Evolving Boxes as Flexible Tools for Teaching High-School Students Declarative and Procedural Aspects of Logic Programming

Bruria Haberman[1] and Zahava Scherz [2,*]

[1] Computer Science Dept., Holon Academic Institute of Technology, and
Dept. of Science Teaching, The Weizmann Institute of Science,
Rehovot 76100, Israel
bruria.haberman@weizmann.ac.il
[2] Dept. of Science Teaching, The Weizmann Institute of Science,
Rehovot 76100, Israel
zahava.scherz@weizmann.ac.il

Abstract. During the last decade a new computer science curriculum has been taught in Israeli high schools. The curriculum introduces CS concepts and problem-solving methods and combines both theoretical and practical issues. The Logic Programming elective module of the curriculum was designed to introduce to students a second programming paradigm. In this paper we describe how we used *evolving boxes*, when teaching abstract data types (ADTs), to introduce the interweaving declarative and procedural aspects of logic programming. The following types of evolving boxes were used: (a) black boxes that could be used transparently, (b) white boxes that could be modified to suit specific needs, and (c) grey boxes that reveal parts of their internal workings.

We conducted a study aimed at assessing students' use of ADTs. The findings indicated that the students demonstrated an integrative knowledge of ADT boxes as programming tools, and employed unique autonomous problem-solving strategies when using ADTs in programming.

1 Introduction

During the last decade a new computer science curriculum has been taught in Israeli high schools. The curriculum introduces CS concepts and problem-solving methods independently of specific computers and programming languages, along with the practical implementation of those concepts and methods encountered in actual programming languages [5, 6]. One elective module of the curriculum- *Logic Programming*, was designed to introduce a (second) declarative programming paradigm.

Logic programming (LP) enables programmers to concentrate on the declarative and abstract aspects of problem solving, and usually liberates them from dealing with

* Corresponding author.

R.T. Mittermeir (Ed.): ISSEP 2005, LNCS 3422, pp. 156 – 165, 2005.

the procedural details of the computational process. However, sometimes the procedural aspects of logic programming, besides the declarative ones, are also encountered, especially when manipulating compound data structures. Therefore, it is important to use suitable instructional tools to teach the interweaving declarative and procedural aspects of programming. One way that this can be accomplished is by using *evolving programming boxes*.

We developed a two-stage "Logic Programming" course, implemented in the Prolog programming language, which was designed for high-school students. One main goal of the course was to expose students to different aspects of logic programming and to enhance their problem-solving and design skills in the context of the LP paradigm. The 90-hour basic module was designed, as part of the CS curriculum, for beginners and covers the following topics: introduction to propositional logic and predicate logic, including logic programming, data base programming, compound data structures, recursion, lists, introduction to abstract data types (ADTs), and basic methods of problem solving and knowledge representation. The 60-hour advanced module, designed for advanced students who had already learned the basic module, introduces advanced methods of problem solving and knowledge representation, advanced generic abstract data types, and advanced programming techniques [11].

Being a declarative language, logic programming is suitable for knowledge representation and content formalization [16]. Abstract data types are considered as useful tools for CS problem solving and knowledge representation [1]. Since logic programming abstracts the manipulation of compound data structures by hiding procedural aspects and details of their implementation [2], it is convenient for implementing and utilizing abstract data types; hence, it is a suitable programming environment for teaching the notions of ADTs [11].

The abstract data type, which is discussed in both modules of the "Logic Programming" course as a recurrent CS concept, is introduced to students as a *mathematical model with a set of operations* [1]. S*pecification* of an ADT is achieved by formally and verbally defining its use as a model and its operations. *Implementation* of an ADT is achieved by means of the logic programming language by formulating rules to define general predicates for each of the specified ADT operations. The actual implementation of an ADT is achieved by creating a black box. The *use* of an ADT for problem solving is done by defining problem predicates using predefined general predicates.

Here we present how we used evolving programming boxes to gradually introduce ADTs as flexible problem-solving and programming tools. We demonstrate how evolving boxes may be employed to foster students' ability to organize declarative and procedural programming knowledge. We employed our instructional approach to teach declarative and procedural aspects of logic programming. However, these tools can be adopted to introduce similar aspects of any programming paradigm.

1.1 Evolving Programming Boxes

In this section we describe three typical types of evolving programming boxes that can be used in different layers of abstraction.

Black Boxes: A black box is a fully implemented component with predictable functionality and pre-defined interface. Every black box has two components: (a) an interface visible to the user, which describes the implemented operations; in the context of logic programming, each general predicate is characterized by its name, its arguments, its meaning, and assumptions that relate to the way the predicate should be invoked during a programming process; (b) An implementation component that encapsulates the details of how the operations (general predicates in the case of logic programming) were implemented.

The underlying idea of using black boxes, according to the information hiding principle, is that the end-user is only permitted to know what the black box does, and is not allowed to know how the operation is done. Accordingly, the end-user does not need to know how predefined operations are implemented within the black box. The access to source code is therefore denied, and the use of black boxes is done by transparently invoking the encapsulated predefined operations to define new operations.

White Boxes: Black boxes are ready for use without modification but cannot be customized to satisfy the requirements of a particular application. In contrast, white boxes are *visible* modules with accessible source code, and the user is supposed to read and understand thoroughly their internals, with the possibility of copying and modifying them to suit his needs.

Accessibility to the code has pedagogical as well as practical aspects. More specifically, it enables the student to learn and practice programming by: (a) understanding how a given code was implemented according to a given specification, (b) learning from examples how to create new and similar modules, (c) practicing debugging and modifying a given code to suit individual needs.

Grey Boxes: When black boxes provide too little information and white boxes reveal too much, we need to go for a middle ground, which we termed grey boxes. A grey box reveals parts of its internal workings, not just the relations between the input and output. The information can become as detailed as necessary where needed. Revealing some internal information might also help the client (programmer) improve the performance of the complete system [3].

Black, white, and grey boxes are used in programming, especially in the development of object-oriented systems [7, 17]. White and black boxes are used to formally define behavioral compositions expressed via contracts [12].

Educators have stated that integrating black, grey and white boxes into the process of instruction has pedagogical benefits [7, 9, 11, 13, 15, 17]. For example, Eckstein [7] describes how various techniques emphasize different aspects of the architectural design of a framework, and how these techniques can be combined into a general paradigm for instruction. Specifically, she recommends integrating the following instructional methods: *black box teaching*, *white box teaching*, and *incremental teaching*, to explain a complex object technology-based framework by using smaller and simpler frameworks and patterns. She claims that in this way, the students become progressively more familiar with the context of the learned framework and its possibilities, and will recognize the overall picture and the functionality of the framework [7]. Haberman and Ben-David Kollikant demonstrated how black boxes can be utilized to introduce basic programming concepts to novices [9]. Haberman

used black boxes to teach beginners how to use lists in Prolog, thus avoiding the burden of the implementation details, which were found to be very complicated [8]. Here we describe how we used evolving ADT boxes to emphasize the declarative and procedural aspects of logic programming.

2 The Instructional Approach

According to our instructional approach, we recommend that the ADT concept be gradually presented in 8 consecutive stages, as illustrated in Table 1. Stages 1-4 are designed for all students (beginners and advanced); we integrated them in the basic module of the logic programming course. We suggest that stages 5-8, which appear to be complicated [8], should be taught exclusively to advanced students. Accordingly, we integrated those stages into the advanced module of the course. Stages 1-3 deal with one specific type of problem solving, namely by using predefined tools to solve problems and to write programs. However, stages 4-7 deal with various aspects of implementation and with the development of new tools.

Stage 8 integrates both types of problem solving and provides a set-up for learning and using ADTs in the context of knowledge integration similar to the one described in [3]. We suggest that in order to foster integrative knowledge, besides learning new aspects of ADTs, students should progress in each stage, using all the tools and methods that they acquired in previous stages. Next, we describe the activities associated with each stage.

Stage 1 - Acquaintance with given specifications of ADTs: Initially students become acquainted with the specification of generic abstract data types (e.g., lists, sets, multi-sets, trees, and graphs). Suitable examples of concrete problems should be used to illustrate the presented ADTs. Students should realize that the specification of an ADT is independent of the implementation (programming) stage, and of the programming environment.

Stage 2 - Use of ADTs to solve a given problem: Next, students should practice how to choose "known" ADTs to solve a given problem. For example, students should be able to determine that the *tree*- ADT is the most suitable one to present the family parenthood relationship between the females (or males), whereas the *graph*-ADT should be used to present that relationship between all the family members (without referring to a specific gender).

Stage 3 - Use of ADT black boxes in programming: One of our main pedagogical goals was to emphasize the declarative aspects of programming: To the end, the black boxes are presented in terms of *what they do* and not *how it is done*. In this stage, we emphasize the following declarative aspects: (a) the use of a black box is independent of its implementation and therefore does not require becoming acquainted with the implementation details; (b) the use of a black box binds to its interface. Moreover, the use of black boxes has declarative aspects in the sense that the definition of problem predicates is done declaratively in terms of general ADT predicates. For example, the definition of a student in a specific class is phrased as follows: “a person is a student in a class if he is a member of the list of students who belong in that class”. However,

procedural aspects must be also taken into account when using black boxes to implement declarations in order to accomplish a working program.

Accordingly, we suggest that at this stage students should practice using predefined ADT black boxes to write computer programs that solve given problems. Specifically, students are taught to define new problem predicates by transparently invoking predefined general predicates. In addition, ADT black boxes should also be used by students simply to define new general ADT predicates in terms of the predefined ones.

Stage 4 - Specification of new ADTs: At this stage the student plays the role of a consumer who specifies and orders a new ADT black box from his teacher. The teacher implements the required ADT according to the student's specifications in terms of a black box, which is then used by the student to write his program.

Table 1. Gradual presentation of the ADT concept

	Stage	Emphasized Aspects of programming	Target Population
	Acquaintance with given specifications of ADTs	declarative	beginners and advanced
	Determination of ADTs to solve a given problem	declarative	
	Use of ADT black boxes in programming	declarative and procedural	
	Specification of new ADTs	declarative	
	Acquaintance with ADT grey boxes	procedural	advanced only
	Manipulation of ADT white boxes	procedural	
	Implementation of new ADTs	procedural	
	Knowledge integration and autonomous problem solving	declarative and procedural	

Stage 5 - Acquaintance with predefined ADT grey boxes: After students became familiar with the specifications and the use of ADTs, we suggest that they gradually learn how to implement an ADT according to its specifications. Initially, students become acquainted with the implementation of familiar ADTs. At this point the black boxes that have been transparently used in the previous stage become unfolded, i.e. the code within the black box is no longer hidden. Actually, at this point the black box becomes a grey box – visible yet *only read*, and the students perform operations such as reading the code, running the code and following up its execution in order to understand "how it works". At this stage students are also exposed to new procedural aspects of data implementation in terms of the language constructs (e.g., recursive

data structures) and new techniques of data manipulation (e.g., recursive list processing).

Stage 6 - Manipulation of Predefined ADT White Boxes: At this stage the *read only* boxes turn out to be white boxes and the code becomes "more" accessible in the sense that it can also be modified. Here the following procedural aspects of programming are emphasized: students learn advanced programming techniques and efficiency aspects, and practice code debugging, code modification, and writing new code from scratch.

Stage 7 - Implementation of New ADTs: After becoming acquainted with the implementation of predefined ADT boxes, the students experience how to implement new ADT boxes according to a defined specification. At this stage they eventually become independent of the teacher in terms of supplying built-in programming tools. The following procedural aspects should be emphasized: (a) an ADT is implemented according to its specification; (b) the implementation of an ADT is encapsulated in terms of a black box; and (c) an ADT may have alternative black box implementations.

Stage 8 - Knowledge Integration and Autonomous Problem Solving: At this stage students make a significant step toward attaining proficiency, and they practice solving advanced and complex problems. To succeed in these complex missions, students need to understand how the problem-solving patterns that they have already acquired are connected to specific examples and to new problems; they also need to adapt their patterns to suit more complex situations [3]. Moreover, they have to integrate the knowledge that they have gained when learning, creating, and using ADTs in previous stages, and to successfully incorporate it into their solving-program processes.

On the one hand, the students start acting like autonomous standalone developers, reusing their own tools, and on the other hand, they experience sharing tools with peers and reuse others' tools. Actually, they employ ADTs to solve a given problem in the following process: They try to determine familiar ADTs suited for the given problem and use the relevant predefined black boxes. When the predefined ADTs do not suit their needs, they specify new ADTs from scratch or modify the specification of other ADTs, implement them in terms of black boxes, and then use them to develop their programs. The implementation of new black boxes is done based on the knowledge acquired when manipulating grey and white boxes.

3 Fostering Integrative Programming Knowledge

During the last few years, we have conducted an ongoing study aimed at assessing various aspects of students' use of ADTs in the Prolog environment: (a) one part of the study focused on students' strategies for using ADTs to develop Prolog programs [11]; (b) another part of the study focused on the role of ADTs in the project development process [15]; and (c) another part was concerned with students' views toward ADTs [10].

We found that students adapted various strategies for using ADTs, some of which proved that they correctly grasped ADT as a formal CS concept. Other students

improvised alternative strategies, which indicated that their conception of ADT did not match the correct CS definition. Nevertheless, the use of ADTs for problem solving and knowledge representation helped many students to develop correct programs regardless of the strategies they used [11]. The findings also revealed that for most students, ADTs served as a project development organizer [15], and they mostly expressed positive attitudes toward ADTs as problem solving and programming tools [10].

Based on those findings, here we discuss the students' perception of ADT boxes from another perspective–the use of predefined modules of code as multifunctional components for composing and editing a program.

3.1 Students' Perceptions of ADT Boxes

We found that students had gained various perceptions of ADT boxes and of their role in programming. Figure 1 illustrates the types of boxes that reflect students' perceptions in terms of code transparency and accessibility. The less opaque the box is, the more it is accessible and changeable.

Perception of box	Type of box	Associative activities
Sealed, inaccessible	Black Box	Transparent use
Visible, yet incomprehensible "Copy and paste"	Unfolded Grey Box	Code cloning (duplication)
Visible, comprehensible, yet unchangeable	Read Only Grey Box	Comprehension of implementation details
Problem-oriented "Cut and paste"	Flexible White Box	Deleting code, Asserting code
Generic Templates for defining new predicates	White Box	Code modification, rewriting, creating new boxes

Fig. 1. Perception of ADT boxes

Sealed inaccessible black boxes: Beginners who had studied how to use ADT black boxes but were not acquainted with their implementation, perceive the boxes as an integral part of the programming language. Most of them consider the black box as a sealed entity whose content is inaccessible. They believe that it is impossible to examine the contents of the box or to change it; accordingly, they *transparently invoke* general predefined predicates in order to define new problem predicates. Most of the advanced students also use familiar generic black boxes transparently when defining new predicates, even though they have access to the context of those boxes and are familiar with their implementation. These students demonstrate the ability to decide when to use a predefined code as a black box or as a white box.

Unfolded black boxes: We found that students define problem predicates by cloning (non-transparently invoking) general predicates, and actually copy their implementa

tion from the black box to the main program. Advanced students who are familiar with the content of the boxes usually use this strategy. Interestingly, we also found that some of the beginners used this strategy as well, even though they were not familiar with the box's implementation. They unfold the black box and reveal its code only for *copy and paste* purposes. Most of them do not try, nor do they demonstrate any willingness to understand the actual code inside the box. They just copy a selected part of the code and insert it, as is, in their programs. Actually, they perceive the ADT black box as a collection of predicates that can be duplicated and inserted in other programs. The findings indicated that these students are convinced that a correct program should contain all the definitions of the predicates involved. Moreover, they believe that copying the definition of the invoked general predicate contributes to a better understanding of the meaning of the newly defined predicates.

Read only boxes: We found that students use a white box as *read only* scaffolding tool for implementation purposes. They do not copy or rewrite definitions from the given predefined box. Instead, they first try to define problem predicates on their own, according to the conceptual patterns they had gained through the learning process and then check whether their definitions are compatible with those of the relevant general predicates in the box.

Flexible problem-oriented white boxes: Many advanced students perceive the predefined ADT box as a *flexible box* that can be reduced or expanded according to the problem to be solved. The reduction of the box is done by deleting redundant predicates. Students justify this approach by arguing that there is no point in overloading the computer's memory by the implementation of predefined predicates that are not used in the problem-solving process. The expansion of the box is accomplished by additionally implementing new, necessary general predicates that are used to solve the given problem.

White boxes as tools for defining new predicates: Many advanced students rewrite the definitions of general predefined predicates (instead of transparently invoking the general predicate) to define new predicates. Actually they use them as templates and rewrite their definitions by making small changes.

3.2 Construction of Integrative Knowledge

The findings of our study indicated that the students had constructed integrative declarative and procedural knowledge of ADT boxes, and they employed them in unique ways to develop programs. The use of predefined black boxes for ADT enabled them to concentrate on high-level cognitive tasks such as problem analysis, problem solving, and knowledge representation without the burden of knowing complex implementation details. In contrast, the white boxes enabled students to learn, through examples, how to implement ADTs according to a given specification, and to practice code reuse and modification. The students defined their own rules of using ADT boxes and demonstrated a variety of strategies of using them while writing their programs. Those who learned and comprehended the notions of the formal ADT concept, used it the way expert programmers do: They first try to determine the suitable predefined ADT for the given problem and then transparently use the relevant ADT black box. Only when the familiar predefined black boxes are insufficient to

solve the problem, do they unfold a relevant box and make the minimal necessary changes, or specify and implement a new ADT. Once the new ADT box is implemented, they use it transparently as is common among professionals. In contrast, students who are immature, and are still in the middle of the learning process, interpret in their own way the roles of the ADT boxes. Some of them avoid using black boxes because they believe that the encapsulation of the general predicates they used reduces the meaning, clarity, and completeness of their programs. Others, although beginners, transparently used predefined black boxes, and temporarily avoided using them when they started learning about their implementation [11].

4 Conclusion

In this paper we demonstrated how evolving ADT boxes can be employed to teach the interweaving declarative and procedural aspects of logic programming. We believe that the suggested instructional model can be adopted to emphasize various aspects of any programming paradigm, and can also be used to guide the students toward proficiency in programming based on abstraction and code reuse.

We recommend that the suggested instructional model be employed while providing the students with an appropriate learning environment that promotes learning processes in the context of knowledge integration [4]. Various aspects of the learning concept should be introduced in different ways by repetition through simpler frameworks [7]. Scaffolding examples should be used to demonstrate the activities associated with each stage of the model; appropriate exercises and support activities should be developed to motivate students to use black boxes, comprehend the code of white boxes, reuse code provided by others, modify code, and choose the appropriate boxes to solve given problems. Moreover, in order to foster integrative knowledge, students should continue, in each stage of learning, to practice and meaningfully utilize the tools and the methods that they have previously acquired.

References

1. Aho, A.V. & Ullman, J.D. (1992). *Foundations of Computer Science*, W.H. Freeman and Company.
2. Ben-Ari, M. (1995). *Understanding Programming Languages*. John Wiley.
3. Buechi, M. & Weck, W. (1997). A plea for Grey-Box components. *Workshop on Foundations of Object-Oriented Programming*, Zürich, September 1997 Available:http://www.cs.iastate.edu/~leavens/FoCBS/buechi.html
4. Clancy, M.J. & Linn, M.C. (1999). Patterns and Pedagogy. *ACM SIGCSE Bulletin*, 31(1), 37-42.
5. Gal-Ezer,J., Beeri, C., Harel, D., & Yehudai, A. (1995). A high-school program in computer science. *Computer*, 28(10), 73-80.
6. Gal-Ezer,J., Harel, D. (1999). Curriculum and course syllabi for high school CS program. *Computer Science Education*, 9(2), 114-147.
7. Eckstein, J. (1999). Empowering framework users. In *Building Application Frameworks: Object-Oriented Foundations of Framework Design*. Mohamed E. Fayad, Douglas C. Schmidt, and Ralph E. Johnson (Eds.). John Wiley & Sons, 505-522.

8. Haberman, B. (1990). Lists in Prolog. M.S. Thesis. The Weizmann Institute of Science, Rehovot, Israel. (in Hebrew)
9. Haberman, B. & Ben-David Kollikant, Y. (2001). Activating "black boxes" instead of opening "zippers" – A method of teaching novices basic CS concepts. *ACM SIGCSE Bulletin*, 33(3), 41-44.
10. Haberman, B. & Scherz, Z. (2003). Abstract data types as tools for project development – High school students' views. *Journal of Computer Science Education online*, January 2003. Available: http://iste.org/sigcs/community/jcseonline/
11. Haberman, B. Shapiro, E. & Scherz, Z. (2002). Are black boxes transparent? – High school students' strategies of using abstract data types. *Journal of Educational Computing Research*, 27(4), 411-236.
12. Helm, R., Holland, M. & Gangopadhyay, D. (1990). Contracts: Specifying behavioral compositions in Object-Oriented systems. In *Proceedings of the European Conference on Object-Oriented Programming on Object-oriented programming systems, languages and applications (ECOOP/OOPSALA).* 25, Ottawa Canada, October 1990, 169-180.
13. Kiczales, G. (1994). Why are black boxes so hard to reuse? Invited talk, OOPSLA'94. Available: http://www.parc.xerox.com/spl/projects/oi/towards-talk/transcript.html
14. Resnick, M., Berg, R. & Eisenberg, M. (2000). Beyond black boxes: bringing transparency and aesthetics back to scientific investigation. *Journal of the Learning Sciences*, 9(1), 7-30.
15. Scherz, Z. & Haberman, B. (2003). The role of abstract data types in the project development process. Submitted to *Journal of Computer Science Education*.
16. Sterling, L. & Shapiro, E. (1994). *The art of Prolog* (2nd ed.). Cambridge, MA: MIT Press.
17. Warford, J.S. (1999). Black Box: A new Object-Oriented Framework for CS1/CS2. *ACM SIGCSE Bulletin,* 31(1), 271-275.