.4, 7.0	$U = 4347$, $p = 0.042$
92	variable already defined[3]	Yes	7.3, 4.0	5.0, 3.0	$U = 4351$, $p = 0.043$

[1]class, interface, or enum expected
[2]incompatible types OR incompatible types: *type* cannot be converted to *type*
[3]variable *variable name* is already defined in method *method name*

nine had a statistically significant reduction in the number of errors per student. These are presented in Table 5. Of these nine CEMs, all but one are enhanced by Decaf. This is an important finding. Only one recent study (Denny *et al.*, 2014) investigated individual errors and reported no significant results for the three investigated: *cannot resolve identifier* (CEM 12), *type mismatch* (CEM 57), and *; expected* (CEM 7). Although we did not find a significant difference for CEM 7, we did for CEMs 12 and 57. The fact that an insignificant difference for CEM 7 was found in both studies is somewhat expected, as this unenhanced javac CEM is particularly straightforward.

CEM 5, which has a statistically significant difference, but is not enhanced by Decaf, is *'.class' expected*. There are several possible explanations for this. First, it could be a false positive. Second, there could be a genuine reason that intervention students committed this error with a lower frequency—perhaps a pedagogical difference between the semesters, although significant efforts were made to avoid any. Third, it is not known if helping students by enhancing some CEMs has a 'knock-on' effect of helping with errors generating other CEMs (which are not enhanced).

Program 2 defines an empty method on line 6, which is called by the main method on line 4. If this line is changed from `go(a, b);` to `go(int a, b);`, CEM 5 *'.class' expected* is generated. This is because the type of the method parameter is already known. If `go(int x, int y)` on line 6 is changed to `go(int x, y)`, CEM 12 *<identifier> expected* is generated because no type is given for `y`. This CEM is enhanced by Decaf and does have a significant reduction for the intervention group. Table 6 summarizes this example.

Program 2. Java program exemplifying *<identifier> expected* and *'.class' expected* CEMs.

```
1 public class Test{
2    public static void main(String[] args){
3       int a = 1, b = 2;
4       go(a, b);
5    }
6    public static void go(int x, int y){
7    }
8 }
```

Table 6. Comparison of *<identifier> expected* and *'.class' expected* CEMs. Line numbers and code correspond to Program 2.

Code	Comment	CEM	Enhanced by Decaf?
go(a, b)	Correct (line 4)	-	-
go(int a, b)	Error (line 4)	5 '.class' expected	No
go(int x, int y)	Correct (line 6)	-	-
go(int x, y)	Error (line 6)	12 <identifier> expected	Yes

Given the similarities between how these two errors are generated, it would not be entirely unreasonable to find that helping students with CEM 12 has a knock-on effect of helping them with CEM 5. Both errors can occur due to incorrectly stating (or not stating) the types of method arguments/parameters, in calling (line 4) or defining (line 6) a method. However these CEMs can arise in different situations and a full investigation of this potential knock-on effect is beyond the scope of the present work.

4.3. *Do ECEMs reduce the incidence of repeated errors?*

It is important to note that the number of errors a student commits is not a guaranteed measure the student is struggling, although a high number of errors is certainly an indication that something may be wrong. For repeated errors it is a different situation. Jadud (2006) found that how often errors are repeated is one of the best indicators of how well (or poorly) a student was progressing. Other studies have also focused on the incidence of repeated errors and what they say about student behavior and performance (Ahadi *et al.*, 2015; Watson, Li, and Godwin, 2013).

In this study, a student is said to have committed a repeated error when two consecutive compilations result in the same CEM and originate from an error on the same line of code. Only the first CEM reported by javac is used in calculating repeated errors, and the method is the same for both groups. A repeated error *string* is an occurrence of at least one repeated error—it could be more than one—provided the repeated errors themselves are consecutive with no other events between them. Such a string ends when a different CEM is encountered or a different line of code causes the same CEM (each indicating that the original error was resolved). Figure 7 shows the number of repeated error strings per student (by group) for the top 15 CEMs. A Mann-Whitney U test (two-tail) showed that the number of strings per student was greater for the control group ($Mdn = 37$) than for the intervention group ($Mdn = 27$, $U = 6437$, $p = 0.012$). Note that this data is not paired—each line in Figure 7 represents a succession of all students in each group, ordered in decreasing number of repeated error strings. This shows that more control students had more repeated error strings and were therefore more likely to be struggling.

This reduction in the number of repeated error strings led to the development of a new metric for quantifying repeated errors called the Repeated Error Density (RED) (Becker, 2016b). It was found using the data from this study that enhancing CEMs results in a statistically significant reduction in RED and Jadud's error quotient (Jadud, 2006), providing further evidence that enhancing CEMs reduces the number of repeated errors.

Figure 8 shows the number of repeated errors per CEM for the top 15 CEMs representing 86.3% of all errors. A Wilcoxon signed-rank test (two-tail) showed

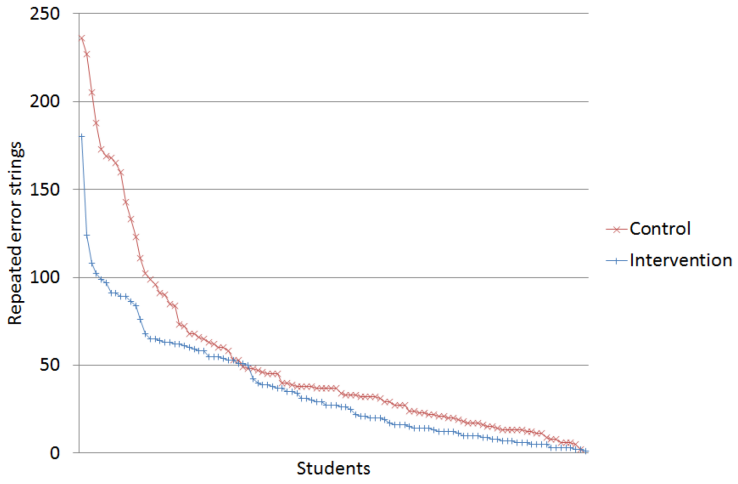

Figure 7. Number of repeated error strings per student (top 15 CEMs).

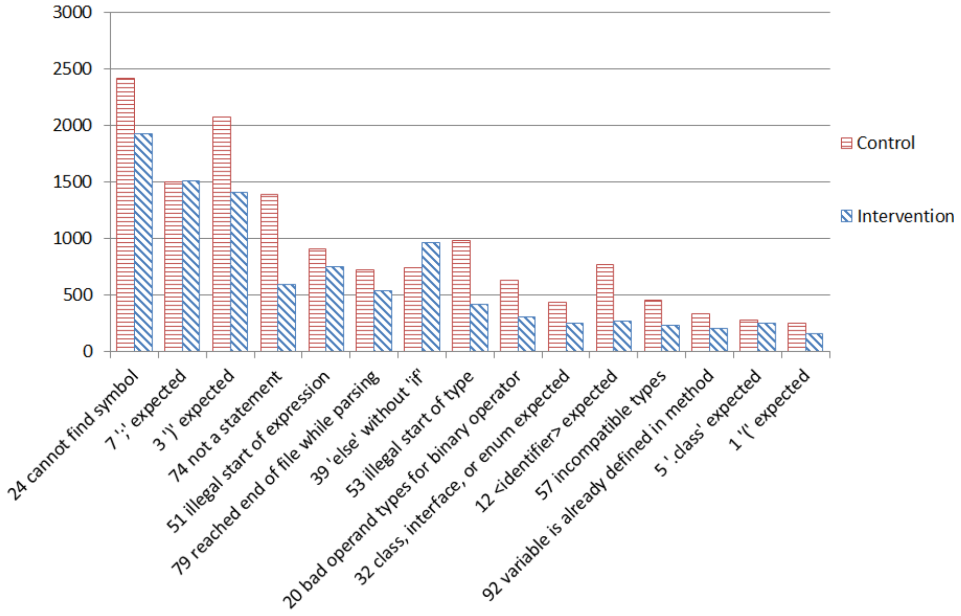

Figure 8. Number of repeated errors per CEM (top 15 CEMs).

that the number of errors was greater for the control group ($Mdn = 742$) than for the intervention group ($Mdn = 416$); $Z = -2.90$, $p = 0.004$. The only CEM with a higher number of repeated errors for the intervention group was CEM 39 *else without if*. This was also the only CEM in the top 15 with a higher number of (overall) errors for the intervention group, and one of the three top-15 CEMs that are not enhanced by Decaf. However in the case of this CEM, this difference was not found to be significant. Additionally, the number of errors for CEM 7 *; expected* was nearly the same for each group. The straightforward nature of this particular unenhanced CEM is a possible explanation for this, as discussed in Section 4.2.

4.4. *Do students find ECEMs beneficial?*

At the end of each semester students were presented with a short optional and anonymous survey relating to their experience using Decaf. The survey was comprised of a number of Likert questions, each with an optional open-ended field asking "Please explain (optional)". An independent-samples t-test (two-tail) was conducted for each Likert question. The response rate was approximately 32% for the intervention group and 20% for the control group. It is interesting to note that for the intervention group, an average of 28% of the optional comments were completed compared to 7% for the control group. This is one indication that the intervention group was more engaged with their learning.

When asked "How much of a barrier to progress do you feel compiler errors are?" students in the intervention group were significantly more likely to report that compiler errors presented less of a barrier to progress than the control group (Figure S1). When asked "How frustrating do you find compiler errors?" the intervention group found compiler errors significantly less frustrating than the control group (Figure S2). This is encouraging, particularly as the intervention students were being presented with both the javac CEMs as well as the Decaf ECEMs, and we had an additional concern that students might find being presented with two versions of the same error messages confusing or frustrating. However this does not appear to have been the case. The full survey results can be found in (Becker, 2015).

4.5. *Basis for generalization*

In this section we analyze the control group and compare it to groups from several other studies on Java and other languages. This is with a view to providing a case for generalizing these results to other groups of students, languages, etc.

The 10 most frequent CEMs recorded in the control group represented 73% of all control errors. These CEMs are similar to those in five other previous studies of CEMs in Java (Table 7). The top 10 CEMs from this study share six CEMs with the top 10 of Brown *et al.* (2014) and Jackson, Cobb, and Carver (2005), five with Tabanao *et al.* (2011) and Dy and Rodrigo (2010), and four with (Jadud, 2006). Despite spanning 10 years and most likely four Java versions, this indicates that the students in the control group of this study are generating very similar errors to students in the other studies. Figure 9 shows the distributions of the top 10 CEMs in all six studies.

Having established that our control group is similar to other studies featuring Java with the motivation of demonstrating that generalization to other Java studies is a potential, we sought evidence for which generalization to other languages is a possibility. Jadud noticed that the top Java errors he collected had a similar distribution to five studies using other languages (Jadud, 2006). Inspired by this analysis, Figure 10 shows the frequency of the nine most frequent errors from this study's control group and those from three languages Jadud reported on: Haskell (Heeren, Leijen, and van IJzendoorn, 2003); FORTRAN (Moulton and Muller, 1967); and COBOL (Litecky and Davis, 1976), as well as the most recent study on Java (Brown *et al.*, 2014).

The errors at each rank are different as the languages are different, with the exception of some of the Java errors (see Table 7), and there is no way of easily

Table 7. Top 10 CEMs from this study (control group) and five other Java studies: A (Brown *et al.*, 2014); B (Jackson *et al.*, 2005); C (Tabanao *et al.*, 2011); D (Dy and Rodrigo, 2010); and E (Jadud, 2006).

Error	% of all errors					
	Control group (this study)	A	B	C	D	E
cannot find symbol*	16	17.7**	14.6	∼18**	18.9**	16.7**
‘)’ expected	11.5	6.5†	3.8	∼10†	9.6†	10.3†
‘;’ expected	10.7	9.5	8.5	∼12	11.7	10
not a statement	7.4	3	2.5			
illegal start of expression	6.3	4.4	5.7	∼5	5.2	5
reached end of file while parsing	4.9					
illegal start of type	4.6					
‘else’ without ‘if’	4					
bad operand types for binary operator	3.9					
<identifier expected>	3.8	3.6	4.5	∼9	3.7	
% Total	73	65.8	51.8	∼69	79.9	71.9
Total errors	28,860	$> 5 \times 10^6$	559,419	24,151	∼14,500	∼70,000

*Some studies broke this CEM down into: unknown variable, unknown method, unknown class, and unknown symbol. As the students in this study had not yet studied methods or classes, it is reasonable to assume that most *cannot find symbol* errors were actually *cannot find symbol - variable* errors. Manually looking at many of these errors in the data supports this.

**_unknown variable_ or _cannot find symbol - variable_ (See * above).

†_bracket expected_.

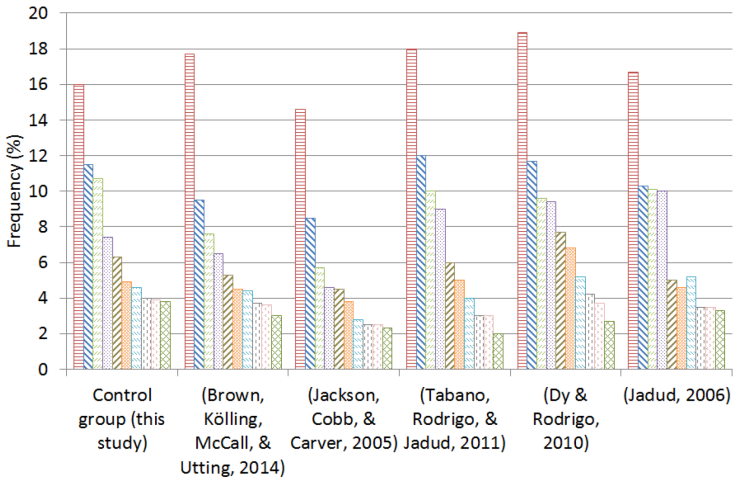

Figure 9. Percent frequency of the 10 most frequent Java CEMs from this study (control group) and five other studies. In each group the left-most bar represents the most frequent CEM and the right-most, the 10th most frequent.

evaluating why this distribution is common across so many languages[3]. Jadud does posit two possible reasons: the programmer and the grammar. If indeed the reason is programmer behavior, it would support the idea that the students in this study are not only similar to those in other studies involving Java, but to those involving many other languages as well. This would be important in generalizing the methods and results of this study. Similarly, if this commonality is due to the grammar of the languages, it could be taken as evidence that results for one language could potentially be generalized to others, with obvious complications involving systematically and reliably generalizing errors in one language to another.

[3]See Jadud (2006), p. 69 for a more thorough discussion.

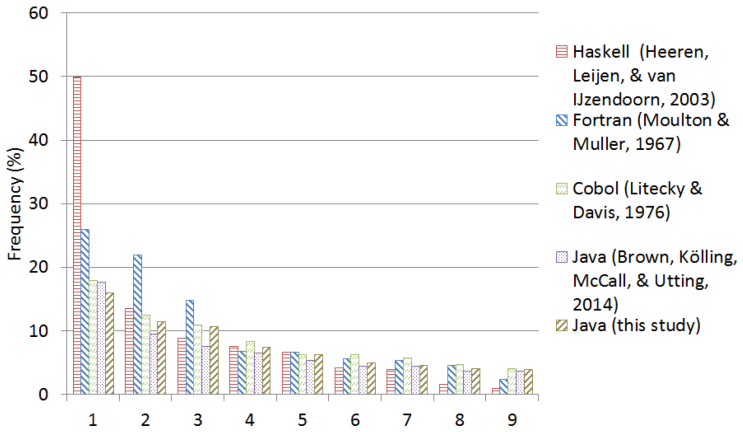

Figure 10. Frequency of the 9 most frequent errors from four different languages.

4.6. *Threats to validity*

Attempts were made to make all environmental and pedagogical factors as similar as possible across the two years of the study. Students learned the same topics in as similar a manner as possible, with the same lecturer, material, labs and environment. 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 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 the results to a large extent for two 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 closer to the expected numbers, and the students in the control group had a similar error profile to students in other studies. Related to the threat just mentioned, students were encouraged to only use Decaf. However, students could choose to use another environment, or use Decaf and another environment concurrently, although the lecturer noted very little evidence of this.

A minor issue is that Decaf does not enhance three of the top 15 CEMs: 39 *'else' without 'if'*, 53 *illegal start of type*, and 5 *'.class' expected*. This however did provide another interesting self-contained control case which spanned both groups. As both groups experienced the same raw Java CEMs in these cases, it would be expected that there would be little variation in their frequencies. Indeed for one of these (CEM 39) Decaf had only a slightly higher frequency, and for CEMs 53 and 5, the frequencies were almost equivalent. For all of the other top 15 CEMs, the number of errors was lower for the control group. Additionally, CEM 5 provided an opportunity to briefly explore the possibility of a knock-on effect of ECEMs (Section 4.2).

The Decaf software was designed before the publication of Denny *et al.* (2014), and shares with their research a threat to validity in that the control students were presented with more than one CEM which may confuse some students and

potentially allow others to correct more than one error simultaneously. This is a direct effect of the design decision of not to interfere with the standard Java CEM presentation in any way for the control group.

In enhanced mode, the original Java CEM is presented unaltered, alongside an ECEM (if one is generated) to the intervention group (however, unlike some other studies, no other information (such as example code) that could lead to validity issues is presented). Being presented with both messages side by side could potentially lead to student confusion because they are being presented with two versions of a single compiler error message. However, the results of survey questions (particularly open-ended responses) did not show any evidence of this.

5. Conclusion

There are many difficulties faced by students learning to program, and few (if any) are as persistent and universally experienced as those in interpreting compiler error messages (CEMs). These difficulties have been documented in the literature for at least four decades and occur with almost all programming languages. These CEMs are extremely important as the student's primary source of information on their work, providing instant feedback intended to help students locate, diagnose and correct their own errors, often made just seconds before. Unfortunately they are often less than helpful. Terse, confusing, too numerous, misleading, and sometimes seemingly wrong, they become sources of frustration and discouragement.

This paper presented the results of an in-depth empirical investigation on the effects of a Java editor called Decaf, specifically written for this research. Decaf features enhanced CEMs (ECEMs) intended to be more understandable and helpful than those provided by the compiler. Only a few systems providing ECEMs exist, and there are even fewer in-depth empirical studies on ECEM effectiveness (Denny *et al.*, 2014).

The aim of this research was to investigate the questions:

(1) Do enhanced compiler error messages reduce the overall number of errors?
(2) Do enhanced compiler error messages reduce the number of errors per student?
(3) Do enhanced compiler error messages reduce the incidence of repeated errors?
(4) Do students find enhanced compiler error messages beneficial?

Two groups were investigated during their semester 1 CS1 module, a control group experiencing standard Java CEMs and an intervention group experiencing ECEMs. Each group consisted of approximately 100 students and together they generated nearly 50,000 errors. The control group was shown to have an error distribution very similar to several other studies on Java and other languages, providing a baseline and grounds for generalization. It was found through several angles of analysis, that the overall number of errors was significantly reduced for the intervention group. Perhaps more importantly, the number of errors per student was reduced, particularly for high frequency errors. Nine CEMs were identified (eight enhanced by Decaf), accounting for 43.2% of all errors, which had a significantly reduced number of errors per student. These errors are amongst the most commonly encountered by students in several other studies.

The number of repeated errors, a key metric in identifying struggling students, was also reduced in addition to the number of repeated error strings. This supports

previous work showing a reduction in the EQ and RED metrics (Becker, 2016b). The data was also analyzed from a group perspective using principal component analysis, finding that both groups had distinct profiles. The intervention group exhibited less variance with a more homogeneous error profile than the control group.

Feedback from students involved in this study showed a positive learning experience with ECEMs. Students that experienced ECEMs reported that compiler errors were not as significant a barrier to progress as those that only experienced the raw CEMs, and students that experienced ECEMs felt that compiler errors were less frustrating than those that only experienced the raw CEMs.

5.1. *Future research directions*

At some point in the future Java may be replaced as the most common language for novice instruction, with Python as a top contender. Although this will bring change, the fact remains that several popular novice teaching languages have come and gone over more than four decades, but the difficulties presented by CEMs have persisted. This makes it seem unlikely that the problems students encounter with CEMs will be alleviated in the short-term by a language change alone. In addition, the manner in which data about the problem is gathered will continue to change. Error detection and aggregation is getting increasingly sophisticated. The Blackbox dataset introduced by Brown *et al.* (2014) contains millions of errors, and has already been used to analyze 37 million of them, from hundreds of thousands of students over at least several hundred institutions (Altadmri and Brown, 2015). However studies like the present one, closer to the students involved than global studies like Blackbox, will remain extremely important in determining how to improve student success in programming.

A solution, if there ever is one, will come first from one of three likely sources. The first is language designers themselves, through languages which by nature are less prone to errors rooted in complex syntax and semantics. The second is compiler designers, who have the possibility of discovering and deciphering error causes differently and presenting more useful CEMs to programmers so they can rectify them more effectively. An example is Eclipse which has a custom Java compiler with its own CEMs, which in some cases are arguably better than those of javac (Ben-Ari, 2007). The third are designers of editors and environments such as Decaf—tools which interpret and address the problems presented by CEMs, most likely through enhancement. Ultimately the solution will probably be a combination of efforts from language, compiler and editor designers in concert. Nonetheless, existing languages already exist and already have their flaws. These languages are immensely popular, running the software that the modern world depends on. In addition, there will always be languages with CEMs more notorious than others, and therefore a likely need for enhanced CEMs.

Directions for future work follow two avenues. The first is further into the data already gathered by applying the rubric of Marceau *et al.* (2011a) designed to identify specific error messages that are problematic for students. Although the present research identified specific error messages, these were based on frequency, not analyzing the actual issues students encountered when they committed particular errors. The second direction is a new and improved editor that will take into account lessons learned here. A web-based editor is envisioned, requiring no download or installation of software. This will provide scope for future study by

including more institutions, greater student diversity, and a greater overall number of participants.

It is perhaps unreasonable to think that enhancing compiler error messages will completely alleviate the problems students have with them. However it has been shown that Decaf reduced student errors, reduced indications of struggling students, and provided a positive learning experience. It is hoped that the results of this work and those to come will help in providing assistance in one of the many hurdles computer programming students face in learning an extremely important skill.

Supplemental material

Table S1 Most frequent Java compiler error messages and errors from eleven studies.

Figure S1 Student feedback: How much of a barrier to progress do you feel compiler errors are?

Figure S2 Student feedback: How frustrating do you find compiler errors?

References

Ahadi, A., Lister, R., Haapala, H., and Vihavainen, A. (2015). Exploring machine learning methods to automatically identify students in need of assistance. In *Proceedings of the Eleventh Annual International Conference on International Computing Education Research*, pages 121–130. ACM.

Altadmri, A. and Brown, N. C. (2015). 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.

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

Becker, B. A. (2016a). An effective approach to enhancing compiler error messages. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*, pages 126–131. ACM.

Becker, B. A. (2016b). A new metric to quantify repeated compiler errors for novice programmers. In *Proceedings of the 21st Annual Conference on Innovation and Technology in Computer Science Education*, pages 296–301. ACM.

Becker, B. A. and Mooney, C. (2016). Categorizing compiler error messages with principal component analysis. In *Proceedings of the 12th China - Europe International Symposium on Software Engineering Education*.

Ben-Ari, M. M. (2007). Compile and runtime errors in Java. `http://introcs.cs.princeton.edu/11cheatsheet/errors.pdf`.

Bennedsen, J. and Caspersen, M. E. (2007). Failure rates in introductory programming. *ACM SIGCSE Bulletin*, **39**(2), 32–36.

Bergin, J., Agarwal, A., and Agarwal, K. (2003). Some deficiencies of C++ in teaching CS1 and CS2. *ACM SIGPlan Notices*, **38**(6), 9–13.

Bergin, S. and Reilly, R. (2005). Programming: factors that influence success. *ACM SIGCSE Bulletin*, **37**(1), 411–415.

Brown, N. C. C., Kölling, M., McCall, D., and Utting, I. (2014). 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.

Brown, P. J. (1983). Error messages: the neglected area of the man/machine interface. *Communications of the ACM*, **26**(4), 246–249.

Burgess, M. (1999). C programming tutorial 4th edition (k&r version). http://markburgess.org/CTutorial/C-Tut-4.02.pdf.

Caspersen, M. E. and Bennedsen, J. (2007). Instructional design of a programming course: a learning theoretic approach. In *Proceedings of the Third International Workshop on Computing Education Research*, pages 111–122. ACM.

Cass, S. (2015). The 2015 top ten programming languages. `http://spectrum.ieee.org/computing/software/the-2015-top-ten-programming-languages`.

Chamillard, A. and Hobart Jr, W. C. (1997). Transitioning to Ada in an introductory course for non-majors. In *Proceedings of the Conference on TRI-Ada'97*, pages 37–40. ACM.

Coull, N. J. (2008). *SNOOPIE: development of a learning support tool for novice programmers within a conceptual framework*. Ph.D. thesis, University of St Andrews.

Davies, S., Polack-Wahl, J. A., and Anewalt, K. (2011). A snapshot of current practices in teaching the introductory programming sequence. In *Proceedings of the 42nd ACM Technical Symposium on Computer Science Education*, pages 625–630. ACM.

Denny, P., Luxton-Reilly, A., Tempero, E., and Hendrickx, J. (2011). Codewrite: supporting student-driven practice of java. In *Proceedings of the 42nd ACM Technical Symposium on Computer Science Education*, pages 471–476. ACM.

Denny, P., Luxton-Reilly, A., and Carpenter, D. (2014). Enhancing syntax error messages appears ineffectual. In *Proceedings of the 19th ACM annual Conference on Innovation and Technology in Computer Science Education*, pages 273–278. ACM.

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

Farragher, L. and Dobson, S. (2000). Java decaffeinated: experiences building a programming language from components. Technical report, Trinity College Dublin, Department of Computer Science.

Flowers, T., Carver, C. A., and Jackson, J. (2004). Empowering students and building confidence in novice programmers through gauntlet. In *Frontiers in Education, 2004. FIE 2004. 34th Annual*, pages T3H–10. IEEE.

Guo, P. (2014). Python is now the most popular introductory teaching language at top us universities. `http://cacm.acm.org/blogs/blog-cacm/`.

Guzdial, M. (2011). Predictions on future CS1 languages. `https://computinged.wordpress.com/2011/01/24/predictions-on-future-cs1-languages/`.

Hagan, D. and Markham, S. (2000). Teaching Java with the BlueJ environment. In *Proceedings of the Australasian Society for Computers in Learning in Tertiary Education Conference*.

Hartmann, B., MacDougall, D., Brandt, J., and Klemmer, S. R. (2010). 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.

Heeren, B., Leijen, D., and van IJzendoorn, A. (2003). Helium, for learning Haskell. In *Proceedings of the 2003 ACM SIGPLAN workshop on Haskell*, pages 62–71. ACM.

Hristova, M., Misra, A., Rutter, M., and Mercuri, R. (2003). Identifying and correcting Java programming errors for introductory computer science students. In *ACM SIGCSE Bulletin*, volume 35, pages 153–156. ACM.

Hsia, J. I., Simpson, E., Smith, D., and Cartwright, R. (2005). Taming Java for the classroom. In *ACM SIGCSE Bulletin*, volume 37, pages 327–331. ACM.

Jackson, J., Cobb, M., and Carver, C. (2005). Identifying top Java errors for novice programmers. In *Proceedings of the 35th Annual Frontiers in Education Conference*, pages T4C24–T4C27. IEEE.

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

Jadud, M. C. (2006). *An exploration of novice compilation behaviour in BlueJ*. Ph.D. thesis, University of Kent.

Kelleher, C. and Pausch, R. (2005). Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Computing Surveys (CSUR)*, **37**(2), 83–137.

Kölling, M. (1999). *The design of an object-oriented environment and language for teaching*. Ph.D. thesis, Department of Computer Science, University of Sydney.

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

Kummerfeld, S. K. and Kay, J. (2003). The neglected battle fields of syntax errors. In *Proceedings of the Fifth Australasian conference on Computing Education*, pages 105–111. Australian Computer Society, Inc.

Lang, B. (2002). Teaching new programmers: a Java tool set as a student teaching aid. In *Proceedings of the inaugural conference on the Principles and Practice of programming, 2002 and Proceedings of the second workshop on Intermediate Representation Engineering for Virtual Machines, 2002*, pages 95–100. National University of Ireland.

Liston, M., Frawley, D., and Patterson, V. (2016). A study of progression in Irish higher education 2012/13 to 2013/14: A report by the Higher Education Authority (Ireland). `http://hea.ie/sites/default/files/hea-progression-irish-higher-education_final.pdf`, ISBN 1-905135-46-7.

Litecky, C. R. and Davis, G. B. (1976). A study of errors, error-proneness, and error diagnosis in COBOL. *Communications of the ACM*, **19**(1), 33–38.

Marceau, G., Fisler, K., and Krishnamurthi, S. (2011a). 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.

Marceau, G., Fisler, K., and Krishnamurthi, S. (2011b). 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.

Matthíasdóttir, Á. and Geirsson, H. J. (2011). The novice problem in computer science. In *Proceedings of the 12th International Conference on Computer Systems and Technologies*, pages 570–576. ACM.

McCall, D. and Kölling, M. (2014). Meaningful categorisation of novice programmer errors. In *2014 IEEE Frontiers in Education Conference (FIE) Proceedings*, pages 1–8. IEEE.

McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y. B.-D., Laxer, C., Thomas, L., Utting, I., and Wilusz, T. (2001). A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. *ACM SIGCSE Bulletin*, **33**(4), 125–180.

Moskal, B. and Lurie, D. (2004). Evaluating the effectiveness of a new instructional approach. In *Proceedings of the 35th ACM Technical Symposium on Computer Science Education*, pages 75–79. ACM.

Motil, J. and Epstein, D. (n.d.). JJ: a language designed for beginners. `http://www.ecs.csun.edu/~jmotil/TeachingWithJJ.pdf`.

Moulton, P. and Muller, M. (1967). Ditran - a compiler emphasizing diagnostics. *Communications of the ACM*, **10**(1), 45–52.

Murphy, C., Kim, E., Kaiser, G., and Cannon, A. (2008). Backstop: a tool for debugging runtime errors. *ACM SIGCSE Bulletin*, **40**(1), 173–177.

Nielsen, J. (1994). Heuristic evaluation. In *Usability inspection methods*, volume 17, pages 25–62.

Orsini, L. (2013). Why programming is the core skill of the 21st century. `http://readwrite.com/2013/05/31/programming-core-skill-21st-century`.

Pane, J. and Myers, B. (1996). Usability issues in the design of novice programming

systems. Technical report, Carnegie Mellon University.

Porter, L., Guzdial, M., McDowell, C., and Simon, B. (2013). Success in introductory programming: What works? *Communications of the ACM*, **56**(8), 34–36.

Rey, J. S. (2009). *From Alice to BlueJ: a transition to Java*. Master's thesis, Robert Gordon University.

Rodrigo, M. M. T., Baker, R. S., Jadud, M. C., Amarra, A. C. M., Dy, T., Espejo-Lahoz, M. B. V., Lim, S. A. L., Pascua, S. A., Sugay, J. O., and Tabanao, E. S. (2009). Affective and behavioral predictors of novice programmer achievement. In *ACM SIGCSE Bulletin*, volume 41, pages 156–160. ACM.

Schäfer, A., Holz, J., Leonhardt, T., Schroeder, U., Brauner, P., and Ziefle, M. (2013). From boring to scoring–a collaborative serious game for learning and practicing mathematical logic for computer science education. *Computer Science Education*, **23**(2), 87–111.

Schorsch, T. (1995). CAP: an automated self-assessment tool to check Pascal programs for syntax, logic and style errors. In *ACM SIGCSE Bulletin*, volume 27, pages 168–172. ACM.

Shlens, J. (2003). A tutorial on principal component analysis: derivation, discussion, and singular value decomposition. `https://www.cs.princeton.edu/picasso/mats/PCA-Tutorial-Intuition_jp.pdf`.

Siegfried, R. M., Chays, D., and Herbert, K. (2008). Will there ever be consensus on CS1? In *FECS*, pages 18–23.

Siegfried, R. M., Greco, D., Miceli, N., and Siegfried, J. (2012). Whatever happened to Richard Reid's list of first programming languages? *Information Systems Education Journal*, **10**(4), 24.

Sloan, R. H. and Troy, P. (2008). Cs 0.5: a better approach to introductory computer science for majors. *ACM SIGCSE Bulletin*, **40**(1), 271–275.

Tabanao, E. S., Rodrigo, M. M. T., and Jadud, M. C. (2011). 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.

Thompson, S. M. (2004). *An exploratory study of novice programming experiences and errors*. Ph.D. thesis, University of Victoria.

TIOBE (2016). TIOBE index for May 2016. `http://www.tiobe.com/tiobe_index`.

Toomey, W. and Gjengset, J. (July, 2011). Arjen: A tool to identify common programming errors. `http://minnie.tuhs.org/Programs/Arjen/`.

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

Watson, C., Li, F. W., and Godwin, J. L. (2013). Predicting performance in an introductory programming course by logging and analyzing student programming behavior. In *Advanced Learning Technologies (ICALT), 2013 IEEE 13th International Conference on*, pages 319–323. IEEE.

Yadin, A. (2011). Reducing the dropout rate in an introductory programming course. *ACM inroads*, **2**(4), 71–76.

Appendix I

All CEMs recorded during study. Gaps in numbering are due to CEMs included in software but not recorded in results.

CEM number	Enhanced by Decaf?	CEM description
1	yes	‘(’ expected
2	yes	‘(’ or ’[’ expected
3	yes	‘)’ expected
4	yes	‘.’ expected
5	no	‘.class’ expected
6	no	: expected
7	yes	‘;’ expected
8	yes	‘[’ expected
9	yes	‘]’ expected
10	yes	‘{’ expected
11	yes	‘}’ expected
12	yes	<identifier> expected
13	no	> expected
14	no	> expected
15	no	array dimension missing
16	yes	array required, but *type* found
19	yes	bad operand type *type_name* for unary operator ’*operator*’
20	yes	bad operand types for binary operator ’*operator*’
22	no	break outside switch or loop
23	no	cannot assign a variable to final variable *variable_name*
24	yes	cannot find symbol
25	no	cannot return a value from method whose result type is void
27	no	‘catch’ without ‘try’
29	yes	class *class_name* is public, should be declared in a file named *class_name*.java
31	no	class expected
32	yes	class, interface, or enum expected
34	no	constructor *constructor_name* in class *class_name* cannot be applied to given types
36	no	double cannot be dereferenced
38	no	duplicate class: *class_name*
39	no	‘else’ without ‘if’
40	no	empty character literal
43	no	exception *exception_name* is never thrown in body of corresponding try statement
46	no	illegal ‘.’
47	yes	illegal character: ‘*character*’
48	no	illegal escape character
49	no	illegal initializer for *type*
50	no	illegal line end in character literal
51	yes	illegal start of expression
52	no	illegal start of statement
53	no	illegal start of type
54	no	illegal static declaration in inner class *class_name*
55	no	illegal underscore
56	no	incomparable types: *type* and *type*
57	yes	incompatible types: *type* cannot be converted to *type*
58	no	inconvertible types
59	no	*type* cannot be dereferenced
60	no	integer number too large: *value*
61	yes	invalid method declaration; return type required
63	no	malformed floating point literal
64	no	method *method_name* in class *class_name* cannot be applied to given types
65	no	method *method_name* is already defined in class *class_name*
66	no	missing method body, or declare abstract
67	yes	missing return statement
69	no	modifier static not allowed here
70	no	no suitable constructor found for *method_name*
71	no	no suitable method found for *method_name*
72	no	non-static method *method_name* cannot be referenced from a static context
73	yes	non-static variable *variable_name* cannot be referenced from a static context
74	yes	not a statement
77	yes	package *package_name* does not exist
78	yes	possible loss of precision
79	no	reached end of file while parsing
81	no	repeated modifier
83	yes	‘try’ without ‘catch’, ‘finally’ or resource declarations
85	no	unclosed character literal
86	yes	unclosed comment
87	no	unclosed string literal
89	yes	unexpected type
90	no	unreachable statement
91	yes	unreported exception *exception type*; must be caught or declared to be thrown
92	yes	variable *variable_name* is already defined in method *method_name*
93	no	variable *variable_name* might not have been initialized
94	no	‘void’ type not allowed here
95	no	while expected