

Introduction to Programming for Engineers following the Parachute Paradigm

Gerardo M. Sarria M.
Universidad Javeriana-Cali, Colombia
gsarria@cic.puj.edu.co

Abstract – Each generation of students poses a great challenge to teaching/learning methodologies. The current generation of students always tries to avoid the abstract things; they need concrete things in order to understand the world. In this paper we propose a new approach of how and what to teach in the introduction to programming course for engineers. We believe the classic approach of teaching programming by teaching a programming language is no longer appropriate for the new generations of students. We use the idea of the parachute paradigm used in specification of models to gradually introduce the notions of programming instead of the syntax and semantics of a programming language.

Index Terms - Introduction to programming, Notions of programming, Parachute paradigm, Teaching/Learning.

INTRODUCTION

Programming became a basic science for any engineer. We can see an industrial engineer programming the best path between two places in order to save things such as time and money, or programming a production chain; we can also see a civil engineer programming a GPS device to take some measures; and we can continue naming different situations in which engineering professionals must use notions of programming. This is why many universities have changed their curricula to incorporate an introductory programming course in the first year of all engineering students. However, this has brought some problems such as lack of motivation, low grades, and university desertion. For this reason, there have been many proposals of how and what to teach in the introduction to programming course (we may see one of the first publication of this matter in [1]). The classic approach is the “teach-a-language” approach which through the years has reduced its effectiveness. Other approaches include using different (interactive) spaces [2], integrating software and hardware topics [3], using languages never tried [4] or proposing new ones with graphic tools like Alice [5].

We propose a new approach changing the idea of the course. Since we believe in formal methods, we propose an introductory course of programming based on the notions of programming instead of tools and languages. We use the idea of the parachute paradigm used in specification of models with the Event-B and Rodin platforms [6]. We go through all the notions starting on that of a system, then the

notions of observation, state, condition, abstraction, recursion, iteration and ending with the notion of data abstraction. We do not teach them how to write a line of code, they learn that in a logical way. It is necessary to use a programming language for the practice (we use Python [7] because we think it enhances the learning process, the fourth principle of an initial programming course [8]) but it is not the core of the course since many of the engineers never use a programming language after the course; they just use the notions of programming. The classes are supported by readings, practices and tools similar to micro world-based educational software. We have followed this approach for 2 years now and the results have been satisfactory, especially in increasing motivation and reducing desertion.

The rest of the paper is organized as follows: section 2 explains the idea of the parachute paradigm, section 3 illustrates the contents and methodology of the course describing every notion of programming and how we teach them, and section 4 shows some results, and gives some conclusions and future work.

THE PARACHUTE PARADIGM

The *Parachute Paradigm* [9] is an approach of system studies by means of one or more models which are refining in order to make them more accurate. Models are built from the observations that can be made about the system. These observations determine the states and events of the system.

To better understand the parachute paradigm we must place ourselves mentally at certain level above the system to be modeled. We can take a football game, for instance: if we are too high in the sky the observations we can make about it are just the field and people moving around inside it. Sometimes a person gets out and sometimes another gets in. This is the first approach of the system. Then, if we step down a little (closer to the system) we may be able to observe more interesting things like the ball (the players pass it to each other) and the colors of both uniforms. This way we refine the model. We can step down more and more and thus the model is gradually refined offering all the details and properties of the system.

We can see the parachute paradigm in action by considering the evolution of the video games. In the Atari 2600 console (1977), for instance, the football game had a rectangle as a field and the players were squares; even the ball was a little brown circle. But through the years this

approach has been enhanced by adding more and more details. For the time being we can see the faces of the popular players.

CONTENTS AND METHODOLOGY OF THE COURSE OF PROGRAMMING

We used the idea of the parachute paradigm to develop a first course of programming. We know that changing a person's way of thinking is difficult, and even more difficult if the new manner is an algorithmic one. This is why we split the knowledge of programming in eight notions:

1. System
2. Observation
3. State
4. Condition
5. Abstraction
6. Recursion
7. Iteration
8. Data Abstraction

We believe that programming is beyond computers. In this sense, the use of a computer in the course should start with the third notion (State), not the first. We are also convinced that at the beginning we require a graphic tool in order to maintain the attention of every student (the textual environments in this time of 3D video games, touch screens, interactive television, etc. is no longer convenient). We used software based on the micro-worlds paradigm.

As any other course we conceived this one as a sequence of sessions. We consider the duration of a course as a period of 16 weeks; then we count with 32 sessions of 2 hours. We take one or more sessions for each notion including theory and practice (more practice than theory). Now we will explain in detail the idea of the above notions of programming.

I. Notion of System

We teach the notion of system as a set of interacting elements in a very specific domain. We don't go deep in the definition of the word but in the application of the concept. We give a set of examples starting with the most known such as the solar system, the digest system, and the economic system. Then we move to the currently known (which students use as part of their lives) such as ATMs, television, game consoles and computers.

The idea here is to show that everything among us is a system, and most (if not every) systems are suitable to be modeled. We show that computer programs are computational models of real systems. We also ask students to identify the elements of the systems and to describe their interactions. If we empty most students' pockets we will find out a lot of devices able to be used as material for the class: iPods, cell phones, credit cards, CDs/DVDs, USB flash drives, etc. Additionally, there is a webpage very useful for this notion: *How Stuff Works* [10]. In that page we can find a

lot of articles of those devices and systems mentioned above which can be used to motivate students.

We must not expend more than one session in this notion.

II. Notion of Observation

Once students understand a system they are able to describe it. At this point we are interested in the elements and not the system as a whole. Following the parachute paradigm we step down in order to be closer to the system and we ask the students to identify what is static and what changes. What remains static will be called "a constant" and the changing elements will be called "variables".

Additionally, for each element we detect the values that it can take. The set of these values defines the data type of the element.

At the end of this session we introduce a very basic software written in Python that will help us the next couple of notions. This software was developed to assist the learning process, to aid understanding the notions and to begin programming with graphic results. It was inspired by the micro-worlds paradigm and it is similar to programming languages such as Logo and Karel. Figure 1 shows one example of a world in this software.

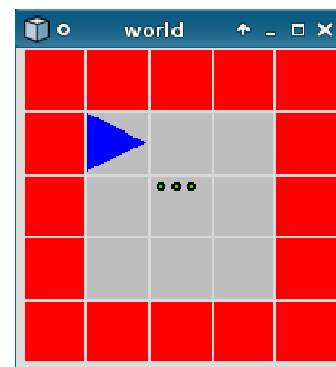


FIGURE 1
EXAMPLE OF THE WORLD

This world consists in a matrix where in each location there is a wall (red squares) or a squirrel (blue triangle) and/or food (little green circles). We can observe, for instance, that for this system the walls are constants, and variables are: the location of the squirrel, her direction, the number of food in the squares and the number of seeds the squirrel carries in his mouth.

Worlds are the key of the programming learning exercise because we describe problems and we graphically visualize the execution of the solution programs inside them.

This notion needs no more than one or two sessions to be understood.

III. Notion of State

Stepping down once more in a system we make the students noticing that sometimes some variables have one value and sometimes they have another value. The valuation of all variables defines a state of the system. Then we ask them

how the system changes from one state to another. They realize that some operations must be executed to change the values of the variables (and consequently change the state). Here is where the definition of the concept of *algorithm* arises: the sequence of operations leading from one state to another.

Operations are described in Python notation. At this point we consider them black boxes with inputs and outputs. Since Python is an implicit typed language, this characteristic avoids the noise from data type declaration in function calls. Figure 2 illustrates an example of a change of state in the squirrel world.

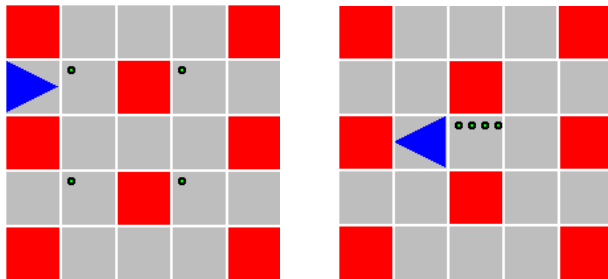


FIGURE 2
EXAMPLE OF A CHANGE OF STATE

In these worlds we have only seven operations:

- `MOVE()`: the squirrel moves one location in the current direction
- `TURNLEFT()`: the squirrel turns left (counter-clockwise)
- `eat()`: puts one seed in the squirrel's mouth
- `putFood()`: takes one seed from the mouth of the squirrel and puts it in the current location
- `hasFood()`: answers if there is food in the current location
- `hasWallInFront()`: answers if there is a wall on the next location in the current direction
- `getMouth()`: says the number of seeds the squirrel has in her mouth.

From these operations the student should describe the sequence of actions leading from one state to another state.

We can spend a couple of sessions in this notion.

IV. Notion of Condition

If the state of the system is unknown students feel the necessity of having a control structure to consider various cases. Then we introduce the notion of condition. However to better understand the mechanism of this notion we must start exercising the very basic foundations of the propositional logic: truth tables and connectives. As previous notions this must be done using expressions closer to the students' knowledge.

At the beginning we just use the "if <condition>:" structure. When this is understood we include the "else:" expression.

Since the logic expressions are abstract things we should spend around four sessions in this matter.

V. Notion of Abstraction

The more complex the problem to solve is, the larger the code becomes. Once more students feel the necessity of having another way to program that allows writing less lines of code. Then we teach them how to organize various operations into a single one. Black boxes become white boxes. Now students can build their own operations by calling the basic ones. With the new operations they can build others and so on. This brings the notion of levels of abstraction in a natural way. The first example we give is the expected `TURNRIGHT()` operation (students always hate the idea of turning left three times in order to turn right):

```
def TURNRIGHT():
    TURNLEFT()
    TURNLEFT()
    TURNLEFT()
```

One thing to take into account is that here we start separating ourselves from our micro-world software. We begin to solve engineering problems (mathematical, most of all) using other libraries.

Additionally, at this point we make a first evaluation. This evaluation has two parts: one part with the computer and other without it (we call the part with the computer a "mini-project"). We consider this separation important because it shows us that the notions are well-conceptualized. If we make just a practice exam then we cannot be sure that students are developing the problems by trial and error.

VI. Notion of Recursion

We believe recursion is the basic form of repetition in computer science. However this notion seems to be very difficult to understand. We simplify the idea by considering only numeric recursion and teaching this notion from what it is: a function call. We teach them, of course, all about the basic case and the rules to reduce all other cases but, as the other notions, with real life examples students know, and using the squirrel world once in a while.

This notion gives us the possibility of solving many more problems. In this sense we can spend four to six sessions (depending on students) practicing recursion.

VII. Notion of Iteration

Although we think recursion is the simplest form of repetition because there is no change of state in the variables, it seems more natural for students to apply other structures of repetition that do change the state of variables. They are called iterations. There are various forms of iteration: `for`, `while`, `do-while`, we just consider the "while <condition>:" form.

In this notion we stay for several sessions. Since the notions are gradually introduced, with the inclusion of a new notion we continue using and practicing the previous ones. Students have now all the tools to solve most problems.

Here we make a second evaluation.

VIII. Notion of Data Abstraction

This final notion comes out at the necessity of grouping large amount of information. We ask, for instance, to develop a function computing the numeric grade point average of all classes they have taken. If the number of classes is just a few then there is no problem, but if students have taken seven or more classes, handle that amount of variables is inconvenient (imagine the amount of input variables).

From all kinds of data structures we choose lists. This has two reasons: they are basic data structures and their use mechanism in Python is very comfortable.

We use the remaining sessions for this notion and we make a third and last evaluation.

RESULTS, CONCLUSIONS AND FUTURE WORK

In this paper we have shared our approach of the introduction to programming teaching/learning process. After two years of applying this idea our students not only increased their motivation (70%) and grades (30%), but they are able to

- to model computationally a problem,
- to apply the notions of programming in the solution of a problem, and
- to interpret, to develop, and to explain algorithms in a programming language.

The decision of using Python as a programming language was decisive. We needed a language that was interpreted to avoid the noise of the compilation step, had a very simple syntax and a lot of documentation, and that was used in the industry just in case the future engineers use the language in their work. We previously tried C and Scheme but they do not fulfill all the requirements.

We have proved that the use of graphic programs in the micro-world paradigm is effective. We plan to enhance our software by upgrading it to a 3D program. The program that inspired us to make the jump to 3D was *Light Bot*, the puzzling gameversion of Lego Mindstorms [11].

ACKNOWLEDGMENT

We want to thank Camilo Rueda for his brilliant idea of using the Parachute Paradigm in our course. We also want to thank Gustavo Gutierrez and Alejandro Arbelaez for developing the squirrel world, and Andrés Becerra, Mario Mora, Salim Perchy and all people above for their suggestions and actions to make this a better course.

REFERENCES

- [1] Gries, David. "What should we teach in an introductory programming course?" 1974. *Proceedings of the 4th SIGCSE Symposium on Computer Science Education*. Vol. 6 (1), pp.81 – 89.
- [2] Nagurney, Ladimer S. "Teaching Introductory Programming for Engineers in an Interactive Classroom." 2001. *Proceedings of the 31th ASEE/IEEE Frontiers in Education Conference*. Reno, NV.
- [3] Parish, Allen, et al. "An Integrated Introductory Course for Computer Science and Engineering." 1999. *Proceedings of the 29th ASEE/IEEE Frontiers in Education Conference*. San Juan, Puerto Rico.
- [4] Azemi, Asad and Laura L. Pauley. "Teaching the Introductory Computer Programming Course for Engineers Using Matlab." 2008. *Proceedings of the 38th ASEE/IEEE Frontiers in Education Conference*. Saratoga Springs, NY.
- [5] Conway, Matthew J. "Alice: Easy-to-Learn 3D Scripting for Novices." 1997. PhD Thesis. University of Virginia. Available online: <http://www.alice.org/>
- [6] Abrial, Jean Raymond, Michael Butler, Stefan Hallerstede, and Laurent Voisin. "A Roadmap for the Rodin toolset." 2008. *Proceedings of the 1st international conference on Abstract State Machines, B and Z*. LNCS Vol. 5238, London, UK.
- [7] Rossum, Guido van. February 2009. *Python Tutorial*. Fred L. Drake, Jr., editor. Python Software Foundation.
- [8] Scheneider, G. Michael. "The introductory programming course in computer science: ten principles." 1978. *Proceedings of the SIGCSE/CSA Technical Symposium on Computer Science Education*. Vol. 10 (1), pp. 107 – 114.
- [9] Abrial, Jean Raymond. Guidelines to formal system studies. November, 2000. MATISSE project.
- [10] How Stuff Works. 1998. <http://www.howstuffworks.com/> Accessed: 22 March 2009.
- [11] Light Bot. 2008. <http://www.gameroo.com/games/light-bot?r=nl>. Accessed: 22 March 2009.