Example-based Programming: a pertinent visual approach for learning to program.

Nicolas Guibert
Laboratoire d'Informatique Scientifique et Industrielle,
ENSMA - University of Poitiers
BP 40109, 86961 Futuroscope Cedex, France.
33 (0)5 49 49 80 70
guibert@ensma.fr

Patrick Girard
Laboratoire d'Informatique Scientifique et Industrielle,
ENSMA - University of Poitiers
BP 40109, 86961 Futuroscope Cedex, France.
(33) (0)5 49 45 37 24
girard@ensma.fr

Laurent Guittet
Laboratoire d'Informatique Scientifique et Industrielle,
ENSMA - University of Poitiers
BP 40109, 86961 Futuroscope Cedex, France.
(33) 05 49 49 80 67
guittet@ensma.fr

ABSTRACT

Computer Science introductory courses are known to be difficult for students. Kaasboll [1] reports that drop-out or failure rates vary from 25 to 80 % world-wide. The explanation is related to the very nature of programming: "programming is having a task done by a computer"[2]. We can notice three internal difficulties in this definition:

- The task itself. How do we define it, and specify it?
- The abstraction process. In order to "have it done by…" students need to create a static model covering each task behavior.
- The "cognitive gap". It is difficult for novice programmers to model the computer, and its "mindset", which is required to express the task model in a computer-readable way. The bad usability of programming languages increases this difficulty.

The lack of interactivity in the editing-running-debugging loop is often pointed as an important aggravating factor for these difficulties. In the mid-seventies, Smith [3] introduced with Pygmalion another programming paradigm: Programming by Examples, where algorithms are not described abstractly, but are demonstrated through concrete examples. This approach involves several advantages for novices. It allows them to work concretely, and to express the solution in their own way of thinking, instead of having to embrace a computer-centered mindset. The programming process becomes interactive, and as PbE languages are "animated" languages, no translation from the dynamic process to any static representation is required.

In this paper we investigate both the novice programmer and existing PbE languages, to show how visual and example-based paradigms can be used to improve programming teaching. We give some elements of a new Example-based Programming environment, called Melba, based on this study, which has been designed to help novice programmers learning to program.

Categories and Subject Descriptors

D.1.7 [**Visual Programming**].

H.1.2 [**User/Machine Systems**]: Human Factors.

H.5.2 [**User Interfaces**]: Graphical user interfaces (GUI).

I.3.6 [**Methodologies and Techniques**]: Interaction techniques, Languages.

General Terms

Design, Human Factors, Languages.

Keywords

Metaphors, Example-based Programming, Visual Programming, Didactics for Computer Science.

1. INTRODUCTION

The programming process traditionally involves the static and abstract description of a dynamic task in a dedicated language, in order to teach a computer how to perform this task. At this point, the system checks the syntactic correctness of this algorithm and then allows the programmer to test the correctness of the program itself. But this type of interaction between the computer and the programmer proved to be inappropriate for beginners: as related by Kaasboll between 25 and 80% of students world-wide either fail or give up introductory courses.

Prior works investigating novice programmers troubles have allowed us to summarize their errors in a simplified taxonomy based on three layers of programming expertise. Pragmatics is the definition of the task. Semantics define the computer performer, and syntax refers to the medium used by the programmer to

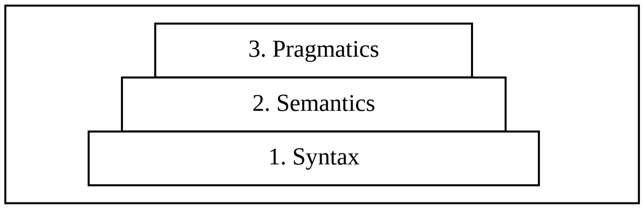
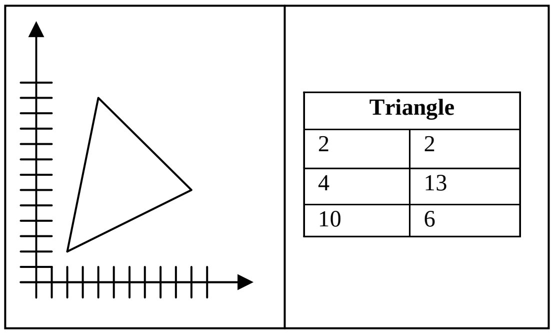


Figure1 . A common decomposition of programming knowledge : the "semiotic ladder"

express the abstraction of the task behaviors in computer readable way.

When starting experiencing programming, novices first encounter syntactic errors raised by the compiler. When (at last!) their program syntax is corrected, they run it, mostly often to discover it doesn't perform as they expected it to; in beginners case this generally relates to misconceptions of the machine model. We propose to summarize them in three classes: *temporal* errors are characterized by misconceptions of the control structures (including the most basic : sequence); *anthropomorphic* relate to the common hidden belief that the computer is able to understand what is meant, not only to perform what is written; the last class of semantic troubles refers to the difficulties to cross the cognitive gap of data representation between the task domain and the computer domain. Most often, data in the domain is *analogical*, that is, the shape of data reveals the concept. On the other hand, computer world requires numerical data representations that lack this analogy and are said *fregean*, see figure 2.



Triangle	
2	2
4	13
10	6

Figure 2. On the left, an analogical representation of a triangle ; on the right, an associated fregean model used in programming.

Finally, students are confronted with pragmatic errors; that is they realize their abstraction of the task was not complete, resulting in bugs; plus, they have trouble modeling non-trivial tasks (the gap between programming and software engineering skills).

In an attempt to reduce the inner difficulties of programming task, Smith introduced with Pygmalion [3] the concept of "Programming by Examples". Its main idea is that knowing how to perform a task should be enough to program the task: the programmer "plays the role" of the computer-performer and demonstrates concrete examples of task behaviors in a direct manipulation interface; while the programmer role-plays, the PbE system records the associated program. We will demonstrate an example of PbE in Pygmalion (since pygmalion is the PbE pioneering system, it can be used to show the core of PbE, without interfering concepts), the minimum of two numbers.

First we define this function by creating a function icon, figure 3 (in Pygmalion a box contains a value, which can be either a litteral or the result of an expression).

Then we define our concrete case by filling the arguments. After that, we call the "if … then … else" control structure. We create a "<" test and put it in the "if" icon by drag and drop. After that, we drag and drop the "6" box to fill the first argument of the "<" operator (figure 4). By doing this, we inform the system that the first argument of "<" is the first argument of "Min". We do the same for "4", and then the system automatically evaluates the test as false. We drag and drop the second argument into the return box and thus fill execute the return statement. And it's done, we

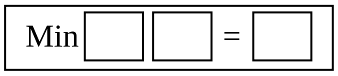


Figure 3. A function icon in Pygmalion, with two parameters.

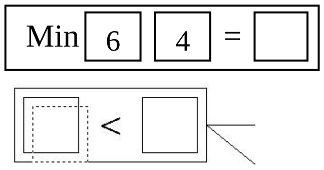


Figure 4. Defing by filling arguments with a litteral value or the evaluation of an expression

have programmed:

```
"function Min (integer a, integer b)
 return integer
begin
      if  not(a<b) then return b
                  else ????
end"
```

This incomplete algorithm is already usable, in the case (a>b). To complete the definition, we call the function in the case (a<b), the system drives us back in the edition environment to program this case. We've defined the function in two concrete examples, with the programmer playing the role of the computer-performer.

In the next parts, we will relate programming by Example systems to the different layers of programming knowledge, and we will show how such concepts can be integrated to the teaching of programming, and what benefits it can bring to students.

2. A Typology of Programming by Examples in a teaching programming perspective

If we refer once again to Duchateau's definition ("programming is having a task done by a computer-performer") and relate pragmatics to modeling the task, semantics to knowledge of the computer-performer, and syntax to the language which interfaces the programmer who models the task and the computer which performs the program, we can already notice an important difference between programming by examples and classical programming paradigms. On one hand, the programmer creates a static plan to command the computer, and on the other hand, the programmer role-plays the actions of the computer-performer, and the system "learns" by generalizing. This "First-Person Programming" style reduces in itself the gap between the user and the computer-performer. The programmer has no difficulty to infer the abilities of the system, because all the actions the computer can perform are displayed in a graphical interface. We can classify the different PbE systems in three classes, depending on what this graphical interface exactly displays.

2.1 Syntactic PbE systems

In syntactic PbE systems, such as Pygmalion, the computer displays a set of instructions, and programming is done by creating and filling these instructions (with results of expression or litterals). For instance, in the previous example, the interaction objects where the "<" expression, the "if … then … else" branching statement, and the "Min" function we were editing. The object displayed by the PbE system is therefore the program. As the programming model is a manipulation of statements, we label this first class of systems as "Syntactic".

2.2 Semantic PbE systems

A second class of PbE systems does not display the program itself, but the program context only. Usually, such system-centric representations use a metaphor for increased usability. We label these systems as "semantic" as they model the computer-performer. A good example of such system is Ken Kahn's ToonTalk [4], which uses on a lego-looking micro-world to represent the objects of the program. Graphical operators (a dusty vacuum which can "draw up" values to generalize them as variables, scales are used for tests… figure 5) provide a metaphorical knowledge of the computer-performer state and abilities.

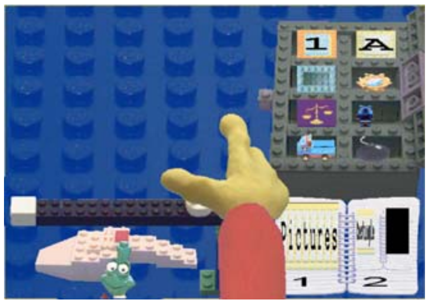


Figure 5. Using a metaphor to display the « computer algebra » (typed system data and associated operators)

In a teaching perspective, semantic PbE might be useful to provide support to the concepts of variable and data types, when powered by the appropriate metaphor.

2.3 Pragmatic PbE systems

Whereas semantic PbE systems use metaphors to provide to an intended audience understanding of the computer-performer, the PbE systems we label as "pragmatic" have for objective to keep the programming process inside the task domain. Pragmatic systems do not try to provide a comprehensive representation of the computer; they put the computer world in a "black box", and try to make the programmer forget the box exists. A canonical Pragmatic PbE system is Smith and Cypher's StageCreator [5], whose goal is to enable kids to write animations, simulations or 2D games by Programming by Examples techniques: "programming is kept in domain terms, such as engines and track, rather than in computer terms, such as arrays and vectors".

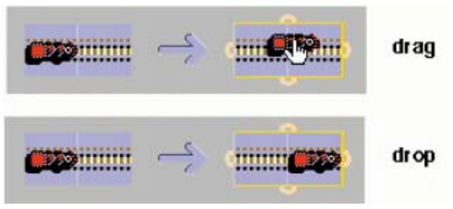


Figure 6. Pragmatic systems, as StageCast creator, take a domain-centric perspective, and allow the programmer to remain in the task domain while programming.

In a didactical issue, pragmatic systems could allow a complete novice programmer to learn the control structures of imperative programming, and how the system interprets an imperative program, without having to understand concepts like (fregean) data types, or variables. Of course, such pragmatic system must fit to an imperative programming style (unlike Creator).

2.4 Pedagogical issues

If we refer once more to Duchateau's definition of programming, using an adapted pragmatic PbE enables to teach the "abstract and complete modeling of the task" part of programming, putting the computer-performer in a black box. Once the students has acquired a correct temporal model of algorithm processing, using an adapted metaphor in a semantic PbE interface helps to understand what is the computer-performer; finally, using a advanced syntactic programming with examples interface helps the student to cross the "having done by…" obstacle without being disturbed by misspelling. We believe that dividing programming difficulties in separate phases has obvious didactical advantages. In the next parts, we present the MELBA example-based system, which implements such approach in a learning tool for programmers.

3. Programming by Examples in learning programming: the Melba system.

3.1 Introducing control structures: example of the robot with the dropper

Using pragmatic programming by examples in learning algorithms control structures is best illustrated with an example. We will demonstrate how this works this example of the robot with the dropper [2], see figure 7.

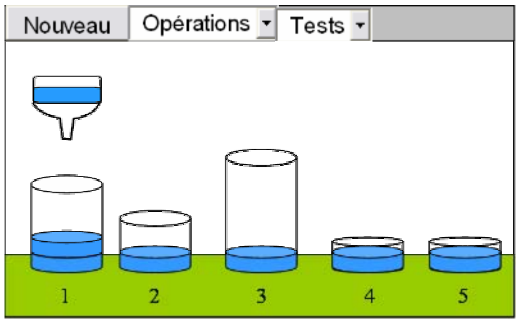


Figure 7. Pragmatic PbE in MELBA: the example of the dropper exercise.

Its working environment is: an alignment of glasses, and a dropper the robot can use. The computer-performer is able to (menu "operations") : position the dropper on the first glass, step to the following glass, press a drop into the current glass, fill the dropper. It 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 (Command New) can 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.

Now, let's demonstrate how the environment can be used to program the task of filling all glasses. The novice student selects the "operators" menu, and the "On_first_glass" command. The system creates an history in the shape of a movie tape, with the before and after states of the action, and writes the code on a contiguous frame. Then he/she selects the "press_drop" and the

"On_next_glass" and continues by selecting the "redo" command, specifies the group of actions he/she want to redo, and sees options to : redo <n> times, or redo until <?>. Student chooses : "until <on_last_glass>", because he/she noticed thenumber of glasses is variable from example to example. This creates a loop in the program frame, and starts running it … but soon generates an error message : "Error : -dropper_empty-". So the student steps back one action, and inserts in the history : "fill_drop". But the system knows that part of a loop has been edited, and therefore prompts the students if this action should be executed (a) each time, (b) on a special condition. Choosing the last one creates a "if … then …". Similar problems will appear when the system will try "on_next_glass" when it is on the last one, or try "press_drop" on a full glass. This shows how Pragmatic PbE can be integrated in teaching programming, to show on concrete interactive examples when to use control structures and how they work . Using several examples to see how the created algorithm performs in other initial states is also supported and very important in this step of learning. Let us notice that the system prompts the user if editing the program makes it non-compliant with previous examples. An interesting approach is to ask other students if they can generate a counter-example with the "new" command.

3.2 Semantic PbE as a support of a correct modeling of the computer-performer: the desktop metaphor.

A difficulty novice students commonly encounter is trouble modeling what the computer can and cannot do. In a situation of linguistic communication this leads to the "superbug" (Spohrer 1986) or "anthropomorphic bug", the hidden belief the computer is able to infer their intents from incomplete specifications. This problem demonstrates the need, in order to learn programming, to define a good model of the computer-user. This is where semantic PbE enters in action. We chose the computer desktop for representing the computer-performer, because it had several advantages in this perspective. First, the Window-Desktop metaphor is familiar to our audience, and is the classical way to display what's "inside" the computer. Second, there is a strong mapping (table 1) between programming objects and concepts and desktop objects or operators: the document is a named box that contains one typed data (= variable). The window metaphor also provides many applications (= libraries) to manipulate numbers, text, trees and tables. These applications are composed of many procedure or functions (the concepts of input or output parameters are also supported in the metaphor: "Open …", "Save as…").

Table 1. Links between the concepts of imperative programming and the metaphor

ADA Programming Language	**MELBA semantic Programming by Demonstration**
A numeric variable (types Integer, Natural, Positive, Float, String, Boolean, Arrays et Records, access)	A document (documents use icons of their default application)
Assigning to a variable	Editing the document, then saving it as… , or drag and drop.
Calling a procedure or a function	Pressing a button or clicking a menu item
The program context	The computer desktop

The metaphor helps understanding the "computer's mind", and we link pragmatic and semantic representations in order to help the student to understand the relationships between the two.

4. Conclusion

In this paper, we introduced the problems of learning programming. We raise the idea that the poor means of interaction (non interactive, abstract, require to learn a difficult syntax, require to build complex representations of the machine state, and to mentally animate them, require to adopt a machine-centric mindset) used in nowadays programming are strongly linked to these difficulties. We suggest to adopt another learning model for programming, which allows to divide the students difficulties in successive learning steps, and study the usability of an alternative programming paradigm, programming by examples. We build a taxonomy of PbE systems relaying on their demonstrational interface, and link each type of interface to a particular step in the learning process. Then we present a new visual programming by examples system named MELBA, which was built for learning programming. This system, for now a prototype, is going to be transformed into a completely functional environment, in order to lead a study on the effects of the system and its associated approach with actual students.

5. 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] Smith, D.C., *A Computer Program to Model and Stimulate Creative Thought*. 1977, Basel: Birkhauser. 187p.

[4] 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.

[5] Smith, D.C., *Novice Programming comes of Age.* Communications of the ACM, 2000. 43(3): p. 75-81.