NoobLab: An Intelligent Learning Environment for Teaching Programming

Paul Neve, Gordon Hunter, David Livingstone and James Orwell
Kingston University
London, UK
paul@kingston.ac.uk

Abstract*— Computer programming is a highly practical subject and it is essential that those new to the discipline engage in hands-on experimentation as part of the learning process. However, when faced with large cohorts and an increasing demand for distance and student flexible learning, incorporating this into a programming course can be difficult. There is a dynamic that exists between tutor and student in a real-world programming workshop session that is not easily replicated online. In this paper we describe an online learning environment that begins to create an analogue of this dynamic and its successful integration into an undergraduate programming module. Ultimately, the potential exists to not only improve the student learning experience but also investigate and inform programming pedagogy itself.

Keywords-component; computer programming; pedagogy; e-learning; virtual laboratories; educational analytics

I. INTRODUCTION

The refrain of Jenkins' students' that programming is "boring" and/or "difficult" [1] is all too familiar. Tutors also face their own difficulties and drudgeries presented by both logistical realities and the nature of the discipline. Programming is a complex subject that cannot be taught through explanation alone, a fact that informs constructivist approaches to teaching the discipline. One such approach is Wulf's [2], which places the onus on the student to learn through practical experimentation and in collaboration with peers. Lecture-style teaching is reduced to a minimum and the tutor's role defined as a "guide on the side".

Unfortunately, practical considerations often intrude upon the constructivist ideal, and the ease with which conventional lectures can be delivered to a large cohort is a compelling argument in their favour. A common approach is to use lectures to establish concepts then follow up with practical lab/workshop sessions. This is relatively simple to manage, as a large cohort can attend one lecture then be divided into smaller, more manageable groups for the hands-on sessions.

It is important that a tutor or other expert be available during workshops to guide students' practical experimentation. Guidance from a tutor will prompt a reaction from a student: they might produce new code, engage the tutor in a dialogue, or do something else entirely. In turn, this reaction alters the tutor's future feedback particularly if the response was unexpected. However, the dynamic between tutor and student goes beyond the *domain contingency* [3] defined by David and Heather Wood, lasting not for just a single engagement but the duration of a workshop session, probably during a module as a whole and possibly throughout their entire relationship. It is a self-reinforcing cycle that we call "the learning loop", and is a key part of an effective programming workshop.

Unfortunately, even if a large cohort is subdivided, finding enough tutors to cover all the groups can be problematic, and the quality of feedback during workshops can be adversely affected. Large cohorts also introduce problems with respect to assessment. Marking hundreds of student submissions is extremely time consuming, so course designers fall back on assessment strategies which can be marked quickly and/or automatically. For this reason multiple choice questions are often used, but these must be carefully selected and/or constructed if they are to genuinely assess a student's ability. It is particularly important to avoid the assessment becoming a "pop quiz" on terminology trivia; for example, a question that offers several definitions for a term such as "overriding" or "overloading" does little to demonstrate an ability in the student to actually utilise this knowledge in any practical or useful sense. Students who achieve a good overall grade in such assessments often remain utterly incapable of creating an original program without assistance: the skills and knowledge they need in order to pass bear no resemblance to the skills required for real-world programming.

Meanwhile, the demand for distance and flexible learning is increasing, and such a mode of study does not always lend itself to the real-time dialogue that is a crucial part of the learning loop. These practical realities once again interfere with the ideal pedagogic world.

II. THE *PRACTICAL PROGRAMMING* MODULE

Practical Programming is a second semester, first year undergraduate module at Kingston University. It follows a first semester module that uses Java to introduce programming to complete beginners.

Practical Programming sets out three aims:

- To develop students' enthusiasm for practical programming,
- To enhance students' experience with programming environments,
- To develop students' confidence in their ability to write programs.

There are two occurrences of the module delivered concurrently: one for Computer Science students and one for Information Systems (IS) students. In 2011 half of the IS cohort failed to achieve a passing grade even after retakes had been taken into account. The IS module team felt that changes had to be made, and in 2012 the decision was made to redesign the IS occurrence, to switch from Ruby to Javascript, and to make use of learning technology to promote student engagement via flexible, self-paced study.

A. *The vision*

Three key words were taken from the module aims to inform the redesign process: *enthusiasm*, *experience* and *confidence*, three things that students often lack when it comes to programming. In discussing the difficulties of integrating learning technology into a constructivist learning context, Gance sets out several principles [4]: the student must "cognitively engage" with their learning environment; new learning should involve active exploration of the environment; students should engage in a "hands-on, dialogic interaction"; any practical exercises should be "authentic in nature"; and a social component should include dialogue with "mentors" and other learners.

A constructivist approach was adopted that would focus on practical programming activities and be based around three overarching games programs. Abstract programming problems would be avoided – activities that did not directly involve the games would remain "authentic" and involve a related skill or concept. Students would be "cognitively engaged" throughout the module, engaging in hands-on, programming tasks on a continuous basis. Students would also be given a measure of control over their assessment parameters so that success (the definition of which differs between students) becomes an attainable goal, promoting student motivation as per Gregory and Jenkins' [5].

B. *Developing the learning environment*

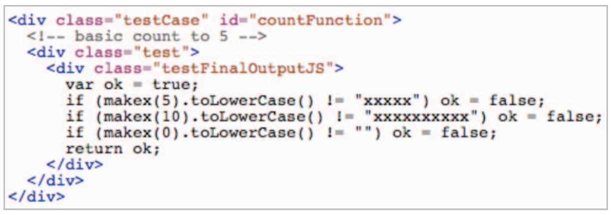
NoobLab was an existing learning environment for programming already in use at Kingston University [6]. It presented learning content alongside an area in which simple Javascript-based programs could be composed and run in a virtual console. A number of open source tools were combined with bespoke code to create the environment, the most important of which was Oni Labs' implementation of Stratified Javascript, Apollo (http://onilabs.com/apollo). This allowed infinite loops – a common beginners' mistake – to execute within the environment without causing the host browser to hang.

The first iteration of NoobLab was designed to present learning content and exercises designed to bootstrap programming naïfs to a level where they could at least understand basic programming terms and concepts prior to undertaking a computing-related Masters degree. The first iteration was tightly coupled to this content. It had no facilities for assessment, and no ability to test students' code against an exercise's desired outcome, The environment did not record the student's code as they worked, nor did it record any interactions with or feedback from the environment. Ultimately it was a useful preparatory resource but inherently limited.

Thus the first task was to decouple the tool from the existing content, and introduce a more flexible means for defining new content. Some consideration took place as to whether to adopt an established standard for describing e-learning content, such as eLML or CNXML. However, these were rejected due to the fact that both have a steep learning curve, and that editors available are limited. Additionally, both standards would need to be extended to accommodate functions such as automated testing of submitted program code.

Consequently, the decision was made to use HTML, but to add several semantic extensions to support the context of programming pedagogy. The advantages of this approach were twofold: those authoring content for a computer programming course are likely to be familiar with HTML; secondly, although any interactive aspect would be dependent on the NoobLab environment, content would still be at least readable in a conventional browser. The most important of these extensions introduced test criteria, which could be expressed either as a simple string comparison against which program output would be compared, or as Javascript code to be run after the student's own.

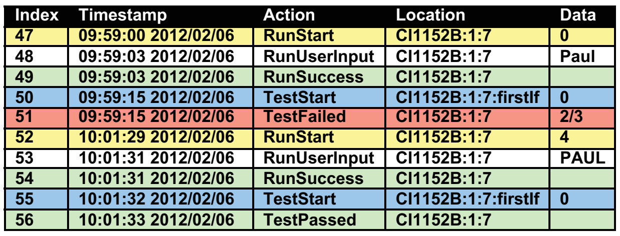
Figure 1 shows an example of how test criteria is specified; here the student's task is to write a function called *makex* that returns a string of X's of the length specified in the parameter. The test case is expressed as Javascript code that is run after the student's, with a return value set to true or false indicating test success. Although the example shows a single test condition, multiple test conditions can be used. Note that the extensions are semantic and use the existing *class* attribute, so do not invalidate the HTML specification.



```
<div class="testCase" id="countFunction">
  <!-- basic count to 5 -->
  <div class="test">
    <div class="testFinalOutputJS">
      var ok = true;
      if (makex(5).toLowerCase() != "xxxxx") ok = false;
      if (makex(10).toLowerCase() != "xxxxxxxxxx") ok = false;
      if (makex(0).toLowerCase() != "") ok = false;
      return ok;
    </div>
  </div>
</div>
```

Figure 1. Semantic extensions in HTML to support test criteria

The environment was also extended to gather data about students' navigation through and interactions with learning content, e.g. moving to a new piece of content, responding to a quiz question or pasting a code exemplar into the editor. The environment would also log any code that is executed and its outcome (e.g. a successful run, syntax or runtime error), and a difference index based on Levenshtein's algorithm [7] which indicates how much the student's code has been altered between runs. If code is run against test criteria, the level of success against the criteria would also be logged.

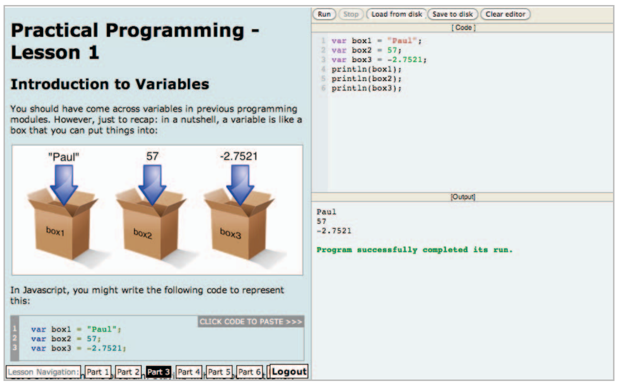


Index	Timestamp	Action	Location	Data
47	09:59:00 2012/02/06	RunStart	CI1152B:1:7	0
48	09:59:03 2012/02/06	RunUserInput	CI1152B:1:7	Paul
49	09:59:03 2012/02/06	RunSuccess	CI1152B:1:7	
50	09:59:15 2012/02/06	TestStart	CI1152B:1:7:firstIf	0
51	09:59:15 2012/02/06	TestFailed	CI1152B:1:7	2/3
52	10:01:29 2012/02/06	RunStart	CI1152B:1:7	4
53	10:01:31 2012/02/06	RunUserInput	CI1152B:1:7	PAUL
54	10:01:31 2012/02/06	RunSuccess	CI1152B:1:7	
55	10:01:32 2012/02/06	TestStart	CI1152B:1:7:firstIf	0
56	10:01:33 2012/02/06	TestPassed	CI1152B:1:7	

Figure 2. An excerpt from the NoobLab usage logs.

Figure 2 shows a log excerpt from the implemented system, taken during a practical exercise where the student was tasked with creating a program to do a case insensitive comparison of two strings. In index lines 47-51 the student believes they have a correct solution. It runs without syntax or runtime errors, but is logically flawed. Thus in lines 50 and 51 they attempt to test it against the test criteria, and it fails. In line 52 we see a second run attempt. In this line, the *Data* column indicates a Levenshtein difference index of 4. Upon the second test attempt in lines

55-56, the code passes the test criteria. Thus we can conclude that the changes made by the student at line 52 were the ones that successfully solved the exercise task.



Practical Programming - Lesson 1

Introduction to Variables

You should have come across variables in previous programming modules. However, just to recap: in a nutshell, a variable is like a box that you can put things into:

In Javascript, you might write the following code to represent this:

```
var box1 = "Paul";
var box2 = 57;
var box3 = -2.7521;
```

```
var box1 = "Paul";
var box2 = 57;
var box3 = -2.7521;
println(box1);
println(box2);
println(box3);
```

```
Paul
57
-2.7521

Program successfully completed its run.
```

Figure 3. The NoobLab Environment.

Figure 3 shows the user interface that the student sees. The basic, static material for a given piece of learning - “a lesson” – is shown on the left hand side. In many respects this can be considered analogous to the lecture component of the dual lecture/workshop pedagogic model discussed previously. It incorporates exemplar code, images, videos and quiz questions. On the right hand side the student can modify and run code, which, along with the practical exercises and test criteria specified as part of the learning content, form an analogue of the workshop component.

C. *Course and assessment design*

The three games chosen were Hangman, Tic-Tac-Toe and Connect 4. Each of these was intended to present an increasingly challenging programming task. Completing Hangman and making a good start on Tic-Tac-Toe would achieve a basic pass but to achieve a high grade all three games had to be at least attempted if not completed. Students were encouraged to be strategic about where they focused their efforts, which became their method of controlling their assessment. Effectively, students self-selected ability groups, similar to Jenkins and Davy [8], and Davis et al [9]. Weaker students could focus on a simple implementation of one of the easier games; stronger students could flex their muscles and create something more impressive or tackle more of the games.

Marks were awarded to discourage plagiarism and encourage regular engagement. Up to 30% would be awarded for the games programs ultimately submitted but 50% would be awarded for a “Big Test” during which students would be expected to make alterations to the three games under examination conditions. This had a similar difficulty-banded approach, with 9 activities, three for each game, and each activity within a game increasing in difficulty. Once again, students could select activities that best fit their ability level and control the parameters of assessment, but ultimately attain a mark appropriate for the effort and expertise demonstrated. Students who chose to focus on Hangman would achieve the similar, bare-pass level as those who chose to do a selection of “easy” questions across all three games; students who completed the harder Connect 4 or selected a range of harder questions across the games would do much better.

The final component of assessment consisted of fortnightly “Small Tests”, 4 in total, each worth 5%. These were presented not as regular examinations but as a means by which students could be credited for their week-to-week efforts. Although these were not specifically associated with the coursework games, they were deliberately designed to allude to them, e.g. adding items to a Tic-Tac-Toe-style 3x3 grid. Despite Jenkins’ argument against continuous assessment [1], it was felt that deferring assessment until the end of a module risks students disengaging until final assessment is upon them – by which point they have little chance of catching up.

D. *Course delivery*

There were two scheduled sessions lasting two hours each per week. In the previous version of the module this involved a lecture followed by practical workshop, but the boundaries between the two were deliberately blurred in the module redesign. In the redesign, the morning session opened with an overview of any interesting developments that arose from the previous week’s activities, followed by a short introduction to the current week’s learning material. This took approximately 15-20 minutes, after which students were expected to embark on a self-paced exploration of new material presented within the NoobLab environment. This included formative practical programming exercises against which the student could run embedded NoobLab test cases, and thus get immediate feedback on their work. The afternoon session was billed as optional, and students were given the choice to attend if they felt they needed to. In order to discourage students retreating entirely into the virtual world and becoming solitary in their learning, the tutor would roam and act as a “guide on the side”. While the environment would provide the majority of feedback, the tutor would observe, interject and offer commentary, but would also return to the lectern and engage the group as a whole whenever an interesting discussion point emerged from students’ activities.

III. RESULTS

Although two staff members were allocated to the scheduled sessions, it was common for one of them to leave before the end simply because they were not needed – students received much of the required feedback directly from the environment itself. Thus the use of the environment had a positive effect on human resource requirements, and provided some elements of the learning loop.

Just as with the formative exercises, Small Tests were also framed as NoobLab test criteria, and this simplified the marking process considerably. A student receiving what came to be known as “the green box of success” from the environment could be awarded full marks, with no further examination of their work required. Usually this was also the case for near misses, with only submissions at the weaker end of the spectrum requiring human assessment to determine whether some partial credit was merited. In most cases, students received their marks for a Small Test on the same day it was sat.

Based on the 46 students who completed more than half the summative tasks, 78% of them achieved a passing grade. At the module’s conclusion, students were given a series of positive statements about the NoobLab environment and the module as a whole. They were asked to assign a mark where 6 indicated strong agreement and 1 indicated strong disagreement. For each assertion, a "satisfaction index" was generated, based on the total score expressed as a percentage of the maximum possible score. The assertion "I did better at this module than I expected to" scoring a high satisfaction index of 83%. There was a noisy but significant correlation ($R = 0.749$, $p < 0.001$) between time spent in the NoobLab environment and a student's final mark. Further investigation of two of the outliers (see Figure 4) also yielded interesting results, with one proving to be a student with prior Javascript experience, whereas the other had "crammed" for the final Big Test but done very little work beforehand.

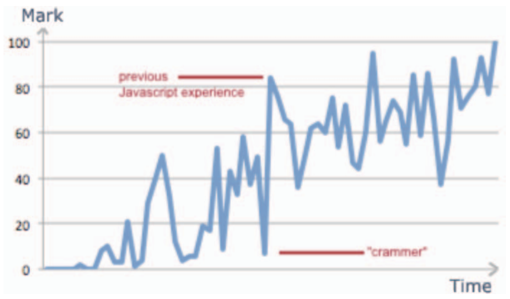


Figure 4. Final mark against total time spent in NoobLab environment

General feedback was very positive. The NoobLab environment scored extremely highly, with related assertions averaging over 90% Additionally, the environment was repeatedly cited in the free text question *"what was the best thing about the module?"* and often praised in other free text responses. Students were also complementary towards the module structure and delivery approach. Contrary to Jenkins' objections to continuous assessment [1] the assertion "*it was a good idea to have regular Small Tests*" scored an extremely high satisfaction index of 94.12%. Small Tests also had a positive impact on the attendances of the optional sessions, and in their immediate aftermath there was a clear increase in formative usage of the learning environment.

IV. Related work

Some commonalities exist with previous research and/or tools. RoboProf [10] can assess code against test criteria and also logs students' usage, but is restricted to a simple comparison of program output against a set of test data and lacks the seamless presentation of NoobLab. InSTEP [11] presents code editing facilities alongside associated learning material, but is limited to exercises based around small modifications to templates. ASAP [12] provides comprehensive testing capabilities, and could be integrated with a web-based code editor, but is primarily concerned with assessment and lacks the immediacy of NoobLab's feedback. Coursemarker [13] assumes the student has access to a functional compiler which does not lend itself to supporting beginners or distance learners. The online version of Havebeke's *Eloquent Javascript* [14] has a browser-based editor similar to the first iteration of NoobLab, but equally has similar limitations.

A key difference is that the NoobLab/Practical Programming combination adopts a more holistic approach where both course and technology design has been part of an integrated process, and one has informed the other. Arguably the same could be said about CodeAcademy (www.codeacademy.com). This uses gamification [15] and adds social components to create an immersive, engaging learning experience, with a similar approach to expressing test criteria to NoobLab. Their social focus is also very much in keeping with Gance's 4th principle [4]. However, by its very nature CodeAcademy targets the public as a whole; one cannot limit a course to a specific institution or cohort. Although proponents of the Open Educational Resource movement might consider this a good thing, institutional regulations and requirements could be incompatible with this position. Additionally, CodeAcademy does not provide a means for tutors to access the detailed level of usage data as that stored by NoobLab. This is expected to be a crucial resource in the progression towards a more complete automated implementation of the learning loop.

V. Future work

After the successful deployment on Practical Programming, this approach and the NoobLab environment will be rolled to other programming modules, particularly those targeted at beginners. One possible target module is called *Fundamentals of Programming Concepts*, and among other learning outcomes seeks to establish the skill of decomposing a programmatic problem into its constituent parts. In the past students have been presented with logical problems for which they must express a solution in pseudocode. However, students find what is essentially "programming on pen and paper" tedious and struggle to understand its relevance. To make the activities more akin to "real" programming, some of these activities were reformulated as tasks that can be solved using a BBC Basic interpreter. The integration of a BASIC interpreter within NoobLab would be a logical next step.

A more innovative approach might be to make problems and exercises more visual in nature, as per Pattis' *Karel the Robot* [16], and particularly later projects that wrap a modern "host" language around the Karel command set [17]. Such exercises can seem like recreational puzzle solving, but this approach mean that students begin learning the syntax and some foundations of a real-world language before they are actively conscious of doing so.

There is also a demand for NoobLab to support other, "real world" languages for modules beyond beginner level and where course content becomes language specific. Java is emerging as the most urgent requirement in this regard. Other requirements have been articulated to provide facilities that ease the burden of summative assessment and feedback, particularly for large cohorts.

A. Patterns and programming pedagogy

The importance of the logs that are gathered by the environment cannot be overstated, as they can be inspected manually and/or analysed quantitatively to obtain insights into both individual students and general trends. Perhaps the most exciting area for future work lies in analysis of patterns within these statistics.

Consider the pattern "run program" followed by "syntax error" repeated many times, with little or no difference in the code between run attempts. We call this the "SOS Pattern" - a student unable to make the jump from error message to diagnosis, and who is desperately hoping that the problem will go away on its own. A logical extension would be for this pattern to trigger the display of material related to debugging techniques and interpreting error messages.

The SOS Pattern is commonly observed in those new to programming, and is thus easily predicted. However, a more thorough analysis of the statistics will likely yield other patterns that are common across students. Statistical pattern recognition approaches and data visualisation techniques will be used to identify "clusters" and/or "signatures" across learners. Visualisation of students' routes through learning material and formative practical exercises has already exposed another pattern, the "Rosetta Stone". This is seen in students having difficulty with a practical exercise who, upon revisiting a certain part of learning material, are then able to complete the exercise. It is only a short jump to suggest that future students who struggle with the same exercise should be automatically directed by the environment to the associated Rosetta Stone material.

Using these signatures to prompt unsolicited, impromptu feedback would permit the implementation of a richer, more "intelligent" analogue of the learning loop. Signatures could also be used to inform pedagogy, course design and student management. For example, a student who is exhibiting a pattern that previously resulted in a negative outcome could be flagged for remedial action, or an exercise that is causing difficulty for many students could be flagged for review.

VI. Conclusions

The use of online delivery and automated assessment tools can greatly enhance the learning experience in computer programming courses. If used correctly, these tools can deliver flexible, self-paced learning in a discipline that is fraught with difficulties both pedagogic and logistic.

It is possible to design an online learning environment for programming that is in keeping with a constructivist pedagogic philosophy. A holistic approach is key – the design of the technology must proceed hand-in-hand with the design of the pedagogy and course content as a whole. The learning experience must be integrated, with no sudden jumps between different modes of study – the experience should be as consistent as possible regardless of whether a student is simply reading through material or actively engaged with a practical exercise. When this approach was used on a first-year undergraduate programming module, student feedback was overwhelmingly positive. The merits of this approach would also seem to be confirmed by the increasing popularity of the CodeAcademy website.

However, the true potential of this research lies not in improving delivery of programming courses, or making material more engaging. By using the statistics gathered from students as they work to drive feedback, and specifically, common patterns or "signatures", a closer analogue of the learning loop that exists between a human tutor and student may be implemented. Ultimately, one might envisage a truly adaptive learning environment for computer programming, that has the capability to learn and adapt to each student, advising them and providing feedback not only upon request but also based on impromptu "observation" of their work, just as a roaming human tutor might during a physical, real-world workshop session.

VII. References

[1] T. Jenkins, "On the difficulty of learning to program", *3rd Annual Conference of the LTSN Centre for Information & Computer Sciences*, Loughborough, U.K., 2002, pp. 53–58.

[2] T. Wulf, "Constructivist approaches for teaching computer programming", *6th Conference on IT Technology Education*, ACM, Newark, USA, 2005, pp. 245–248.

[3] D. Wood and H. Wood, "Vygotsky, Tutoring and Learning," *Oxford Review of Education*, vol. 22, no. 1, Carfax Pub. Co, Oxford, U.K., pp. 5–16, 1996.

[4] S. Gance, "Are Constructivism and Computer-Based Learning Environments Incompatible?" *Journal of the Association for History and Computing*, American Association for History and Computing, Oregon, USA, 2002.

[5] T. Jenkins and P. Gregory, "Motivating Computing Students," in *Effective Learning and Teaching in Computing*, RoutledgeFarmer, London, UK, 2004, pp. 21–28.

[6] P. Neve and D. Livingstone, "NoobLab: An online workshop environment for programming courses" in *Quality enhancement in learning and teaching: How Kingston University is improving the student experience*, Kingston University, Surrey, UK, 2011, pp. 52–53.

[7] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," in *Soviet physics doklady*, 1966, vol. 10, pp. 707–710.

[8] T. Jenkins and J. Davy, "Dealing with diversity in introductory programming," in *1st Annual Conference of the LTSN Centre for Information and Computer Sciences*, Edinburgh, UK, 2000, pp. 81–87.

[9] H. C. Davis, L. A. Carr, E. C. Cooke, and S. A. White, "Managing diversity: Experiences teaching programming principles". *2nd Annual Conference of the LTSN Centre for Information and Computer Sciences*, London, U.K. 2001.

[10] C. Daly and J. M. Horgan, "An automated learning system for Java programming," *IEEE Transactions on Education*, vol. 47, no. 1, IEEE, pp. 10– 17, Feb. 2004.

[11] E. Odekirk-Hash and J. L. Zachary, "Automated feedback on programs means students need less help from teachers," *32nd SIGCSE Technical Symposium on Computer Science Education*, New York, NY, USA, 2001, pp. 55–59.

[12] C. Douce, D. Livingstone, J. Orwell, S. Grindle, and J. Cobb, "A technical perspective on ASAP–Automated System for Assessment of Programming," *9th Computer Assisted Assessment (CAA) Conference*, Loughborough, U.K., 2005.

[13] C. A. Higgins, G. Gray, P. Symeonidis, and A. Tsintsifas, "Automated assessment and experiences of teaching programming," Journal on Educational Resources in Computing, vol. 5, no. 3, ACM, New York, USA, 2005.

[14] M. Haverbeke, *Eloquent JavaScript,* No Starch Press, California, USA, 2011.

[15] S. Deterding, D. Dixon, R. Khaled, and L. Nacke, "From game design elements to gamefulness: defining 'gamification'," *15th International Academic MindTrek Conference*, MindTrek Association, New York, NY, USA, 2011, pp. 9–15.

[16] R. E. Pattis, *Karel the Robot: A Gentle Introduction to the Art of Programming*. John Wiley & Sons, Inc. New York, NY, USA, 1981.

[17] B. W. Becker, "Teaching CS1 with Karel the Robot in Java," *SIGCSE Bulletin*, vol. 33, no. 1, pp. 50–54, ACM, New York, USA, 2001.