The Programmer's Learning Machine: A Teaching System To Learn Programming

Martin Quinson and Gérald Oster
Université de Lorraine, France.

ABSTRACT

The Programmer's Learning Machine (PLM) is an interactive exerciser aimed at learning programming and algorithms. It targets students in (semi-)autonomous settings, using an integrated and graphical environment that provides a short feedback loop. This generic platform also enables teachers to create specific programming microworlds that match their teaching goals. This paper discusses our design goals and motivations, introduces the existing material and the proposed microworlds, and details the typical use cases from the student and teacher point of views.

Categories and Subject Descriptors

K.3.2 [**Computer and Information Science Education**]: Computer science education, Information systems education, Self-assessment

General Terms

Experimentation, Algorithms, Human Factors.

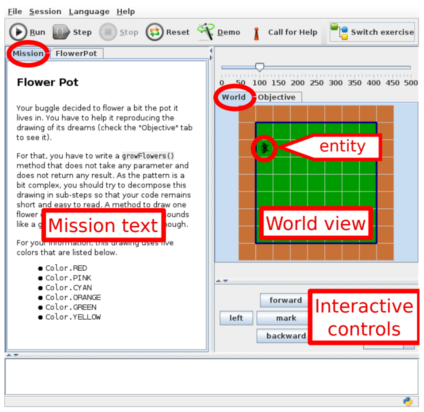
Keywords

CS1, Tools, Teaching Programming, Teaching Algorithms.

1. INTRODUCTION

The Programmer's Learning Machine (PLM) is a free integrated educational software environment aimed at learning and teaching programming. The main target user group is college and university students learning programming in loosely tutored practical sessions, but it can be used with younger pupils. It comes with 160 exercises covering the basics of programming, sorting algorithms and recursion.

Figure 1 shows PLM's main window when discovering a new exercise. It is composed of two main panels: The left one presents the mission text, that provides any information needed by the student to solve the exercise (ranging from the theoretical background and generic body of knowledge



Flower Pot

Your buggle decided to flower a bit the pot it lives in. You have to help it reproducing the drawing of its dreams (check the "Objective" tab to see it).

For that, you have to write a `growFlowers()` method that does not take any parameter and does not return any result. As the pattern is a bit complex, you should try to decompose this drawing in sub-steps so that your code remains short and easy to read. A method to draw one flower ounds like a g ough.

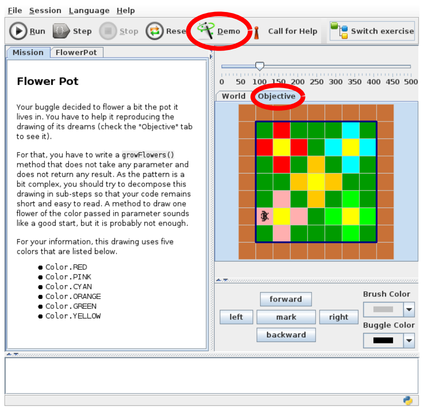
For you colors that are listed below.

- `Color.RED`
- `Color.PINK`
- `Color.CYAN`
- `Color.ORANGE`
- `Color.GREEN`
- `Color.YELLOW`

Figure 1: Discovering a new exercise in the PLM.

to practical details and hints) while right panel displays the current state of the exercise's world. Each world in PLM entails at least one active entity that executes the student code. Under the world view, interactive controls allow preliminary experiments with the exercise.

A very effective way to understand the exercise goal is to switch to the second tab of the world view as in Figure 2. It displays the world as it should be by the end of the exercise. Pressing the "demo" button starts an animation of the operations leading from the initial state to the objective. The



Flower Pot

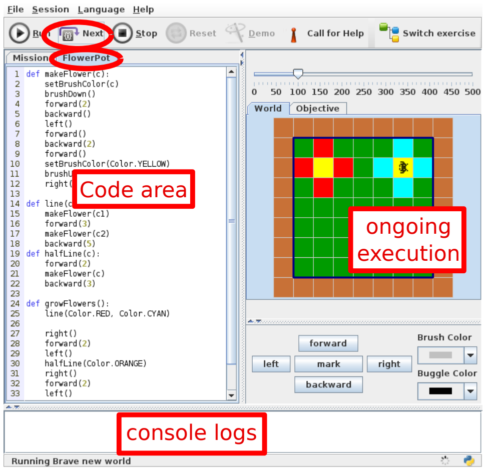
Your buggle decided to flower a bit the pot it lives in. You have to help it reproducing the drawing of its dreams (check the "Objective" tab to see it).

For that, you have to write a `growFlowers()` method that does not take any parameter and does not return any result. As the pattern is a bit complex, you should try to decompose this drawing in sub-steps so that your code remains short and easy to read. A method to draw one flower of the color passed in parameter sounds like a good start, but it is probably not enough.

For your information, this drawing uses five colors that are listed below.

- `Color.RED`
- `Color.PINK`
- `Color.CYAN`
- `Color.ORANGE`
- `Color.GREEN`
- `Color.YELLOW`

Figure 2: Understanding the exercise goal through the demo mode.



```
def makeFlower(c):
    setBrushColor(c)
    brushDown()
    forward(2)
    backward()
    left()
    forward()
    backward(2)
    forward()
    setBrushColor(Color.YELLOW)
    brushU
    right(

def line(c
    makeFlower(c1)
    forward(3)
    makeFlower(c2)
    backward(5)
def halfLine(c):
    forward(2)
    makeFlower(c)
    backward(3)

def growFlowers():
    line(Color.RED, Color.CYAN)

    right()
    forward(2)
    left()
    halfLine(Color.ORANGE)
    right()
    forward(2)
    left()
```

Figure 3: Writing and executing code.

speed of this animation can be changed at will.

To pass an exercise, the student must enter the code in the second tab of the left panel, as shown in Figure 3. Any potential compilation or execution error is displayed in the console located at the bottom of the window. The code can then be executed either continuously or step by step. If the world state does not matches the objective after the execution, an informative message explains the observed problem.

Exercises can be solved in either Java, Python or Scala, under the standard settings. The PLM uses the standard JVM with either the standard Java or Scala compiler, or the standard Jython implementation of python.

Empirical evidence show that this short and graphical feedback loop improves the students' motivation and make their learning more effective in practice. In addition, the PLM makes it easy for teachers to adapt the provided teaching material to their teaching goals, or to create new original material.

This paper is organized as follows: Section 2 discusses the design goals and rationals of the project. Section 3 situates the PLM within the state of the art. Section 4 shows the effectiveness of our educational tool through the presentation of the teaching material developed on top of it. Section 5 presents how teachers and resource authors can adapt and create more resources. Section 6 discusses some evidences of the tool usage. Finally, Section 7 concludes the paper and presents some envisioned future works.

2. DESIGN GOALS AND RATIONAL

The overall rational of the PLM is that programming a learn-by-doing activity. Given the major role of practice in the learning of programming, the PLM intends to be an automated exerciser allowing the student to practice with the programming tasks. The PLM tries to be useful to three categories of users, each of them mandating specific goals.

2.1 Toward Students

The PLM should facilitate autonomous learning and make programming engaging and satisfying. This major objective can be decomposed and detailed in several goals: Autonomous learning mandates *a consistent set of resources* with which the student can interact and experiment. Open-ended exercises enable free experimentation, but more guided exercises are precious to beginners that get easily paralyzed when facing choices. A *short feedback loop* provides a clear goal to the student's experimentation while an automated evaluation leading to quick success experiences reinforces the student's motivation. A *visual feedback* allows to focus on algorithmic concepts through the program effects, reducing the importance of syntactic elements. Finally, the tool should be *intuitive, easy to use and available* on most systems to avoid that technical details hinder the user experience. Along the same line, the environment must be translated and adapted to be usable by non-native English learners. This is particularly true for younger pupils.

Another important design goal of the PLM is to ease the transition to other programming contexts through the usage of standard tools and languages. *Several programming languages are proposed* in the environment, so that the choice is not forced on the user. Generic purpose programming languages are however known to be less adapted to novice programmers [2]. Java programs must for example use `public static void main(String[] args)` as an entry point, exposing beginners to advanced notions on day one [14]. This code can be provided to the students, but the extra code induce an extra cognitive burden that can drown the beginners. It is then preferable to *template the provided code and hide* the parts that are less relevant to the teachings.

2.2 Toward the authors of resources

The PLM should provide the technical support to ease the creation of innovative learning situations adapted to each context. In particular, it should be easy to build a new form of microworld providing a specific learning situation. In addition, the environment should provide all non-functional code, for example related to user interactions, the student's code compilation or the handling of persistent sessions. An integrated instrumentation infrastructure would allow to assess the efficiency of the developed pedagogical resources.

2.3 Toward the teachers

The environment should provide convenient and ready to use tools without hindering the practitioner's pedagogical freedom. It is expected that most teachers will leverage existing resources. This is similar to the classical use of existing books that is common in most teachings. Nevertheless, it should be easy for the teachers to adapt the existing resources and assemble them to build their own learning path on top of the existing resources.

3. CONTEXT AND STATE OF THE ART

The PLM builds upon the concept of Programming Microworld, introduced in the eighties through the LOGO programming language [11]. In such settings, the student controls an *entity* or actor that interacts with its *environment.* In the initial Logo microworld, the entity is a robotic turtle that leaves trails on a sheet of paper as it walks. Many systems build upon this idea, such as LogoBlocks [1], the Lego Mindstorm System or Roamer [5].

One drawback of the LOGO microworld is the scarce interactions between the entity and its environment: there is no way for the turtle to check the state of the sheet. Subsequent microworlds provide richer interactions. The most popular is certainly *Karel the robot* [12], where the actor, a robot named Karel, evolves in a world consisting of intersecting streets and avenues. It can pick and drop "beepers" (objects laying on the ground) but cannot pass the walls. This leads to richer interactions through predicates that test the presence of walls and beepers. Kara [8] is another microworld where a ladybird picks leafs while avoiding trees. Many other similar microworlds have proposed in the literature, as reviewed in [2].

In addition to such generic microworlds, specialized microworlds can introduce specific concepts. In [16], the authors leverage three microworlds: Most of the teaching are conducted with the *BuggleWorld*, a generic microworld with rich entity-environment interactions. Recursion is first introduced using the *PictureWorld*, that allows the construction of complex quilt patterns by combining simple shapes with basic transformations. Recursion is then further exercised through the drawing of polygons, spirals and trees in a LOGO-inspired microworld. This idea is pushed further in [3], where each exercise leverages a specific microworld. The limit of this approach is that the microworlds of both projects are developed separately, without any code reuse. The non-functional code then induces an important extra burden. By contrast, the PLM factorizes this code to ease the authoring of pedagogical resource.

Many other similar environments have been proposed in the literature. In the taxonomy of Kelleher and Pausch [9], the PLM is within the category *Teaching System/Mechanics of Programming*. It covers several sub-categories: Simplify Typing Code; Making New Models Accessible; Tracking Program Execution and Make Programming Concrete.

Scratch, Alice and Greenfoot are very renowned educational projects to teach and learn programming. Scratch [13] enables kids to develop interactive stories and animations. Since scripts are constituted of building blocks that are visually assembled, it is accessible to kids even before they master reading and writing. The Alice [6] environment learn older pupils about object-oriented programming through the design of scripted 3D animations. Greenfoot [10] uses the standard Java language for a similar goal.

The biggest difference between the PLM and these projects consists in the ability to propose auto-evaluated resources. Scratch, Alice and Greenfoot can be considered as IDEs that are specifically tailored to learners. In the PLM however, teachers can prepare specific exercises on which the students work autonomously, at their own pace. Both approaches are complementary: learners could first start with guided resources on the PLM to learn the concepts they need, and then move to an educational IDE to build their own projects using these concepts.

4. EXISTING TEACHING MATERIAL

This section presents the teaching resources developed on top of the PLM and distributed with the environment itself. The main goal of this discussion is to demonstrate the effectiveness of the PLM as an educational environment. The presented resources are interesting per see, even if we have no formal evidence of their intrinsic effectiveness. Many of them were previously existing in the literature. The practical effectiveness of the PLM (and to some extend of these resources) is discussed in Section 6.

4.1 The PLM's Universes

Since the PLM entails several kind of microworlds, we refer to each family of microworlds as an *universe*. We now detail the existing universes, focusing on their pedagogical features and on their typical use in exercises.

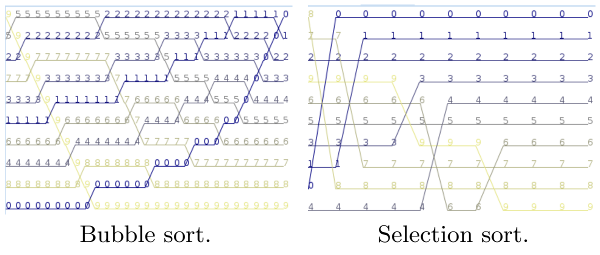

Bubble sort. Selection sort.

Figure 4: Temporal view of the sorting world.

The Buggles Universe.

This generic universe was originally the only existing one in the PLM. It relies on an original idea of Franklyn Turbak, at Wellesley College [16]. It features *the buggles*, that are able of many interactions with their environment: they can pick and drop objects, paint the ground, read and write messages on the ground, hit walls, etc.

Thanks to its versatility, this universe is used in many proposed exercises. The space invaders that depict the buggles became the mascot of the PLM project with the years.

The Turtles Universe.

This universe constitutes a clone of the classical LOGO microworld. It is used in two of the proposed lessons. In the recursive lesson, the students are asked to draw recursively polygons, spirals, trees and fractals. The "turtle art" lesson presents several classical LOGO drawings to inspire the students, that are expected to then play in creative mode where their solution is not checked against any objective.

Unit testing universe.

This universe is very specific as it does not provide any graphical representation. Each exercise proposes a method prototype that the students must fill. This method is then tested against a comprehensive test of parameter values. These exercises, inspired from the work of N. Parlante[1], are probably less motivating than graphical microworlds, but they are efficient to practice on a particular point. The introductory lesson provides for example short exercises on conditionals where the students must write simple boolean expressions matching the provided textual descriptions.

Sorting Algorithms.

This original universe targets the exploration of the sorting algorithms. The demo mode can be used at full speed to get a practical idea of the differing algorithm complexities, or to understand the algorithm behavior step by step.

Interestingly, it is not sufficient to sort the provided array of data to pass a typical sorting exercise, but the proposed solution must access the data exactly the same amount of time than the expected solution. This is enforced through adapted primitives that mediate and count any data access. Using `isSmaller(i,j)` to compare cells does account for two reads. Using `copy(i,j)` or `swap(i,j)` to modify the data accounts respectively for one read plus one write, or for two reads plus two writes. This constraint on the amount of

[1] `http://codingbat.com/`

data accesses forces the student to strictly implement the expected algorithm, which in turn requires a very good level of understanding of the presented algorithms.

When the amount of data accesses does not match, understanding the difference between the proposed code and the expected solution can reveal difficult. To ease this process, the student can graphically explore the history of their sorting algorithm, as shown in Figure 4. The time is represented as abscissa while ordinate represent the array cells. The lines represent the values in these cells; When two lines cross, this means that two values were swapped at this timestamp. This representation, introduced by A. Cortesi[2], reveals precious to understand the behavior of sorting algorithms.

The Rainbow Baseball.
This universe applies the classical sorting algorithms in a funny and original setting. It builds upon the Pebble Motion Problem [4], where objects must be placed on the right vertex of a graph under several movement constraints. This problem is similar to the 15-puzzle problem. This engaging universe can be played by clicking on the pebbles to move them, but the provided lesson require the student to reimplement the major sorting algorithms. A temporal view allows to compare the proposed code with the expected one.

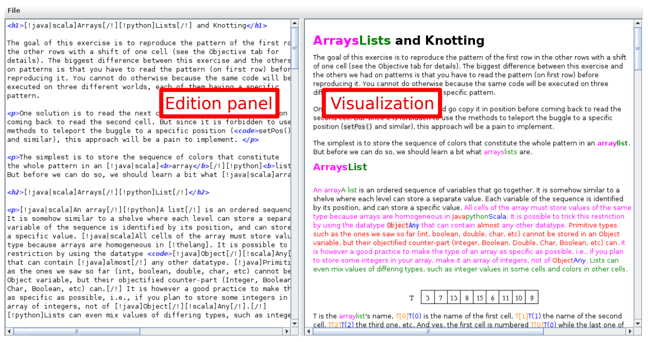
The Pancakes Universe.
This universe builds upon the classical pancake problem, where a stack of pancake should be sorted only by flipping pancakes at the top of the stack. The corresponding lesson entails several existing algorithms to solve this problem, some of them being non-trivial. A funny fact is that one of these algorithms was initially published by Bill Gates before he invented Windows [7].

4.2 Proposed Learning Path

The exercises are not isolated in the PLM, but grouped by thematic lessons providing a coherent progression on each topic. This is particularly important to self-learners using the unmodified tool, even if the teachers are welcome (or even expected) to adapt this path to their settings.

The first lesson introduces the basic concepts of programming to absolute beginners. After a quick tour of the environment, the notion of instructions is presented through very simple Buggle exercises. The lesson then introduces conditionals and while loops. The concepts are presented as part of simple exercises and reinvested immediately in several application exercises that leverage the universes' versatility. An extra complexity comes from the fact that the pattern matching of Scala is more powerful than the switch-case construct of Java, while Python offers no switch construct. The mission text are then dynamically adapted to the chosen programming language so that the right concept and syntax are presented to the student.

Variables are the next introduced concept, before for loops and do-while loops. In python, we present the idiomatic way toward do-while loops since this construct does not exist in this language. 20 exercises are proposed for these notions, 15 of them being engaging and situated application exercises that follow uncluttered introduction exercises presenting the concepts. Some exercises induce more complex loop settings, with non-trivial loop conditions, limit cases and modeling



```
<h1>[!java|scala]Arrays[/!][!python]Lists[/!] and Knotting</h1>

The goal of this exercise is to reproduce the pattern of the first ro
the other rows with a shift of one cell (see the Objective tab for
details). The biggest difference between this exercise and the others
on patterns is that you have to read the pattern (on first row) befor
reproducing it. You cannot do otherwise because the same code will be
executed on three different worlds, each of them having a specific
pattern.

<p>One solution is to read the next c                               on
coming back to read the second cell. But since it is forbidden to use
methods to teleport the buggle to a specific position (<code>setPos()
and similar), this approach will be a pain to implement. </p>

<p>The simplest is to store the sequence of colors that constitute
the whole pattern in an [!java|scala]<b>array</b>[/!][!python]<b>list
But before we can do so, we should learn a bit what [!java|scala]arra

<h2>[!java|scala]Arrays[/!][!python]List[/!]</h2>

<p>[!java|scala]An array[/!][!python]A list[/!] is an ordered sequenc
It is somehow similar to a shelve where each level can store a separa
variable of the sequence is identified by its position, and can store
a specific value. [!java|scala]All cells of the array must store valu
type because arrays are homogeneous in [!thelang]. It is possible to
restriction by using the datatype <code>[!java]Object[/!][!scala]Any[
that can contain [!java]almost[/!] any other datatype. [!java]Primiti
as the ones we saw so far (int, boolean, double, char, etc) cannot be
Object variable, but their objectified counter-part (Integer, Boolean
Char, Boolean, etc) can.[/!] It is however a good practice to make th
as specific as possible, i.e., if you plan to store some integers in
array of integers, not of [!java]Object[/!][!scala]Any[/!].[/!]
[!python]Lists can even mix values of differing types, such as intege
```

ArraysLists and Knotting

The goal of this exercise is to reproduce the pattern of the first row in the other rows with a shift of one cell (see the Objective tab for details). The biggest difference between this exercise and the others we had on patterns is that you have to read the pattern (on first row) before reproducing it. You cannot do otherwise because the same code will be executed on three diff specific pattern.

On d go copy it in position before coming back to read the second cell. But since it is forbidden to use the methods to teleport the buggle to a specific position (`setPos()` and similar), this approach will be a pain to implement.

The simplest is to store the sequence of colors that constitute the whole pattern in an **arraylist**. But before we can do so, we should learn a bit what arrayslists are.

ArraysList

An arrayA list is an ordered sequence of variables that go together. It is somehow similar to a shelve where each level can store a separate value. Each variable of the sequence is identified by its position, and can store a specific value. All cells of the array must store values of the same type because arrays are homogeneous in JavapythonScala. It is possible to trick this restriction by using the datatype `ObjectAny` that can contain almost any other datatype. Primitive types such as the ones we saw so far (int, boolean, double, char, etc) cannot be stored in an Object variable, but their objectified counter-part (Integer, Boolean, Double, Char, Boolean, etc) can. It is however a good practice to make the type of an array as specific as possible, i.e., if you plan to store some integers in your array, make it an array of integers, not of ObjectAny. Lists can even mix values of differing types, such as integer values in some cells and colors in other cells.

T	3	7	13	8	15	6	11	10	9

T is the arraylist's name, T[0]T(0) is the name of the first cell, T[1]T(1) the name of the second cell, T[2]T(2) the third one, etc. And yes, the first cell is numbered T[0]T(0) while the last one of

Figure 5: Mission text editor.

task to identify the repeated instructions, as advised in [15].

Functions are then introduced, followed by methods and parameters. Functional decomposition is introduced through several exercises requesting to repeat a given pattern an increasing amount of time. About 15 exercises introduce and apply these concepts. Most of our students need 4 to 6 hours to reach the end of this unit.

About 40 unit testing exercises to practice with writing of simple conditionals take place at this point of the progression. They could theoretically be placed earlier, but our unit universe requires the student to know about methods. Sequence of elements (arrays in Java and Scala, lists in Python) are then explored through 18 exercises.

This concludes the introductory lesson, that is usually completed by absolute beginners at university level in 10 to 20 hours. It can be followed by two application lessons: the first one quickly explores classical maze algorithms while the other introduces a simple form of cellular automaton called Langton's ant. This is easy to implement for the students, and leads to interesting drawings.

After the language syntax and basic constructs introduction, we move on to more algorithmic lessons. One lesson explores the classical sorting algorithm, while two other lessons situate these algorithms in the engaging microworlds presented in previous section. Another lesson explores recursion through logo-based figures.

5. EXTENDING THE PLM

The teachers can reuse the proposed material as-is, or they can adapt and extend it. One current limitation of the PLM is that such adaptation can only be done in Java for now, forcing the teachers to have some notions of this language even if they teach Scala or Python.

5.1 Creating new lessons

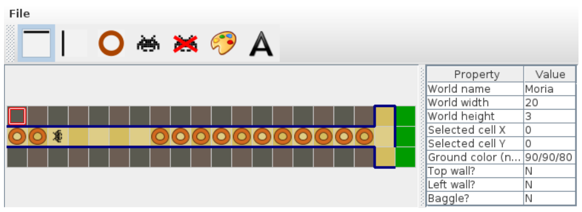
The easiest PLM adaptation is to create a lesson that builds a new learning path from existing resources. This is trivially done by building a Java collection of the selected resources during the lesson's initialization. This could be further simplified in the future by using json configuration files instead of Java code.

5.2 Creating new exercises

A typical exercise is composed of the mission text, a description of the initial environment's state, and a correction entity implementing the requested work.

The mission text is an HTML file, possibly with condi-

[2]http://corte.si/posts/code/visualisingsorting/



Property	Value
World name	Moria
World width	20
World height	3
Selected cell X	0
Selected cell Y	0
Ground color (n...	90/90/80
Top wall?	N
Left wall?	N
Baggle?	N

Figure 6: Buggle world map editor.

tional inclusions to adapt the mission to the used programming language. Since such conditional texts tend to be intricate and burdensome to edit, the PLM provides an adapted text editor (Figure 5). The text is edited in the left panel and previewed in the right panel. A color code depicts graphically the parts that are specific to each programming language. The mission text can also entail optional *hints*, that are shown only on explicit student request.

One way to build the initial state of the microworld is through a dedicated Java class, which complexity naturally depends on the world to instantiate. A graphical editor is provided for the buggle worlds, that can become rather complex. This intuitive tool is depicted in Figure 6.

Most exercises test the students' code on multiple instances of the problem to detect more errors. From the author perspective, this simply requires to attach several world instances to the exercise.

The correction entities have several roles: they are applied to the initial worlds to compute the objective worlds; They are executed when playing the demo, and they provide an hosting scaffold around the student's code chunk to produce valid files. This last usage requires to automatically edit the entity. To this end, the entity source code must be annotated as shown in Figure 7. When compiling the student's code, the section marked with BEGIN/END TEMPLATE is removed and replaced with the student code while the other parts are left unchanged. Once the BEGIN/END SOLUTION section is removed, the BEGIN/END TEMPLATE section is also used as initial content in the source code editor. Here, this mechanism preseeds the editor with the function's prototype that must be filled by the student.

Since they are used as a scaffold around the student's code, it is necessary to write one correction entity in each programming language for each exercise.

```
package lessons.welcome.methods.slug;

import java.awt.Color;
import plm.universe.bugglequest.SimpleBuggle;

public class SlugTrackingEntity extends SimpleBuggle {
  public void run() { // called when executing the entity
      while (! isOverBaggle()) {
         if (isFacingTrail()) {
            brushDown();
            forward();
            brushUp();
         } else {
            left();
         }
      }
      pickupBaggle();
   }

   /* BEGIN TEMPLATE */
   boolean isFacingTrail() {
      // Write your code here
      /* BEGIN SOLUTION */
      if (isFacingWall())
         return false;
      forward();
      boolean res = getGroundColor().equals(Color.green);
      backward();
      return res;
      /* END SOLUTION */
   }
   /* END TEMPLATE */
}
```

Figure 7: Example of annotated entity source code.

5.3 Designing a New Microworld

Adding a radically new learning context to the PLM requires to extend the classes defining an universe: the *World* class contains the microworld state and data; the *Entity* class is the ancestor of all correction entities. This class is mainly responsible to adapt the World interface and provide to the user convenient primitives to alter the world. The *WorldView* provides a graphical representation of a given World. Any universe should also be documented through an HTML following the same conventions than the mission texts of exercises. Finally, most universes should entail a *WorldPanel* class that provides the entities' interactive controls. The buggle universe also provides a graphical editor.

Universe	Amount of lines
Buggle	1400 (+600 for the editor)
Sorting	900
Baseball	900
Tuttle	700
Panecake	500
Units	500
Hanoi	200

Table 1: Complexity of each universe.

Table 1 presents the code complexity of each universe. The numbers remain surprisingly low despite the wealth of the proposed universes, proving the effectiveness of the code factorization enabled by the PLM.

6. TOOL ADOPTION

We use the PLM since five years in our institution. Thanks to this tool, absolute beginners become quickly confident in the programming syntax, allowing to take on more advanced content in the same time than before its introduction.

We never publicized the PLM until this first publication. The environment was only spread by word of mouth so far, we are not tracking the amount of downloads from our web page and we did not even have a users' mailing list until recently. We know that it is or was used for practical sessions and proposed as a complement to the regular teaching in a few institutions within our personal social network.

The only quantified estimations that we have come from an activity tracker added to the PLM in 2011 as a preliminary gamification feature. This tracker twits automatically when the student passes an exercise, spreading the good news to the world. This feature can easily be opted-out to respect the student's privacy, so the reported values constitute lower bounds. Since the tracker inception, we know that 12,000 exercises were solved while we estimate that our own students produced only between 4,000 and 8,000 tweets. We thus suspect that the PLM is used by other institutions and individuals who freely downloaded the project without keeping us informed.

7. CONCLUSION AND FUTURE WORK

In this paper, we described the Programmer's Learning Machine, an integrated educational software dedicated to the learning and teaching of programming. It is designed for users with no or little programming experience and aims at providing practice in a friendly and attractive environment. An extensive but coherent set of exercises is presented to the students, that can progress at their own pace. Several microworlds are leveraged, some of them specialized in specific programming concepts. The exercises can be solved using either the Java, Scala or Python programming language, using the standard compiler and implementation. The tool and all teaching materials are fully translated to English and French. The proposed lessons entail over 160 exercises, covering the imperative kernel (basic syntax and control structures), sorting algorithms and recursion.

The PLM is freely available from the project's web page[3], under an open-source license (GPL and CC-BY-SA). It is known to work on Windows, Linux and MacOS. We are now actively fostering the emergence of a user community, in the hope that others will enjoy this project.

Future work naturally entails the extension of the proposed teaching material and the addition of new microworlds to explore other concepts such as backtracking, dynamic programming, object-oriented programming or others. We would also like to introduce an simple instrumentation infrastructure that could be used to quickly detect the students requiring the teacher's help. Such infrastructure could also be used to study the effectiveness of specific teaching strategies and approaches. This will enable formal studies of the apparatus efficiency that will in turn help improving further this educational environment.

8. REFERENCES

[1] Andrew Begel. Logoblocks: A graphical programming language for interacting with the world. Master's thesis, Massachusetts Institute of Technology, 1996.

[2] Peter Brusilovsky, Eduardo Calabrese, Jozef Hvorecky, and Philip Miller. Mini-languages: A way to learn programming principles. *Education and Information Technologies*, 2(1):65–83, 1997.

[3] Nadya Calderon, Jorge Villalobos, and Camilo Jimenez. Developing programming skills by using interactive learning objects. In *14th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE'09)*, pages 151–155, Paris, France., July 2009. ACM.

[4] Gruia Calinescu, Adrian Dumitrescu, and Janos Pach. Reconfigurations in graphs and grids. In *LATIN 2006: Theoretical Informatics*, volume 3887 of *Lecture Notes in Computer Science*, pages 262–273. Springer Berlin Heidelberg, 2006.

[3]http://www.loria.fr/~quinson/Teaching/PLM/

[5] Dave Catlin. The roamer robot, 1989. Valiant Technologies.

[6] Matthew Conway, Steve Audia, Tommy Burnette, Dennis Cosgrove, and Kevin Christiansen. Alice: lessons learned from building a 3d system for novices. In *CHI '00: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 486–493, New York, NY, USA, 2000. ACM.

[7] William H. Gates and Christos H. Papadimitriou. Bounds for sorting by prefix reversal. *Discrete Mathematics*, 27(1):47 – 57, 1979.

[8] Werner Hartmann, Jurg Nievergelt, and Raimond Reichert. Kara, finite state machines, and the case for programming as part of general education. In *Proceedings of IEEE Symposium on Human-Centric Computing Languages and Environments*, page 135, Stresa, Italy, 2001. IEEE Computer Society.

[9] Caitlin Kelleher and Randy Pausch. Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Computing Survey*, 37(2):83–137, 2005.

[10] Michael Klling. The Greenfoot Programming Environment. *ACM Transactions on Computing Education (TOCE)*, 10(4):21, November 2010.

[11] Seymour Papert. *Mindstorms: Children, Computers, and Powerful Ideas.* Basic Books, 1980.

[12] Richard Pattis. *Karel the Robot: A Gentle Introduction to the Art of Programming with Pascal.* John Wiley and Sons, London, 1981.

[13] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and Yasmin Kafai. Scratch: Programming for all. *Communications of the ACM*, 52(11):60–67, nov 2009.

[14] Eric Roberts, Kim Bruce, Kim Cutler, James Cross, Scott Grissom, Karl Klee, Susan Rodger, Fran Trees, Ian Utting, and Frank Yellin. The acm java task force project rationale. Technical report, ACM, 2006.

[15] Mara Saeli. *Teaching Programming for Secondary School: a Pedagogical Content Knowledge Based Approach.* PhD thesis, Technische Universiteit Eindhoven, 2012.

[16] Franklyn Turbak, Constance Royden, Jennifer Stephan, and Jean Herbst. Teaching recursion before iteration in CS1. *Journal of Computing in Small Colleges*, 14(4):86–101, may 1999.