A STUDY OF THE EFFICIENCY OF AN ALTERNATIVE PROGRAMMING PARADIGM TO TEACH THE BASICS OF PROGRAMMING.

Guibert, N.[1], Guittet, L.[1] and Girard, P.[1]

[1]LISI / ENSMA, 86961 Futuroscope Chasseneuil Cedex,France. E-mail: {guibert, guittet, girard}@ensma.fr

ABSTRACT

Although computers and programs have now become essential in many sciences as analysis or measurement tools, many still find the learning of computer science extremely difficult. Kaäsboll reported in 2002 [1] that between 25 and 80% of students in first-year programming courses either failed or gave up world-wide.
A synthesis of different works in Cognitive Psychology and Didactics allowed us to summarise the intrinsic difficulties of programming in two main objectives:

- to build a viable model of the structure of the task.
- to build a viable model of the computer which will perform the task.

The reason that makes learning programming that difficult is that students, contrary to physics for instance, have no effective starting model, and therefore no ground to understand the brutal error feedback they are faced to in their early experiences of programming.
These internal difficulties are even more stressed by the use of "professional" languages and environments as learning environments, because they:

- are built on rather complex models of the computer.
- give support to evaluation but few to conception, leading to "blank page" syndromes.
- make it very difficult to separate the construction of these two models.

We plan to study the efficiency of an alternative programming paradigm, "programming by examples". Programming by examples is neither static nor abstract, but enables the programmer to interact with a graphical model of the state of the task, through concrete examples.
We detail the MELBA learning environment, built on a constructivist approach and example-based techniques to provide a visual and interactive support to construct the models of the task and of the computer, and analyse the efficiency of the system and of its underlying method through a study on 65 first year students in bio-informatics.

1. INTRODUCTION

Programming has been from its early days regarded as a difficult discipline to teach. Even nowadays, j. Kaasboll reports that, world-wide, first-year programming courses in university suffer a rate of failure between 25% and 80%.
On the other hand, programs play an essential role in most sciences nowadays, in analysis, measurements, and so on. Therefore, programming skills have become part of the training of many physicists, and biologists (especially in genetics), which gives to the problem of learning to program a wider scope than mere computer science students.

Why can programming be so difficult to learn? We will first try to answer this question by proposing a decomposition of a classical programming activity, and by linking to each step its associated psychological and didactic issues in the literature.

2. DECOMPOSITION OF THE PROGRAMMING ACTIVITY AND CLASSIFICATION OF THE STUDENTS' TROUBLES.

According to Duchateau [2] & [3], a programming task can be defined by its purpose : to "have a task done by a computer". Chronologically, the activity of conceiving a program starts by the accurate analysis of the task, when the programmer asks himself the question "What should I do?". This step is usually not pertinent in initiation to programming, because the target tasks are either trivial, either already well-known, and is more related to software engineering courses. Their first difficulties in conception therefore appear when they try to abstract all the different behaviours of the task in a strategy which summarises them all (figure 1).

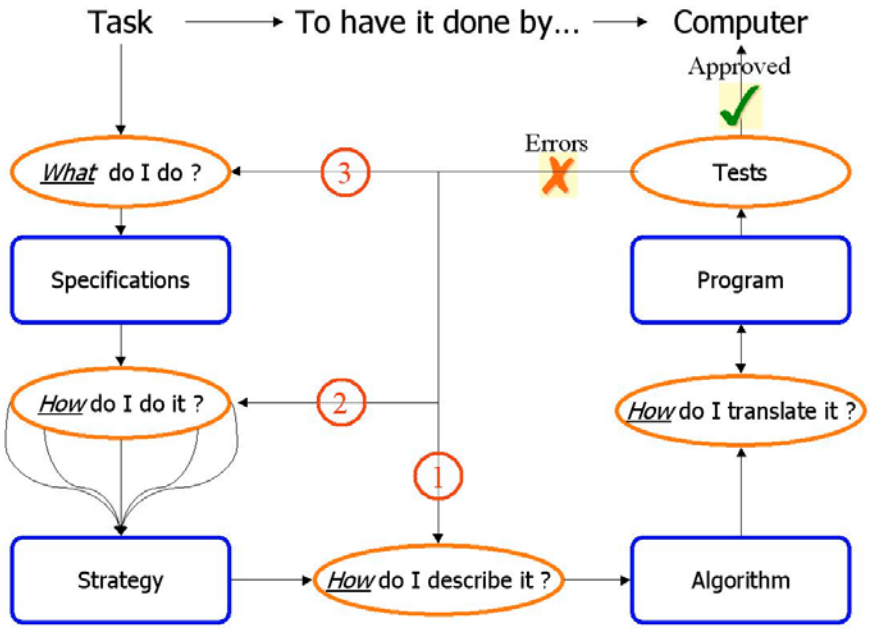

Figure 1: Decomposition of programming activity.

To do this, he/she must build a viable model of the temporal structure of the task, which will allow him to understand and predict the dynamic behaviour of the program. The next problem is to describe this strategy in an understandable way for the computer performer.

A specific difficulty of programming courses is that, contrary to other sciences, like physics, the novice student in programming has no "naïve" starting model of the "inside" the computer, which is a great obstacle to abstract more elaborate models [4]. A person "programming" a VCR describes a sequence of operations that he already performed in direct-manipulation style and that are, therefore, already familiar to him. A person "programming" in HTML uses a particular mark-up language to abstract and describe a concept which is, once again, already familiar : the presentation of his document. On the other hand, in "genuine", "classical" programming, a programmer writes programs which abstract some behaviour "inside" of the computer, whose basic objects and operations are completely unfamiliar to a novice. This leads, in practice, to two different types of novice mistakes:

- "Anthropomorphic" mistakes ([5]; [6]; [7]): seeing initially the computer as a "black box", the student acts has if there were a "magical" hidden mind inside, which would allow the computer to infer what is meant but left unsaid, instead of blindly performing what is written.
- Errors linked to the digitization of the objects of the task [8], caused by the cognitive distance which separate the natural representation of these objects from their representation inside the program [9].

Usually, objects from the domain of the task are said "analogical" as their representation are similar in structure to the things they describe. On the other hand, the descriptions of objects in a programming language are abstract representations, said "fregean" as they have no such similarity; consider the description of a triangle as "int [][] abc={{2,2},{4,13},{10,6}};" (figure 2)!

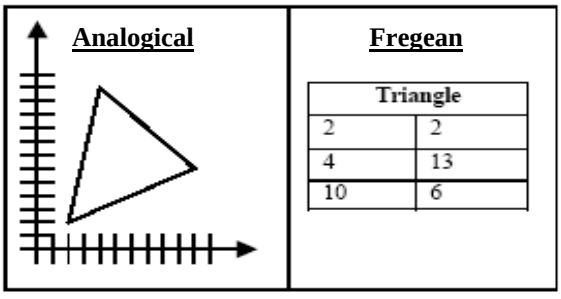

Figure 2: The "analogical gap" of data.

The last step of the process consists in evaluating the results of the executed program, which involves exactly the same difficulties; and as it is easy for the novice programmer to make false assumptions about the state of the system this usually results in introducing new errors when correcting.

A specific problem of this conception cycle is that the programmer is confronted to the lacks of his conception in a reverse order: he first debugs its algorithm to find sometimes that his strategy is not working in every case, and eventually that his specifications are false or incomplete. This explains why the common "top-down" approaches of learning to program, which involve learning activities associated to each step of the conception cycle, have not succeeded, as reports Rogalsky [10]:

> *« The observation of students beginning in Computer Science in a class context seems to stress that the analysis methods taught to them are not perceived as a tool to resolve their problems, but as a contract (with the teacher) they have to respect.*
> *The expression of the "analysis phase" most frequently appears as a paraphrase of the program, which can append before, at the same time, or even after typing the program in a particular programming language».*

Another common difficulty, which appears at the very beginning of the conception of a program, is the "blank page syndrome"; students don't know where to start from, as reported by [11] :

> *"… when problems are presented … you start at the top and do so and so and so. All is very logical and simple and then you sit down and ouch. Where do I start? It may be that it's easy, but the problem is that you don't know in which end to start when you have to solve a problem. And they have told it for certain, but it's, you know, important to do in the tutes instead of only going through the solution quickly."*

This common class of problems shows there might be a problem, in the professional environments used for teaching, and in other pedagogical tools as well, with the support of the conception steps: programming environments only support evaluation, of the syntax and of the behaviour of the program once it is written (Arrows 1, 2, 3 in figure 1) and do not support its conception.

This classification demonstrates that learning to program cannot be reduced to learning the syntax of a particular language, as well as it takes more than learning musical notation to be a composer. More important, it stresses the fact that novices difficulties and errors are not only related to the (often difficult and tedious) syntax of these languages, but to the need of student to construct a correct model of how the programs are executed, and the difficulty of this task. At the novice level, the claim is supported by many studies. Du Boulay ([7], p. 285) notes that "Even if I no effort is made to present a view of what is going on 'inside', the learners will form their own." Perkins et al. ([12], p. 162) "attribute students' fragile knowledge of programming in considerable part to a lack of a mental model of the computer...." Sleeman et al. ([13], p. 251) found that "even after a full semester of Pascal, students' knowledge of the conceptual machine underlying Pascal can be very fuzzy."
Computer science is unlike school physics: intuition and manipulative facility are no use, and the consequences of misconceptions are immediately exposed, causing bugs. This brutal feedback is certainly the cause of much psychological grief experienced by (Goold et Rimmer [14])'s or (Wilson [15])'s students (table 1)

	Basic Concepts	**Data Structures and Algorithms**	
	Total	Exam	Total
Variable	Regression coefficients		
Average sore in other disciplines	0,32	0,89	0,71
Has done programming before university.	-	-	12,32
Disgust of programming	-12,50	-28,38	-22,40
Problem Solving ability	1,36	-	-
Sex (m|f)	7,52	-	-

Table 1: Factors contributing to success in introductory CS courses.

3. From the "black box" to the "glass box"

If we try to abstract these difficulties, we can conclude than the hardness of programming is clearly related to how programs are conceived and edited. The classical edition paradigm for programs (compile-run-debug-edit loops) turns the computer into a black box, whereas a good novice programming environment should be able to show students how everything is going on inside, so they could infer a correct execution model.

3.1 "Example-based" Programming : Definitions and Implementation.

In an attempt to reduce the inner difficulties of programming task, Smith introduced with Pygmalion [8] the concept of «Example-based Programming» (EbP) [16, 17]. Its main idea is that knowing how to perform a task should be enough to program the task. The edited program is associated with a concrete example, which allows an active feedback of the program behaviour. Two different techniques are associated with this concept: Programming with Example, and Programming by Demonstration. In programming with example, the user edits a program in a particular language, and this program is interpreted in real time in the context of the example. In programming by demonstration, the user demonstrates, by manipulating the example, the desired behaviour, and the program is constructed automatically. In both cases, several examples are often needed to construct most non-trivial programs.

In the topic of Novice Programming, EbP seems to have several advantages on a classical approach. The most important is active feedback: contrary to the classical "compile-run-debug-edit" loop, the novice programmer can immediately see how is interpreted what he has written. It should therefore be easier for him to build an "execution model" of the inner of the computer. To measure the effects of EbP in an initiation tool for programming we built the MELBA (Metaphor-based Environment to Learn the basics of Algorithmics) EbP environment. Figure 3 displays this environment and how the concepts of EbP are implanted.

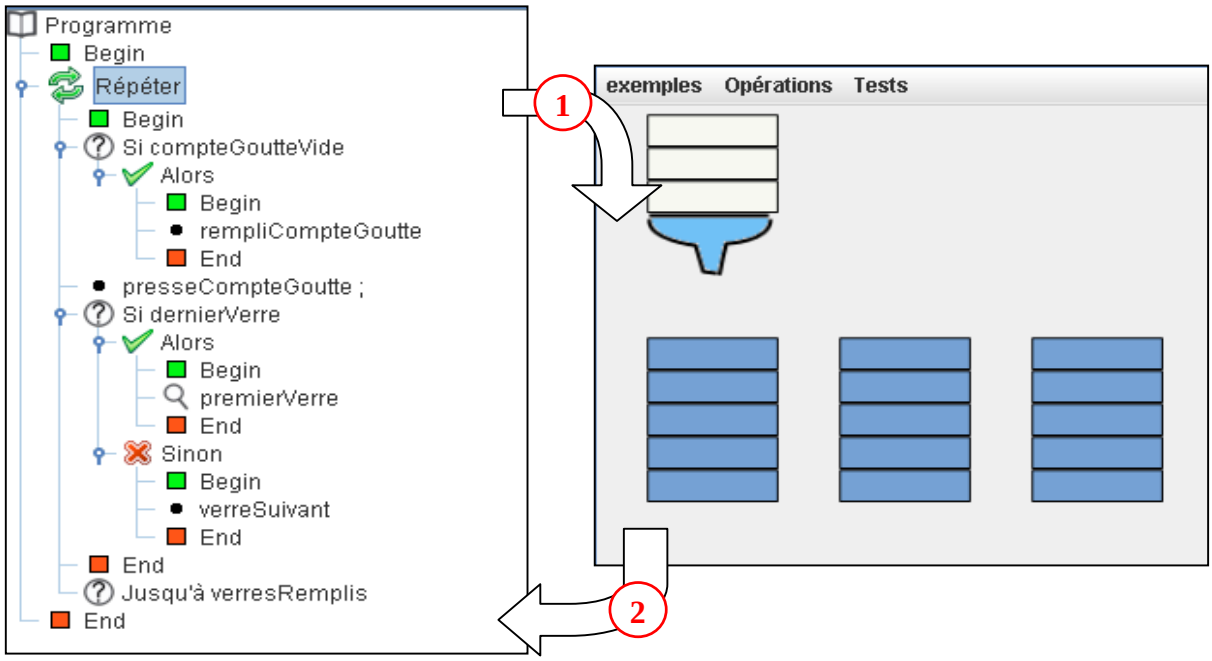

Figure 3: Example-based Programming in MELBA.

The example above (figure3) displays two different components of the Melba EbP system, in the context of a particular exercise, whose goal is to write a program which fills glasses with a dropper. The available instructions are:

- *fillDropper* ,
- *firstGlass* which positions the dropper on the first glass,
- *nextGlass* which steps to the following glass,
- *pressDropper* (a single drop falls into the current glass)

The computer is able to test: if all glasses are full, if the current glass is full, if the dropper is empty and if it is on the last glass. The programmer can also create a new example with a given capacity and initial content of the dropper, a given number of glasses with given capacities and initial contents, or load an existing example.

The left frame in figure 3 displays the program, and allows the programmer to interact with it : when he/she clicks a statement (in figure 3, the REPEAT statement) this statement is highlighted, and the system runs the program until it executes the highlighted statement. This run is animated on both left and right frame. On the program, the magnifying glass icon shows the instruction being performed; the example, which is displayed analogicaly, is synchronised to the current state of the program. When the programmer wants to edit the program, he clicks on a node in the program tree on the left, and the system goes to that particular state. There, he/she can add a statement, and the example synchronises on the state after the execution of the newly added statement. This mechanism implements the "Programming with Example" concept (arrow 1).

"Programming by Demonstration" (arrow 2) mechanism allows him/her to perform simple or composed actions on the example itself (here with the "Operations" menu, alternatively with direct manipulation) which modify the program : "press the dropper" action would add the "*pressDropper*" statement the program, as well as performing it on the example (it is a simple action), while "empty the dropper" (a composed action) would drop the whole content of it in the current glass, and add to the program, at its current state, the following:

"REPEAT UNTIL *dropperEmpty*
BEGIN
pressDropper
END"

3.2 Early measures of the effects of Example-based Programming.

In order to measure the influence of EbP as a tool to learn programming, we lead an experiment on 65 first year students in bio-informatics. The students were split in several groups, some using Melba to introduce the concepts of control flow, others not. Randomisation process took care of keeping the same rate of students having a prior experience of programming in all groups. A specific part of their exam was specifically dedicated to the comprehension and writing of programs in an algorithmic language. Table 2 shows the results of the different groups.

	Class	**EbP groups**	**Non EbP groups**
Average score (arithmetical mean)	59,74358974 %	64,39393939 %	57,36434109 %
Standard Deviation	37,60313453	33,84349857	39,55812908
Deviation of low* scores from 50% * <=50%	30,76004804	23,57022604	33,92334964
Percentage of scores >=50%	63,07692308	76,19047619	58,13953488

Table 2: results of students with and without EbP support in first programming course exams.

The results of the non-EbP groups can be regarded as control results, which witness of the general results of initiation to programming courses, and especially a great disparity, as suggests the combination of the fair arithmetical mean and of the important standard deviation. These results are encouraging, since not only did the EbP support enhance the average results, but did it reduce the number of really (close to 0) low scores, as shown by the lower deviation of scores <=50%,and increased the rate of scores over 50%.

4. DEEPER DESCRIPTION OF THE ENVIRONMENT.

This experiment also gave us some keys to upgrade the general usability of the environment, and encouraged us to develop some other important features, in order to support the acquisition of more programming skills. In the following sections, we will first describe two particular components that have been added since this experiment, and enlighten the pedagogical need they support. Then we will conclude by describing the future experiments we will make on the upgraded environment.

4.1 Graphical history of the program run.

The first component was added because the experiment exhibited a particular lack in the existing environment; whereas the program frame gave the student a global view of the algorithm, no component displayed such general information about the course of the example, the "example" frame displaying only a local information in time: the current state of the example. To help the students to backtrack the origins of the bugs in their programs, we therefore designed a component whose role is to display general information about the course of the example (figure 4).

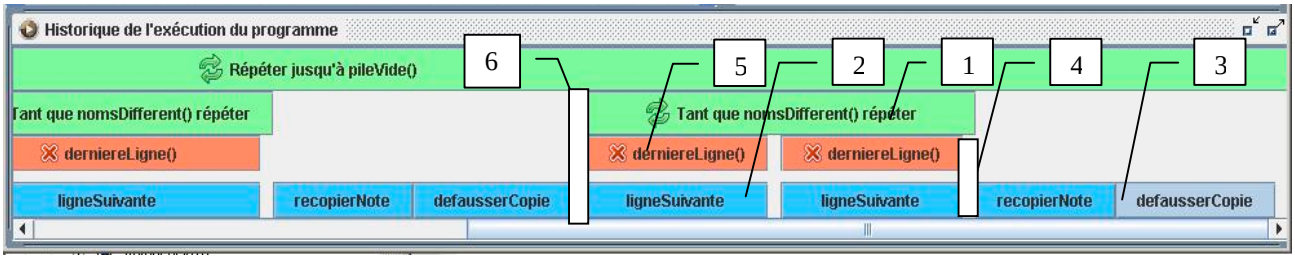

Figure 4: graphical history of the example.

This history takes the shape of a tree, allowing the programmer to trace faster where in the program frame is the statement currently performed. Here, the Y axis gives us the information that statement 2 is inside an "else" branch, labelled by its condition (5), which is in inside a while loop (1), itself inside the highest level "Repeat" loop. The X axis give a temporal information. The latest-executed instruction is shown as selected (3), and the blanks show the end of a loop turn : blank 4 shows the end of the second turn of loop 1, whereas blank 6 shows an end of turn for the loop at the highest level.

This component also enables direct interaction : by clicking a particular low or higher level statement, the environment goes back/forward to the corresponding state of the system, and the example and program frames are synchronised automatically. This feature allows accurate backtracking, enabling the novice programmer to answer the question : "why did it pass by that particular branch ?". We also believe these features will facilitate student-student and student-teacher dialog, serving as a tool for explaining the behaviour of the program.

4.2 A metaphor-based approach to explain the concepts of variables and data types.

The second component added to the environment has for objective to display the current context of the system, not in terms of user or task object, but with a system-centric point of view. We plan to use it as a tool to fill the abstraction gap between these two different descriptions of data, and therefore to increase the understanding of "variable" or "data type" concepts.

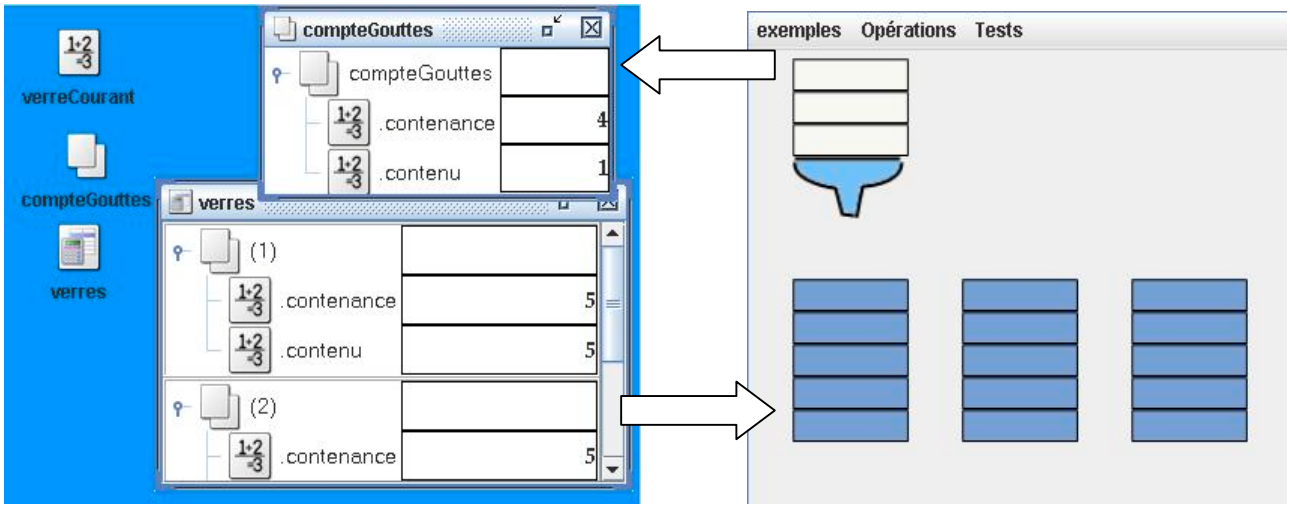

Figure 5 : System-centric and Human-Centric representations of the state of the system.

To do that, we chose to visually represent all the imperative programming objects and concepts, but through a metaphor (figure 5). This approach has already been tried out in an end-user programming context (the goal was then to empower users to easily write programs, not to teach specific Computer Science concepts) with another paradigm of programming ("parallel", multi-threaded, programming) by the ToonTalk system [18]. We wanted to use it to make quantitative measures of the effect of a metaphor-based learning system to teach programming concepts. We chose the desktop metaphor, for two principal reasons.

First, this metaphor is familiar to our audience, and is the classical way to display what's "inside" the computer. Second, there is a strong mapping between programming objects and concepts and desktop objects or operations. For instance, the document is a named box that contains one typed data (= variable). The desktop metaphor also provides a mapping for the concepts of input or output parameters ("Open …", "Save as…"), and for the concept of assignment (copy-paste mechanism).

Contrary to rote learning where using a metaphor (ex. the analogy variable = box [7]) might lead students to create false mental models, here the interaction with the concrete visual model can be used to avoid misinterpretations of the analogy (when a student pastes (=assign) the content of a variable in another variable, he does *see* that this operation actually *replaces* the older content of the target, and as the system does only enable it when the target that has the same data type).

Our objective is to teach students a correct model of the computer, by using the metaphor itself as a teaching tool, and by making the novice programmers study the relationships between the analogical view of the example frame, and the system-centred view in the desktop metaphor. The ultimate goal of this study is teach which to correctly chose the data required to solve a particular programming problem.

5. CONCLUSION

In this paper, we have summarised and classified the different difficulties students are faced to in the beginning of their learning of programming. We raise the idea that the human-computer interaction style involved by the classical conception approach (which is non interactive, abstract, requires to build complex representations of the machine state, and to mentally animate them, requires to adopt a machine-centric mindset) used in nowadays programming are strongly linked to these difficulties.

We present an alternative approach for conceiving programs, known as “Example-based Programming” (EbP). Although an early tentative did exist for the LISP language (Lieberman’s Tinker [19]), no EbP programming system had ever been conceived in the scope of teaching imperative programming. No quantitative study has ever been made of the impact of an example-based approach to the learning of programming concepts. In this paper we describe a system implementing the different concepts involved in EbP, that has not been conceived in the purpose to construct programs, but in the purpose to explain the programming process in imperative languages. We present quantitative measures of the impact of this system in understanding control flows of algorithms (which embeds the skills of understanding and of writing an algorithm). We describe extensions of this system built to support other programming skills.

We plan two other series of experiments, in the purpose to study globally the impact of the environment on the learning of concepts such as data types and parameters, on one hand, and on the other hand to study the particular impact of each interaction concept used in the environment.

6. REFERENCES

1. Kaasboll, J., *Learning Programming*, . 2002, University of Oslo.
2. Duchâteau, C. *From "DOING IT ..." to "HAVING IT DONE BY ...": The Heart of Programming. Some Didactical Thoughts.* in *NATO Advanced Research Workshop "Cognitive Models and Intelligent Environments for Learning Programming"*. 1992. S Margherita, Italy.
3. Duchâteau, C., *Images pour programmer*. Vol. 1. 2000, Namur: Facultés Universitaires Notre Dame de la Paix. 166.
4. Ben-Ari, M. *Constructivism in Computer Science Education*. in *29th ACM SIGCSE Technical Symposium on Computer Science Education*. 1998. Atlanta Georgia: ACM press.
5. Pea, R.D., *Language-Independent Conceptual “Bugs” in Novice Programming.* Journal of Educational Computing Research, 1986. **2**(1): p. 25-36.
6. Spohrer, J.G.a.S., E. *Analysing the high frequency bugs in novice programs.* in *First Workshop on Empirical Studies of Programmers.* 1986.
7. Du Boulay, B., *Some Difficulties of Learning to Program*, in *Studying the Novice Programmer*. 1989, Lawrence Erlbaum Asssocites. p. 283-299.
8. Smith, D.C., *Pygmalion, An Executable Electronic Blackboard*, in *Watch What I Do : Programming by Demonstration.*, A. Cypher, Editor. 1993, The MIT Press: Cambridge, Massachusetts.
9. ARSAC, J., *Préceptes pour programmer*. 1991, Paris: Dunod.
10. ROGALSKY J., S.R., HOC J-M., *L'apprentissage des méthodes de programmation comme méthodes de résolution de problème.* Le travail humain, 1988(51): p. 309-320.

11. Carbone, A., Hagan, D. L. & Sheard, J. *Consolidate, preserve, and build: a tutor training program for a new school.* in *Australasian Conference on Computer Science Education*. 1998. Brisbane, Queensland, Australia.

12. D. Perkins, S.S., and R. Simmons, *Instructional strategies for the problems of novice programmers.*, in *Teaching and Learning Computer Programming*, R.E. Mayer, Editor. 1988, Lawrence Erlbaum Associates. p. pages 153-178.

13. D. Sleeman, R.T.P., J. A. Baxter, L.Kuspa, *An introductory pascal class: A study of student errors.*, in *Teaching and Learning Computer Programming.*, R.E. Mayer, Editor. 1988, Lawrence Erlbaum Associates. p. 237-257.

14. Goold, A.a.R., R., *Undergraduates in business computing and computer science (poster session).* SIGCSE Bulletin, 2000. **32**(3): p. 188.

15. Wilson, B.C. *Contributing to Success in an Introductory Computer Science Course: A Study of Twelve Factors.* in *SIGCSE*. 2000: ACM.

16. Cypher, A., ed. *Watch What I Do: Programming by Demonstration*. . 1993, The MIT Press: Cambridge, Massachusetts. 604.

17. Lieberman, H., *Your Wish is my command*. 2001: Morgan Kaufmann. 416.

18. Kahn, K., *How Any Program Can Be Created by Working with Examples*, in *Your Wish is My Command*, H. Lieberman, Editor. 2001. p. 21-44.

19. Lieberman, H., *Tinker: A Programming by Demonstration System for Beginning Programmers*, in *Watch What I Do: Programming by Demonstration*, A. Cypher, Editor. 1993, The MIT Press: Cambridge, Massachusetts. p. 49-66.