Learning Programming Languages through Corrective Feedback and Game-Based Concept Visualisation

Abstract. In this paper, we address common issues faced by students in programming courses by combining implicit and explicit corrective feedback to provide real-time assistance in coding tasks and introducing a novel game-based concept visualisation technique to allow students to visually obtain knowledge of coding concepts and execution state. The mapping between game actions and source code forms an implicit example-based environment allowing concepts and feedback to be more clearly conveyed than in e-learning systems, integrated development environment (IDE), or through static course materials. An experimental evaluation of a prototype system confirmed the potential of this approach for programming education by scoring highly in terms of both user satisfaction and its potential pedagogical capability.

Keywords: Programming Education, Corrective Feedback, Game-Based Learning.

1 Introduction

Learning to program is a fundamental skill required to obtain a Computer Science degree throughout the world. However, each year many students find the process of learning a programming language to be a difficult and even unpleasant task [4,16]. Several reasons have been put forward to explain this. Some are related to traits of the students themselves. For instance, they may lack problem solving or logical reasoning skills or have a low mathematical ability. Other reasons include no interest in the subject matter, perceived high difficulty of the course [2,16], lack of immediate results from little syntax [2] or even wanting to avoid the social stereotype of being perceived as a “nerd” [11]. Other reasons are associated with the quality of the teaching process. For instance, the use of traditional static lecture materials generally fails to convey program dynamics to students. Instructors may focus upon teaching the syntax and semantics rather than necessary problem solving skills. Additionally, a traditional one size fits all approach ignores the individual student’s learning style and ability, making it very easy for less able students to rapidly fall behind the rest of a cohort [11]. Clearly, there are many varying theories concerning why introductory programming courses are perceived to be difficult, but the majority of literature tends to share a common sense of panic about their perceived high dropout rates [11,16]. This is perhaps justified considering that a recent survey of 63 international institutions found that on average 33% of students fail an introductory programming course [4]. The question is how to address this problem. One-to-one tutoring has been
demonstrated to be one of the most effective pedagogical approaches [6] but it is practically unfeasible to use this method at university level especially for larger cohorts. A more suitable method could be to increase the pedagogical effectiveness of existing instruction by using adaptive e-learning systems to provide individualized guidance and feedback to a student. Another approach could be to try and increase the student’s motivation to learn programming through the use of a more stimulating and less intimidating learning tool than a traditional IDE.

In this paper, we address common issues faced by students in programming courses by combining implicit and explicit corrective feedback to provide real-time assistance in coding tasks. We have also developed a novel game-based concept visualisation technique, allowing the student to visually obtain knowledge of coding concepts and execution state. The mapping between game actions and source code forms an implicit example-based environment allowing concepts and feedback to be more clearly conveyed than in a traditional e-learning system, IDE, or through static course materials. The remainder of this paper is organized as follows. Section 2 summarizes existing approaches to improve the effectiveness of programming instruction. Section 3 presents our method and implementation. Section 4 discusses findings of a user evaluation. Finally, Section 5 concludes the paper.

2 Related Work

Throughout the years, numerous techniques have been used to make the teaching of introductory programming skills to novices more effective by adopting a simplified learning environment. General methods include simplifying the language (e.g., BASIC) or reducing the need to type syntax by allowing program construction using visual objects (e.g., Alice2) [17]. Whilst visual and block-based programming languages such as Scratch or Alice2 have shown some success in teaching basic programming concepts, a student is still required to eventually scale-up to a more complex language and the use of these introductory tools does not imply this will automatically follow. In fact, it could be argued that removing a language’s syntax defeats the point of learning a programming language to start with [2]. Another approach to make the learning process more effective is to teach programming through Computer Games. The ability of games to teach problem solving skills through a repeated cycle of expertise [10] makes them ideally suit programming education, as students are required to develop a hierarchy of skills through repeatedly extending their existing problem solving techniques and knowledge of syntax [16]. Game usage can be divided into two main categories. The first is the constructivist technique of game development. Although this method has shown notable success in terms of both learning performance and student satisfaction [3,18], it is likely that novices will still encounter usual difficulties such as IDE complexity, vague compiler messages and no visualisation of state and syntax execution until a basic game structure is completed. The second approach addresses the need for visualisation by demonstrating concepts through turtle-based challenges (e.g. [2,5]). However these systems are generally restricted in terms of feedback provided and their teaching capability limited to function calling. Adaption to learner characteristics is even rarer
Learning Programming Languages through Corrective Feedback and Game-Based Concept Visualisation

and an outdated appearance means their appeal is rapidly diminishing with the current generation of students [5]. To address the need for guidance, several systems have been developed over recent years. Although functionalities provided vary from system to system, most share several common limitations. The range of exercises provided is usually limited to either completing missing lines in a skeleton program [12,14,23], satisfying a program specification [8,26] or computing the value of a variable after code execution [1,13]. Correctness of the student’s answer is verified using case tests and then feedback is generated. Feedback is usually limited consisting of a set of test cases on which the students code has passed and failed [1,13-14], possibly based on the ratio of fails and passes [8] and accompanied with instructor comments on specific test cases based on constraint violation [26]. More detailed feedback was provided by the JITS system which included a correction algorithm to correct student syntax mistakes [23]. Some solutions are limited in terms of the range of topics which they cover. For example, JEE-Tutor [20] was limited to teaching operator precedence, associatively, and expression evaluation only. A final limitation is that material and concepts are presented in an un-stimulating environment providing no visualization of the code execution state. Considering that many errors which novice students make can be attributed to an inadequate understanding of program state [9] visually being able to monitor an effect of code execution could enhance the learning experience.

3 Our Method

We believe that a more effective method of teaching programming concepts can be obtained by merging the reasoning capabilities with a highly motivational game-based environment. Previous research has suggested that environments of this nature can be more effective [25]. However, examples of such systems for programming education are rare. To support novices in developing an understanding of syntax and logic our approach includes different forms of corrective feedback: implicit, explicit and program logic level. To convey coding concepts in a clearer manner than syntax alone, we provide game-based concept visualisation. Although we apply our method to Java, the approach and techniques could be applied to any language.

3.1 Scene Modelling and Code Assessment

Content is disseminated to students through a set of learning scenes. We describe a learning scene - LS by using a number of properties. The most important is the tuple of n concepts \{C_1, C_2 ..., C_n\} enumerating concepts, which LS is designed to teach, ranked in the order of importance. A set of m scenes \{S_1, S_2 ..., S_m\} defines prerequisite scenes of LS which a student is required to complete before LS is made available. A set of k pairings \{{T_1, O_1}\{T_2, O_2\}, ..., \{T_k, O_k\}\} describe a collection of test criteria and required outcomes. A boolean flag s indicates whether or not learning performance should be assessed using code similarity analysis. A set of r fragments \{F_1, F_2 ..., F_r\} lists code fragments which are required to be present in a valid
solution. Finally, a set of $x$ model solutions $\{A_1, A_2, ..., A_x\}$ represents valid solutions to the problem. To determine whether or not a student’s code is a valid solution to $LS$ the code is analysed over a number of stages (Fig. 1).

**Fig. 1. Overview of Corrective Feedback and Code Assessment Features.**

The first two stages (of the Interpreter) involve checking the code for syntax and runtime errors. If errors are detected at either of the first two stages then the code is passed to the corrector module where our corrective feedback functionalities are invoked. The third stage verifies the validity of the student’s code against the set of test criteria (and model solutions only if $LS\{s = true\}$) to ensure it functions correctly. The final (optional) stage is requirement check where presence of the set of required fragments is verified, e.g., $LS[r=\{"for"\}]$. If the student's code passes all test stages but is not an exact match of any of the scene's model solutions then it is stored pending further analysis by an instructor. If the solution is deemed valid then it is added as a model solution. This reduces the instructor’s workload considerably considering that they are not required to develop and author every possible solution to a programming problem.

### 3.2 Explicit Corrective Feedback

Debugging is a key skill required when programming. However, many novices can struggle with ‘ambiguous’ and unspecific compiler messages [20]. In other words, the feedback returned does not always accurately describe the fault in the code (Table 1). Considering that novices are heavily reliant upon IDE feedback and guidance to compensate for their lack of experience, unspecific compiler messages pose a serious issue. Simply acting upon the feedback provided could result in the introduction of even more errors with student confusion. The loss in lab time through correcting syntax can be both costly to students (less problem solving time) and instructors (debugging rather than teaching programming techniques). To address this issue, we
Learning Programming Languages through Corrective Feedback and Game-Based Concept Visualisation

extend the rewriting technique of traditional tutoring systems further to provide the student explicit corrective feedback. By using source code analysis techniques to tailor error messages to the student’s code, we can explain the problem to the student more clearly and therefore allow them to develop a deeper understanding of its cause. From an implementation perspective, our system includes a database of common errors (Table 1) grouped by class. If an error is reported from the javac compiler, its class is identified and a set of possible causes are retrieved from the database. For simplicity, we only consider the first error thrown. Using line and column information provided, the corresponding fragment at fault is extracted and compared to the fragments in the database. If a match is found, tailored advice is supplied to the student. If there is no match, then a generalised layman’s terms message is returned and the fragment added to the database pending an expert instructor to specify advice. Always providing the standard compiler error along with the generalised advice ensures that any experts assisting the student can perform diagnosis. As with the reuse of student solutions to programming tasks, this allows our system to become semi self-populating rather than requiring an instructor to manually author examples and feedback of all possible causes of error. Additionally, it allows the systems to continually expand to cover a greater range of errors.

Table 1. Common Java Errors from a Programming Course [15] and possible causes.

<table>
<thead>
<tr>
<th>Error Class</th>
<th>Code Fragment</th>
<th>Actual Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>cannot resolve symbol</td>
<td>String = new String(); return s.Length();</td>
<td>No variable name.</td>
</tr>
<tr>
<td>; expected</td>
<td>x = x + 3; String y = “is” x;</td>
<td>Misspelt method name.</td>
</tr>
<tr>
<td>class or interface</td>
<td>For(int i=0; i&lt;5; i++) {}</td>
<td>Case on keyword ‘for’.</td>
</tr>
<tr>
<td>expected</td>
<td>calc(int x, int y);</td>
<td>Adding parameter types in call.</td>
</tr>
<tr>
<td>&lt;identifier&gt; expected</td>
<td>public void int getX()</td>
<td>Extra keyword in signature.</td>
</tr>
<tr>
<td>not a statement</td>
<td>String x = “a” + “p” ; ”c” ;</td>
<td>Extra semi-colon.</td>
</tr>
<tr>
<td>} expected</td>
<td>// close loop }</td>
<td>Accidental commenting.</td>
</tr>
<tr>
<td>{ expected</td>
<td>public class My Class</td>
<td>Space in the class name.</td>
</tr>
</tbody>
</table>

3.3 Implicit Corrective Feedback

As with [23], we attempt to provide automatic error correction of some of the student’s syntax errors, in particular automatically correcting spelling and case mistakes involving Java keywords (e.g. int) and common Java classes (e.g. String). This implicit corrective feedback instructs the students how to correct problems on the syntax level. Currently, this functionality is only invoked if the compiler produces a ‘cannot find symbol’ error. When an error of this type is detected the system first extracts the token at fault using line and column information provided by the compiler. Next the Levenshtein distance of the token and each of possible
replacements is computed to produce a ranked list of substitutions ordered by least distance. The Levenshtein distance can be calculated with a run time of $O(|n|m|)$. The token at fault is replaced with the highest ranked substitution and the code is recompiled. If the ‘cannot find symbol’ error has been removed then the system asks the student if they think the correction is valid. This ensures that the student is actively made aware of the error in their code and is guided to correct it. If the error is still present then the next ranked substitution is applied until the list is exhausted. If the system was unable to correct the student’s error through substitutions, then the code is passed to the compiler message customiser for further analysis.

3.4 Program Logic Corrective Feedback

Our final form of corrective feedback supports the student in correcting problems at the program logic level. As novices can encounter difficulties in completing coding tasks, our system is capable of suggesting possible solutions based upon their similarity to the student’s code. Defining similarity of code fragments performing the same under all possible conditions is unfeasible to verify, so we consider a simpler definition. As we want to compare the structure of a student’s code to a range of possible solutions, we have chosen to use the tree-based approach [21] by comparing the Abstract Syntax Tree (AST) of the student’s code to a scenes set of model solutions. This allows us to identify the most appropriate code resembling what the student has already created through precise node-by-node comparison, rather than supplying a single model answer. Specifically, we use a tree edit distance algorithm [24], which has performed well in a similar project to determine the similarity between classes [22]. Let $V$ be a set of vertices, and $E$ a set of edges. For two ordered trees $Z_1 = (V_1, E_1)$ and $Z_2 = (V_2, E_2)$, we define three valid elementary operations on $Z_1$ and $Z_2$: the deletion of a leaf vertex $v \in V_1$, the substitution of a vertex $w \in V_2$ for a vertex $v \in V_1$ and the insertion into $Z_2$ of a new vertex $w \in V_2$ as a new leaf. A cost function is applied to each operation. The edit distance is the lowest cost sequence of operations needed to transform $Z_1$ into $Z_2$. By constructing an edit graph of $Z_1$ and $Z_2$, the problem of finding a valid transformation from $Z_1$ to $Z_2$ can be reduced to finding the shortest path through the graph. This has a run time of $O(|V_1||V_2|)$ requiring $O(|V_1||V_2|)$ additional space [24]. From an implementation perspective, model solutions are first retrieved from the database and along with the student’s code, are converted into simplified AST’s using the Eclipse JDT project. These are passed to a reducer class, which prunes the search space based upon the average number of vertices in a model AST. For instance, if on average a model AST contained 50 vertices and the student AST contained 30 vertices, then all model AST’s with the number of vertices greater than a pre-defined threshold could be removed, as an excessive amount of edit operations would be required to perform the transformation. The edit distance is then calculated on the reduced set of model AST’s to determine the closest match to the student’s code.
Learning Programming Languages through Corrective Feedback and Game-Based Concept Visualisation

3.5 Game-Based Concept Visualization

To convey programming concepts and constraints to novice students in a more visual and interactive manner than a conventional e-learning system [1, 7-8, 12-14, 23, 26] we use a game-based learning interface as a support. Unlike standard applications of computer games within programming courses, our approach does not simply include games as a method of sugar-coating the learning process. For example, we do not simply combine games with traditional programming tasks to make them more entertaining and appealing to students. Neither do we simply add irrelevant games to the learning process as a means of rewarding student progress. Instead, the uniqueness of our approach is to transform well designed game tasks into programming tasks, such that the student can learn how to program by completing the tasks. We implement this concept through a game-based learning interface. This interface allows us to convey programming concepts graphically to students, thus forming an implicit example-based learning environment. Through observing the game state, novices can obtain a greater understanding of concepts and the execution state [9] than through the use of traditional static course materials alone. Furthermore, the use of games for concept visualisation provides a strong motivational tool, where novices can see immediate results of code execution without first having to master a large amount of syntax [2]. The game scenario can also be customised, thus allowing for a greater variety of different examples and increasing the motivational capability further. For instance, to teach object creation one of our implemented learning scenes is set in a garage near a racing track (Fig. 2). The player assumes the role of the mechanic’s apprentice. When the scene commences, the mechanic explains how his car (object) was created from a specific blueprint (class). To demonstrate the concept, the student is required to create two new wheel objects and then add them to the car. This is performed by right clicking on a box of ‘Wheel’ in the corner of the room and selecting ‘new Wheel()’ from the popup menu. When the student performs these actions the corresponding fragments of course code are printed in the box below the game interface. This provides the student with an implicit example of the relation between classes and objects and demonstrates the corresponding syntax required.
Fig. 2. Sample Learning Scene Demonstrating Object Creation.

In comparison to verifying the validity of student’s solutions or adding additional error messages, the authoring cost of this stage to an instructor is naturally more expensive, requiring the specification of a mapping from game actions to source code as well as creating graphics. However, we believe that the potential educational benefits of providing students with a motivational environment of this nature, where they can perform game actions and view corresponding code fragments or vice-versa outweighs the authoring cost. Future work will include the development of a pedagogical interface [27] to further assist ordinary instructors in game creation.

4 Evaluation

Preliminary Findings. To determine whether students would be interested in using a game-based approach to learn programming, we have conducted a preliminary survey of students who recently completed the Introduction to Programming (IP) course at our university. A total of 54 students completed the online survey, of which 29 completed the course in 2011 and 25 in 2010. Findings indicated a high level of student interest in using a game-based approach, with only 11% of students indicating no interest or felt the approach would have been unsuitable for the IP course. 82% felt that a game-based approach would have been more interesting than current laboratory tasks, with 63% indicating they would have practiced programming more in their own time if a game-based approach was used. Additionally, 74% reported that unspecific error messages had been a hindrance at some point in the course.

User Experience. To assess the suitability of our system, we have conducted a user study on a prototype. The main research questions we asked included: Would the techniques presented provide better support to students in understanding programming concepts than traditional tools such as BlueJ or Eclipse? Does the approach convey concepts such as method calling in a clear manner? Would the system be more appropriate for a novice programmer than IDE’s? The participants included six members of the universities Learning Technologies Team (LTT) and six demonstrators. The LTT regularly provides training and advice to staff on aspects of e-learning and is involved in developing tools to support education, such as Blackboard extensions. Participants were invited to either a group or individual session which lasted on average for 30 minutes. Each session included discussion and two demonstrations of each of our support methods and game-based tasks. Participants were asked to provide an overall grade out of five indicating whether or not a feature would aid the students understanding of coding concepts and syntax. The results are shown in Table 3.
Learning Programming Languages through Corrective Feedback and Game-Based Concept Visualisation

Table 3. Results from the User Study.

<table>
<thead>
<tr>
<th></th>
<th>Error Correction</th>
<th>Message Customisation</th>
<th>Code Suggestion</th>
<th>GBL Use</th>
<th>GBL Tasks</th>
<th>Actions to Source Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>LTT (n=6)</td>
<td>4.17</td>
<td>4.50</td>
<td>4.00</td>
<td>4.50</td>
<td>4.33</td>
<td>4.50</td>
</tr>
<tr>
<td>Demonstrator (n=6)</td>
<td>4.00</td>
<td>4.17</td>
<td>3.33</td>
<td>4.33</td>
<td>4.17</td>
<td>4.67</td>
</tr>
<tr>
<td>Overall Average (/5)</td>
<td>4.08</td>
<td>4.33</td>
<td>3.67</td>
<td>4.42</td>
<td>4.25</td>
<td>4.58</td>
</tr>
</tbody>
</table>

The qualitative feedbacks collected were mainly positive. Most participants believed that the auto-correction of errors would be a useful support to novices. Several suggested that the functionality should be invoked after a student has attempted to correct the mistake instead of immediately. Error message customisation was deemed to be the most useful corrective feedback technique as it could provide students more practical advice than what they currently receive through javac. The game-based visualisation of coding concepts such as method calling proved to be a popular idea to support novices (10/12 positive) and the mapping between game actions to output source code was viewed as an effective method of demonstrating the effect of code execution to novices (9/12 positive). Overall participants indicated that they saw value in the techniques for teaching novice programmers with 8/12 indicating the system would be more appropriate for novices than BlueJ and 10/12 more appropriate than a conventional IDE.

User criticism. The main criticisms of our approach were related to the scaling up issue, as when a student migrates to a regular IDE the additional supports would be lost. Suggested solutions to this issue were to integrate the functionality directly into an IDE rather than using a standalone system or to always provide the generic error message along with customised support. Other comments were related to the code presentation (lack of syntax highlighting).

Video demonstration. As part of this paper, we have compiled video demonstrations of our game-based interface along with the corrective feedback techniques outlined. They are available at youtube.com/duicwl11.

5 Conclusion

In this paper, we have presented a novel method of teaching programming concepts by generating different forms of corrective feedback with a motivational game-based interface for the visualisation of coding concepts and code execution. Unlike static lecture materials, game-based concept visualisation allows novices to rapidly develop an understanding of coding concepts through a set of clear, visual examples which conveys more information about the execution state than static syntax alone. It also provides a strong source of motivation by allowing novices to obtain ‘instant results’ of code execution without first having to master a large amount of syntax. Unlike traditional programming teaching, our approach goes beyond simple ‘fill in the gap’
exercises by providing real-time context-specific guidance and feedback on student’s codes through source code analysis. We have evaluated and confirmed the feasibility of the approach from both instructor (educational potential) and student perspectives (interest). Future work will include extending the game-based interface to support problem-based learning and to evaluate educational effectiveness using a set of non-programmers. Additionally, the potential benefits of using a token-based approach to evaluate coding similarity will be investigated in terms of both runtime and accuracy.

6 References

Learning Programming Languages through Corrective Feedback and Game-Based Concept Visualisation