International Symposium on End User Development
Schloss Birlinghoven, Sankt Augustin (Bonn), Germany - October 6-7, 2003

Teaching and Learning Programming with a Programming by Example System

Nicolas Guibert, Patrick Girard
{guibert, girard}@ensma.fr

LISI/ENSMA
Téléport 2 - 1 avenue Clément Ader - BP 40109
86961 Futuroscope Chasseneuil cedex

1. Introduction

The programming process traditionally involves the static and abstract description of algorithms in a dedicated language; then the system checks the syntactic correctness of these description and allows the designer to run his program to test if it does what it is supposed to. But this approach has proved to be difficult for many novices (studies in didactics for programming reveal rather high failure rates – between 25 % and 80 % world-wide according to [1] – in programming introductory courses). That's because it has to face two well-known problems :

- Bad usability of the system tools used for the human-computer communication (lacks of usability in editing phase, and lacks of interactivity in the editing-compiling-debugging cycle).
- Mistakes in the abstraction process which links the computer's dynamic tasks and the developer's static description of them with algorithms.

In the mid-seventies, D.C. Smith introduced with Pygmalion [2] another programming paradigm, Programming by Example(s), where algorithms are not described abstractly, but are demonstrated in concrete examples. Since then, Programming by Example(s) and other advanced HCI techniques, as metaphors and microworlds, have been used in several experimental tools for novice programmers, but up to date, none of them managed to equal classical languages expressiveness. Worse again, whereas the PbE approach proved to be an innovative and elegant technique to handle human-machine interaction, it doesn't handle at all the programmer-programmer communication aspect of a programming language. In ToonTalk [3], for instance, no static representation is provided, and thus the only way to "read" a program is to analyse the dynamic interactions of this program's objects while it runs. Other tools use dedicated textual languages, or a comic-strip metaphor to provide a static presentation.

In this position paper, we will discuss about the learning styles and didactic models associated with each programming paradigm; the differences involved in the student's learning style, and we will try to explain how both programming paradigm can be combined to grant the designer a better support on his ongoing programming process. As an example, we will shortly describe the Melba system, which embeds a "classical" visual language and a concrete examples handled with a desktop metaphor and an "internal" PbE engine to teach novice programmers imperative programming concepts.

2. Studies of the "conventional" programming learning process compared with a PbD learning process.

Comparative analysis of didactics studies referring to programming (for instance [1]) has led us to identify two generic classes of mistakes in traditional programming:

- "Temporal" mistakes, caused by a misunderstanding of the algorithm's dynamic process (typically inside a loop)

- "Anthropomorphic" mistakes, where the novice programmer acts as the interpreter I compiler had a "contextual" understanding of his algorithm, bypassing the written statements.

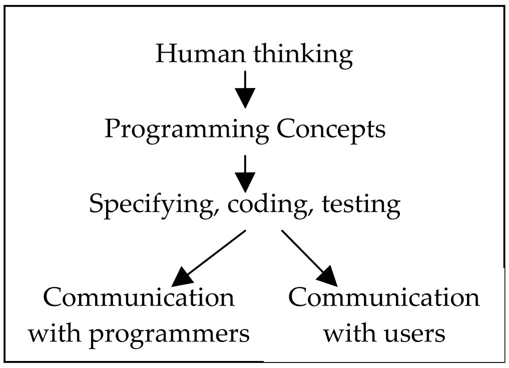

Figure 1. A summary of programming learning process

Moreover, most studies on novice programmers (as [4]) agree that the main difficulty for learners is to construct big programs from existing bricks. This seems to prove that there is an important step between understanding programming concepts and the mastery of the coding skill.

According to Booth's [5]studies, it exists an even more advanced level of competence, related to usability and extensions of programs, which is to consider programs as a mean of communication with users and with programmers (Figure 1).

If we try to use Booth's schema to model a PbD learning process, we can find many matches. The cognitive tasks of decomposition of the problem is the same (it refers to human thinking), and both paradigm use "programming concepts", as variables, parameters or conditions (although PbD may not use if-then-else and loop structures, neither Boole's Algebra explicitly) . In both paradigms, the user needs to specify its problem, and needs to produce a set of test by identification of each problem case and instantiation into a compliant example. In Programming by Demonstration however, the coding and testing tasks are both done on concrete examples (let's notice they are not merged: as the developer might have provided not enough examples, or not significant ones, a PbD-created program still needs testing and debugging).

Now let's look back at the two great classes of low-level errors in imperative algorithms. PbD offers obviously the great advantage upon classical programming style to nearly avoid temporal bugs, as it uses a dynamic description of the task. However, the "anthropomorphic" bugs would probably grow in number, as it might be easy for the programmer to forget to explain his direct manipulations. Plus, PbD makes it generally harder to handle "big" programs, because of the decomposition of the task in a list of many concrete (and often redundant) examples . With many PbD tools it's quicker to write an abstract algorithm in a classical language, than to generate a (set of) complient(s) example(s) and demonstrate it step by step.

To be complete, we have to mention that obviously abstract and example based programming do not use the same learning style. On a later analysis of his early PbD system Pygmalion, Smith relates classical programming with a "symbolic" learning style, whereas PbD is linked to an "enactive" learning style (where learning is accomplished by doing), according to Bruner's classification.

According to Bruner's view, these two different learning styles should all be preserved since they can often be efficiently combined. As we've seen, Programming by demonstration and abstract programming involve complementary learning style, and they share several cognitive tasks. Both envisage programming as problem solving, and both use the implicit metaphor of "teaching" the computer (with laws, or with examples).That's why we argue that both programming paradigms can be efficiently used in combination.

As an example, the MELBA (Metaphor-based Environment to learn to Build Algorithms) system, which uses concrete examples as well as abstract description to learn imperative programming concepts to university students, is divided in two parts : the context manipulation space and the program space.

Direct manipulations on the context modify the program (the abstraction process) whereas the program is executed on the context space (the instantiation process). We can notice the combination of both paradigms is stronger than use of only one: in this bubble-sort algorithm, the immediate feedback provided by the instantiation process would have granted the detection of the two temporal

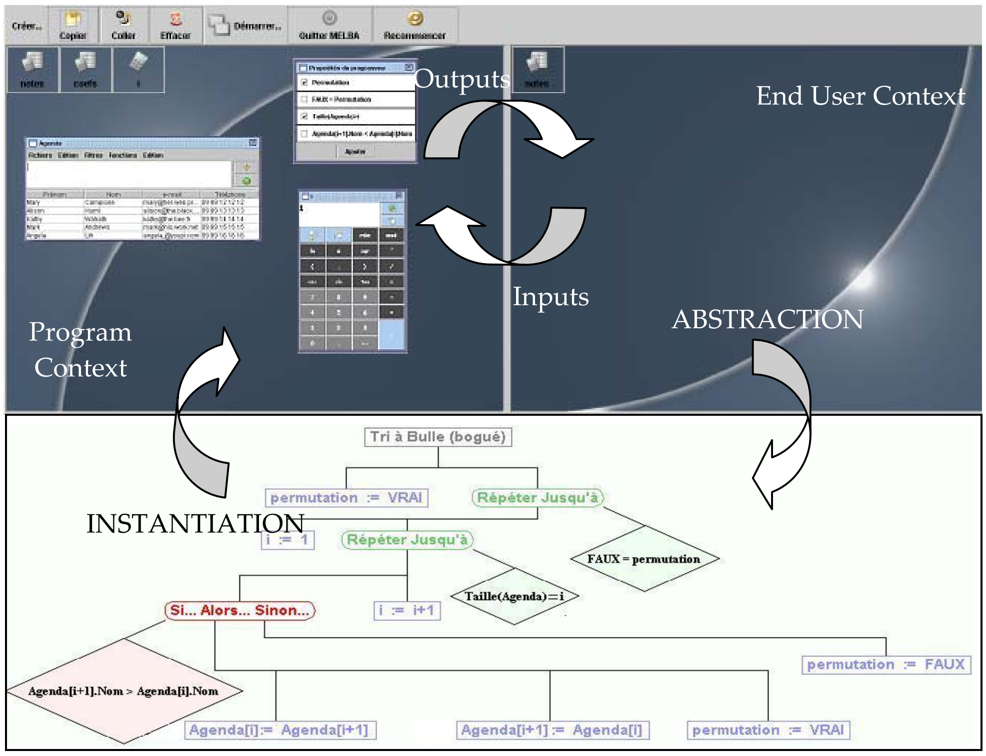

Figure 2. Interactions between contexts and program spaces in Melba.

bugs in the inner loop (the bug with line permutation is also anthropomorphic, since it is exactly what you would ask to a two-handed human interpreter), and the abstraction process gives the user an immediate feedback on the meaning for the system of its interactions within the PbD engine, avoiding most anthropomorphic bugs, as well in abstract as in concrete example mode (Figure 2).

3. Conclusion and Topics for discussion

In this paper, we reported the classical types of errors and difficulties in imperative programming learning. We argued that using classical abstract programming and programming by examples in combination helped to deal with many of these problems, whereas it didn't import specific PbD-related problems. Referring to Booth classifications of learning programming states, we argued that PbD learning involved basically the same cognitive tasks in the same order. Thus, how HCI techniques may support the cognitive tasks of dividing the problem, and how they can help the user structuring a large PbD task requiring several examples by supporting variables definition, and use of conditions seems to us good topics for discussion.

4. Bibliography

1. Kaasboll, J., *Learning Programming*, . 2002, University of Oslo.

2. 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.

3. 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.

4. Pea, R.D., *Language-Independent Conceptual "Bugs" in Novice Programming*. Journal of Educational Computing Research, 1986. 2(1): p. 25-36.

5. Booth, S., *Learning to program: a phenomenographic perspective*, . 1992.