A Didactic Interface in a Programming Tutor

Marilza Antunes de LEMOS [1, 2]
FACENS - Faculty of Engineering of Sorocaba [1]
Caixa Postal 355 - 18001970 - Sorocaba - SP - Brasil
PSI-EPUSP - University of São Paulo [2]
Av. Prof. Luciano Gualberto, 380 - 05508-900 - SP - Brasil
mlemos@lsi.usp.br

Leliane Nunes de BARROS
DCC-IME-University of São Paulo
R. do Matão, 1010 - 05508-090 - São Paulo - SP - Brasil
leliane@ime.usp.br

Abstract. This paper describes an interface that provides accessible cognitive components to programming learners during problem solving. We show how an Intelligent Tutoring System can keep the epistemological fidelity between its interface and its internal representation. For this purpose, we allow the student, throughout the interface, to select high level actions (cognitive knowledge components) and primitive actions mapped to *elementary patterns*, i.e., pieces of code commonly used by experienced programmers and educators.

Introduction

The goal of an Intelligent Tutoring System (ITS) for programming learning is to help students to acquire skills on problem solving. The main difficulties found by programming apprentices seem to be related to the cognitive reasoning and human computer interaction principles (HCI) [9][13][19]. However, most of the programming ITSs do not tackle the real problems found by apprentices. In short, some features that are usually pointed as the weaknesses of programming ITSs are:

(1) few tutoring systems have a programming environment with the appropriate components for the apprentices, such as a guided editor and an interpreter;
(2) most of the available tutoring systems interfaces do not communicate to the student their domain model in use, neither their reasoning process;
(3) generally, aspects of visibility are not considered in a suitable way: the memory overload is a problem for all programmers, however, this problem is particularly uncomfortable for beginners since they still don't have the necessary skills to deal with it. This suggests that a good interface of a programming ITS should provide visible and easily accessible information at any time;
(4) programming is a process of creating and translating a *mental plan* to a *program plan*, i.e., a solution plan that can be executed by the computer [18]. Therefore, a learning environment should minimize the difficulty of this translation process by providing high level actions to compose high level mental plans, that can be mapped to the target program by the student. When the environment (interface) does not supply such high level elements, learners are forced to build solutions to achieve the

problem goals composed with low-level actions, i.e., to reason about programming language instructions [8].

Empirical studies on Programming Psychology [3], show that apprentices use their background knowledge while trying to build their solution programs. However, by doing that they are able to create only *natural plans* which are different from *programming plans*, since they do not consider computational aspects. This problem is considered one of the greatest cognitive barriers for programming learning [8]. The purpose of this work is to bridge the gap between those plans.

Epistemical Fidelity in ITSs for Programming

Most of the intelligent tutoring systems do not follow what Wenger [19] called *epistemical fidelity* of representational mapping. On the one hand, there is the source of real knowledge, called epistemic knowledge. This knowledge is the one that we wish to represent in the ITS to allow the system: (i) to reason about it; (ii) to diagnose the student solution; (iii) to generate explanations; (iv) to communicate with the student. On the other hand, the student should manipulate and learn about this epistemic knowledge by means of an interface. For example, the syntax of a programming language can be an example of epistemic knowledge source. Other example can be, as we will see in this work, programming cognitive knowledge that corresponds to the programmers ability to solve problems.

Generally, it is not possible to directly introduce such real world knowledge in the tutoring system. Therefore, it is necessary to map epistemological knowledge source with its representation model: internal (for the system) and external (for the student). According to Wenger [19], the interface of an ideal tutoring system is rigorously an external representation of the expertise that the system has internally. When this happens, we say that the system has reach a communication power that guarantees its intelligence features, such as:
(1) the system can communicate its problem solving knowledge to the student;
(2) the system can control each communication step and monitor its effects;
(3) problem solving activities can be easily monitored and compared with the internal system reasoning;

To reach epistemic fidelity, the design of the external representation, the interface, must be strongly committed with a deep understanding of the domain. In this proposal, we include in the interface, the same knowledge embedded in the system, as we show in the next sections.

Goals and Contributions

In this work, we believe that a good interface can expand the communication power of an ITS and therefore helps students to learn the cognitive knowledge of the programming task. We show how an interface can work as a tool that provides support during the construction of programming mental models. Such an interface can help the student to build the target cognitive knowledge by making him to solve selected problems. We believe that by using this tool, the student can create his own programming mental models, i.e., his ability to construct high level programs (or mental plans) to solve problems. In order to reach this goal, the proposed interface contains cognitive components, based on programmers mental models [15]. By interacting with those components through an interface, the student can acquire problem solving abilities on programming. This assumption is based on the Norman's principle [7] :

"In interacting with the environment, with others, and with the artifacts of technology, people form internal mental models of themselves and of the things with which they interact. These models provide predictive and explanatory power for understanding the interaction."

1. Programming Task as Problem Solving

There is a general agreement that the main difficulty in introductory computer programming courses is acquiring skills on problem solving in the computer context. Although a learner knows how to solve very well a problem, such as a second order equation, he can find difficulties on building a computer program to solve it. Such activity includes understanding the problem, identifying its goals, finding programming plans that reach these goals and, finally translating the plans to an executable [18]. In short, given a problem to be solved by the computer, a programmer must acquire planning and computer encoding abilities. An intelligent tutoring system for learning these programming skills must have planning components and make them available to the students.

In this work we adopt the *Problem Based Learning* approach (PBL) [5], as the didactic strategy since successful experiences have been reported in the literature, showing that through problem solving, learners have expanded their programming skills and knowledge in a short period of time [2]. The process of solving programming problems consists of [10]:

(i) identification of the problem goals;
(ii) selection of programming plans that reach these goals;
(iii) selection of a set of actions (abstract actions) that will compose the program plan;
(iv) decomposition of the abstract actions in subplans (recursively), of the selected plan (step (ii)). This process finishes when there are no more abstract actions to be decomposed, i.e., the plan is only composed by primitive actions;
(v) codification of the primitive actions using a programming language.

1.1 Programmers Cognitive Components

Researchers agree that understanding mental models of expert programmers can be a valuable mean to teach programming students to construct their own mental models [6][11][12][14]. The programming learning interface proposed in this paper, makes *cognitive components* available to the student so he (she) can assemble them to form a computer program in two levels: (1) **planning level** and (2) **implementation level**. At the planning level a *Mental Program Model* can be generated [15], and at the implementation level, the programmer generates an *Implemented Program Model*. The components shared by the internal and external representations of the tutor are described below.

Programming Goal is a programming problem goal that must be achieved to solve the whole problem. Goals can be decomposed into subgoals. Goals are achieved by *programming plans*. It is possible to have different plans to solve the same goal. An example of programming goal is "*search for an element of a list* " (figure 1).

Programming Plan is a high level plan or strategy to achieve a particular programming goal. Plans can be decomposed into subplans and actions. A programmer can use different programming plans to reach the same goal. Different plans can involve different conceptual topics on programming in different levels of difficulty. Figure 1 shows how the "*search for an element of a list*" goal can be achieved by means of several plans, for example, the "*end-list"* plan or the "*count-times"* plan.

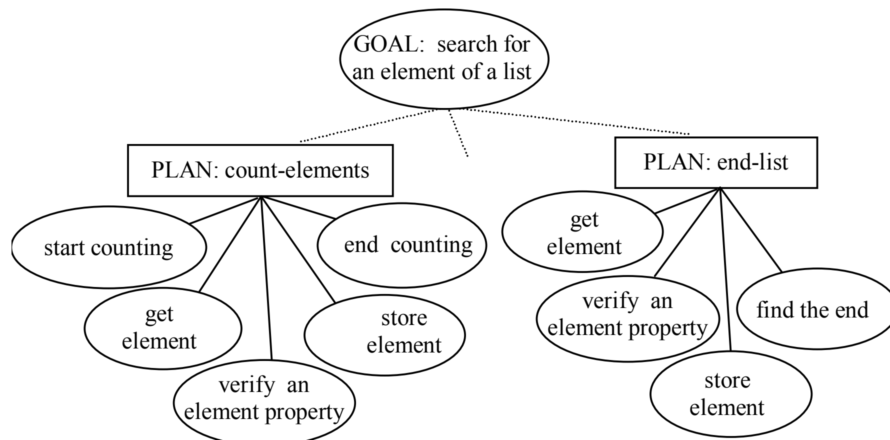

Figure 1. Count-times plan decomposition from the cognitive tree.

Mental Program Model is a solution represented in a higher abstraction level, the cognitive level. It is composed of *programming goals*, *plans* and *actions*.

Implemented Program Model is the *program mental model* encoded in a programming language.

Primitive Action is a programming mental action that can be easily translated to a programming language code. A primitive action can be seen as a *programming plan* composed by one action.

Elementary Pattern is a fragment of an *implemented program* or pattern code that corresponds to a *primitive action or a set of actions that compose a higher level plan* [17][1]. Brooks [4] named such patterns as *beacons*. Beacons can be used as building blocks in the construction of a program [9]. Table 1 shows some *elementary patterns*.

Table 1. Examples of plans with its elementary patterns

Programming Plan	Elementary Pattern

Unrelated choice	if (condition-1)
	    action-1
	if (condition-2)
	    action-2
	if (condition-3)
	    action-3

Sequential choice	if (condition-1)
	    action-1
	else
	    if (condition-2)
	        action-2
	    else
	        action-3

2. Interface Architecture

Figure 2 shows the interface during a resolution of a programming problem. The Learner Interface components are:

- Boxes for selecting the topic, level or problems to be solved;
- Cognitive Tree;
- Implemented Program Model;
- Mental Program Model;
- Control Buttons: Add, Delete, Clear, Diagnose and Exit (x).

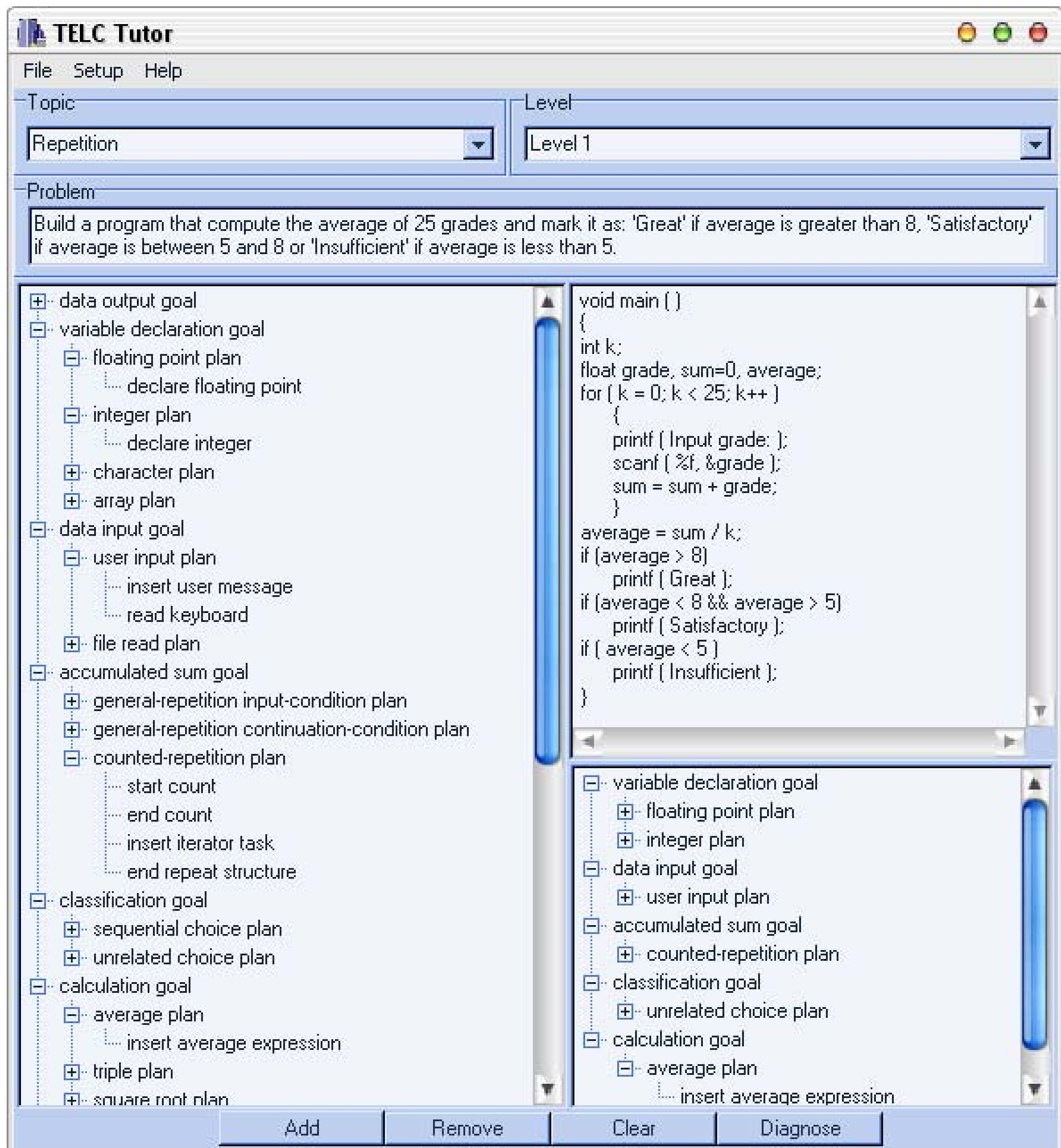

Figure 2. Programming interface components

Boxes of topic, level and problem selection are available for the student to decide the topic he (she) would like to study, the level of difficulty and the problem. The ***Cognitive Tree*** is located in the left side of the screen and it contains a hierarchical decomposition of cognitive components used by programmers during problem solving, as such: *programming goals*, *plans* and *actions*. The tree can be expanded, by selecting the symbol (+), or retracted by selecting the symbol (−) in each one of its branches. Goals are located in the higher level of the tree (root) and actions are located in its lower level (leaves). Intermediary elements are: (sub)plans and sub(goals).

The *Mental Program Model*, located at inferior right window, corresponds to the tree built by the student. The construction of this model is done by the selection of goals, plans and actions from the *Cognitive Components Tree*. The buttons *Add*, *Delete* and *Clear* allow the student to modify the model.

The *Implemented Program Model*, located at superior right window, contains the construction of the final program in C code. Note that the student does not need to worry

about syntax details of the programming language since the *elementary patterns* are automatically gathered and placed on the screen every time the student selects a primitive action.

Note that the *mental program* is still a hierarchy of cognitive components and therefore is different from a C implemented program, which is a total-order sequence of instructions. This is part of the unsolved translation gap there exists between mental program and implemented program and thus it is left to the student to make the final association.

3. Interacting with the Learner Interface

Following, we describe the main operations that learners can accomplish in the interface during the program construction.

Building a Program

The student decides the topic, level and problem that he (she) wishes to solve. When a problem is selected, a set of new functions become available to the student. Then, he (she) can create his high level program, by selecting programming goals, plans and actions from the *Cognitive Tree*. During the selection phase, the student has to: (i) analyze and identify goals and subgoals (recursively) from the problem description; (ii) try to find them in the tree (possibly in different levels) and (iii) select plans or primitive actions to achieve the goals. To solve a particular goal, alternative plans can be found in the tree. When the learner chooses one plan, the interface hides the alternative ones that were not chosen. So, the tree navigation is simpler and now, adapted to the learner choices or knowledge. Two important facts occur during the student selections:

(i) the code block in C language (*elementary pattern*) is automatically inserted in the *Implemented Program Model*;

(ii) the complete branch of a *programming plan* is copied to update the *Mental Program Model*;

Sometimes, the learner has to interact with the tutor to supply additional information to complete an elementary pattern, such as, a text or a value for a variable. In these cases, appropriate interfaces are displayed to acquire the additional information. The student can submit his implemented program to be diagnosed (*Diagnose* button) by the tutoring system or to a C compiler.

4. Methodology

The specification of the cognitive tree was done for a selected set of problems taken from a typical introductory course on computer science. The cognitive tree was carefully designed: the components on the top of the tree are more related to problem goals while the components closer to the leaves are more related to programming control. As previously described, for each primitive action in the tree there is a corresponding elementary pattern that "*implements*" it. Such patterns are considered to be correct, since they have been specified by an experienced programmer.

***Definition**: a cognitive tree that solves a set of programming problems Σ is correct if for any problem $p \in \Sigma$, there is at least one implemented program that can be generated from the cognitive components that solves p.*

The tests used to evaluate the correctness of the cognitive tree, and its mapping to the elementary patterns, were done by compiling and executing the programs that were generated by the interface. The wrong selection of goals and plans can generate a program that is not the correct solution of a given problem. Therefore, it is necessary to have a diagnostic module to detect these errors. Although in this work we do not focus on this issue, we are currently developing a diagnostic module to allow tutorial interventions. Since the system expertise knowledge is directly handled by the student through the interface, we can specify suitable hints and explanations to the student.

5. Discussion

There are strong evidences that computer programmers use their internal mental model of problem solving to help them on building correct programs. However, the translation process of the mental model to a computer program is not well known by the Programming Psychology area. Thus, it is important to help a programming apprentice to make his own association between the mental model and the implemented program. In this work we have proposed one way to help students to make this association by explicitly representing:

(i) the *mental program model*, described as a hierarchy of cognitive components selected by the student (inferior right window of figure 2) and

(ii) the *implemented program model*, the C program, built by the tool according to the mental program model constructed by the student, (superior right window of figure 2).

While the mental model is composed of a tree of goals, plans and primitive actions, the C program is composed of a sequence of instructions. The translation from one model to the other is not accomplished by the student. This skill is the one that we want the student to acquire by using this interface. We believe that from the visualization of these two different knowledge representations, the student can learn by example a mental model of the translation process and, in the future, he will be able to program without using the tool. Such hypothesis is based on the fact that some programmers learn by "*looking*" and trying to comprehend programs that were built by other programmers. The tool has yet to be evaluated in a group of students that has never attended computer classes before.

6. Conclusions

The idea of using knowledge patterns (*beacons*) in an Intelligent Tutoring System is not new [4]. However, we have not found in the literature a programming tutoring tool that applies the use of *elementary programming patterns* in an ITS.

Elementary patterns were originally proposed by the *Pedagogical Patterns* community and computer science educators [10]. Therefore, those patterns have been constructed to be used as a powerful tool in programming learning activities. Although most of them are written for object-oriented programming language, few of them are written in other computer languages, such as functional or procedural languages [17][16]. Since we were interested on procedural languages, great part of our work was dedicated to:

- write patterns in C language;
- select the best collection of problems;
- identify goals, plans and primitive actions for each problem;
- map patterns to problem goals, plans or primitive actions;
- select the appropriate terms to appear in the interface in such way that students could be able to relate those terms to her/his internal mental model of the problem and

- build a teacher interface to introduce his own collection of problems with the corresponding set of goals, plans or primitive actions.

We are currently developing a diagnostic module to allow tutorial interventions. We believe that since the system expertise knowledge is directly handled by the student through the interface, we can specify suitable hints and high level explanations to students.

References

[1] Astrakhan, O. and Wallingford, E. Loop Patterns. In *Proceedings of the Fifth Pattern Languages of Programs Conference*, Allerton Park, Illinois. 1998. http://www.cs.uni.edu/~wallingf/research/

[2] Beaumont, C. and Sackville, A. Identification of best practice in the use of problem based learning in the teaching of Computing. The 3rd Annual Conference of the LTSN Centre for Information & Computer Sciences. Loughborough University, UK August, 2002. Obtained in: http://www.ics.ltsn.ac.uk/pub/conf2002/Beaumont.html

[3] Bonar, J. and Soloway, E. Preprogramming knowledge: a major source of misconceptions in novice programmers, *Human-Computer Interaction*, Vol. 1, 1985, pp. 133-161.

[4] Brooks, R. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, 18:543–554, 1983.

[5] Fink, F.K. Integration of Engineering Practice into Curriculum – 25 years of experience with Problem Based Learning. FIE'99: Frontiers in Education Conference – Engineers Designing the Future, Puerto Rico, November 1999. Http://www.elite.auc.dk/fkf/publications.htm

[6] Moström, J. E. and Carr, D. A. Programming Paradigms and Program Comprehension by Novices. *PPIG'10*, 1998. Obtained in the Internet: http://www.mostrom.pp.se/folk/jem/art01.pdf

[7] Norman, D. Some Observations on Mental Models. MENTAL MODELS, eds. D. Gentner and Albert Stevens, LEA, pp. 7-14, 1983. Obtained in the Internet: http://www.cogs.susx.ac.uk/users/christ/crs/atcs/norman.html

[8] Pane, J. F. and Myers, B. A. The Influence of the Psychology of Programming on a Language Design: Project Status Report. In *Proceedings of the 12th Annual Meeting of the Psychology of Programmers Interest Group*, A. F. Blackwell and E. Bilotta, Eds. Corigliano Calabro, Italy: Edizioni Memória, April 10-13, 2000, pp. 193-205.

[9] Pane, J. F. and Myers, B. A. Usability Issues in the Design of Novice Programming Systems. Human-Computer Interaction Institute Technical Report CMU-HCII-96-101, Carnegie Mellon University, Pittsburgh, August 1996. Obtained in: http://www-2.cs.cmu.edu/~pane/publications.html

[10] PPIG. *Psychology of Programming Interest Group*, 1987. Obtained in: http://www.ppig.org/

[11] Ryan, C. and Al-Qaimari, G.A. Cognitive Perspective on Teaching Object-Oriented Analysis and Design. In *Proceedings of the Annual Computer Science Postgraduate Conference*, Technical Report (TR-96-36), RMIT University, Melbourne, Australia, 1996. Obtained: http://citeseer.nj.nec.com/13036.html

[12] Smith, P. A. and Webb, G. I. Reinforcing a Generic Computer Model for Novice Programmers *ASCILITE'95*, Melbourne, Australia, 1995.

[13] Soloway, E. and K. Ehrlich. Empirical Studies of Programming Knowledge. *IEEE Transactions on Software Engineering*, vol. SE-10, no. 5, p. 595-609, September, 1984.

[14] Storey, M.A.D.; Fracchia, F.D.; Muller, H.A. Cognitive Design Elements to Support the Construction of a Mental Model during Software Exploration. *The Journal of Systems and Software*,1999. Obtained in: http://citeseer.nj.nec. com/270012.html

[15] von Mayrhauser, A. and Vans, A. M. Program Understanding: A Survey. *Technical Report CS-94-120*, August 23, 1994. Obtained in: http://citeseer.nj.nec.com/vonmayrhauser94program.html

[16] Wallingford, E. Functional Programming Patterns and Their Role in Instruction. 2002 Obtained in: http://www.cs.uni.edu/~wallingf/patterns/functional/

[17] Wallingford, E. Elementary Patterns and their Role in Instruction. *OOPSLA'98* Educators Symposium Notes,1998. Obtained in Internet: http://www.cs.uni.edu/~wallingf/patterns/elementary/chiliplop98/summary.html

[18] Weber, G.; Brusilovsky, M. S., Steinle, F. ELM-PE: An Intelligent Learning Environment for Programming, 1996. Obtain in: http://www.psychologie.uni-trier.de:8000/projects/ELM/elm.html

[19] Wenger, E. Artificial Intelligence and Tutoring Systems. Los Altos, CA: Morgan Kaufmann, 1987.