BlueFix: Using Crowd-Sourced Feedback to Support Programming Students in Error Diagnosis and Repair

Christopher Watson, Frederick W.B. Li, and Jamie L. Godwin

School of Engineering and Computing Sciences, University of Durham, United Kingdom
{christopher.watson, frederick.li, j.l.godwin}@durham.ac.uk

Abstract. Feedback is regarded as one of the most important influences on student learning and motivation. But standard compiler feedback is designed for experts - not novice programming students, who can find it difficult to interpret and understand. In this paper we present BlueFix, an online tool currently integrated into the BlueJ IDE which is designed to assist programming students with error diagnosis and repair. Unlike existing approaches, BlueFix proposes a feedback algorithm based upon frameworks combined from the HCI and Pedagogical domains, which can provide different students with dynamic levels of support based upon their compilation behaviour. An evaluation revealed that students' viewed our tool positively and that our methodology could identify appropriate fixes for uncompilable source code with a significantly higher rate of speed and precision over related techniques in the literature.

Keywords: Programming Education, Feedback, Compiler Errors, Crowd Fixes

1 Introduction

Feedback is regarded as one of the most important influences on learning and motivation [2][3]. To satisfy learning outcomes in an introductory programming course, a student has to develop a range of programming knowledge: syntactic, semantic, schematic, and strategic [18]. When learning to program students' are guided on the correctness of their syntax by compiler feedback. However standard compiler feedback is designed for experts - not novices, and often fails to match their current level of conceptual knowledge, making it difficult to understand. [9] considers the effect poor quality compiler feedback can have on programmers from a HCI perspective. They conclude that although programmers can often encounter cryptic messages which are difficult to resolve, most related disciplines have not paid much attention to this aspect, because it is felt that programmers should adapt to compilers. In contrast most pedagogical theory places a strong emphasis on adaption to the individual to make instruction most effective. Clarity and elaboration are fundamental principles of good feedback from both a HCI [9] and pedagogical perspective [2][3], yet most standard compiler messages fail to adhere to this [10]. Additionally due to parsing limitations, compilers often present the same feedback for a range of distinct errors [8][9][10]. This ambiguity poses a problem considering that novice programmers lack

the experience and expertise to identify the actual fault in their syntax [8]. Unsurprisingly it is not uncommon to observe novices applying almost random strategies to resolve compiler feedback which they struggle to comprehend [11], possibly blindly acting on the feedback provided with the belief that the computer is always right [9]. In this paper we present BlueFix, an online tool currently integrated into the BlueJ IDE which is designed to assist programming students with error diagnosis and repair. Our contributions include:

- Unlike [10][17], BlueFix proposes a feedback algorithm based upon frameworks combined from the HCI and Pedagogical domains, which can provide different students with dynamic levels of support based upon their compilation behaviour.
- Unlike standard compilers, BlueFix also provides combined feedback measures, supporting a learner with appropriate levels of elaborative feedback, rather than simply assuming a one size fits all approach.
- Also unlike standard compilers, BlueFix places an emphasis on teaching programming students how to resolve errors by example, and therefore suggests methods to resolve syntax errors using a database of crowd-sourced error fixes.
- BlueFix also shows a substantial improvement in performance of existing solutions, both in terms of the time taken to identify appropriate feedback and code fixes [10] and precision [17].

The remainder of this paper is organised as follows. Section 2 discusses related work. Section 3 presents the BlueFix architecture. Section 4 discusses initial findings on novice compilation behaviour and further motivations for the work in this paper. Section 5 presents an evaluation of BlueFix. Finally, Section 6 concludes the paper.

2 Related Work

Prior research has demonstrated that poor quality compiler feedback can have a negative effect on learning performance. [1] propose an algorithm to quantify the extent to which a student struggles to resolve syntax errors during a programming session. Using compiler data gathered from students taking an introductory Java course, [1] identified a significant relation between a student's mean *error quotient* and their overall course mark. Implying that students who struggle to resolve syntax errors perform worse on assessments than those who do not. In a related study [7] found that programming students experienced syntax issues regardless of their ability. However, the students who performed less well on a programming course were more likely to have been unable to produce syntactically correct code for programming exercises.

In general, two main approaches can be used to make compiler feedback more appropriate for novices [9]. The first approach consists of enhancing and/or rewriting standard feedback in laymans terms, and/or adding additional elaborations to clarify the feedback provided [3]. [6] developed a pre-compiler for Java based upon this approach. Although the system was not formally tested, instructors reported they were spending less time explaining compiler feedback to students over the duration of a course. A similar approach was used by [14]. However, elaborative feedback was only provided after a student had made the same error multiple times - thereby reduc-

ing the likelihood they would become reliant on the support and fail to develop a "feel for syntax" [9]. In contrast, [4] found additional elaborations did not necessarily help novices to resolve errors more quickly or correctly. However the authors question the validity of their own study, as it used students who had programming experience. The weakness of these techniques is that they do not address the possible inaccuracy of the reported error message. Even with additional elaborations, it is possible that a student will fail to understand the feedback provided, as it fails to align with their current level of programming knowledge [9]. Also the elaboration provided is usually generic, which is substantially less effective than response-contingent feedback [3].

The second approach involves directly tailoring the feedback provided based upon a static analysis of a student's uncompilable source code. Approaches such as [12][13] provide implicit feedback [3][10] to students, by spell-checking identifiers. Whilst this technique can provide Knowledge of Correct Response (KCR) feedback [2], it can only be applied on a limited subset of error types. An alternative approach, which has been gaining momentum in recent years, builds upon the observation that novices will often seek debugging help from online forums. However, standard search algorithms are based upon string literals rather than code semantics - making it more difficult for novices to identify relevant error fixes. A solution to this problem is to identify and provide a student with examples of how other programmers have resolved similar errors in the past [15]. By comparing the similarity of a student's code against a database of crowd-sourced error fixes [10][16][17], standard compiler feedback could be substantially enhanced with elaborative feedback [5] - effectively demonstrating to a student how to fix an error. This technique also addresses the issue of possible message inaccuracy - as even if the compiler feedback was inaccurate, fix suggestions could demonstrate how to transform damaged code into compliable code.

3 BlueFix: Methodology and Implementation

BlueFix is aimed at supporting a programming student in acquiring fundamental syntactic and semantic [18] Java knowledge, by demonstrating how syntactic errors can be corrected through two forms of elaborative feedback: enhanced error messages and crowd-sourced example error fixes. Unlike prior work [10][17], the BlueFix approach (Fig. 1) has been developed by combining sound feedback principles from both a HCI [9] and pedagogical perspective [3] (Table. 1). The originality of BlueFix is to combine these frameworks by proposing a set of techniques which provide a student with a progressively increasing level of elaborative compiler feedback, based upon the extent to which they struggle to resolve a particular error. In contrast, prior solutions [10][17] simply provide the student with all available elaborative feedback simultaneously - which is unlikely to be processed effectively by a novice and risks cognitive overload [3]. Instead the approach of BlueFix is to first encourage a student to resolve an error themselves, and then to dynamically adjust the level and type of elaborative feedback provided if they fail to do so. We believe this approach will support students' in developing debugging expertise - therefore allowing them to focus upon program logic and developing schematic knowledge [18] rather than syntax issues.

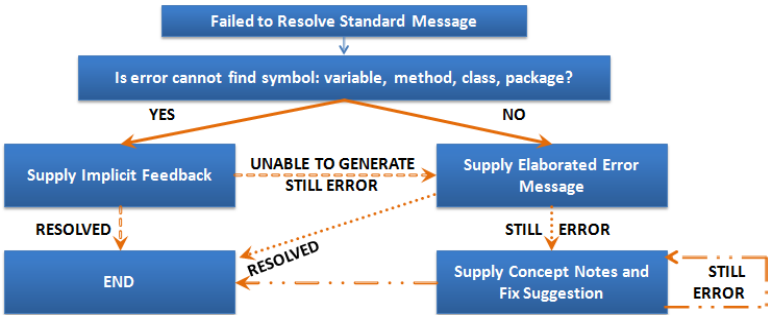

Fig. 1. Flow chart of the BlueFix process. Interventions are based on a student's error state.

From a HCI perspective, [9] suggests that in order to be most effective, compiler feedback should be divided into three increasing levels of elaboration:

1. Provide the programmer with a short message of the problem.
2. Provide brief explanations or generic examples.
3. Provide a further level of support based upon potential corrective actions.

These principles along with pedagogical feedback principles [3] (Table. 1) are tightly integrated into the BlueFix approach (Fig. 1). BlueFix initially provides a student with the standard compiler error - even though it may be inaccurate. This is an intentional choice; so that a student can develop a "feel for syntax" [9] and recognise errors in different environments without BlueFix support. If the student is unable to resolve an error on the second attempt (where resolve is defined as obtain either a different message or compiled code), then in line with [3][9] the level of elaborative feedback is increased to either implicit KCR [2] feedback or additional elaborations based upon the type of error message. If the error still persists on a third successive compilation, then the student is supplied with concept notes and crowd-sourced error fixes to demonstrate how others have resolved a similar syntactic problem in the past.

Table 1. Selection of pedagogical feedback principles [3] applied in BlueFix

#	Principle	BlueFix Application
1	Provide elaborated feedback to enhance learning.	BlueFix provides elaborated feedback in the form of error message explanations and fix suggestions.
2	Present elaborated feedback in manageable units.	BlueFix increases the amount of feedback supplied gradually responding to a sequence of student errors.
3	Keep feedback as simple as possible, but no simpler	Students are supplied with increasing levels of feedback abstraction when they fail to resolve errors.
4	Provide feedback after students have attempted a solution.	BlueFix provides feedback each time the student attempts compilation.
5	For difficult tasks, use immediate feedback.	As programming is a higher order cognitive task, BlueFix provides feedback immediately on compilation.
6	For low-achieving students, use correct response and elaboration.	Supplies the student with more elaborations to aid understanding in response to increasing compile fails.

An initial database of fixes was constructed using the compiler logs gathered (see Section 4) from students on the Introduction to Programming course at our university. BlueFix also allows additional fixes to be collected and added to its database during a lab session, by using additional classes which we have integrated into the BlueJ IDE. If BlueFix determines that the student requires an error fix, a list of fixes are queried and retrieved from an online MySQL database based upon a generalised error message (no identifiers), ranked, and presented to the student within a JFrame (Fig. 2).

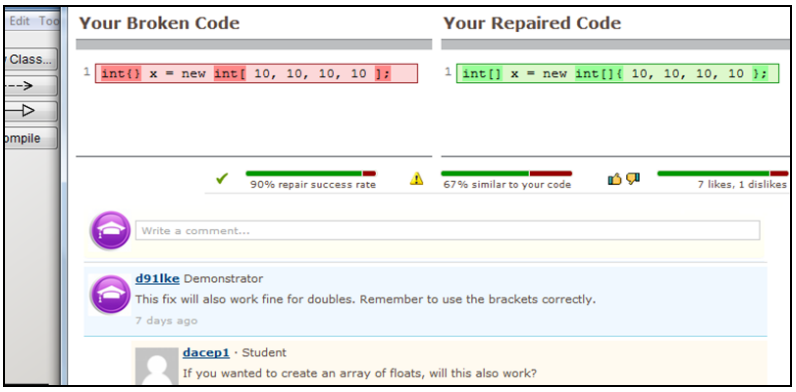

Fig. 2. BlueFix interface, showing a sample fix and student-demonstrator discussion.

Unlike [10], BlueFix also includes social media aspects. Students can rate fixes using 'like' buttons, allowing the better fixes to be more prominently ranked and displayed. Additionally, students are able to engage in anonymous discussions about particular error fixes with their peers and instructors - providing students who are usually reluctant to ask for assistance an opportunity to clarify issues and collaborate [5] (Fig. 2).

3.1 BlueFix Process: Determining relevant feedbacks at each stage.

In this sub-section, we outline the process and techniques BlueFix uses to determine the most appropriate feedback mechanisms at each stage of execution (Fig. 1).

Stage: Supply Implicit Feedback. The most likely cause of a "cannot find symbol" error is the misspelling of a variable, method, class, or package identifier. For this type of error, suggesting a fix which has been applied in the past is not appropriate. User defined types and naming within the Java language mean that although two classes or methods may share the same name, they may not share the same semantics. It is worth noting that in the case of our current dataset, these techniques can be applied to a substantial percentage of errors recorded (27%). Unlike [17] we therefore have constructed special handlers for the following unknowns.

1. *Unknown class*. This error is usually caused by either misspelling a class name or failing to import a required class from the API or a local package. BlueFix contains a Map of all Java SE6 and Java SE7 packages indexed on the class name. When

this error is reported, BlueFix first recursively scans the students BlueJ project and extracts the packages and classes within. The unrecognised class name is extracted from the compiler message, and compared against each of the names classes in the project, along with API classes. This comparison is performed using String matching techniques (Jaro-Winkler algorithm [19]). If a match is found (score >0.90), then it is added to an internal list of possible matches. The list is then sorted on score, and the closest match returned. If an import statement is missing from the project, this is also returned to the student. An example is shown in (Fig. 3).

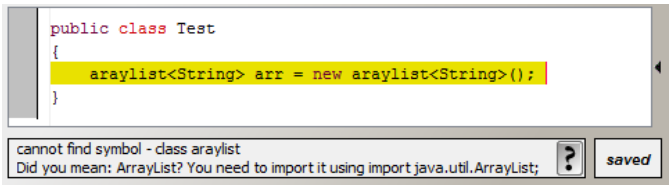

Fig. 3. BlueFix correcting an identifier misspelling and providing the required import line.

2. *Unknown package*. Essentially the same approach as unknown class, however attempts to suggest a package path using the last word on the import line as the class.
3. *Unknown method*. The internal parse tree of BlueJ is used to first determine the type of object calling a method and the class containing the method call. Through the internal debugger and reflection, lists of method signatures for the object type (and its super-classes) are retrieved and each name is compared against the name of the unknown method. As with unknown class handler, Jaro-Winkler is used and the closest match suggested to the student as a possible fix.
4. *Unknown variable*. The internal parse tree is used to extract the names of all identifiers within a class between the error line location and the closest method signature before it. Field names of the class are then added to a list of identifiers and Jaro-Winkler similarity computed.

Stage: Supply Elaborated Error Message. If either the initial error type was not an unknown identifier, or if BlueFix was unable to determine suitable implicit feedback, then the next level of elaborative feedback is generated. Although prior solutions [6][14] have supplied elaborative feedbacks in the form of supplementary error explanations, these explanations lacked grounding in pedagogical or HCI principles - therefore increasing the risk that a student would still not be able to comprehend the cause of an error from the feedback provided. Another criticism of this approach is that it fails to address the lack of locality or specificity of standard messages, thereby supplying the student with additional feedback regardless as to whether the underlying message (or feedback) is correct or not. The resulting student confusion from inaccurate elaborations can clearly have a negative effect on the learning process [2][3]. To address this issue, [9] presented a set of eight principles of effective compiler message design: clarity, specificity, context-insensitivity, locality, proper phrasing, consistency, suitable visual design, and extensible help. To enhance the effectiveness of standard compiler feedback, BlueFix contains a database of 92 distinct compiler error

messages and rewritten versions that conform to the sub-principles of clarity and proper phrasing: using positive tone, providing constructive guidance and reducing jargon. Future enhancements will attempt to address the specificity, context-insensitivity and locality principles by performing additional parsing checks to tailor the feedback supplied based directly to the student's damaged source code.

Stage: Supply Concept Notes and Fix Suggestion. If the elaborated error messages fail to assist a student in resolving an error, then BlueFix will attempt to locate and present a code fix. This is a stored piece of code which is similar to the student's uncompilable source code (and the same error message). A fix is defined as the changes made on the stored uncompilable code, to transform it into the stored fixed code.

Construction of a Database of Fixes. The compiler events in our dataset (Section 4) were processed on a per-week, per-student, and per-filename basis. For each week, the successive *n* compiler events that a student performed, were classified into a set of tuples $\{\{c_1, c_2\},.., \{c_{n-1}, c_n\}\}$, so that the errors the student fixed during a session could be identified. Unlike existing solutions [10][17] which consider all tuples to be a valid fix if the compile success status of $\{c_i, c_j\}$, i = false, and j = true, we only consider such events to be a *possible* fix. In our solution, we first assess the quality of a possible fix before adding it to our database. We believe this will help to improve the quality of available fixes. For example, a student could resolve an error by simply deleting or commenting out blocks of code. Although this can transform the code into a compliable state, it is unlikely to resolve the underlying error. Therefore unlikely to show a student how to resolve an error which they are struggling with. To detect deletion fixes we compute the diff ratio (ignoring whitespace) between the source code of each c_i and c_j. If the number of insertions and changes = 0, and the deletes > 0, we classify the fix as a deletion fix. Commented fixes are detected by using a regex expression on the error line. However we are currently experimenting with performing additional comment checks by using the patches (text differences between the two files) generated by a diff algorithm. Analysis on the precise metrics is still required.

Identifying Appropriate Fixes. The first step used to determine fixes for the student's uncompilable source code is to retrieve all possible fix tuples from the database, which have the same generalised error message (no identifiers) as the student has.

However not all of these retrieved fixes can be applied to the student's code. We therefore need a measure of similarity between a student's uncompilable source code and the uncompilable source code in a fix tuple. That way we can determine the changes that were performed on a similar piece of code, to make it compliable. Previous work [10] proposed using a structure-based similarity technique, where the parse tree of a student's uncompilable source code was generated, then compared against the parse trees of possible fixes. Whilst this approach benefits from high precision, performing node-by-node comparison of multiple parse trees incurs a substantial time cost; making the technique unfeasible for larger databases or longer code fragments. As with [17] we therefore chose to compute similarity by using a string matching algorithm. These methods in general have the advantage of a low time cost. A range

of measures were considered: Levenshtein, Dice Coefficient and Needleman-Wunsch. However, we found the best performance in terms of balancing speed and accuracy came from calculating the Jaro-Winkler distance [19] between the source files.

The problem is that edit distance algorithms can over penalise for different method, variable, or class identifiers. We therefore use the BlueJ lexical analyser to first tokenize both the students uncompilable source code, and the uncompilable source code of each of the fix tuples retrieved so far. As with [17] we then compute the line similarity between the error line of the student's uncompilable code against the error line of each fix tuples uncompilable code. However unlike [17] and due to our preliminary findings on location inaccuracy (Section 4), we also incorporate region similarity into an overall appropriateness measure. This is calculated in the same was as line similarity, but includes lines up to 4 lines behind the reported error location and 4 lines after.

As BlueFix allows a student to rate fixes using a simple like/dislike system we also include student ranking in our appropriateness measure, to allow more 'stable' or 'popular' fixes to be higher ranked than fixes have not been successful in the past. BlueFix takes all this information into account in its scoring function. The appropriateness of each fix is scored as:

$$\text{Fix Appropriateness} = (5L + 2R + 1T) / 8 \quad (1)$$

where *L* is the Jaro-Winkler similarity between the tokenized error line in the student's code against the error line in the candidate fix; *R* is the Jaro-Winkler similarity between the tokenized error region of the student's code and candidate fix; *T* is the % of student 'likes' of a fix. The exact coefficients have been derived through trial and error. We wanted to place a greater emphasis on an individual line match rather than region, as a region may contain many irrelevant lines. The possible fixes are then ranked using a Comparator based upon fix score and the fixes with a score higher than a threshold (currently set at 0.8) are presented to the student. Future work will include reassessing the coefficients and threshold, as the size of the fix database increases.

Substituting Fixes. Before presenting a fix to a student, as with [17][10] we apply a token-based substitution of variable names and values in the selected fix for those in the student's original code. This ensures that the elaborative feedback in the form of a fix suggestion is tailored to the student's coding context, thereby increasing the likelihood of understanding [2][3]. The student can directly request BlueFix to apply the fix to their code by using the 'apply fix' button. After this, BlueFix will invoke the java compiler to determine whether or not the fix has resulted in compliable code or not, and update a successful substitution counter in the database accordingly. This function is currently restricted to single line fixes.

4 Further Motivations For Work

To explore aspects of novice compilation behaviour we used a sample of students who studied the 2011/2012 Introduction to Programming course at our university. The course was designed to teach Java to student's of varying abilities and assumed no

prior programming experience. Lectures were supported by a weekly practical session where students would practice programming problems using the BlueJ IDE. We decided to directly gather information on student compilation behaviour by creating an extension to BlueJ. Each time the student compiled their code on a university PC, the extension would log a snapshot of their program source code along with information on the result, timestamp, error message and line number. A total of 39 students' provided us written consent to use their logged data. In terms of prior programming experience, we note that 10 students indicated they had prior experience; however when asked the size of the largest program they had written, the majority indicated a small program (<1000 lines). 6 of these students indicated they had previously used Java, but on average had less than a year's experience. Although we collected logs for 15 practical sessions (teaching weeks 4-19), we restrict our analysis to only 11 of these sessions due to student examination and assignment work.

During the 11 logged practical sessions, our plugin recorded a total of 22,993 distinct compiler events of which 11,412 (49.6%) were compiler errors. As 2,937 distinct messages were recorded, we classified errors into a hierarchy based upon message abstraction level and type. This yielded eight top level groupings (Table. 2). We found that 85% of the errors that students encountered came from three categories: identifiers (38%), syntax (26%) and computation (24%).

Table 2. Classification of Java Compile Time Errors.

Group	Description	Example Subgroup
Syntax	Violate the fundamental syntax rules of Java	; expected
Computation	Program logic definition, flow control.	illegal operations
Identifiers	Unknown, re-declaring variables / methods	unknown method
Scope	Access violations: public, private, packages	method is private
Exceptions	Error handling, try-catch keywords	try without a catch
Inheritance	Method / variable overriding, super	super-type not called
Abstract	Misuse of abstract keyword	cannot have body
Static	Relate to use of class and object types	cannot be referenced

Syntax Errors as a Predictor of Student Performance. To analyse a possible link between syntax errors and student performance we applied the error quotient algorithm proposed by [1] to our data set. As our students had not yet completed their final exam, we compute an interim ability score as a function of five completed assessment components. These are the total marks (%) that the student gained on: (A) weekly exercises, (B) fault injection report, (C) multiple choice concepts exam, (D) paper based programming exam, (E) computer based programming exam. Each task is weighted in terms of the amount of programming involved and the module weighting. An ability score is computed as: $(1A + 1.5B + 2C + 2.5D + 2.5E)$, normalized into the range 0-100. Applying this function to the 39 participants of our study yielded a normal distribution of ability scores (Shapiro-Wilk's $p > .05$) ranging from 22.69 to 84.91. Consistent with the findings of [1], a linear regression revealed that a student's mean error quotient could statistically significantly predict their ability score, $F(1, 37)$

= 8.695, $p < .005$ and the student's mean error quotient accounted for 16.8% of the explained variability in ability scores (adjusted $r^2 = .168$) - a medium effect.

Accuracy of Error Messages and Location. The fault injection assignment required students to inject 30 random faults (change variable, delete line / character) into a small Java project (~600 lines) and evaluate the hypothesis: "it is difficult for compilers to identify which programming errors have been made, and the location of the problem". Out of the 39 students who participated in this study, 22 (56%) believed it was difficult for a compiler to identify which programming errors had been made, and 23 (59%) believed it was difficult for the compiler to isolate the location of the error. The average distance between reported error location and actual error location was 4.02 lines (*SD* = 1.19), however students found this distance varied considerably depending upon the type of error. The average number of reported messages that the students thought were inaccurate was higher than anticipated, with a mean of 29.5% (*SD* = 14.83). Concluding opinions by the students were mixed. Some students commented that they had "*no issues*" with the error messages as they "just get used to them". Other students were more critical, stating that "*the advice it* (the compiler) *gives often ends up breaking the programs even more. As such, it is best not to rely on it too much*". Nevertheless these findings indicate that over half of our students were not satisfied with the lack of appropriate guidance from standard compiler feedback.

5 Evaluation of BlueFix

5.1 Student Feedback

We have conducted an evaluation of a prototype system using a focus group of 11 students from the Introduction to Programming Course at our university. Students were provided with a demonstration and discussion of prototype BlueFix functionality, and asked to complete a short questionnaire. All students viewed the enhanced error messages and fix suggestion capabilities of BlueFix as a useful aid to help with error resolution. 63% viewed fix suggestions as the most useful form of support, and 37% viewed enhanced error messages as the most useful. 81% of the students perceived value of including a social aspect which would allow them to discuss particular fixes with peers and instructors. The interesting finding of our evaluation is that 72% of the students believed that error fixes should be tailored to their broken code; however the fixes should not be able to be directly applied, for example, by clicking "apply this fix". This is possibly due to the opinion that applying a fix may damage the code further, or transform it to the point where the student no longer comprehends it.

5.2 BlueFix Precision Compared to HelpMeOut

To evaluate the precision of our approach, we chose to run BlueFix on 20 test cases for each of the error groups in Table 2. Faults such as renaming a variable, deleting a random line, switching parameters, and removing a random character were performed

on a small Java project consisting of 7 Java classes and approximately 6,000 lines of code. The project was not part of the BlueFix database of fixes. At the time of running, BlueFix contained 7,645 distinct error fixes, covering 842 distinct error types. To compare BlueFix to related work, we have implemented the methodology presented in [17]. Fixes were returned to an expert programmer who would determine whether or not they provided an example of how to repair the mutation. Findings on the precision and recall of both techniques are shown in Table 3.

Table 3. Comparing BlueFix precision, recall and F_1 to HelpMeOut [17]

		BlueFix			**HelpMeOut**		
Error Group	**Cases**	**Precision**	**Recall**	F_1	**Precision**	**Recall**	F_1
Syntax	20	51.98	25.49	27.67	34.60	31.36	28.33
Computation	20	47.37	10.25	13.51	35.42	13.57	13.99
Identifiers	20	86.84	0.70	1.38	5.21	22.18	2.67
Scope	20	46.93	45.20	34.00	36.67	50.86	31.93
Exceptions	20	38.52	36.38	19.60	27.92	24.39	17.86
Inheritance	20	32.46	36.36	24.88	28.21	42.63	26.04
Abstract	20	40.35	27.81	23.73	28.75	33.25	20.85
Static	20	40.70	84.21	51.64	32.22	88.33	41.60
Overall *M*		**48.14**	**33.34**	**24.55**	**28.62**	**39.32**	**22.91**
Overall *SD*		**16.77**	**25.18**	**14.72**	**10.06**	**23.35**	**11.88**

A Wilcoxon Signed-Rank test was performed to determine if there were any differences in the precision of fixes suggested by BlueFix and HelpMeOut [17]. We found a statistically significant difference in the precision rate of BlueFix (*Mdn* = 50.0) compared to HelpMeOut (*Mdn* = 33.4), $z = 6.29$, $p < .0005$. Out of 160 test cases, BlueFix had a higher precision on 82 cases (51.3%), a worse precision rate on 11 cases (6.8%) and no difference on 67 cases (41.9%).

6 Conclusion & Future Work

In this paper we have presented BlueFix: an online tool integrated into the BlueJ IDE which is designed to assist programming students with error diagnosis and repair. The original contribution of BlueFix is to propose an algorithm and methodology to supply individual students with dynamic levels of feedback support, based upon combining feedback frameworks taken from both the HCI and Pedagogical domains. We have also added to the growing body of knowledge on novice compilation behaviour by presenting preliminary findings from analysing quantitative compiler data gathered over a one year Java programming course. We have conducted an evaluation of BlueFix, which revealed that students viewed the tool positively, and an initial evaluation of BlueFix precision suggests an improvement of 19.52% over a previous technique [17]. Future work will consist of evaluating BlueFix accuracy on a larger data set, and possibly expanding the tool to provide advice on runtime errors through analysing

stack traces. Additionally we plan to evaluate the effectiveness of BlueFix by deploying it during the 2012/2013 academic year and comparing its effects on learning and compilation behaviour [1] to the 2011/2012 cohort.

7 References

1. Jadud, M.C.: Methods and Tools for Exploring Novice Compilation Behaviour, pp. 1-5 Proc. ICER (2006).
2. Jaehnig, W., and Miller, M.A.: Feedback types in programmed instruction, a systematic review, The Psychological Record 57(2): 219-232 (2007).
3. Shute, V.J.: Focus on Formative Feedback. Review of Educational Research 78(1): 153-189 (2008).
4. Nienaltowsi, M., Pedroni, M. and Meyer B.: Compiler error messages: what can help novices? pp. 168-172 Proc. SIGCSE (2008).
5. Sykes, E.R., and Franek, F.: Presenting JECA: A Java Error Correcting Algorithm for the Java Intelligent Tutoring System, pp 151-156 Proc. IASTED (2004).
6. Flowers, T., Carver, C.A., and Jackson, J.: Empowering students and building confidence in novice programmers through Gauntlet, pp. T3H/10 - T3H-13 Proc. FIE (2004).
7. Denny, P., et al.: Understanding the Syntax Barrier for Novices, pp. 208-212 Proc ITiCSE (2011).
8. Marcrau, G., Fisler, K. and Krishmanurthi, S.: Measuring the Effectiveness of Error Messages Designed for Novice Programmers, pp. 499-504 Proc. SIGCSE (2011).
9. Traver, V.J.: On Compiler Error Messages: What They Say, and What They Mean. Advances in Human Computer Interaction, 2010 (1).
10. Watson, C., Li, F., and Lau, R.H.: Learning Programming Languages through Corrective Feedback and Concept Visualisation, Proc. ICWL (2011).
11. Kummerfeld, S.K., and Kay, J.: The Neglected Battle Fields of Syntax Errors, pp. 105-111 Proc ACE (2003).
12. Burrel, C., Melchert, M., Mann, S. and Bridgeman, N.: Augmenting Compiler Error Reporting in the Karel++ Microworld, pp. 41-46 Proc NACCQ (2007).
13. Sykes, L.: Process Model for the Java Intelligent Tutoring System, Journal of Interactive Learning Research 18(3): 399-410 (2007).
14. Murphy, C., Kaiser, G.E., Loveland, K., and Hasan, S.: Retina: Helping Students and Instructors Based on Observed Programming Activities, pp. 178-182 Proc. SIGCSE (2009).
15. Brandt, J., et al.: Example-Centric Programming: Integrating Web Search into the Development Environment, pp. 513-522 Proc. CHI (2010).
16. Mujumdar, D., et al.: Crowdsourcing Suggestions to Programming Problems for Dynamic Web Development Languages, pp. 53-56 proc. CHI EA (2011).
17. Hartmann, B., MacDougall, D., Brandt, J., and Klemmer, S.R.: What would other programmers do? Suggesting Solutions to Error Messages, pp. 1019-1028 Proc. CHI (2010).
18. Meyer, R.E., From Novice to Expert, Handbook of Human-Computer Interaction, pp. 781-795, Prentice-Hall (1997).
19. Cohen, W.W. and Ravikumar, P. and Fienberg, S.E. A comparison of string distance metrics for name-matching tasks, pp. 73-78, Proc. IIWeb (2003).