NEWTJAVA: LEARNING BASIC ALGORITHMIC PRINCIPLES WITHOUT PAIN

Patricia Fernández Garrido (1), Jose Jesus García Rueda(1), Raúl V. Ramírez Velarde(2)

(1) Carlos III University of Madrid
Madrid, Madrid, (Spain)
(2) I.T.E.S.M.
Monterrey, Nuevo León, (Mexico)
rueda@it.uc3m.es

Abstract

In this paper we introduce NewtJava, a tool we have recently developed for the teaching and learning of basic algorithmic principles, to be used by students without previous experience in programming. For many years now, university teachers of courses on programming have been complaining about their student's bad performance when learning elementary programming concepts. Students manifest serious difficulties for the understanding and use of basic flow control structures in programs, the assimilation of data types and the decomposition of simple problems into computer-oriented instructions sequences, for instance. In response to these problems we have develop a teaching software tool that combines the three fundamental characteristics of the programming practice into an easy to use interface: abstract (algorithmic) thinking, language independent implementation and dynamic behaviour observation. And this without involving the intricacies of a programming language. As explained in the following paper, the potentialities of this tool and its pedagogical grounds are immense.

Keywords

Programming teaching, visual programming, algorithmic principles, multimedia teaching, interactive learning.

1. INTRODUCTION: ANOTHER WAY OF LEARNING ALGORITHMICS

When forming programming students, their future performance and the quality of the programs these students will develop lately depends largely on their first contact with the world of programming and on their initial personal experiences on it. Did they feel comfortable when they had to program a computer for the first times? And what is more important, did they learn the basic concepts properly? In accordance with constructivism theory, pupils get in their early experiences basic "thinking objects" (concepts, intuition, related logic, ways of thinking…) to begin to reason in that new discipline; objects to build their own knowledge. As Papert says in [1]: "Learning laws must refers to the way intellectual structures appear one from another and how, in that process, these structures take logical and emotional form". Or, in other words (Piaget's words): "Understanding in the learning process must be general".

But our students do not seem to be good builders, in terms of programming concepts, as it can be noticed if we evaluate overall academic results. They have serious difficulties designing really simple algorithms (conceiving the abstract solution to a problem and translating it into computable code), or using the basic general flow control structures. And that, after having studied a complete course on programming. So, what's the matter? What are the causes of all those problems students find when learning to program? After developing several studies in which we tested our students on basic algorithmic principles, we concluded that most of the problems they find are due to the confusing way in which concepts are commonly presented to them. Nowadays, students learn programming by directly studying a specific programming language. That way, they have to deal with a complex syntax, and many other difficulties (like compiling and linking errors, pre-processor instructions or functions definitions) right from the beginning, making it almost impossible for them to identify what concepts are general programming concepts (like algorithms), and which ones are exclusively language/computer-oriented. Moreover, if the programming language they are exposed to is object-oriented (such as the useful and popular Java), complexity increments highly, and students are unable to focus their attention in the fundamentals of the programming techniques because they simply cannot find them among that whole lot of things to take care of at the same time: classes, instances, methods and so on.

Because of all the exposed above, we decided to change the way our students learn to program. First of all, students must focus on understanding the natural dynamism of a computer, as we would say: "the discrete nature of the computer". It means that the student must transform his continuous and heuristical thinking flow (his human way of thinking, something like: "I know but I didn't know how I knew") into a set of elemental and ordered steps that conform together the solution to a problem (a computer way of thinking). After that, he should learn the general flow control structures belonging to all programming languages (in its abstract form). According to this, we can have our students work just with pseudo-code, explaining them the algorithmic principles with an almost natural language, at least at the first stages of learning. This is the traditional way to solve the problem. However, it lacks a feature quite interesting and desirable in the learning process nowadays: interactivity.

When writing programs with pseudo-code, a student cannot make any change in a given algorithm and then check the new functionality, nor can create algorithms from scratch and test them. That does not seem a very sound way of teaching such a dynamic activity as programming, where professionals make their routines, subroutines and methods work by continuously changing them till they are correct. That kind of process is not available when using a piece of paper and a pencil. If we wish interactivity, we need to pay attention to the paradigm of the interactive devices nowadays: the computer. Following this line, there have been deployed some systems based on new programming languages created deliberately for teaching purposes, such as Juan Segovia's SL language [2]. In the system he proposes, students can interact with the computer writing, compiling and running their programs, in an environment carefully designed to suppress the most common difficulties of general-purpose languages. Despite that, it is yet necessary the student to learn a specific syntax (although it is relatively simple). We believe that syntax, as far as possible, should not be present in the early stages of learning to program. Syntax should be secondary if we wish our students not not to be distracted from the learning of essentials concepts: first I understand what a table is, and then I call it "table".

Another premise to take care of is the achievement of an appealing and intuitive learning process (in order to encourage the affective side of that process). It is not necessary to insist in the benefits of making students learn while playing. For many years now, many types of games have been used in training our mind so we could better solve specific kinds of problems. Also some kinds of games have been used in programming. This is the case of the experience carried out by Anany Levitin and Mary-Angela Papalaskari, using puzzles in teaching algothimics [3]. In their experiences, students were proposed to solve games that were quite similar to classic programming problems, like the famous "Hanoi towers" that illustrates the recursion concept. Of course, these smart games keep the students quite interested… at least in the game itself. But, can a beginner with no perspective at all notice the similarity between the game and the computer-oriented algorithm that the teacher actually intends to convey to him? This method helps greatly to someway clarify ideas to an introduced student, but it doesn't smooth the first contact between a beginner and the computer. Although it could make the students decompose their thinking into discrete steps, it does not improve the assimilation of what we called previously "computer way of thinking": telling orders to a computer, in a computer's way. Anyway, it is interesting to maintain this playing-oriented way of learning as far as possible. Trying that, and trying also to take advantage of the customs of a generation that is used to interact with computers graphically, we decided to make use of the visual programming principles (like Glinert's PICT system [4], for instance). We decided to make our students program in a flow diagram mode using a graphical environment. In our application however, visual programming is absolutely oriented to the learning of a classical imperative programming language, and, as a consequence, it is much simpler than a general purpose visual programming language.

As it will be explained in the next sections, NewtJava aims to cover the teaching and learning of the most basic aspects of the programming techniques, keeping always in mind the ideas introduced previously.

2. NEWTJAVA: LEARNING ALGORITHMIC PRINCIPLES VISUALLY

As seen in the introduction, interactivity and visual programming (since it makes the student feel like playing with the computer, someway) are the two fundamental aspects of our project.

It would be great if we could "program" a computer without actually writing dozens of commands. It would be even better if we could program without writing at all. And the best of all would be to have the chance of modifying our work once and another, testing the results of each change, and that way learning the actual way in which programming problems are commonly solved. That is the aim of NewtJava. With this tool, students can develop a program in a graphical way, as the basic instructions and flow control structures needed to solve

elemental problems are available as icons to be dragged and dropped to the main frame of the application, where the program is constructed by assembling these atomic pieces.

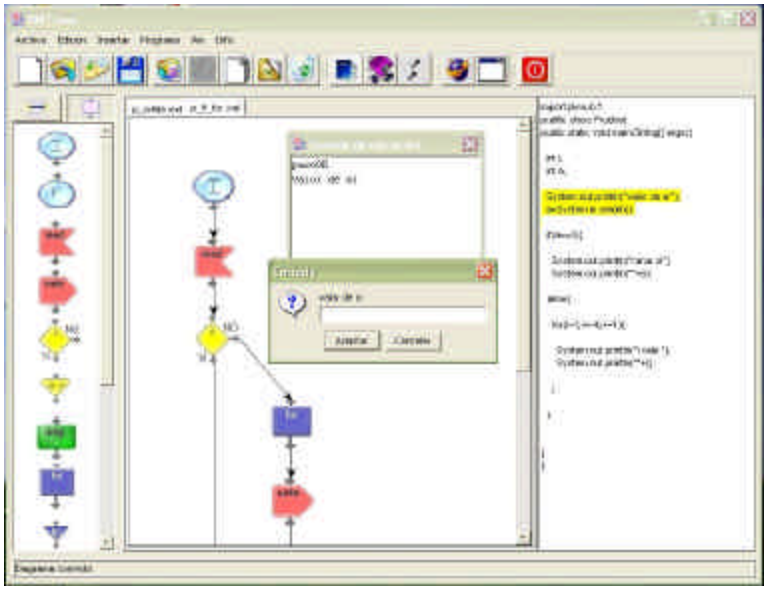

Fig. 1. Graphical user interface of NewtJava.

As we can see in Figure 1, the working area of the application is divided into three sections: a repository of programming elements to the left, a blackboard to draw the flow diagram in the middle, and a section for the translation of the diagram into code to the right.

Students program by drawing flow diagrams, and can keep several of them simultaneously open. Elements in the repository can be dragged and dropped into the central drawing area. Once the elements for a specific program are chosen, the program structure is defined by connecting each element to another, using arrows that highlight the sequential nature of the instructions. In the drawing area, the classic graphics operations like cutting, pasting, undoing, redoing, selection, etc… are available to the student, so the construction of the graph is dynamic and friendly. Each icon representing an instruction or flow control structure can be edited independently to set its properties appropriately (like the counting limits in a FOR statement). This way, the student can deploy his solution without worrying about syntax issues; moreover, the programs are validated by the tool indicating the user if there are structural errors (a WHILE without an END OF WHILE, for instance). That way the student does not have to deal with a compiler. These two points allow the user to think only in terms of the algorithm he is designing. After, and only after, the completion of the algorithm, it is about time to transform it into real code, in order to map in the mind of the student the algorithmic concepts into the actual syntax of a chosen programming language. In our tool, for the time being, the language the diagrams are translated to is Java. Despite this, the application's design allows for the introduction of extensions to other languages in an extremely simple way: the translation functionality is encapsulated in a piece of code (implementing a concrete interface) that only generates separately the string corresponding to each instruction; any other logic needed in the translation is assumed by the application. This way, writing a translator to any other language is trivial.

As a final step in the learning process, the student can simulate the running of the developed code either straight to the end or in a step-by-step basis, allowing the student to test his work and, as a consequence, allowing for feedback (Figure 1). The running of a program is always managed by the application; for example, when the program running requires some data from the user, the application shows a dialog window that asks for it. This application's control over the user's programs allows also for the stopping of an infinite loop in case it would appear (a typical novice error). The red button in the top right implements that functionality.

The lines of code highlighted are the ones being interpreted at the moment. The result of this interpretation is shown to the user in the Running Console (also in the figure).

At anytime, the student can save the program he is developing in a text file (structured as an XML file) or open another program previously saved.

The design of the interface was a milestone for our work, actually: it had to be easy and familiar, appealing, clear and, most of all, easy to learn For example: all the buttons in the application and the elements in the repository have their own tip text tool with a brief description of it. We wanted our students to focus on the algorithms they were deploying, without the cognitive overload of a complex interface.

Summing up, the main idea when creating NewtJava was to obtain a learning tool both cognitively sound and, at the same time, constructivism-based: the learner constructs his own knowledge by interacting once and another with a cognitively attractive and efficient system. In the next section, a couple of experiments are described, in order to show the appropriateness of this model.

3. THE EXPERIMENTS

Once the first version of NewtJava was ready, we proceeded to test it with real students. This testing phase was divided into two steps:

- The technical evaluation.
- The pedagogical evaluation.

One experiment was realized during each step. For the first one we organized a one session experiment with a group of twenty students, all of them enrolled in a course in advanced programming. The fact that these students had a several months long programming background was an advantage at the moment: they conformed a set of experienced beta-testers, able to put our tool under a considerable stress. They were exposed to a brief tutorial and then proposed a couple of problems to be solved with NewtJava. Our aim here was to obtain a first feedback on the suitability of our philosophy, and at the same time to find out some minor mistakes and desirable improvements in the tool. About the former, the program was greatly appreciated by the learners, who found it appealing and useful (many of them told us that they longed for a tool like this several months before, when they were exposed to programming principles and basics for the first time). On the latter, their help was really valuable, as they discovered a couple of important mistakes to be corrected, and proposed several hints to improve the user interface (for instance, thanks to their observations we realized of the need of including the "Stop Running" button previously mentioned)

But from a pedagogical viewpoint, the second experiment, the one in the second step of the testing phase, was the most important. Once the graphical user interface and the main functionality of the tool had been put under test, it was time to evaluate the effect the use of NewtJava could have in our students. As a consequence, we planned a several weeks long experience with students whose algorithmic skills were quite deficient. In fact, they were not completely newcomers in the world of programming, as they were supposed to know the basics of Java language, but in practice we could verify that they were a good example of the kind of learning one can expect when students are taught syntax, object oriented programming and basic algorithmics at the same time. In order to evaluate more accurately the skills of our students, we made them solve an initial test, composed by several simple exercises and problems. This exercises had an increasing difficulty level, and were conceived to cover all of the flow control structures and the most common, and basic, problems to be solved when learning to program: comparing variables, reading arrays, writing on the screen the same sentence several times... They were encouraged to answer the test using pseudo-code, to avoid the complexities of Java. As expected, the results were quite discouraging: almost all of the students showed a worrying lack of capability to solve even the easiest exercises: for instance, when asked to solve a problem whose natural solution was a nested-IF structure, they answered with a FORs-based structure, being clear they did not understand properly the iterative nature of a FOR loop. They seemed to lack an algorithmic way of thinking.

With these results very present, the second part of the experiment began. We organized a course in basic algorithmics, to be taught by using NewtJava. After an initial face-to-face session in which the students were introduced to the software, everyday we proposed them a new exercise to be solved with the tool, providing the solution the very next day. Again, as in the initial test, exercises were thought to cover every aspect of basic algorithmics, and were organized according to a sequence of increasing difficulty. Apart from this work at a distance, we had a two-hour face-to-face session with the students every week, in which we answered the questions we could not answer previously on the email, and then proceeded to propose and solve some exercise, a little more difficult than the ones proposed during the rest of the week.

After several weeks working like this, the improvement in the algorithmic skills of our students was evident and notorious: now they could face and solve programming problems without that feeling of "being lost in the

forest" they showed before the experience with NewtJava. What is more relevant, they felt self-confident, and able to test their recently acquired skills with "real" programming. They found the tool useful and convenient, even though they detected some unnecessary complexities in the use of the tool.

After the completion of the experiment, and feeling reaffirmed on the usefulness of the tool, we are now doing some improvements in the graphical user interface, taking into account our students' suggestions.

4. CONCLUSIONS AND FUTURE WORK

So, it is about time to evaluate the overall results of this work from a more abstract point of view.

Following our theory about how programming should be taught to beginners (in a constructivism-based way), we have developed a tool that facilitates the learning of the essence of programming. In the designing process, besides the pure teaching methodology aspects, we tried to involve some emotional aspects of the learning process (related to the way our students live their learning experience), so we based our application in three basic principles:

- Interactivity without the inconveniences of a typical developing environment.
- Focus on the algorithmic aspects of programming: from the discrete and sequential nature of computers, to the basic flow control structures.
- An appealing and easy to use graphical user interface, in line with the actual tastes and customs of the new generations of students.

These three points make a satisfactory first contact between students and the word of programming easier. In the experiences described above, we found that students with some previous experience in programming improved clearly their comprehension and performance in that subject; so we conclude that such a nice "first contact" would help our students without any previous experience to quickly (much more quickly that with the nowadays methods) understand programming, and to set the bases of a solid and lasting knowledge.

This tool encourages, right from the beginning, a way of conceiving programming as a means to solve problems in a structured fashion. At the same time, it gives the student all the responsibility of taking those elements he is provided with and playing with them to construct complex structures and interact with them, changing them dynamically. Because of all of this, NewtJava is an ideal tool to introduce beginners to programming. Moreover, we believe that it would be advantageous if kids had access to the application, in order to be introduced in such a specific way of thinking as soon as possible, as it is a convenient way of thinking for Mathematics, for instance.

Despite all the described above, NewtJava is yet in an early stage of development. Its functionality is limited, and could be sensibly improved by adding some features like arrays support, functions and methods support or scope of variables. Of course, we intend to implement these functionalities in the near future.

Another important task to do is the planning and realization of new experiments with the application (including some with younger students –not university students-) in order to gain a deeper knowledge into the potential uses of NewtJava as a learning tool.

To sum up, we firmly believe that NewtJava follows a very sound and promising path into the teaching of programming, so… let's Newt!

References

[1] Papert, S., Mindstorms: Children, Computers, and Powerful Ideas, Basic Books, 1980.
[2] Segovia, J., "SL: un Lenguaje para la introducción a la Algoritmia", *Proceedings of the XXV Latinoamerican Conference on Computer Science*, 1999.
[3] Levitin, A., Papalaskari, M., "Using Puzzles in Teaching Algorithms", Proceedings of the 33rd SIGCSE Technical Symposium on Computer Science Education, 2002.
[4] Glinert, E., "PICT: An Interactive Graphical Programming Environment", *Computer*, IEEE Computer Society Press, Los Alamitos, CA, USA, Volume 17, Issue 11, November 1984, pages 7 – 25.