19-8-2016

Effective Compiler Error Message Enhancement for Novice Programming Students

Brett Becker Dr
University College Dublin

Graham Glanville
CCT College Dublin et al.

To appear in *Computer Science Education*
Vol. 00, No. 00, Month 20XX, 1–26

RESEARCH ARTICLE

Effective Compiler Error Message Enhancement for CS1 Students

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

(*May 18, 2016*)

Programming is an essential skill that all computing students must master. However programming can be difficult to learn. Compiler error messages are crucial for correcting errors, but are often difficult to understand and pose a barrier to progress for many novices. High frequencies of errors, particularly repeated errors, have been shown to be indicators of students who are struggling with learning to program. This study involves a custom IDE that enhances Java compiler error messages, intended to be more useful to novices than those supplied by the compiler. The effectiveness of this approach was tested in an empirical control/intervention study of approximately 200 students generating almost 50,000 errors. The design allows for direct comparisons between enhanced and non-enhanced error messages. Results show that the intervention group experienced reductions in the number of overall errors, errors per student, and several repeated error metrics. This work is important for two reasons. First, the effects of error message enhancement have been recently debated in the literature. This study provides substantial evidence that it can be effective. Second, these results should be generalizable at least in part, to other programming languages, students and institutions, as we show that the control group of this study is comparable to several others 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 *et al.*, 2001) which is also a core competency for employment in several IT industries (Orsini, 2013). However learning to program is difficult for many. CS1, the first-year programming course in a degree program, often has high failure rates (Porter, Guzdial, McDowell, and Simon, 2013). Further, difficulty with computer programming has been shown to contribute to well-documented dropout rates in computer science programs (Caspersen and Bennedsen, 2007).

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

[*]Corresponding author. Email: brett.becker@ucd.ie
[†]Now at University College Dublin, Belfield, Dublin 4, Ireland

(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 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 (Tabanao, Rodrigo, and Jadud, 2011).

To the authors' knowledge the only recent study that looked at ECEMs in an empirical control/intervention manner was by Denny, Luxton-Reilly, and Carpenter (2014), who 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 focusses 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 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 inspired by pioneers such as Michael Kölling, the developer of BlueJ, 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. 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 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 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.

2. Background and related work

2.1. *Compiler error messages*

As far back as the 1970s it became evident that many CEMs were not adequate. Litecky and Davis (1976) investigated CEMs in COBOL, determining that their feedback was not optimal for users, particularly for 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 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 importance of poor error messages. Bergin, Agarwal, and Agarwal (2003) pointed out numerous issues with C++ in its use as a teaching language, including CEM issues. 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 still 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, while McCall and Kolling (2014, p. 2589) 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.

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, and the time other students must wait for help is therefore extended. In addition students tend to make mistakes similar to those of their peers at similar stages, and 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, Baker, Jadud, Amarra, Dy, Espejo-Lahoz, Lim, Pascua, Sugay, and Tabanao (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 behaviour to performance, but the mechanisms at work, and whether this was just a special case, warrant further research.

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 analyse naturally accumulating programming process data (NAPD) to identify students in need of assistance. Similar data is analysed using principal component analysis in (Becker and Mooney, 2016) and here in section 4. To the authors' knowledge, these studies are the first to utilize these respective data analysis techniques in the context of NAPD. As the amount of data 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 errors. These often included sample/example code, and did not shy away from personal touches such as humour, similar to Gauntlet (Flowers *et al.*, 2004) described later. It was reported that the quality of student programs was increased through using CAP.

Hristova *et al.* (2003) introduced Expresso, a pre-compiler which scans 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 later.

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 ten 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. 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 provided ECEMs. After targeting the top 50 beginner 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 increased, 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 3.2. Only three systems categorised by Coull met this requirement, CAP being one. Coull also developed SNOOPIE 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 are Argen (Toomey and Gjengset, 2011), HelpMe-Out (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 study was the first recent control/intervention work 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:

(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; and
(3) the number of attempts needed to resolve the most common kinds of errors.

Their analysis concluded that, with reference to the points 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 number of compiles needed to resolve three common syntax errors.

The authors identify 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.

In previous work (Becker, 2016a) 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. 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, and the intervention group received 30 enhanced CEMs while the remainder were raw (as not all CEMs are enhanced by the software). The present study directly measures how control and intervention groups interact with raw CEMs and ECEMs by investigating how each group interacts with these separately.

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 available information (the erroneous line of code and the raw CEM generated) 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. Figure 2 and Figure 3 show screenshots of Decaf in pass-through and enhanced modes respectively.

Table 1 shows all 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 analysing the offending line of user code. One such example is

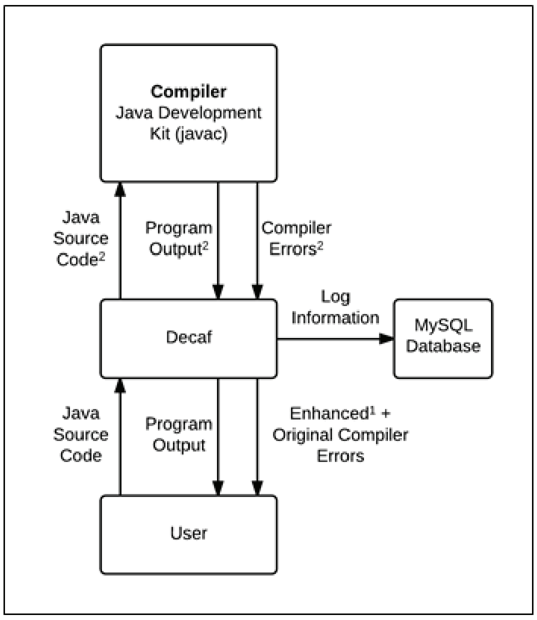


Figure 1. Schematic of Decaf and interactions with user, JDK/javac and database. [1]In pass-through mode, the enhanced error is omitted. [2]Through the runtime environment.

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

Figure 4 shows a small Java program containing one syntax error of type (1) above, followed by the resulting CEM and ECEM.

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.

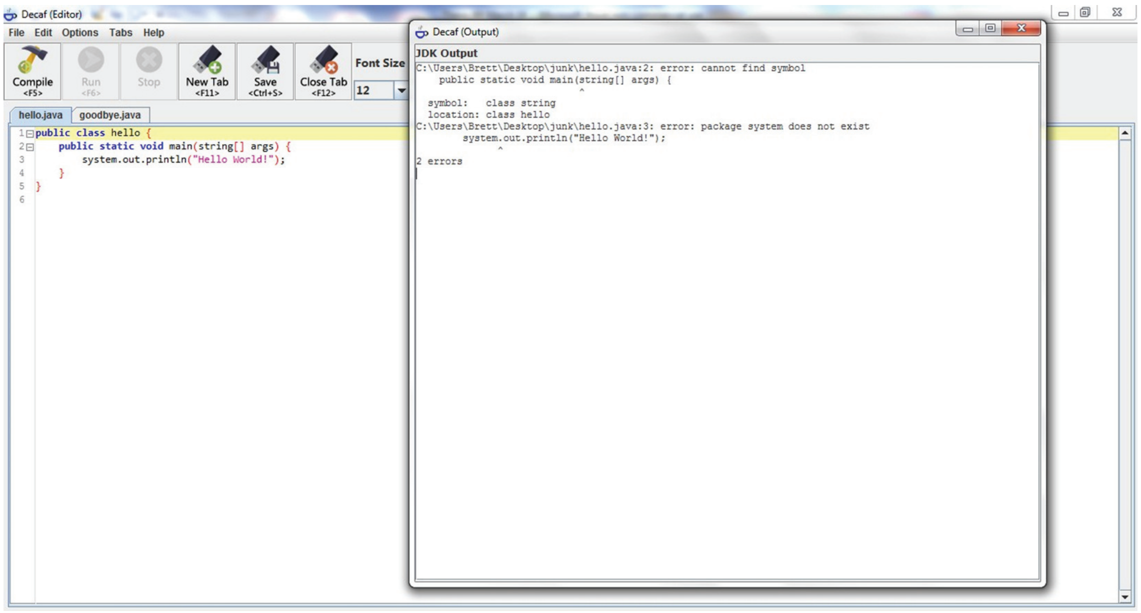
- *java.lang.ArrayIndexOutOfBoundsException*
- *java.lang.NullPointerException*
- *java.lang.ArithmeticException: / by zero*
- *java.lang.StringIndexOutOfBoundsException*
- *java.util.InputMismatchException*
- *FileNotFoundException*
- *NumberFormatException*

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. There was no statistically significant difference

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 detailed in Table S1. Details of some individual CEMs including likely causes from Ben-Ari (2007) were also used. Finally, we included 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>*
- *reached end of file while parsing*
- *unclosed character literal*
- *unreachable statement*
- *array required, but <type> found*



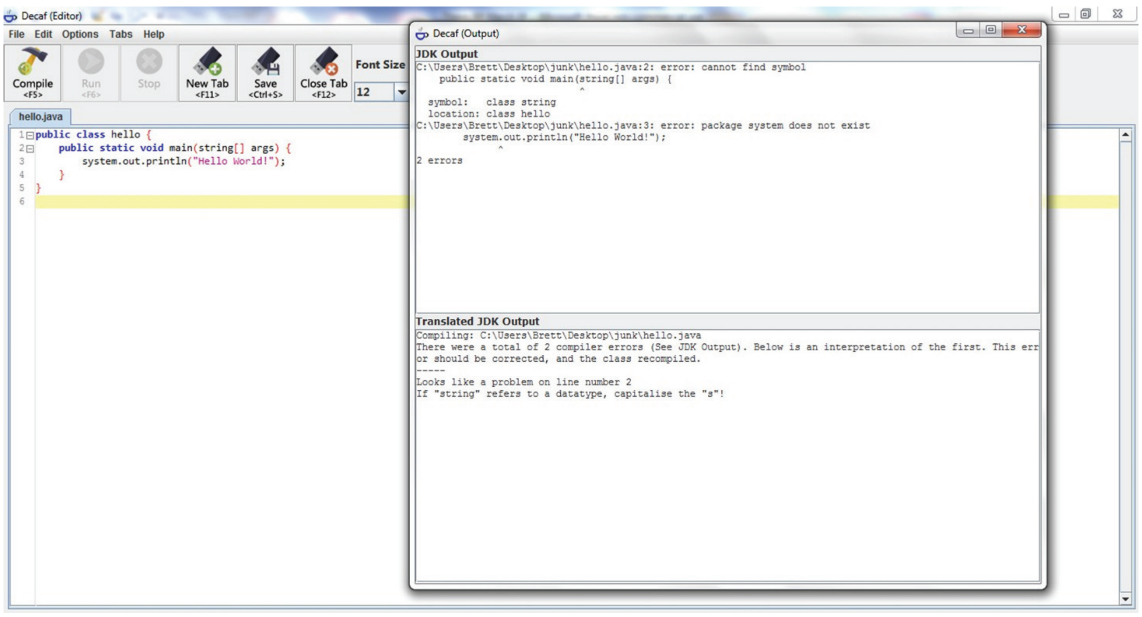
```
public class hello {
    public static void main(string[] args) {
        system.out.println("Hello World!");
    }
}
```

```
C:\Users\Brett\Desktop\junk\hello.java:2: error: cannot find symbol
    public static void main(string[] args) {
                            ^
  symbol:   class string
  location: class hello
C:\Users\Brett\Desktop\junk\hello.java:3: error: package system does not exist
        system.out.println("Hello World!");
              ^
2 errors
```

Figure 2. Decaf seen in 'pass-through' mode, where CEMs are not enhanced, but passed straight on to the user. Here two CEMs have occurred, *cannot find symbol* and *package system does not exist*.

between the groups in terms of age or sex. 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



```
public class hello {
    public static void main(string[] args) {
        system.out.println("Hello World!");
    }
}
```

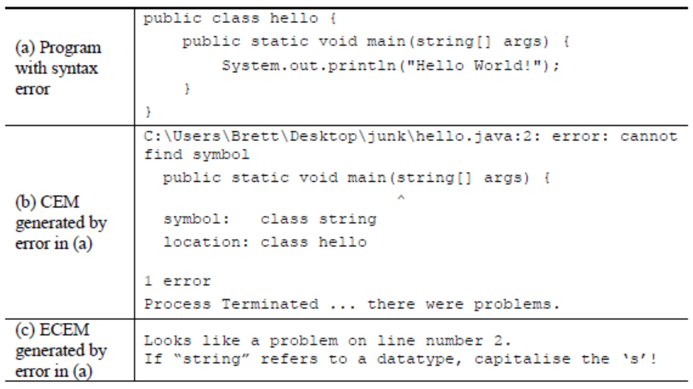
```
C:\Users\Brett\Desktop\junk\hello.java:2: error: cannot find symbol
    public static void main(string[] args) {
                            ^
  symbol:   class string
  location: class hello
C:\Users\Brett\Desktop\junk\hello.java:3: error: package system does not exist
        system.out.println("Hello World!");
              ^
2 errors
```

```
Compiling: C:\Users\Brett\Desktop\junk\hello.java
There were a total of 2 compiler errors (See JDK Output). Below is an interpretation of the first. This err
or should be corrected, and the class recompiled.
-----
Looks like a problem on line number 2
If "string" refers to a datatype, capitalise the "s"!
```

Figure 3. Decaf in enhanced mode. Two raw CEMs are presented at the top, while the first is enhanced and presented below.

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*



(a) Program with syntax error

```
public class hello {
    public static void main(string[] args) {
        System.out.println("Hello World!");
    }
}
```

(b) CEM generated by error in (a)

```
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.
```

(c) ECEM generated by error in (a)

```
Looks like a problem on line number 2.
If "string" refers to a datatype, capitalise the 's'!
```

Figure 4. Java program with one syntax error (a), the resulting javac (unenhanced) CEM (b), and Decaf's (enhanced) ECEM (c).

weeks 2-5 for analysis so that we were only analysing data while each group was working at steady-state since during week 1 Decaf was being installed and during week 6 students were transitioning to another editor. Each group had the following data logged, for each CEM generated:

- Compiler ID (anonymous)
- Line of code and class generating CEM
- CEM
- ECEM (for intervention group)
- Date / time

Denny *et al.* (2014) provided the only other recent empirical study investigating ECEMs, providing evidence that enhancing compiler error messages is not effective. The present study differs from their study in the following ways:

(1) We analyse 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 (and values of repeated error metrics) generated while completing laboratory exercises as well as when practicing programming.
(3) Our ECEMs do not provide examples of code, only enhanced versions of the raw CEMs.
(4) We do not provide any code to students.
(5) Decaf only presents one ECEM at a time compared to two.
(6) Our study is over four weeks compared to two.
(7) 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. For CEMs not enhanced by Decaf the control and intervention groups both only see the raw javac CEMs.

This provides us with an important internal control group – the intervention

Table 2. Profiles of control and intervention groups.

Group	Number of errors	Number of compiler IDs
Control	29,015	122
Intervention	19,785	120
Total	48,800	242

Table 3. Group profiles filtered for inactive students.

Group	Number of errors	Number of compiler IDs
Control	28,861	108
Intervention	19,628	104
Total	48,489	212

group, when they experience errors that are not enhanced by Decaf. We hypothesized that there would be no significant difference between the control and intervention groups when looking at errors generating the unenhanced CEMs. On the other hand, if enhancing CEMs has an effect on student learning, we would see a significant difference between the two groups when looking at errors generating the 30 enhanced CEMs.

4. Results and analysis

We recorded 48,800 errors, generating 74 distinct CEMs, including all 30 for which Decaf provides ECEMs. The full list of CEMs generated is shown in Table S2. Table 2 shows the total number of errors recorded and the number of compiler IDs for both groups. The intervention group logged 32% fewer errors overall. We noted that a number of compiler IDs generated very few errors, most of these occurring in the first week of the study 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 discussed later as a threat to validity.

We filtered the data removing Compiler IDs recording less than an average of ten errors per week (Table 3). 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 which generated a representative and useful amount of data. Other studies such as Jadud (2006), have 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 $p < 0.05$.

Figure 5 shows the 15 most frequently encountered CEMs for both the control and intervention groups. There is a minor shift in rank for some CEMs: CEM 7 moves from 12% in control to 16% in intervention, CEM 39 from 5% to 8%, and CEM 74 from 9% to 6%, however the overall the pattern is similar. This is to be expected – we do not expect Decaf to fundamentally the nature of the errors that students make, other than possibly reducing their frequencies.

In Becker and Mooney (2016) we analysed 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

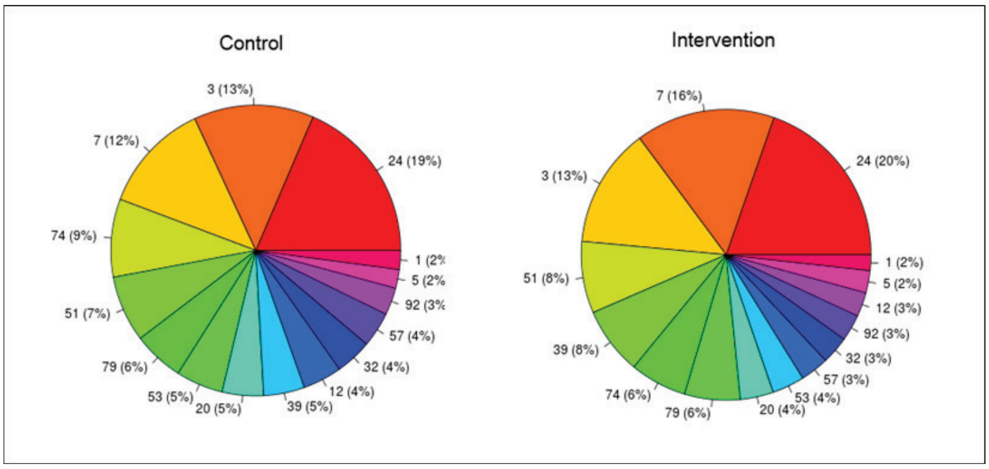


Figure 5. Frequency of the top 15 CEMs enhanced by Decaf, for control and intervention groups.

student who makes errors generating a certain CEM often would also have a high likelihood of making other identifiable errors with high frequency.

PCA is a non-parametric method of reducing a complex data set to reveal hidden, simplified dynamics within it (Shlens, 2003). PCA takes as input a set of variables (which may be correlated) and converts them into a set of linearly uncorrelated principal components (PCs). The number of PCs is less than or equal to the number of original variables. PCA is useful for retaining data that accounts for a high degree of variance, and removing data which does not. The PCA was performed with the `ggbiplot`[1] function for the R statistical/graphical programming language.

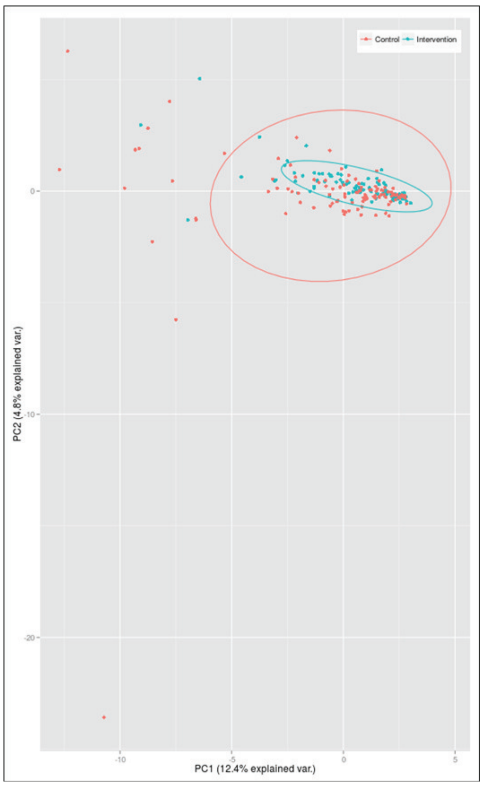
Figure 6(a) shows the results of a PCA taking all errors into account. Each data point represents a student and groups are represented by different colours. The ellipses are 68% probability confidence ellipses. It can be seen that the intervention group exhibits less variance in principal components 1 and 2 (those with the greatest variance) as it has smaller confidence ellipses. In addition, the control group is more widely distributed with more outliers. These 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 is 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 (15 CEM) data is shown in Figure 6(b) . There are three immediate observations to be made:

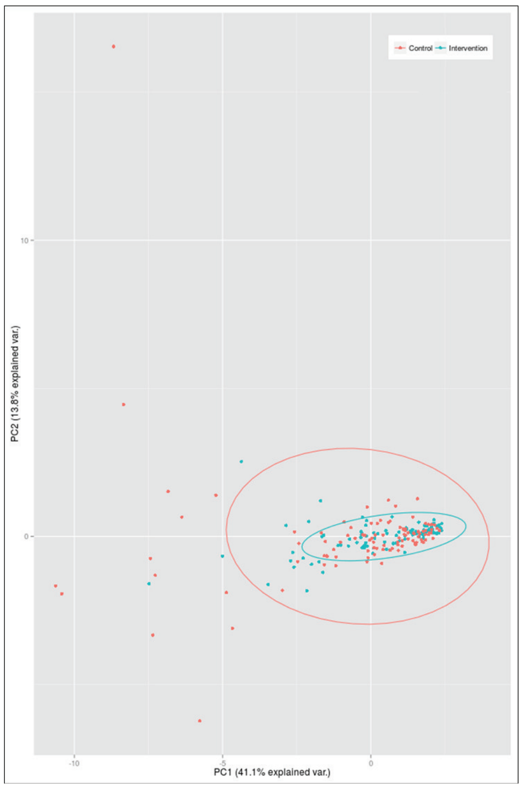
(1) The group profiles remain very similar.
(2) Individual students do not vary much (labels have been removed from the figures for clarity, but they are identifiable with the labels turned on).
(3) The variance of the PCs increase substantially (PC1 from 12.4 to 41.1% and PC2 from 4.8 to 13.8%). Thus for the reduced (15 CEM) data, PCs 1 and 2 account for 53.5% of the variance in each group.

It is important to note when comparing Figure 6(a) and Figure 6(b) that the direction (positive/negative) of the PCs and the resulting correlation with variables

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



(a) All CEMs.



(b) Top 15 CEMs.

Figure 6. Principal component analysis showing clustering of the control and intervention groups based on their error profiles.

(CEMs) is arbitrary, so for instance the fact that the outlying student beyond (-10, -20) in Figure 6(a), is located beyond $y = 15$ in Figure 6(b), 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 CEMs are linked to the PCs, 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 not.

Figure 7(a) shows a strong linear correlation in the number of errors per CEM

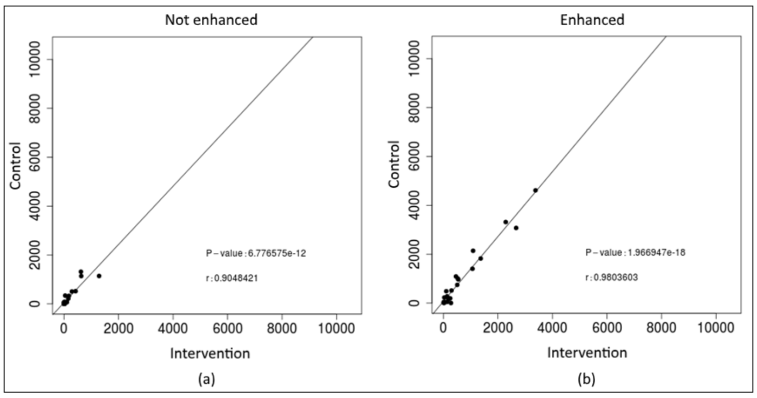


Figure 7. Correlation of errors between the control and intervention groups for errors which (a) do not have CEMs enhanced by Decaf and (b) errors which have CEMs that are enhanced by Decaf.

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 7(b). 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, $p < 0.001$. This is evidence that ECEMs reduced the number of errors made by the intervention group.

Figure 8 shows the number of errors generating the ten 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 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 now 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.

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.

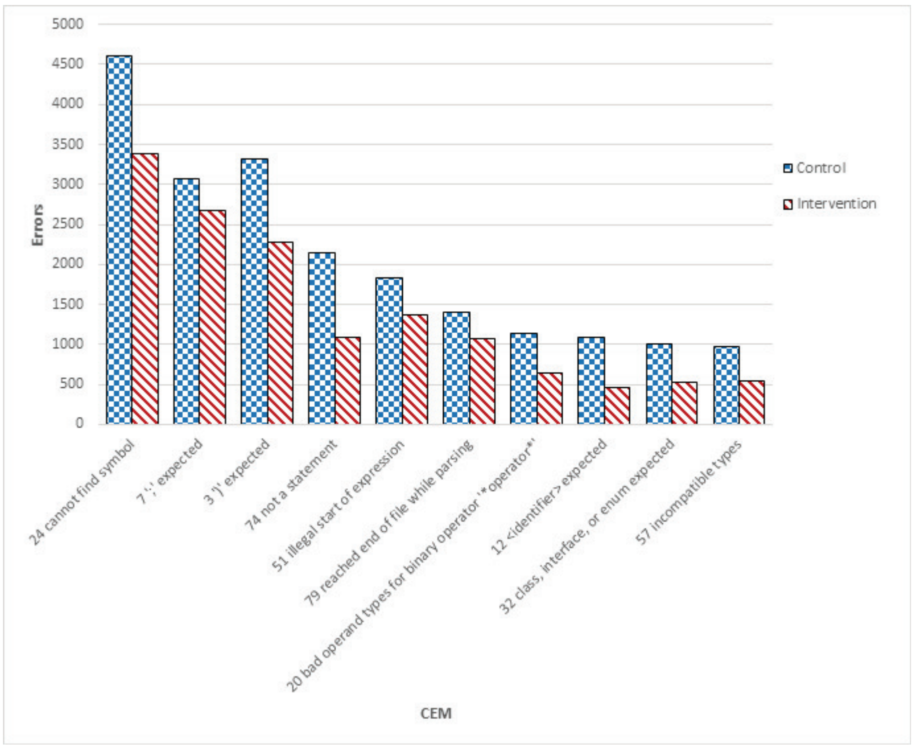


Figure 8. Number of errors per CEM (top 10 CEMs enhanced by Decaf).

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

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

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).

Figure 9 shows a Java program that defines an empty method (lines 6-7), which is called by the main method (line 4). If line 4 is changed to `go(int a, b)`, CEM 5

Table 5. Details of eight 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, or enum expected	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 OR incompatible types: *type* cannot be converted to *type*	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 *variable name* is already defined in method *method name*	Yes	7.3, 4.0	5.0, 3.0	$U = 4351$, $p = 0.043$

```
1 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 }
```

Figure 9. Java program exemplifying *<identifier> expected* and *'.class' expected* CEMs.

Table 6. Comparison of *<identifier> expected* and *'.class' expected* CEMs.

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

'.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.

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.

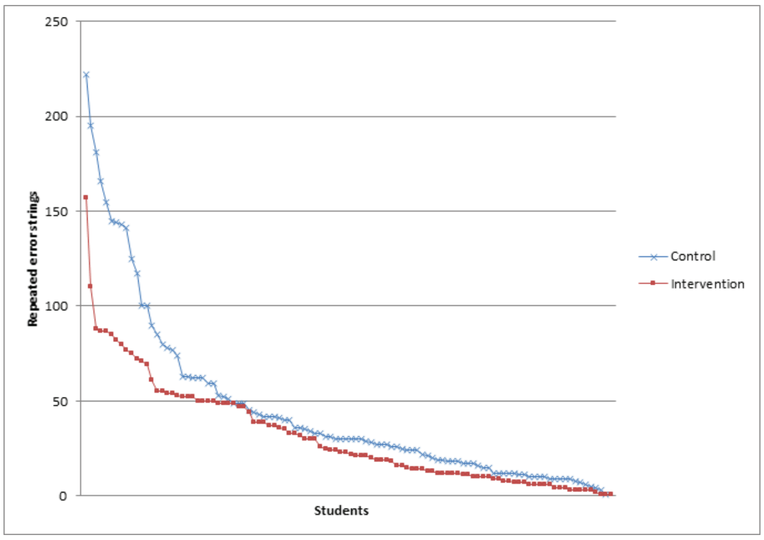


Figure 10. Number of repeated error strings per student (top 15 CEMs).

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. A repeated error *string* is an occurrence of at least one repeated error – it could be more than one repeated error, 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 solved). Figure 10 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 10 represents a succession of all students in each group, ordered in decreasing number of repeated error strings. This shows that more control students made more consecutive repeated errors 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, providing further evidence that enhancing CEMs reduces the number of repeated errors.

Figure 11 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 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.

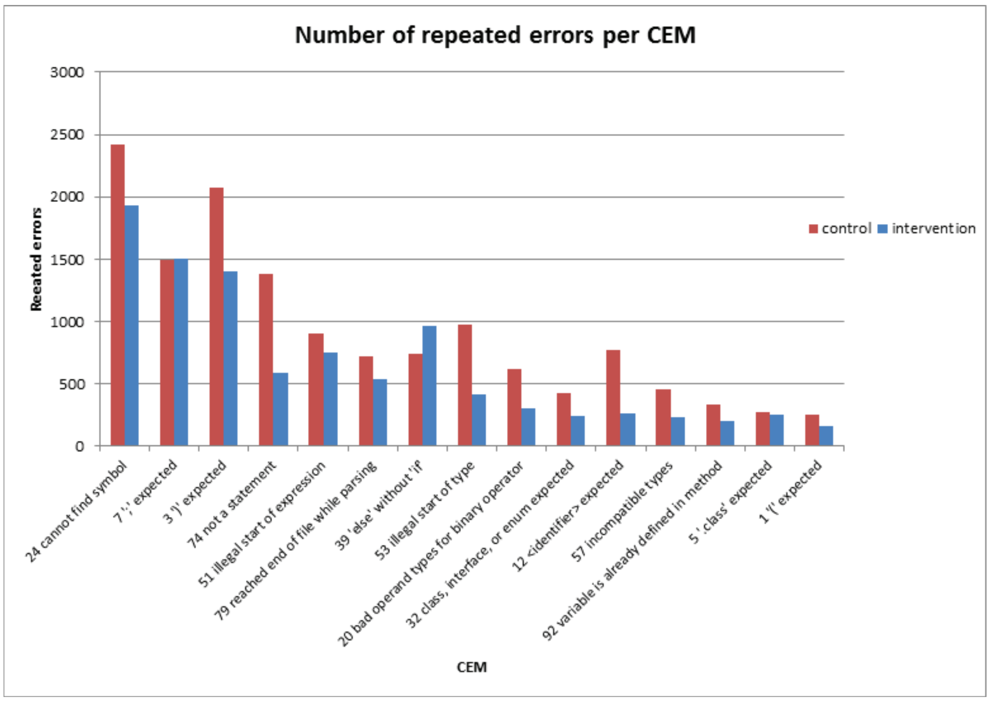


Figure 11. Number of repeated errors per CEM (top 15 CEMs).

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 a concern that students might find being presented with two 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 analyse 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.

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	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.*

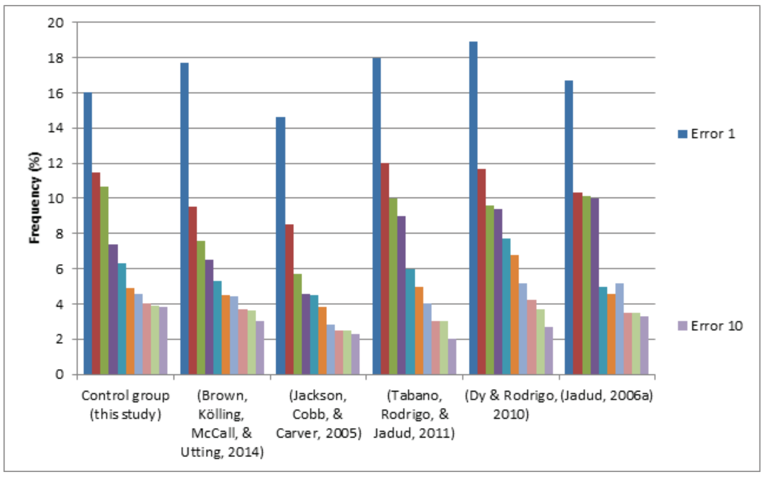


Figure 12. Frequency of the ten most frequent Java CEMs from this study (control group) and five other studies.

The ten 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 ten CEMs from this study share six CEMs with the top ten 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 ten years and most likely four Java versions, this indicates that the students in the control group in this study are generating very similar errors to students in the other studies. Figure 12 shows the distributions of the top ten 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 his

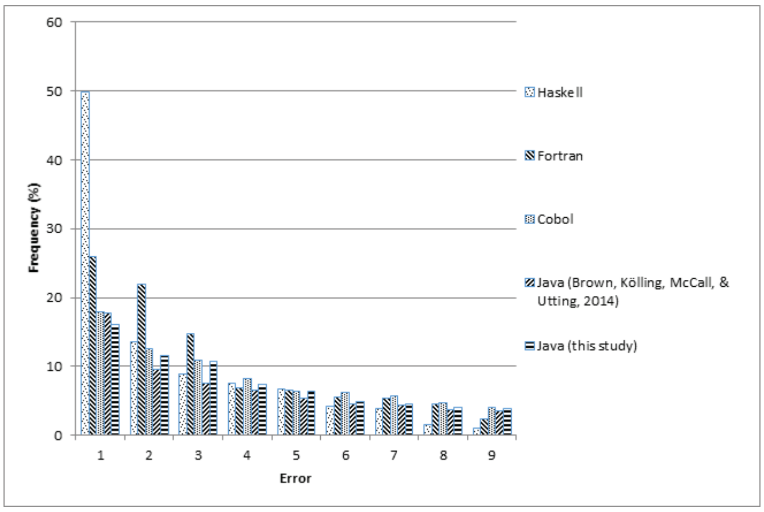


Figure 13. Frequency of nine errors from four different languages.

analysis, Figure 13 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 evaluating why this distribution is common[2] across so many languages. However Jadud posits two possible reasons: the programmer and the grammar. If indeed the reason is programmer behaviour, it would support the idea that the students in this study are not only similar to those in other studies involving Java, but 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.

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 way as possible, experiencing 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

[2]The most common error in Haskell is the most extreme outlier in this data. See Jadud (2006), p. 69 for a possible explanation.

this did not impact the results to a high degree 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, a student 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 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 potentially allows 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 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 cryptic compiler error messages (CEMs). Difficulties students have with CEMs have been present for at least four decades and occur with almost all programming languages. 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. Eight CEMs were identified accounting for 43.2% of all errors, and all enhanced by Decaf, which had a significantly reduced number of errors per student. These eight 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. The data was also analysed from a group perspective using principal component analysis, finding that both groups had distinct profiles. The intervention group exhibited less variance with a more homogenous 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*

In the future Java will most likely be replaced as the novice language of 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, and 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 analyse 37 million of them, from hundreds of thousands of students over at least several hundred institutions (Altadmri and Brown, 2015). However studies like this, closer to the students involved than global studies like Blackbox are and 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 its own 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 this research identified specific error messages, these were based on frequency, not analysing the actual issues students encountered when they committed particular errors. The second direction is a new 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 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 this experience can be shared and help more students, providing assistance in one of the many hurdles computer programming students face in learning an extremely important skill.

Supplemental material

Table S1 Most frequent CEMs (*) or the errors generating them (**) from eleven studies. Some studies are ambiguous, or present both.

Table S2 All compiler error messages logged during the study period.

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

Figure S2 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*, page to appear. 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*, page to appear.

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

Bergin, J., Agarwal, A., and Agarwal, K. (2003). Some deficiencies of c++ in teaching cs1 and cs2. *ACM SIGPlan Notices*, **38**(6), 9–13.

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 2014 conference on Innovation & 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/`.

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 *Frontiers in Education Conference*, volume 35, page T4C. STIPES.

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.

Kummerfeld, S. K. and Kay, J. (2003). The neglected battle fields of syntax errors. In *Proceedings of the fifth Australasian conference on Computing education-Volume 20*, 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.

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.

McCall, D. and Kolling, M. (2014). Meaningful categorisation of novice programmer errors. In *Frontiers in Education Conference (FIE), 2014 IEEE*, 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.

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.

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.

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 reids list of first programming languages? *Information Systems Education Journal*, **10**(4), 24.

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*.

Table S1 Most frequent CEMs (*) or the errors generating them (**) from eleven studies. Some studies are ambiguous, or present both.

Hristova, Misra, Rutter, & Mercuri, 2003**	
1. Using = instead of == or vice-versa	9. Leaving a space after a period when calling a specific method
2. Mismatching, miscounting and/or misuse of { }, [ ], ( ), " ", and ' '	10. >= and =<
3. Wrong separators in *for* loops	11† Invoking class method on object
4. An *if* followed by a bracket instead of by a parenthesis	12. Improper casting
5. Using keywords as method names or variable names	13† Flow reaches end of non-void method
6. Invoking methods with wrong arguments	14. Methods with parameters: confusion between declaring parameters of a method and passing parameters in a method invocation
7. Forgetting parentheses after method call	15. Incompatibility between the declared return type of a method and in its invocation
8. Incorrect semicolon at the end of a method header	16† Class declared abstract because of missing function
†Have a one-to-one mapping with CEMs. The others do not have a one-to-one mapping with student errors (Altadmri & Brown, 2015).	

Flowers, Carver, & Jackson, 2004**	
1. Mismatching curly braces	6. Mismatching parenthesis
2. Mismatching quotations	7. Missing semicolon
3. Misplaces semicolon	8. Misspelling printLine method
4. Improper file name	9. Package does not exist
5. Not initializing a variable before attempting to use it	

Toomey, n.d.	
1. Assignment in *if* statement	16. Missing left brace
2. Use of comparison after Boolean operator	17 Possible loss of precision
3. Use of bitwise operators	18. Malformed *for* loop
4. Cannot find a certain identifier	19. Possible misspelt word or command
5. Please use braces not parentheses	20: Not a statement
6. cannot treat *char* Like a *String*	21. Package does not exist
7. *Else* without a Matching *if*	22. Not enough closing braces
8. Empty statement	23. Right parenthesis expected
9. Empty statement after *if*	24. Missing ;
10. *System.exit()* needs a value	25. Checking for *String* (in)equality
11. Missing identifier	26. Missing " or " in String literal
12. Probable code in wrong place or missing braces / parentheses	27. Unrequired extra type keyword used
13. Probable imbalance with braces	28. Duplicate variable
14. Incomparable types	29. Cannot use something which gives 'void' in an expression

Thompson, 2004	
1. Undefined name	6. Undefined type
2. Type mismatch	7. Parsing error delete token
3. Undefined method	8. Package is not expected package
4. parsing error insert to complete	9. Undefined constructor
5. Should return value	10. Parameter mismatch

Jackson, Cobb, & Carver, 2005	
1. Cannot resolve symbol 2. ; expected 3. Illegal start of expression 4. class or interface expected 5. <identifier> expected 6.) expected 7. Incompatible types 8. Not a statement 9. } expected 10. *class FinalProject*	11. Illegal start of type 12. *java.lang.string* 13. Invalid method declaration; return type required 14. *boolean* 15. *else* without *if* 16. { expected 17. *double* 18. (expected 19. possible loss of precision
Jadud, 2006[*]	
1. Unknown variable 2. Bracket expected 3. Unknown method 4. Semicolon expected 5. Illegal start of expression	6. Unknown class 7. Incompatible types 8. Method application error 9. Private access violation 10. Missing return
Dy & Rodrigo, 2010[*]	
1. Unknown variable 2. ‘;’ expected 3. ‘[‘, ‘]’, ‘(‘, ‘)’, “, ” expected 4. unknown method 5. incompatible types	6. missing return statement 7. illegal start of expression 8. unknown class 9. identifier expected 10. class or interface expected
Tabano, Rodrigo, & Jadud, 2011[*]	
1. cannot find symbol – variable 2. ‘;’ expected 3. ‘(’ or ‘)’ or ‘[‘ or ‘]’ or ‘{‘ or ‘}’ expected 4. missing return statement 5. cannot find symbol – method	6. illegal start of expression 7. incompatible types 8. <identifier> expected 9. class, interface or enum expected 10. cannot find symbol – class
Chan-Mow, 2012[*]	
1. Variable not found 2. Identifier expected 3. Class not found 4. Mismatched brackets/parenthesis	5. Invalid method declaration 6. Illegal start of type 7. Method not found 8. Expected
Denny, Luxton-Reilly, & Tempero, 2011[*]	
1. Cannot resolve identifier 2. Type mismatch 3. Missing ; 4. Token should be deleted 5. Method not returning correct type	6. Missing } 7. Missing) 8. Missing { 9. Using .length as a field 10. Insert “Assignment Operator”
Brown, Kölling, McCall, & Utting, 2014[*]	
1. Unknown variable 2. Semicolon expected 3. Unknown method 4. Bracket expected 5. Unknown class	6. Incompatible types 7. Illegal start of expression 8. Method application error 9. Identifier expected 10. Not a statement

Table S2 All compiler error messages logged during the study period.

CEM number	Enhanced by Decaf?	CEM description
1	yes	'(' expected
2	yes	'(' or '[' expected
3	yes	')' expected
4	yes	'.' expected
5	-	'.class' expected
6	-	: expected
7	yes	';' expected
8	yes	'[' expected
9	yes	']' expected
10	yes	'{' expected
11	yes	'}' expected
12	yes	<identifier> expected
13	-	> expected
14	-	-> expected
15	-	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	-	break outside switch or loop
23	-	cannot assign a variable to final variable *variable_name*
24	yes	cannot find symbol
25	-	cannot return a value from method whose result type is void
27	-	'catch' without 'try'
29	yes	class *class_name* is public, should be declared in a file named *class_name*.java
31	-	class expected
32	yes	class, interface, or enum expected
34	-	constructor *constructor_name* in class *class_name* cannot be applied to given types;
36	-	double cannot be dereferenced
38	-	duplicate class: *class_name*
39	-	'else' without 'if'
40	-	empty character literal
43	-	exception *exception_name* is never thrown in body of corresponding try statement
46	-	illegal '.'
47	yes	illegal character: '*character*'
48	-	illegal escape character
49	-	illegal initializer for *type*
50	-	illegal line end in character literal
51	yes	illegal start of expression
52	-	illegal start of statement
53	-	illegal start of type
54	-	illegal static declaration in inner class *class_name*
55	-	illegal underscore
56	-	incomparable types: *type* and *type*
57	yes	incompatible types: *type* cannot be converted to *type*
58	-	inconvertible types
59	-	*type* cannot be dereferenced
60	-	integer number too large: *value*
61	yes	invalid method declaration; return type required

63	-	malformed floating point literal
64	-	method *method_name* in class *class_name* cannot be applied to given types;
65	-	method *method_name* is already defined in class *class_name*
66	-	missing method body, or declare abstract
67	yes	missing return statement
69	-	modifier static not allowed here
70	-	no suitable constructor found for *method_name*
71	-	no suitable method found for *method_name*
72	-	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	-	reached end of file while parsing
81	-	repeated modifier
83	yes	'try' without 'catch', 'finally' or resource declarations
85	-	unclosed character literal
86	yes	unclosed comment
87	-	unclosed string literal
89	yes	unexpected type
90	-	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	-	variable *variable_name* might not have been initialized
94	-	'void' type not allowed here
95	-	while expected

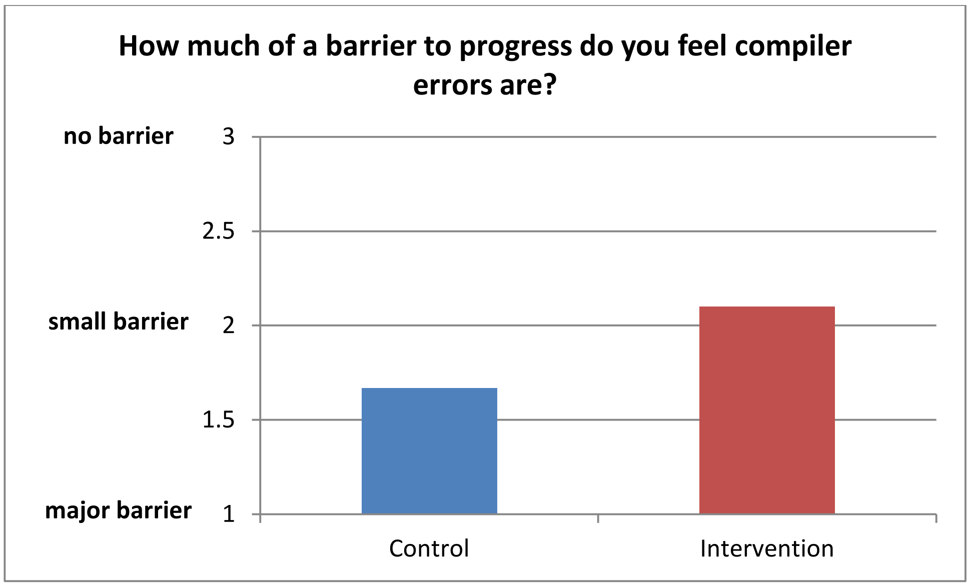


Figure S1 Students in the intervention group were significantly more likely to report that compiler errors did not present a barrier to progress [control (*M* = 1.67, *SD* = 0.64) and intervention *(M* = 2.09, *SD* = 0.72); *t*(51) = 2.21, *p* = 0.032].

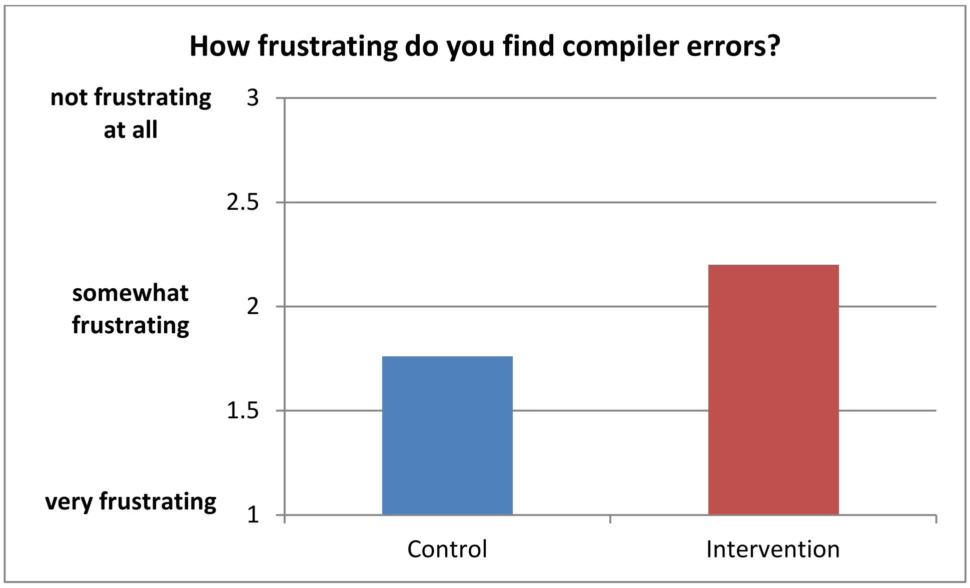


Figure S2 Students in the intervention group were significantly more likely to report less frustration with compiler errors [control (*M* = 1.76, *SD* = 0.61), intervention (*M* = 2.16, *SD* = 0.72); *t*(51) = 2.11, *p* = 0.040].