# The Gobstones method for teaching computer programming

Pablo E. Martínez López, Daniel Ciolek, Gabriela Arévalo, Denise Pari
DCyT - Universidad Nacional de Quilmes
Email: {fidel, dciolek, garevalo, dpari}@unq.edu.ar

***Abstract*—Within the context of Argentina universities that have computer science degrees in their curricula, one of the major challenges is to teach an introductory programming course for the first semester. The problem appears because the students come with serious deficiencies from their previous education level (secondary schools). Among these deficiencies, we identify their lack of abstraction capabilities and mathematical skills, which are usually considered as prerrequisites to take a programming course.**

**This article introduces the GOBSTONES method for teaching an introductory programming course. The method was developed at the *Universidad Nacional de Quilmes* in Argentina, taking into account the described background of the students, and has shown a positive impact by improving the passing rate of those students.**

**GOBSTONES method's main goal was to foster *abstraction* and *abstract thinking* to students that have not developed good abstraction skills. A programming language, also called GOBSTONES, was developed to implement the ideas presented herein. The method focuses on the representation of information, both at the level of code in the form of procedures to express abstraction, and at the level of the universe of discourse, which is a concrete one, but allowing the representation of information in a simple way. Moreover, the tools implementing GOBSTONES have a feature that enables the student to see a visualization of the represented elements.**

***Index Terms*—Teaching programming method, Introductory course on programming, Gobstones language.**

## I. INTRODUCTION

Due to the increasing employment needs of software companies, several universities in Argentina have decided to implement *technical degrees*[1] in computer science, that last 3 years. These degrees are focused on providing the knowledge that students require to be able to be included as junior developers in software development teams. The goals are different compared to traditional (5-years) university degrees, that focus on a more complete curricula, including research aspects and formal methods. Within this context, it is important how the first course in computer programming is given in the first semester. Several methods for teaching computer programming to people without prior experience have been proposed [1, 2, 3]. For example, there are several approaches that start with Scratch and then move to Python, start with Python and use visual libraries, or start with Alice and then move to Python. Even when these approaches have strong foundations, their main lack is that they do not identify explicitly the students' background. Moreover, the environments Alice and Scratch provide concrete models based on a set of predefined characters (such as aliens, magicians, robots, etc.), and all the activities are based on them. The concrete nature of these characters is a good starting point, but their nature makes it difficult to move the focus to more abstract domains (such as text editors, symbol processing, etc.). In the case of Python, the problem is that it is a popular programming language that has a lot of features for industrial applications, and it is difficult to make students focus on the simpler concepts at first – it requires previous knowledge of abstraction from the beginning.

This work describes the foundations of the GOBSTONES method for teaching computer programming as an introductory course. This is a new method designed at the *Universidad Nacional de Quilmes* (*UNQ*) in Argentina, applied since 2009 in order to introduce newcomers to the computer programming and software industry [4]. At *UNQ*, the average student lacks basic mathematical abilities and abstract thinking. This lack appears because the students come with serious deficiencies from their previous education level (secondary schools). For example, they have troubles when applying arithmetic operations over equations, they do not easily understand how to model abstract elements using numbers and mathematical structures, among other problems. GOBSTONES allows the students to understand the concepts involved in programming at a basic level, and as a consequence, improve their passing rate in their exams in the first semester of their university major. Moreover, students passing the first course state that in the further courses (e.g. data structures) they find easy to grasp the new concepts and ideas.

The method is being used in the *UNQ* (in both in-class and virtual courses), in several secondary schools, and has also served as the basis for the Program.AR[2] method for teaching programming as general knowledge (as opposed to professional knowledge) [5].

When designing an introductory programming course, two typical pitfalls are common: either the course assumes a strong mathematical basis and prior abstraction skills, or it oversimplifies the topic making the switch to professional programming languages difficult (e.g. only simple tasks like add-

[1] They are known as "Tecnicaturas" in Spanish.

[2] Program.AR is a initiative from the national goverment of Argentina in order to promote the teaching of computer science as general knowledge in all the levels of teaching in the country, from initial to secondary schools. http://program.ar

978-1-5386-3057-0/17/$31.00 

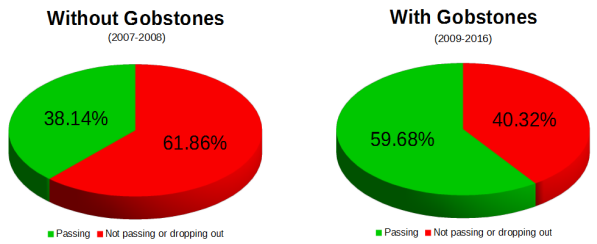


Figure 1. Global passing rates with and without GOBSTONES.

delete-update operations are easy). The GOBSTONES method was designed by taking into account the background of students in order to ease the switch to professional industry programming languages. GOBSTONES' main purpose is to teach *abstraction* and *abstract thinking* to students that have not developed an advanced degree of abstract reasoning.

The method has three main contributions. First, every concept is motivated by proper examples showing their need prior to its introduction (inquiry-based learning). Second, it uses a universe of discourse with concrete elements, that are easy to use as building blocks to represent different abstract elements. And third, it focuses exclusively on selected concepts, excluding more complex ideas (such as input/output commands, complex control mechanisms – e.g. exceptions –, other forms of parameter passing, etc.)

A language, also called GOBSTONES, was designed to embody all these features. The concrete elements included in GOBSTONES are a rectangular *board* composed by cells that can hold *colored marbles or stones*. These elements can be manipulated by a machine called *the head*, which works on a single cell on each operation and is controlled by a program.

The structure of the paper is as follows: Section II explains the background of students starting to learn programming with GOBSTONES, and shows how the passing rate of courses changes before and after the usage of the method. Section III explains the main features of the method, discusses its benefits, and the pedagogical context for GOBSTONES. Section IV describes other methods for teaching programming to novice students, and compare them with ours. Section V discusses some threats to validity we have identified. We conclude in Section VI with some closing remarks and future work.

## II. BEFORE AND AFTER GOBSTONES

GOBSTONES was designed at *UNQ*, in response to the problems found when trying to teach programming with classic approaches in introductory courses, and it was used for the first time in 2009. Prior to it, the course used a classic approach by using the C programming language and focusing on problem solving. In this context, the passing rate of these courses was below 20% (counting number of exams). When inquiring the reasons for these low values, we found that our assumptions about the background of the students were wrong. We identified a lack of mathematical background, with a remarkable deficiency on abstraction skills. The perceived steep learning curve of the course leads to frustration and eventually to drop out the course, and as a consequence, the university degree.

We concluded that it was paramount to introduce programming with a smoother perceived learning curve. Still, the introductory course should require the student to build enough abstraction skills to cope with the increasing complexity of the subsequent courses in the curricula. Given the students' lack of ability to manipulate numbers for modeling and interpreting those manipulations, the approach was conceived to rely on a more concrete computational model to allow the students to build a mental model of a programmable machine.

We analyzed the records of courses on Introduction to Computer Programming from years 2007 to 2016, measuring the passing rates – students passing the course against students not passing or dropping out. We counted students succeeding in the course against those ones that were not succeeding, disregarding the number of attempts it took them to succeed – the average of attempts is 1.6. The reason to consider the students instead of their attempts is that if attempts are measured, a single student failing in several courses counts as several failures. In this case, the passing rate would have been not realistic, because students succeeding are counted only once. The average of attemps shows than more than 50% of students succeed in the first attempt – only a small number of them have more than 3 attempts (before either dropping out or succeeding). We grouped the courses in two: those ones given without using the GOBSTONES method – between 2007 and 2008, 4 semesters – and those using it – from 2009 to 2016, 16 semesters. In the first group, we identified 118 students, from which only 45 passed the course during that period. In the second group there are 812 students, from which 484 of them passed the course (22 of them were students that made some attempts in the previous period and succeeded after the introduction of GOBSTONES). Figure 1 presents these numbers by showing that the passing rate without the GOBSTONES method is 38.14%, while the passing rate with the GOBSTONES method increases to 59.68% – that is, a growth of 21.54%.

## III. THE GOBSTONES METHOD

### III-A. Background

In this section we discuss the details of the GOBSTONES method. In essence, the *didactics*[3] of GOBSTONES was designed to foster abstraction skills. In order to achieve that, we have taken several decisions about which contents to include and in which order.

We require the contents and its order to be based on good principles, and not just on tradition. The principles guiding us are two: the concepts taught must be general, and they must have abstraction and abstract thinking as their main focus. We chose the contents guided by a way to look at programming based on two paradoxes. The first one states:

> "Studying a programming language is not important, but we must study a programming language."

[3]We use the term *didactics* as it is used by Paul Andrews [6]: *"Pedagogy includes didactics, which comprise the strategies and warranted approaches to subject teaching and learning, which may vary from one subject to another, but would necessarily include consideration of the sequencing of ideas and the extent to which the sequence is intellectually coherent."*.

It is a paradox because it appears to be inconsistent. However, it makes sense if we consider that the programming language is just a vehicle to express our ideas, and that those ideas are the really important thing. We want to focus on concepts rather than studying the features of a particular language; while on the other hand, by knowing how to manipulate a language is the only way to describe, conceptualize and communicate our ideas. The second paradox states:

> "We should forget that programs are operational entities, but without forgetting that they are operational entities."

Again, it looks inconsistent. In a first course we must focus on what ideas we are describing, instead on how the machines execute to calculate those ideas, and for that reason we should forget about operational details. However, programs do execute, and a programmer must be aware of this action and be able to consider it when needed. We believe that, while programs are fundamentally operational entities, in an first programming course we must introduce them focusing on their denotational aspect.

In order to focus on abstraction, *the solution strategy* is the most important concept we introduce expressed by spliting the main task into subtasks, and the realization of those subtasks through *procedures*. With this in mind, procedures are introduced as early as possible (i.e. right after the primitive elements of the computational model) in the course.

All language constructs can be classified as *commands* (describing actions) or *expressions* (describing data), or in many languages, as both. GOBSTONES method proposes to make a clear distinction of these two "worlds": commands should **not** describe data, and expressions should **not** describe actions. Procedures are the most important way to express abstraction on the world of commands. This means that when we have to express a complex behaviour in terms of more basic commands, we can group them in a procedure by providing a name, making the execution details abstract, and focusing on the transformation itself rather than on how the transformation is achieved. On the other hand, considering that it is an abstraction mechanism, procedures allow to express the solution of a problem in terms of the problem domain, instead of those of the language domain. Also when we apply the procedures correctly, we can have a top-down approach of the problems' solution, in which we split the problem in several subtasks that are expressed by using procedures. By repeating this process, we tackle every procedure with the same methodology. As a last advantage, if we choose the procedures' names rightly, the program can be made more readable and the solution strategy can be easily detected in a high level reading, without considering the execution details (See Appendix A). Based on this explanation, if the goal is to teach abstraction and promote the abstract thinking, the procedures are the most relevant tool to achieve it.

Additionally, functions are the most important way to express abstractions on the world of expressions, thus being the counterpart of procedures. If we understand the commands as the verbs (because they describe actions), and the expressions, as the nouns (because they describe data), the procedures and the functions are the abstraction mechanisms of each of these two categories, with several analogies among them.

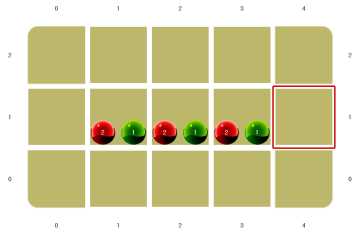

Figure 2. Board of 5 columns and 3 rows, resulting from the program of example 2.

### *III-B. Computational model*

The computational model is simple and concrete. It is based on a rectangular *board* of a given fixed size, made of cells which can hold *colored marbles or stones*, that are manipulated by a device we call the *head* that works on a single cell at a time. Figure 2 shows an example of a board of 5 columns and 3 rows, with some green and red stones and the head positioned at the empty cell at their right.

The primitive commands understood by the head are:

- `Drop`, which deposits a stone of a given color (`Red`, `Green`, `Blue`, `Black`) on the current cell.
- `Grab`, which takes a stone of a given color from the current cell, if exists, and fails otherwise.
- `Move`, which moves the head on a given direction (`North`, `East`, `South`, `West`), if possible, and fails otherwise.

We believe that the concreteness of this computational model allows a student without prior abstraction skills to start building an operational mental model of a programmable machine. For the introductory examples it is key that the state of a program (i.e. the board) can be seen, drawn and easily imagined by the student. It is also important that there are no unexplained or inaccessible mechanisms, such as it is commonly the case in introductory examples that include input-output.

**Example 1.** *Observe the following code with an introductory example of a program that places 3 blue stones to form an horizontal line, assuming there is enough space on the board:*

```
program {
  Drop(Blue); Move(East);
  Drop(Blue); Move(East);
  Drop(Blue);
}
```

*The program is realized just by sequencing simple primitive commands. Despite the simplicity of the task it is important to consider the details of the solution: What happens if the board is not big enough? What happens if the head is initially located close to the east border? At this point we do not expect the students to produce fault tolerant code, but we intend to build a conscience about the importance of taking the initial context into account.*

Shortly after the base computational model is presented and understood, we introduce the concept of *solution strategy*. We motivate the need of explicitly stating the solution strategy for coping with the complexity of programming tasks that can be decomposed into simpler subtasks. We highlight that in order to achieve a good splitting into subtasks, the students have to develop abstraction skills. However, the application of subtasks splitting is resisted by the students while working on the introductory assignments. Still, as the complexity of the assignments grows, the students gradually become aware of the importance of the proper use of procedures to cope with the increasing difficulties. For this reason we explicitly encourage the use of procedures throughout the whole course, from the very beginning.

**Example 2.** *Consider the simple assignment of building a flowerbed with three consecutive flowers, where each flower is represented as two red stones and one green stone. A student resisting the application of subtask splitting could generate the following code:*

```
program {
  Drop(Red);Drop(Red);Drop(Green); Move(East);
  Drop(Red);Drop(Red);Drop(Green); Move(East);
  Drop(Red);Drop(Red);Drop(Green); Move(East);
}
```

*While we expect the students to produce a more readable code where the representation of flowers with stones is properly abstracted in procedures, as follows:*

```
program { MakeFlowerbed(); }

procedure MakeFlowerbed() {
  PlantFlower(); Move(East);
  PlantFlower(); Move(East);
  PlantFlower(); Move(East);
}

procedure PlantFlower() {
  Drop(Red); Drop(Red); Drop(Green);
}
```

*In practice, both pieces of code execute the same primitives and in the same order, and thus are functionally equivalent. However, it is obvious that they differ in readability and extension. We argue that for simple assignments, the temptation of producing the short solution is big and must be answered by the teaching assistant. Had the short solution been accepted at an early point, it would impose greater difficulties for the student to assimilate the application of tasks division as the course advances. We have identified that an early encouragement on subtasks splitting, even for simple assignments where the splitting may seem unnecessary, is the key for developing the abstraction skills that will allow most students to cope with the increasing complexity of programming tasks. A more complete example on the benefits of procedures is presented in the appendix. We are convinced that this, in addition to the simple execution semantics of* GOBSTONES*, is what allows more students to succeed in the course.*

One advantage of the GOBSTONES method throughout the whole course is the unified universe in which the assignments are developed (the board). The computational model provided by the board and the stones is concrete enough to be grasped with ease by students with low abstraction skills, but do not prevent them to develop those skills. While programming tasks based only on a board and stones may appear reiterative, we take advantage of this fact to promote abstraction building skill. We do this by using the stones – and well-named procedures – as the building blocks for representing information of other domains. GOBSTONES colored stones can be used to code elements from many different domains, both concrete and abstract ones. For example, flowers can be represented with two red stones and one green stone. In this way, we can represent all sorts of characters, letters, programs, landmarks, and any other abstract domain that can be thought, even Turing-machines – stones are just basic symbols that can be used to represent any idea. The assignments are chosen so that the represented elements are increasingly abstract, thus developing the abstract thinking as desired. Moreover, the GOBSTONES programming environment allows not only to visualize the board with the stones, but it also allows to map each cell with a given configuration to more representative sprites of a problem's domain – we say that we *attire* the board. Figure 3 shows the same example as in Figure 2, but this time the board is *attired* to show the flowerbed domain instead of stones.

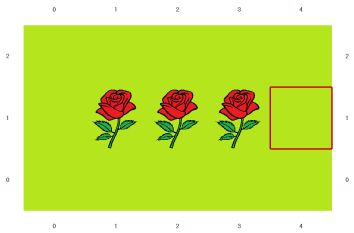

Figure 3. Board of figure 2 but *attired* to show flowers instead of stones.

After presenting the base computational model and the use of procedures, the course continues by gradually introducing new programming constructs. This is done by following an *inquiry-based learning approach* [7, 8, 9], where exercises motivate the need of each concept. That is, we propose an exercise which requires a programming concept that has not been yet introduced, the students are expected to face the challenge and eventually request guidance from the teaching assistant. At this point, the teaching assistant guides a discussion about the generated solutions (if any) or the problems that were found, and after that the teaching assistant can introduce the new concept. In this way, the students feel the need for the concept before learning it, and thus when it is presented, they assimilated it faster than with a classical *definition-example-use* approach.

**Example 3.** *As an example of inquiry-based approach, consider the task of dropping nine blue stones in the current cell. A student can solve the task as follows:*

```
program {
  Drop(Blue); Drop(Blue); Drop(Blue);
  Drop(Blue); Drop(Blue); Drop(Blue);
  Drop(Blue); Drop(Blue); Drop(Blue);
}
```

*Then we can request a program that drops twenty stones, forty stones, and eventually one hundred stones. Clearly sequencing primitive commands soon becomes a strenuous task (even by using procedures). When the concept is understood the teaching assistant presents the simple repetition construct (`repeat`), which repeats the execution of a statement a given number of times. The following code shows that idea:*

```
program {
  repeat (100) { Drop(Blue); }
}
```

Using this scheme, the GOBSTONES approach proposes to introduce the following programming concepts in order. Each concept is materialized in the GOBSTONES language using a keyword to state intention explicitly:

1. Programs as sequence of commands (`program` keyword).
2. Procedures (`procedure` keyword).
3. Simple repetition (`repeat` keyword).
4. Alternative (conditional) statements (`if`, `then`, and `else` keywords).[4]
5. Expressions and functions (`function` keyword).
6. Indexed repetition (`foreach` keyword).
7. Conditional repetition (`while` keyword).
8. Parameters (no special keyword, parameters are listed between parenthesis after a procedure o function definition).
9. Variables (optional `var` keyword).

There are two main reasons to propose variables at the end of the course. Firstly, GOBSTONES method was designed to work as a basis independent of any paradigm. The idea is to learn the main structures common to all the paradigms, and with variables, this is not the case – for example, in Functional Programming. Secondly, due to the didactics by inquiry we use, the variables should be introduced by a specific need. In GOBSTONES, the need appears when the program has to remember a value dependent on a specific cell contents when moved to other cells, in the case of an alternative that has to return two different values, or in the case of accumulators in sequence traversals.

Regarding parameters, we consider that they constitute a more complex concept compared to procedures or functions. Simple functions, similar to simple procedures, have no parameters. They work as a means to abstract complex expressions, similar to how simple procedures abstract complex commands. One example is the function `totalNumberOfStones()` that is defined as

```
function totalNumberOfStones() {
    return (numberOfStones(Blue)
          + numberOfStones(Black)
          + numberOfStones(Red)
          + numberOfStones(Green))
}
```

Another example is the function `isAtTheOrigin()`, that is defined as

```
function isAtTheOrigin() {
    return (not canMove(South)
         && not canMove(West))
}
```

We have identified that parameter passing and the use of variables are the most challenging topics of this sequence, and thus we delay their introduction until the student has enough experience.

### *III-C.* GOBSTONES *as a vehicle for knowledge construction*

Based on Piaget's [10], we consider that GOBSTONES is an adequate method for the development of a thinking process. It focuses on mechanisms that enhance the transition from the instrumental knowledge (know-how through guided exercises) to conceptual knowledge (in our case, abstract meaning of the learned concepts and abstraction itself). Therefore, the conceptualization goes beyond the action, becoming a building process of new concepts based on previous mechanisms by applying reflexive abstraction – the learning is not based only on results.

The use of any language regulates cognitive concepts. A programming language such as GOBSTONES, when it is used as a go-between artifact in knowledge construction, is a necessary step to learn not only "programming" but "the concepts for building programs that can be transferred to other contexts". The challenge is to depart from the mere "doing", as advocated by most approaches, and to achieve an aware assimilation of concepts. This is an innovative feature of our work.

Brousseau [11] states *"A (learning) environment without didactical practices is not enough to make the student induce all the cultural knowledge that is desired for him/her to acquire."* Based on this theory, we consider that GOBSTONES fullfils the prerrequisites of being a well designed environment that supports didactical goals.

Vigotsky's sociocultural theory [12] states that learning is a social process that proceeds in two levels: first through interaction with others, and then integrated into the individual's mental structure, and also that learning is limited to a *"zone of proximal development"* (ZPD), an area of exploration for which the student is cognitively prepared. A teacher, peer or learning artifact can operate as a "scaffolding" to support student's learning guiding him in this ZPD. In that sense, GOBSTONES can be considered as a go-between artifact that leads from the student's autonomous learning to the development of new possibilites.

### *III-D. Closing*

Summarizing, the GOBSTONES method proposes a simple and concrete computational model and an order in which to introduce the basic concepts of programming. We emphasize

[4] Although the usual name used for *if-then-else* is *conditional statement*, we prefer to name it *alternative statement*, to emphasize the fact that the concept is that of *choosing between two alternatives* based on a condition. In this way, we can summarize saying that commands can be combined by sequencing, alternative and repetition, with procedures as its form of abstraction. This pattern of sequence, alternative, repetition, and abstraction appears recurrently in computing (for example in data structures, formal languages, etc.) and thus we value its early presentation.

that the most important concept used is that of solution strategy expressed by subtasks' splitting and realized using procedures, presented as early as possible. On the contrary, parameters and variables are delayed until the student has enough experience to understand them. The board and the stones can be used to model other domains, which further promote abstraction building skills.

## IV. Related Work

Decades of exploratory research on teaching programming have resulted in a plethora of educational tools and pedagogical recommendations [3, 2, 13, 14, 15]. Often, contradictory approaches are reported to achieve similar passing rates. Despite the large volume of studies, there is still little systematic evidence to support any particular approach. In this section, we only discuss the works that most resemble our approach, and we refer the reader to [1] for a survey.

Nowadays the dominant theory of learning is called *constructivism*, which claims that knowledge is actively constructed recursively on knowledge the student already has. In [16] Ben-Ari analyses the theory in the context of computer science education. The article points out that a novice programming student has no effective mental model of a computer and, through the process of learning, he/she builds the model from the ground up. However, each student builds an idiosyncratic model, some of which may turn out to be non-viable for high complexity tasks. Thus, the paper draws on the conclusion that a minimalist model must be explicitly taught and validated.

Making the computer model explicit has implications regarding the level of detail and abstraction. A high-level abstraction – as proposed by the component-first approach [3] – requires to abstract details that were never assimilated by the student, facilitating the construction of non-viable models [17]. On the other hand, low-level abstractions – as presented in lectures on introduction to programming based on the C programming language [2] – may require the student to process an overwhelming amount of information (such as compiler-linker, libraries, input-output, etc.). For this reason, the GOBSTONES approach relies in a simple low-level computational model (the board and the head) and a high-level procedural approach to programming. The goal is twofold: on the one hand, we explicitly present an easy-to-understand concrete operational model, while on the other hand, we encourage the creation of abstraction layers based on concepts captured with procedures. In our experience, the transition from this model to the more abstract model of OOP is less hazardous and reduces the amount of frustration.

From the theoretical point of view, when we make a difference between commands and expressions, we can improve the comprehension of the nature of objects (in object-oriented programming), and the nature of functions (in functional programming). From this viewpoint, both paradigms can be introduced by our approach.

The literature contains different teaching approaches, but their focus can be loosely grouped in three categories: 1) language and syntax, 2) problem solving, 3) abstraction and modeling. We believe that the three categories are important and that they need to be developed simultaneously. The GOBSTONES approach advocates to the guidelines stated by Schneider [18], by focusing on the semantics of simple syntax constructions, presenting a sequence of clearly stated problems for which algorithmic solutions need to be devised, and making emphasis in abstraction through task splitting as a means for coping with complexity. Our goal is to present the introductory material in such a way that we achieve a high rate of knowledge transfer, allowing to progressively apply the acquired skills to more complex programming tasks and concepts.

Research in introductory programming course design often materializes in the form of programming environments with special support for novices. The features of such environments can be classified in three groups: programming support, micro-worlds and automatic assessment.

Programming support tools [19, 20, 21] include interactive code execution, visualization and syntax support (e.g. highlighting and scope indicators) among others. We have implemented some programming support for GOBSTONES, but we chose to exclude detailed runtime visualization features, since they can lead to a trial-and-error way of solving problems instead of an approach where the solution is reasoned.

Micro-worlds [13, 14, 15, 22] are designed to decrease the distance between the students' mental model and the programming language. GOBSTONES presents a micro-world easy to understand, but that also shares common characteristics with a computer. The goal of this rudimentary model is to be more easily refined into low-level computer models in contraposition to anthropomorphic character-based or *turtle-graphics* models.

Automated assessment tools [23, 24, 25] provide quick feedback to a student which can foster the learning process. However, unclear error messages can produce the opposite effect. Currently the GOBSTONES method is supported by the Mumuki [26] classroom management tool. Mumuki allows not only to provide high quality feedback to the students, but also allows the teaching assistant to follow the progress of individual students with great detail.

Considering the quantitative results on students' improvement with the application of a specific method for teaching programming, there is still little evidence to support any particular approach. However, statistical evidence supports the fact that high failure rates in introductory programming courses is a worldwide phenomenon [27].

## V. Threats to validity

In Section II, we presented figures about students' success in a first programming course. Those measurements were calculated on two populations: a group learning with C (2007-2008), and another one learning with Gobstones (2009-2016).

We have identified two threats to validity. The first one is about changes in the evaluation procedures due to the change in the teaching method. We consider that the impact is small, since during the courses, the professors always tried to keep the exams' complexity at a similar level. The second

one is that the statistical population of the first group was smaller than the second one. The course on Introduction to Computer Programming started at 2007, and we decided to change as soon as possible to the GOBSTONES method because the number of students dropping out their studies due to background issues was unacceptable (over 70%). However, we analyzed the reasons behind the low passing rates, and we are convinced that the passing rate would have been similar in a larger population had we not changed the method.

## VI. Conclusions and future work

This article presents the GOBSTONES method for teaching programming in the first semester of a university degree. We highlight the importance of a simple computational model based on concrete elements, and the focus on abstraction building by the early introduction of the concept of solution strategy. Abstraction is reinforced through intensive usage of procedures, and through representation of abstract elements and its visualization. This allows the students to build a viable mental model of computation than can be then transfered to more advanced programming topics.

Another important feature of the method is the use of an inquiry-based learning, because motivation is paramount to engage the students to learn programming. The didactics sequence we recommend enables the use of inquiry-based learning throughout the whole course, since the concepts are ordered in an increasing level of abstraction (e.g. simple repetition before conditional repetition, simple procedures before procedures with parameters, etc.)

The GOBSTONES method has been proved to be very useful for teaching programming. However, its main current drawback is that at present most of the study material is written in Spanish (because the method was designed and mostly used in Argentina). As future work, we are planning to translate the material to English and other languages in order make it available worldwide. Another drawback is that the current environment we are using is desktop-based, and it is difficult to extend it with new features easily – *e.g.* drag and drop blocks. We are working on a web-based environment with the existing features but that may also use drag-and-drop blocks to build the program without syntactical errors.

In addition to the basic ideas described in this article, we have also developed an extension of GOBSTONES with basic data structures: records, lists and variants. So far, we do not have any public documentation of those extensions, and we are only using them in the in-class university course. As an additional future work, we plan to publish these extensions together with teaching materials.

In Section II we have reported an increase of 21% on the course passing rate. Beyond the statistical improvements, we have run a qualitative poll among teachers using GOBSTONES on other institutions, to assess their opinions. They have remarked the benefits obtained by the simple programming constructs allowing students to quickly create programs and visualize the results in a user-friendly interface. They also commended the ease with which the presented programming concepts can be transfered to other programming languages.

For all these reasons, we are convinced that by taking these guidelines into account, the effectiveness of an introdutory programming course for people with a lack of abstract skills can be drastically increased.

## VII. Acknowledgments

The authors wish to thank Fundación Sadosky, and specially the Program.AR team, for their inspiration to write this paper down and their work to develop and spread both GOBSTONES and Program.AR together with us. They also wish to thank the teaching group of Introduction to Computer Programming course at *UNQ* because it was with them that GOBSTONES was born and evolved.

## References

[1] A. Pears, S. Seidman, L. Malmi, L. Mannila, E. Adams, J. Bennedsen, M. Devlin, and J. Paterson, "A survey of literature on the teaching of introductory programming," in *Working Group Reports on ITiCSE on Innovation and Technology in Computer Science Education*, ser. ITiCSE-WGR '07. New York, NY, USA: ACM, 2007, pp. 204–223. [Online]. Available: http://doi.acm.org/10.1145/1345443.1345441

[2] E. S. Roberts, "Using C in CS1: Evaluating the Stanford experience," in *Proceedings of the Twenty-fourth SIGCSE Technical Symposium on Computer Science Education*, ser. SIGCSE '93. New York, NY, USA: ACM, 1993, pp. 117–121. [Online]. Available: http://doi.acm.org/10.1145/169070.169361

[3] E. Howe, M. Thornton, and B. W. Weide, "Components-first approaches to CS1/CS2: Principles and practice," in *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education*, ser. SIGCSE '04. New York, NY, USA: ACM, 2004, pp. 291–295. [Online]. Available: http://doi.acm.org/10.1145/971300.971404

[4] P. E. Martínez López, *Las Bases Conceptuales de la Programación. Una nueva forma de aprender a programar*. El autor, EBook, diciembre 2013, in Spanish. ISBN: 978-987-33-4081-9. URL: http://www.gobstones.org/bibliografia/Libros/BasesConceptualesProg.pdf.

[5] P. Factorovich and F. Sawady O'Connor, *Actividades para aprender a Program.AR*. Fundación Sadosky, EBook, 2015, in Spanish. ISBN: 978-987-27-4161-7. URL: http://programar.gob.ar/descargas/manual-docente-descarga-web.pdf.

[6] PaulAndrews, "Conditions for learning: a footnote on pedagogy and didactics," *Mathematics Teaching*, vol. 204, p. 22, September 2007, uRL: https://www.atm.org.uk/write/MediaUploads/Journals/MT204/Non-Member/ATM-MT204-22-22.pdf.

[7] J. Dostál, "Inquiry-based instruction: Concept, essence, importance and contribution," Ph.D. dissertation, Palacký University, Olomouc, Czech Republic, 2015, iSBN 978-80-244-4507-6, doi 10.5507/pdf.15.24445076.

[8] V. Sampson, J. Grooms, and J. P. Walker, "Argument-driven inquiry as a way to help students learn how to participate in scientific argumentation and

craft written arguments: An exploratory study," *Science Education*, vol. 95, pp. 217–257, 2011, doi: 10.1002/sce.20421.

[9] J. M. Chang, "A practical approach to inquiry-based learning in linear algebra," *International Journal of Mathematical Education in Science and Technology*, vol. 42, pp. 245–259, 2011, first published on: 29 November 2010. DOI: 10.1080/0020739X.2010.519795.

[10] J. Piaget, *La equilibración de las estructuras cognitivas. Problema central del Desarrollo*. Siglo XXI, México, 1978, iSBN 978-84-323-1625-8.

[11] G. Brousseau, *Theory of Didactical Situations in Mathematics (Didactique des Mathématiques, 1970–1990)*. Springer Netherlands, 2002, iSBN 978-0-306-47211-4.

[12] R. Baquero, *Vigotsky y el aprendizaje escolar*. Aique, 2009, iSBN 978-950-70-1333-1.

[13] S. Cooper, W. Dann, and R. Pausch, "Using animated 3D graphics to prepare novices for CS1," *Computer Science Education*, vol. 13, no. 1, pp. 3–30, 2003. [Online]. Available: http://www.tandfonline.com/doi/abs/10.1076/csed.13.1.3.13540

[14] R. E. Pattis, *Karel the Robot: A Gentle Introduction to the Art of Programming*, 1st ed. New York, NY, USA: John Wiley & Sons, Inc., 1981.

[15] P. Henriksen and M. Kölling, "Greenfoot: Combining object visualisation with interaction," in *Companion to the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, ser. OOPSLA '04. New York, NY, USA: ACM, 2004, pp. 73–82. [Online]. Available: http://doi.acm.org/10.1145/1028664.1028701

[16] M. Ben-Ari, "Constructivism in computer science education," in *Proceedings of the Twenty-ninth SIGCSE Technical Symposium on Computer Science Education*, ser. SIGCSE '98. New York, NY, USA: ACM, 1998, pp. 257–261. [Online]. Available: http://doi.acm.org/10.1145/273133.274308

[17] L. Böszörményi, "Why Java is not my favorite first-course language," *Software - Concepts & Tools*, vol. 19, no. 3, pp. 141–145, 1998. [Online]. Available: http://dx.doi.org/10.1007/s003780050017

[18] G. M. Schneider, "The introductory programming course in computer science: Ten principles," in *Papers of the SIGCSE/CSA Technical Symposium on Computer Science Education*, ser. SIGCSE '78. New York, NY, USA: ACM, 1978, pp. 107–114. [Online]. Available: http://doi.acm.org/10.1145/990555.990598

[19] M. Kölling, B. Quig, A. Patterson, and J. Rosenberg, "The BlueJ system and its pedagogy," *Computer Science Education*, vol. 13, no. 4, pp. 249–268, 2003. [Online]. Available: http://www.tandfonline.com/doi/abs/10.1076/csed.13.4.249.17496

[20] L. Malmi, V. Karavirta, A. Korhonen, J. Nikander, O. Seppälä, and P. Silvasti, "Visual algorithm simulation exercise system with automatic assessment: TRAKLA2," in *Informatics in Education*, 2004, p. 048.

[21] T. L. Naps, "JHAVÉ: Supporting algorithm visualization," *IEEE Computer Graphics and Applications*, vol. 25, no. 5, pp. 49–55, 2005.

[22] F. Kalelioğlu, "A new way of teaching programming skills to K-12 students: Code.org," *Computers in Human Behavior*, vol. 52, pp. 200–210, November 2015.

[23] C. A. Higgins, G. Gray, P. Symeonidis, and A. Tsintsifas, "Automated assessment and experiences of teaching programming," *J. Educ. Resour. Comput.*, vol. 5, no. 3, Sep. 2005. [Online]. Available: http://doi.acm.org/10.1145/1163405.1163410

[24] S. H. Edwards, "Improving student performance by evaluating how well students test their own programs," *J. Educ. Resour. Comput.*, vol. 3, no. 3, Sep. 2003. [Online]. Available: http://doi.acm.org/10.1145/1029994.1029995

[25] M. Joy, N. Griffiths, and R. Boyatt, "The boss online submission and assessment system," *J. Educ. Resour. Comput.*, vol. 5, no. 3, Sep. 2005. [Online]. Available: http://doi.acm.org/10.1145/1163405.1163407

[26] F. Aloi, F. Bulgarelli, and L. Spigariol, "Mumuki, una plataforma libre para aprender a programar," in *3er Congreso Nacional de Ingeniería Informática / Sistemas de Información*, ser. CONAIISI, 2015, in Spanish.

[27] C. Watson and F. Li, "Failure rates in introductory programming revisited," in *Proceedings of the 2014 Conference on Innovation Technology in Computer Science Education*, ser. ITiCSE. ACM, 2014, pp. 39–44.

## APPENDIX

### A. Example showing the benefits in the use of procedures

In this appendix we introduce two versions of a program that draws some figure. The purpose is to show the benefits by using procedures, as mentioned in Section III.

The idea is based on one activity called "To program on graph paper"[5] from [5]. This activity proposes a very simple programming language to write programs for drawing pictures in quad paper. The language has primitives for moving (arrows) and for painting a square (some pencil strokes), and ways to express repetition (following some commands between parentheses with a number) and procedure definition and invocation (using name between quotes and square brackets respectively). Except for the primitive to paint a square, the remaining forms exist in GOBSTONES; for the missing primitive, both programs use a basic procedure `PaintSquare`, which is used to express the representation of pencil strokes using stones: a painted cell is represented with one stone of each color. This way of using procedures was already discussed in the body of the paper.

```
procedure PaintSquare() {
   Drop(Blue); Drop(Black);
   Drop(Red}; Drop(Green)
}
```

The way of using procedures we want to illustrate here is to express the *solution strategy*. The first version of the program does not use procedures, and thus it is very difficult to determine, without executing the program, which was the intent of the author. In the case that the program has some

[5] "Programar en papel cuadriculado"

errors (for example, a reversed direction or a wrong number of repetitions in some parts), it is in general very difficult to determine how to correct it, because there is no clue as to what the result should be.

```
program
{
  repeat (7) { PaintSquare(); Move(West) }
  repeat (7) { PaintSquare();
               Move(East); Move(North) }
  repeat (7) { PaintSquare(); Move(South) }
  Move(East); Move(East)
  repeat(6) { PaintSquare(); Move(North) }
  Move(East)
  repeat(3) { PaintSquare(); Move(East);
              Move(South);   Move(South) }
  repeat(4) { PaintSquare(); Move(West) }
  Move(South); Move(South)
  repeat ( 5) { PaintSquare(); Move(East) }
  repeat ( 2) { PaintSquare();
                Move(South); Move(West) }
  repeat (11) { PaintSquare(); Move(West) }
  repeat ( 2) { PaintSquare();
                Move(West); Move(North) }
  repeat (10) { PaintSquare(); Move(East) }
}
```

The second version of the program use procedures to express the *solution strategy*, thus capturing the purpose of the program as part of the code. In this case, it is very easy to imagine what the drawing is, to check if the program behaves as expected, and to correct it in case it does not.

```
program{ DrawSailboat() }

procedure DrawSailboat() {
 DrawLeftSail()
 GoFromLeftSailToRightSail()
 DrawRightSail()
 GoFromRightSailToHull()
 DrawHull()
}

procedure DrawLeftSail() {
  repeat (7) { PaintSquare(); Move(West) }
  repeat (7) { PaintSquare();
               Move(East); Move(North) }
  repeat (7) { PaintSquare(); Move(South) }
}

procedure DrawRightSail() {
  repeat(6) { PaintSquare(); Move(North) }
  Move(East)
  repeat(3) { PaintSquare(); Move(East);
              Move(South);   Move(South) }
  repeat(4) { PaintSquare(); Move(West) }
}
```

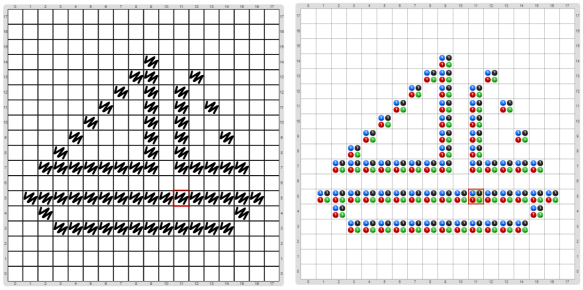

Figure 4. The final board resulting from the drawing programs, both with and without attire.

```
procedure DrawHull() {
  repeat ( 5) { PaintSquare(); Move(East) }
  repeat ( 2) { PaintSquare();
                Move(South); Move(West) }
  repeat (11) { PaintSquare(); Move(West) }
  repeat ( 2) { PaintSquare();
                Move(West); Move(North) }
  repeat (10) { PaintSquare(); Move(East) }
}

procedure GoFromLeftSailToRightSail() {
  Move(East); Move(East)
}

procedure GoFromRightSailToHull() {
  Move(South); Move(South)
}
```

Both programs generate the final board presented in Figure 4, starting from the proper initial board.

The use of procedures with this purpose requires the student to think explicitly in the solution strategy, and to properly name each subtask, thus fostering abstract thinking, as desired. The original activity does not use parameters, but in a later stage when parameters are already introduced, the subtasks of drawing lines can be expressed also by using procedures, resulting for example in the following way to write one of the procedures.

```
procedure DrawLeftSail() {
  DrawLineTo(West, 7) }
  DrawDiagonalLineTo(East, North, 7)
  DrawLineTo(South, 7) }
}
```

This example clearly illustrates our claim about using procedures.