An introduction to object-oriented programming with a didactic microworld: objectKarel

Article *in* Computers & Education · June 2006

DOI: 10.1016/j.compedu.2004.09.005 · Source: DBLP

CITATIONS
62

READS
1,094

3 authors:

Stelios Xinogalos
University of Macedonia
104 PUBLICATIONS **645** CITATIONS
SEE PROFILE

Maya Satratzemi
University of Macedonia
161 PUBLICATIONS **723** CITATIONS
SEE PROFILE

Vassilios Dagdilelis
University of Macedonia
120 PUBLICATIONS **339** CITATIONS
SEE PROFILE

Available online at www.sciencedirect.com
SCIENCE DIRECT®
Computers & Education 47 (2006) 148–171
COMPUTERS & EDUCATION
www.elsevier.com/locate/compedu

An introduction to object-oriented programming with a didactic microworld: *objectKarel*

Stelios Xinogalos [a], Maya Satratzemi [a,*], Vassilios Dagdilelis [b]

[a] *Department of Applied Informatics, University of Macedonia, 156, Egnatia str., P.O. Box 1591, 54006 Thessaloniki, Greece*
[b] *Department of Educational and Social Policy, University of Macedonia, 156, Egnatia str., P.O. Box 1591, 54006 Thessaloniki, Greece*

Received 27 May 2004; accepted 15 September 2004

Abstract

The objects-first strategy to teaching programming has prevailed over the imperative-first and functional-first strategies during the last decade. However, the objects-first strategy has created added difficulties to both the teaching and learning of programming. In an attempt to confront these difficulties and support the objects-first strategy we developed a novel programming environment, *objectKarel*, which uses the language Karel++. The design of *objectKarel* was based on the results of the extended research that has been carried out about novice programmers. What differentiates it from analogous environments is the fact that it combines features that have been used solely in them: incorporated e-lessons and hands-on activities; an easy to use structure editor for developing/editing programs; program animation; explanatory visualization; highly informative and friendly error messages; recordability. In this paper, we present the didactic rationale that dictated the design of *objectKarel* and the features of the environment, including the e-lessons. In addition, we present the results from the use of *objectKarel* in the classroom and the results of the students' assessment of the environment.

Keywords: Programming and programming languages; Teaching/learning strategies; Pedagogical issues

* Corresponding author.
E-mail addresses: stelios@uom.gr (S. Xinogalos), maya@uom.gr (M. Satratzemi), dagdil@uom.gr (V. Dagdilelis).

0360-1315/$ - see front matter © 2004 Published by Elsevier Ltd.
doi:10.1016/j.compedu.2004.09.005

1. Introduction

The three basic strategies for an initial approach to teach programming are: imperative-first, functional-first, and objects-first. The first two strategies have been used for a fairly long period of time, whereas the third one appears to have attracted interest in the last few years. The functional-first strategy initially places emphasis on functions leaving the presentation of state for later, whereas in the imperative-first strategy the emphasis is first given to the state and then the concept of functions is presented.

As far as the objects-first strategy is concerned, according to the ACM curricula report, ''Objects-first emphasizes the principles of object-oriented programming and design from the very beginning. The objects-first strategy begins immediately with the notion of objects and inheritance and then goes on to introduce more traditional structures'' (Chang et al., 2001, p. 30). This means that from the very beginning both the state and functions must be presented.

As the authors of the ACM curricula report acknowledge, the objects-first strategy creates added difficulties to both the teaching and learning of programming. The classic methodology for the teaching of programming began with small and simple programs to be followed by more complex and larger-sized ones. This approach gave novice programmers time to assimilate and to gradually build up new knowledge relevant to the development of programs. However, when using the objects-first strategy, students are required to work with objects from the very beginning. This means that from the very beginning they will have to be taught about objects, classes, methods, constructors, inheritance and at the same time they will have to be taught the concepts of types, variables, values, as well as having to learn the syntax of the language which, as has been shown in the research, comprises one of the biggest sources of difficulties for novice programmers.

In an attempt to support the objects-first strategy various educational software tools have been developed, such as: BlueJ (Kolling, Quig, Patterson, & Rosenberg, 2003), Karel J. Robot (Bergin, Stehlik, Roberts, Pattis, & Karel, 2004), Jeroo (Sanders & Dorn, 2003), JKarelRobot (Buck & Stucki, 2000), and Alice (Cooper, Dann, & Paush, 2000). Certain of these tools, like Karel J. Robot, Jeroo, JKarelRobot, and Alice, constitute programming microworlds which are based on a physical metaphor, while, BlueJ is an integrated programming environment whose main feature is that the user begins with a set of predetermined classes and can create objects and call on methods for those objects in order to examine their behavior.

In our attempt to support the objects-first approach we developed a novel programming environment *objectKarel* (Xinogalos, 2002), which uses the language Karel++, as defined by Bergin, Stehlik, Roberts, and Pattis (1997).

ObjectKarel consists of a programming microworld and thus belongs to the first category mentioned above. However, *objectKarel* incorporates certain characteristics, which do not exist in other software of the same category but which facilitate both the teaching and learning of OOP. The two basic characteristic components of *objectKarel* are the following:

- It is a programming environment, which helps the student to develop programs easily. Program development is accomplished with the help of a structure editor where by selecting the appropriate commands from menus the development/editing of a program takes place.

- It incorporates e-lessons with theory and examples. With the help of e-lessons the beginners familiarize themselves with the concepts of the specific units rather than writing a program from the beginning and they are given the opportunity to experiment via ready examples.

In the sections below the following are described: the didactic rationale, which the design of *objectKarel* was based on the characteristics of the language Karel++ and the environment, and the e-lessons. A presentation of the use of *objectKarel* in the classroom situation is given and finally, the results of the students' assessment of the environment are discussed.

2. The didactic rationale of *objectKarel*

ObjectKarel comprises an educational environment in that it incorporates all those elements, which are necessary for an introduction to OOP. *ObjectKarel* incorporates many teaching/learning strategies, which in accordance with research findings facilitate the learning of OOP.

As has been ascertained by many researchers (Brusilovsky, Calabrese, Hvorecky, Kouchnirenko, & Miller, 1997; Ruckert & Halpern, 1993), one of the most significant difficulties that beginner students face is the *extended instruction set* of programming languages. Also, students have great difficulty in comprehending the general characteristics of the *notional machine*, which they must learn to control and its relation to the *natural machine* (Du Boulay, 1989). Milne and Rowe (2002) report that beginners who are introduced to OOP are unable to comprehend what is happening to their program in memory, as they are incapable of creating a clear mental model of its execution.

In order to deal with these obstacles we chose a small programming language Karel++ that uses the metaphor of a world of robots. We believe that the use of a *programming microworld* which is based on a physical metaphor focuses students' attention, offers the opportunity to solve interesting problems even from the first lessons and contributes greatly to decreasing the "distance" between the mental models or descriptions of algorithms in a natural language and their description in a programming language.

In order to deal with the problems caused by the difficulty in comprehending the general characteristics of the "notional machine" that beginners learn to control and its relation to the natural machine, we created an advanced system of animation and visualization. *Program animation* (Birch et al., 1995) helps students understand the language's semantics and flow of control as well as the way in which the commands of their program are connected with the actions of the robots. Due to program animation but also to the programming microworld used the process of program execution is not hidden and so students do not develop an input–output oriented understanding, as is the case in usual programming environments. In the usual programming environments and the 'real' programming languages, beginners concentrate their attention on tracing the values of the variables during the program's execution and attempt to establish the programs accuracy by controlling the values of the variables. Therefore, as far as the student is concerned, in executing a program they focus on data input and the output of results, which act as "noise" in the comprehension of the language's semantics and flow of control.

In the *objectKarel* environment, the possibility of *tracing* and *step by step execution* implements program animation. *Explanatory visualization* (Brusilovsky, 1993) has also been applied in the

form of explanatory messages written in a natural language on the semantics of the command at hand.

Related research has shown that the syntactic and semantic rules of programming languages create difficulties for beginners (Brusilovsky et al., 1997; Ruckert & Halpern, 1993), which result in the students focusing their attention on these rules and by so doing do not develop programming skills. These difficulties, in combination with syntactic and semantic errors, which in the usual programming environments are presented in coded form, are in no way instructive and often have the effect of disappointing and discouraging students. We, therefore, decided that the development of programs should be realized with the help of a *structure editor* (Miller, Pane, Meter, & Vorthmann, 1994), which is as follows:

- Choosing the appropriate action (class/method declaration, construction of object) or choosing a message to send to an object from a single menu, which is automatically updated whenever the user declares/deletes/edits a class/method.
- Interacting with the system through dialog boxes.

We chose to incorporate this editor with the express aim of helping the beginner to focus on the solution of the problem and the acquisition of concepts rather than on the syntactic details of the programming language. The structure editor reduces the possibility of making syntactic errors, but it does not exclude this possibility at all: for example, the user may omit curly braces or even fragments of code.

Furthermore, our programming environment detects and reports *understandable and highly informative error messages* of all types:

- the line number reported is the actual line of the mistake;
- messages report not only what is wrong but also explain why it is wrong;
- the error messages use a natural language and not codes.

Finally, *objectKarel* incorporates *e-lessons*, which include both theory and hands on activities thus helping in the teaching of programming. The beginners familiarize themselves with the concepts of the specific units rather than writing a program from the beginning and they are given the opportunity to experiment via ready examples. Research has shown that it is wrong to begin the teaching of OOP from scratch (Kolling & Rosenberg, 2001). Writing a class involves design; one has to decide what class(es) should exist and what the methods should be. Instead, a student should start by making small changes to the existing code. In this way, they can go through a sequence of exercises of which they can understand each step. Moreover, via the ready examples that were incorporated in the theory and in the activities, students are given the chance to learn a great deal by studying well written programs and copying style idioms.

3. Language features

The programming language of our environment is based on Karel++ (Bergin et al., 1997), which is closely related to C++ and Java. *ObjectKarel* is based on the metaphor of a world of

robots. The actor of the microworld is one or more robots (object) that are assigned various tasks in a world that consists of:

- crisscrossing horizontal streets and vertical avenues forming one block intervals;
- wall sections between adjacent streets or avenues used to represent obstacles (hurdles, mountains, mazes, etc); and
- beepers – small plastic cones that emit a soft beeping noise – placed on street corners.

Students write programs that instruct robots how to perform their tasks. A task is anything we want the robot to execute based on the capabilities that they have; such as moving the robot to a particular intersection, escaping from a labyrinth, locating a beeper and moving it to another position, etc.

The language has two predetermined classes. The class *Primitive_Robot* and the class *Robot* with predetermined methods those of: *move*(), *turnleft*(), *pickbeeper*(), *putbeeper*(), *turnoff*(). The class *Robot* inherits the methods of the class *Primitive_Robot* and in addition is able to employ methods, which make it possible for the objects of this class to control the environment. The additional methods used by the class *Robot* are *Boolean*: *frontIsClear*(), *nextToABeeper*(), *nextToARobot*(), *facingNorth*(), *facingSouth*(), *facingEast*(), *facingWest*(), *anyBeepesInBeeperBag*(). Each of the objects in these classes has four properties: the street, the avenue, the direction it is facing, and the number of beepers it has in its bag.

The language supports four basic control structures: *if*, *ifelse*, *loop*, and *while*.

If the available class of robots "for the execution of a mission"– is not appropriate, then students can create a new class of robots that inherit the properties and methods of the basic or a previously declared class and extend them. The ability of students to create additional classes and methods is a fairly successful introduction to the concept of modularity at a relatively early stage of teaching objects-first programming.

4. Description of the *objectKarel* environment

The main window of the programming environment (Fig. 1) consists of five structural components which include the menu, the toolbar and three areas.

In the area "*Source Code*" the student can develop a program, modify and save or recall a previously saved program.

In the area "*Visualization of program Execution*" an initial situation of the world is constructed with direct manipulation and the result of the program's execution is also presented here.

In the area "*Compiler Output*", which is found at the bottom of the window and which has a dual function, the following happen:

- When the student compiles the program of the area "*Source Code*" in the area "*Compiler Output*" the result of this compilation is presented.
- During the dynamic visualization of a program's execution the area "*Explanatory Visualization*" is used to present explanatory messages, which provide an explanation of the running command in a natural language.

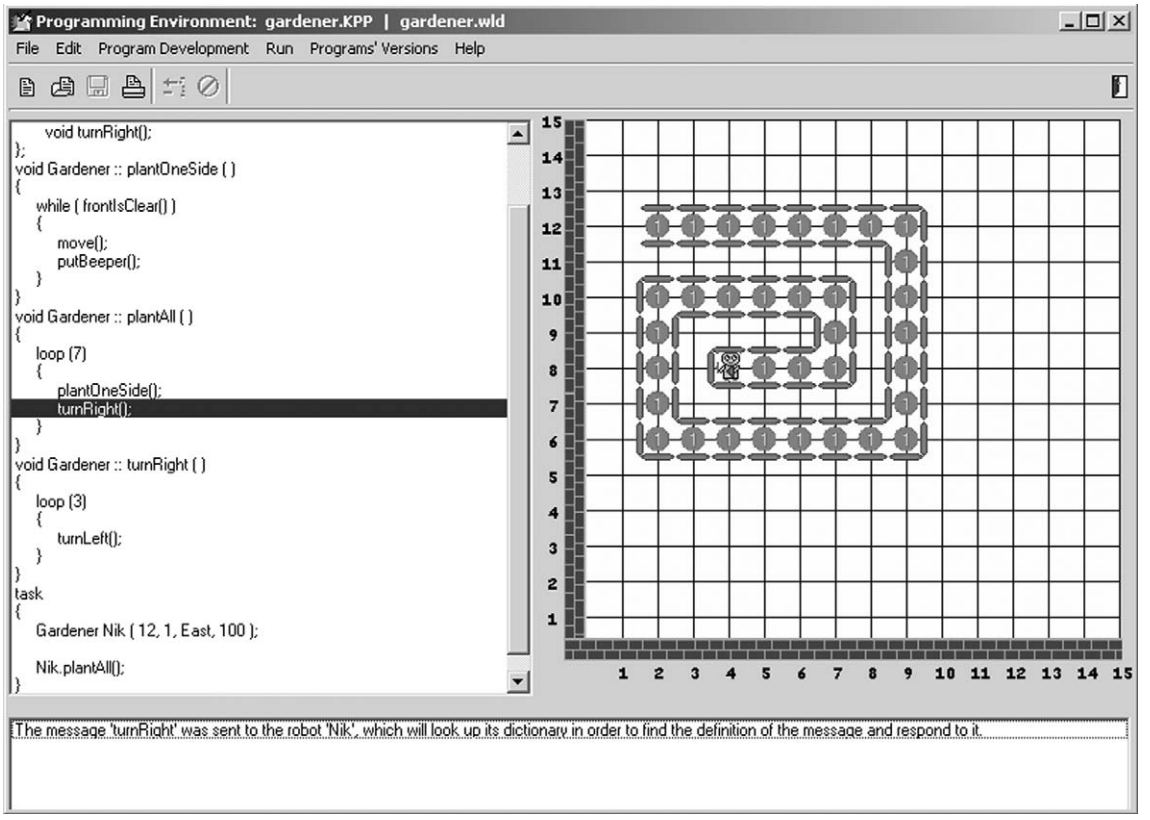

Fig. 1. The main window of the programming environment.

4.1. Development and editing of programs – structure editor

The development of a program is realized using the structure editor, which has been incorporated in the integrated programming environment.

The procedure of declaring a class is simple since the structure editor guides the student through the steps that have to be taken. Specifically, from the selection "*Class/Method Declaration*" of the "*Program Development*" menu, the student is lead to the cards "*Class Declaration*" and "*Methods' Definition*" where, with the help of a template, they declare a new class and method.

The card "*Class Declaration*" (Fig. 2) is comprised of three structural components:

- *A template for the declaration of classes* on the left-hand side of the window.
- At the top on the right the statement of the class is presented.
- At the bottom on the right *a brief description of the way that a class is declared* is presented, which enables any possible queries the student might have to be solved.

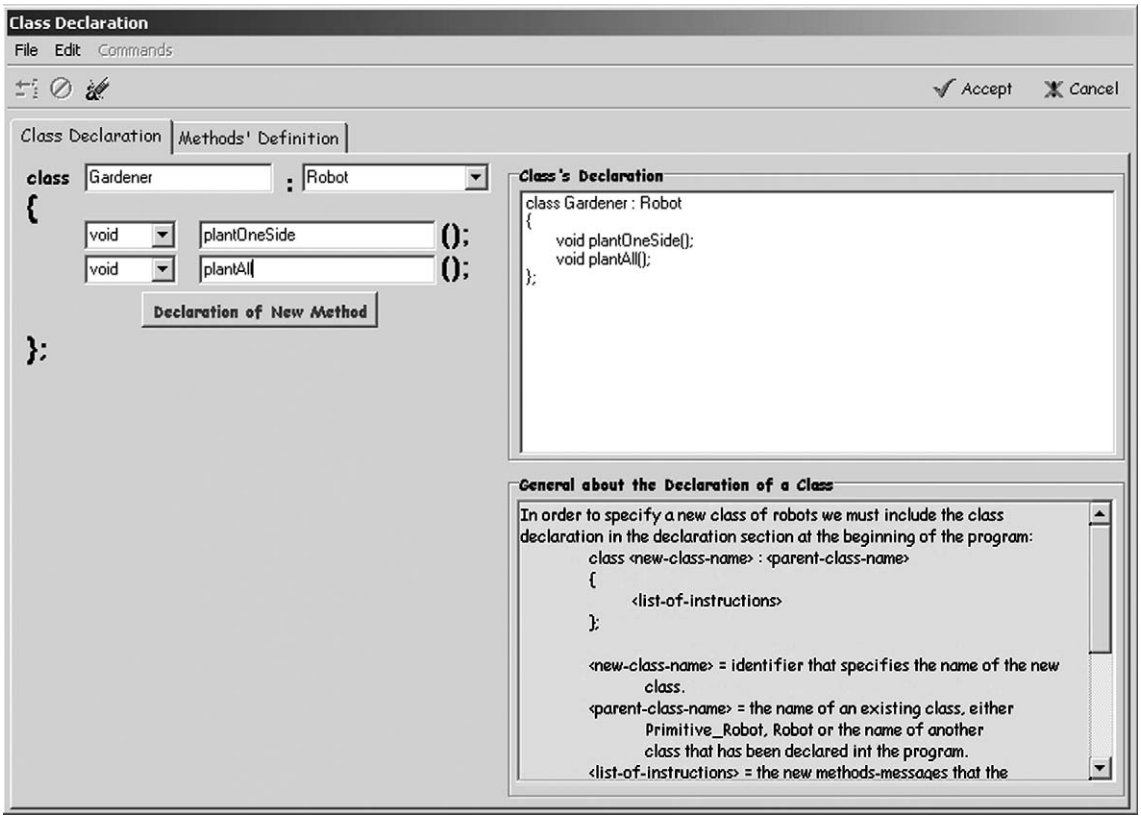

Fig. 2. The card "*Class Declaration*".

When the student completes the declaration of the class, then by using the card "*Methods' Definition*" (Fig. 3) they can define the methods declared in the new class. However, the student does not have to declare all the methods of the new class and then define them.

The possibility to shift from one card to another, for example, enables the student to declare one method in the class each time and to define it. When the student shifts to the card "*Methods' Definition*" a check is done on the statement of the class. If errors are located, the relevant message appears enabling the student to correct it, otherwise the card comes to the foreground and the menu "*Program Development*" is activated.

When the student completes the declaration of the new class and methods, then the area "*Source code* " is automatically updated with the statement of the new class and the menu "*Program Development*" is likewise automatically updated with the names of the new methods.

The student can create objects by choosing *Construction of robot* from the menu "*Program Development*" and then the dialog frame in Fig. 4 appears.

In this frame a template of the construction of an object (robot) is provided, as well as an explanation of the meaning of the command in a natural language. Once they have created the robot(s) needed, the students choose the appropriate messages from the menu "*Program Development*". In

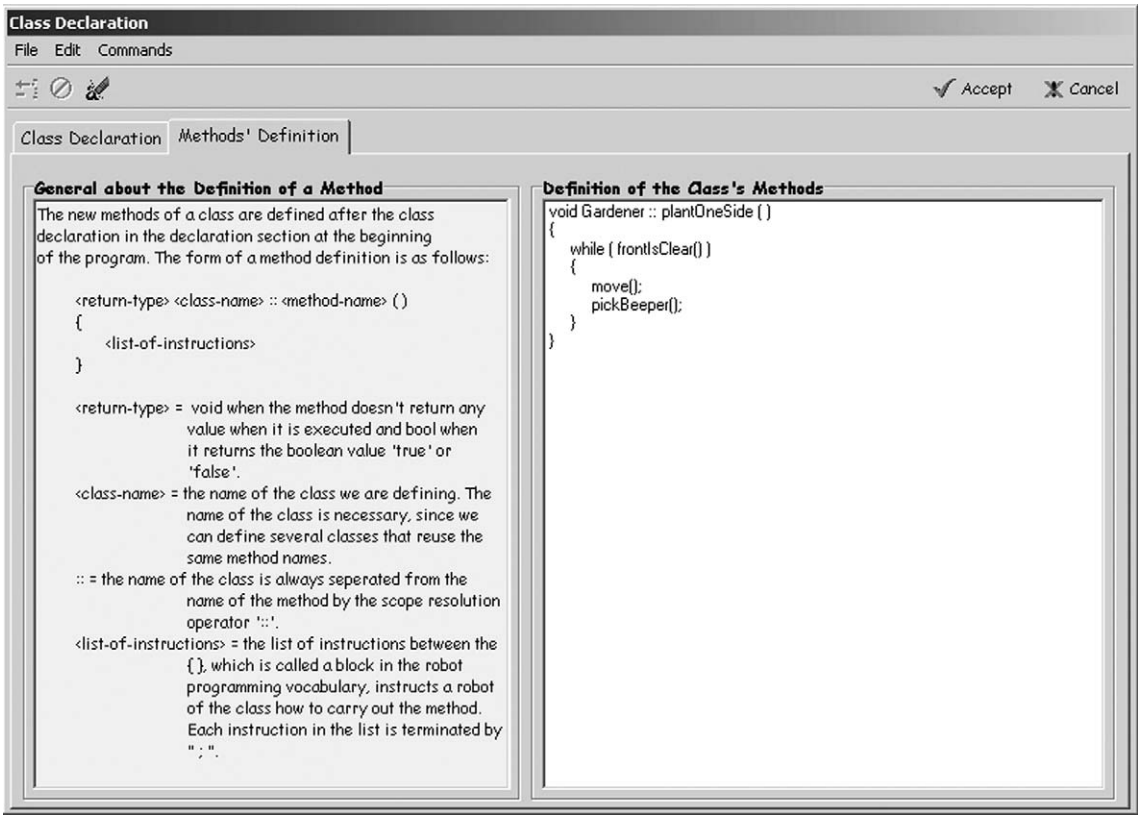

Fig. 3. The card "*Methods' Definition*".

the case where the student has created more than one robot, they can choose which robot they want to send the message to (see Fig. 5).

4.2. Construction – editing of the world situations

For most of the problems that students are called on to solve there must exist a specific initial situation of the world. In order for this initial situation of the world to be created *techniques of direct manipulation* are used.

The student has the possibility to save the situations of the world they create, to print them out and to recall them. In addition, they have *the possibility to intervene in the current situation of the world at any moment*. More specifically, in contrast to similar environments, the possibility of intervention exists even while executing a program. We believe that this possibility is extremely significant both for the student and the teacher for the following reasons:

- *The student has the possibility to experiment and explore* what the result of the execution of a block of code or of a program will be, thus enabling them to get a different situation of the world without being forced to stop the execution of the program, changing the situation of

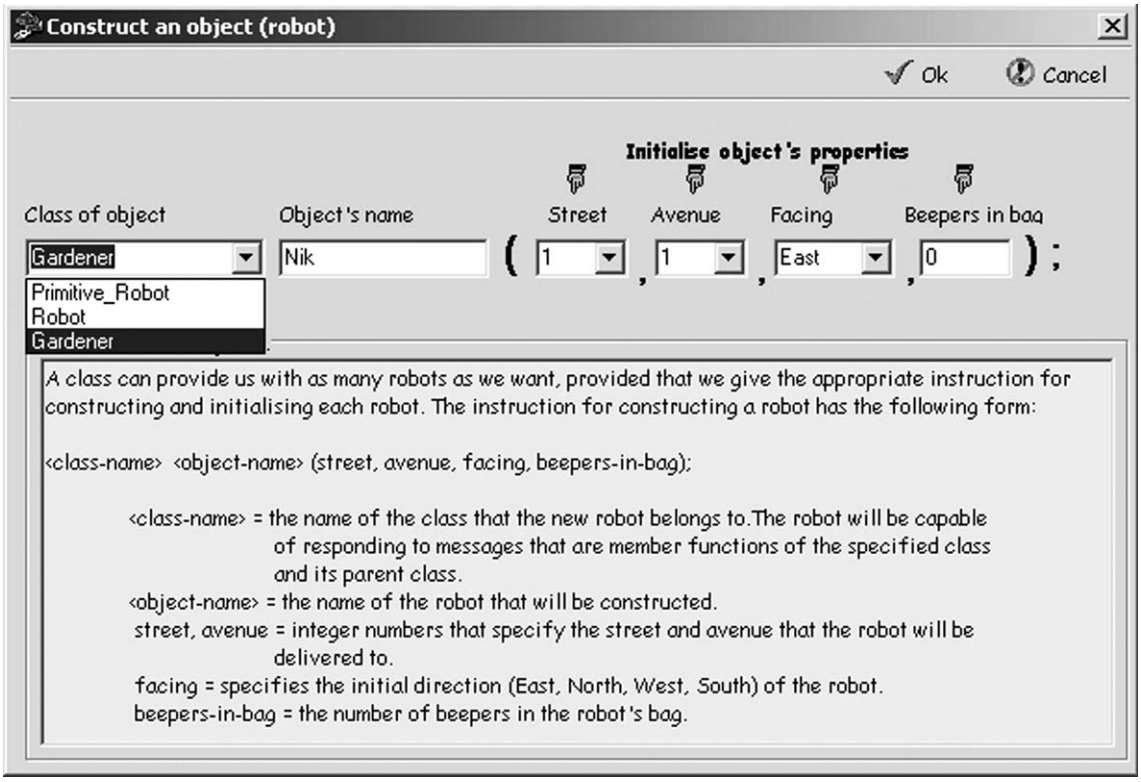

Fig. 4. The dialog frame "*Construct an object (robot)*".

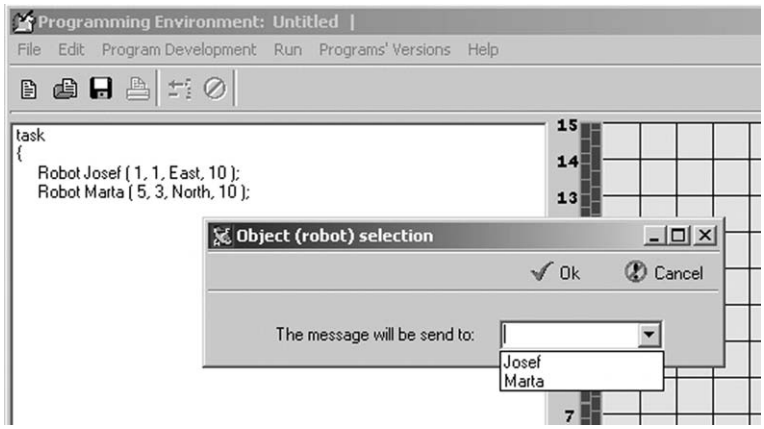

Fig. 5. The dialog frame "*Object (robot) selection*".

the world and then having to execute the program again from scratch. In addition, any chance queries that may arise during the direct execution of a program can be answered. Needless to say that the student would probably not be willing to experiment and explore if they had to undergo the tedious above-mentioned procedure.

- The teacher, within the context of presenting students with examples, has the possibility:
  - To change the situation of the world during a program's execution and to ask the students to determine what the outcome of executing the program will be; and thus checking the level of students' understanding of the concepts being taught.
  - To answer directly and effectively to students' questions which might, for example, concern the way an object responds to the message in relation to the situation of the world, the flow of a program's execution and so on.

4.3. Program compilation

In *objectKarel*, particular emphasis has been given to the presentation of the compiler's results as well as to the possibility of interacting with those results.

The advantages of interacting with the results of the compiler are the following:

- Simultaneous signaling of both the error message and the line of the source code where the error was located helps the student focus their attention on the right spot.
- The student does not need to count the lines of the source code in order to locate the line where the error is, thus devoting their time to more important actions.
- Finally, we believe that students will also take into consideration the warnings and in so doing avoid making the usual errors.

In addition, the error messages and the warnings referred to:

- use a natural language rather than code;
- the line referred to is the actual line where the error is located;
- the error messages do not only state what is wrong but why it is wrong (where it is not obvious).

4.4. Executing – debugging a program

If a program has been successfully compiled, the student has the possibility to execute it in the following ways:

Execution. In this case the program is executed in the usual way that all programming environments offer, that is to say that, the results of the program's execution are presented in the situation of the world.

Tracing. This choice offers the possibility to execute a program step-by-step at a speed, which the student can select from the dialog frame. In this case the current command is indicated by a change of color in both the background and characters, while simultaneously, the result of the execution can be seen in the situation of the world.

Step-by-step execution. In contrast to the previous methods of executing a program, here, the student takes an active role by deciding when the next command will be executed. Each command is carried out in the same way as that described above, for tracing.

The two last choices in executing a program, known as *program animation*, provide substantial help to the novice programmer. More specifically, the step-by-step execution:

- reveals the dynamic nature of a program's execution;
- it provides substantial support to the student in helping them to understand the meaning of the concepts being taught, as well as the control structures;
- it helps in understanding the flow of control and in the elimination of related misconceptions;
- it helps the student not only to locate but also to understand the source of the logical errors;
- it offers the teacher the possibility to cover more material in a shorter period of time, while errors which may arise from the presentation of "dynamic" concepts in a "static" manner, with the use of the blackboard, are avoided.

Furthermore, when the student uses tracing or step-by-step execution, explanations of the meaning of the running command are offered each time. These explanations are in a natural language and are presented, as has already been stated, in the area along the bottom of the programming environment's main window. This possibility, known as *explanatory visualization*, has the following advantages:

- It offers an alternative method of presenting the meaning of the commands and following the flow of the execution of a program. The student can consult the available explanations when they do not understand what is happening by looking at only the visualization of the execution of a program.
- We believe that it can be particularly helpful in understanding and especially in eliminating the misconceptions concerning the concepts and structures, which make life difficult for students, such as the control structures, for example. Even though, with the step-by-step execution, the command being carried out each time is indicated, in certain cases, the student is unable to "follow" the execution of a program. If, for example, the student believes that the block of statements that follows "then" is always executed regardless if the value of the condition is true or not (Sleeman, Putman, Baxter, & Kuspa, 1988), it is most likely that they will not understand why in a similar situation, the section of "then" is not executed and confusion will arise concerning the flow of the execution. We believe that the explanation, in a natural language, as to why the section "then" will not be executed, will help the student understand the meaning of the command and will more easily eliminate the related misconception.
- The student is acquainted with basic concepts and the relevant terminology.

4.5. Recording the students' activities

In the programming environment *objectKarel*, the possibility to record the students' activities during the development and debugging of a program has been incorporated. More specifically, each time a student compiles a program, the system automatically saves not only the source code but also the results of the compilation.

A history of the compilations is presented in a different window by choosing "*Programs' Versions*" from the main menu. The teacher is in a position to see each one of the successive versions of a program and the corresponding error messages, if there were any. This possibility:

- Enables the identification of the difficulties and misconceptions, which the students come up against when being taught object-oriented programming. For example, if the students follow a hit-and-miss approach this is revealed in the sequence of their programs. Thus, the history of compilations provides important new data in the field, since very little research has been conducted on this relatively new programming paradigm.
- It helps the teacher to adjust the lesson to the needs of the students.

4.6. e-Lessons

e-Lessons have been incorporated in *objectKarel*. These have been organized into seven teaching units: *Introduction*, *Classes*, *Inheritance*, *Polymorphism*, *Overriding*, *Selection*, and *Repetition*. Each unit consists of both theory and activities. The basic aims of these lessons are:

- To give the teacher the possibility, while using the programming environment, to teach the basic concepts of objects-first programming and the basic structures of control in a much shorter time span.
- Through the activities, students are given the opportunity to come into contact with ready-made programs, classes and methods, enabling them to experiment with concepts which may cause difficulties.
- To reduce to a minimum the possibility of students forming misconceptions or acquiring erroneous and incomplete knowledge.
- To offer students the possibility of access to theory during the development of a program. Thus, when the students meet with difficulty during the development of a program they will not attempt, as usually happens, to deal with the problem by drawing from the existing knowledge they have from other areas as this, in itself, creates further problems.

5. Using *objectKarel* in the classroom

The environment *objectKarel* was used in a pilot program and evaluated by nineteen (19) students from the Department of Applied Informatics of the University of Macedonia, Greece (Xinogalos, 2002). All students took part in the experiment in a voluntary capacity and all had failed in the examinations on programming subjects (either OOP or imperative programming). These students were chosen since, as we knew, they had met with difficulties in the learning of programming and would thus make up a uniform sample.

Before the start of the course, the students were given a questionnaire whose aim was to record the difficulties they had during their introduction to programming. In addition, our objective was to use the relevant data to investigate to what degree these difficulties would be dealt with at the completion of the course.

The actual course consisted of five two-hour lessons. However, a sixth lesson was held in order for us to evaluate the students' knowledge, and also for the students to give their assessment of the programming environment and the course.

The lessons were structured and carried out in the computer laboratory. The teaching methods were based on the incorporated *objectKarel* e-lessons. In each lesson, the students familiarized themselves with certain notions of OOP by referring to relative theory and in particular using the examples that have been incorporated into the e-lessons as activities. In these activities, the students, as a rule, investigated the proposed code executing it in one go or step-by-step, or by converting the code in order to realize the changes that the transformations had on the robots' behavior. The students were facilitated in their investigations due to the environment with its particular features, including the structure editor described above; the characteristic messages that were presented in each case; and the fact that the robots' behavior was visible. Moreover, the activities provided in stages, familiarized the students with the possibilities that the environment offers. Lastly, in each lesson, a sequence of exercises was proposed whose aim was to clarify or to emphasize certain aspects of each notion. Students solved these exercises in groups of two and without help from the teacher.

For each of these lessons data was collected and analyzed. The data was collected in the following ways:

- the teacher recorded any difficulties that students had in understanding the concepts being taught, the drawbacks and problems that were encountered while using the programming environment;
- the consecutive versions of the students' programs were studied; and
- the recorded actions and dialogues of two groups were analyzed.

Below, the way in which the environment *objectKarel* was used in the classroom for an Introduction to objects-first programming is presented, followed by the results of the study.

5.1. Lesson 1: objects – classes

We believe that in an introductory programming course that uses the OOP paradigm, one should begin directly with the concept of objects. Although there are no related research results, up to now practice has shown that an introduction of this sort is more successful than one which is based on structured programming and which presents the concept of objects at a later stage. Of course, teaching objects as the first basic notion of a programming language is not always easy with commercial programming environments. However, this is possible with *objectKarel* since the objects are robots.

Therefore, in the first lesson, the concepts that are presented are: Object, construction and initialization of an object, message/method, properties and behavior of an object, class, program/task. First, the theory of the unit "Introduction" presents the microworld of *objectKarel* and initiates students in the main principles of OOP.

The activities in the unit 'Classes' enable the students to learn to construct objects, to initialize them and to send messages to them. This construction takes place with the help of buttons with the simultaneous appearance of the syntax of the corresponding statement (Fig. 6). Using the available messages (buttons on the card) the student can solve the problem without having to write a program. The program is developed in stages by clicking the buttons and the corresponding statement is appended to the text area each time the student

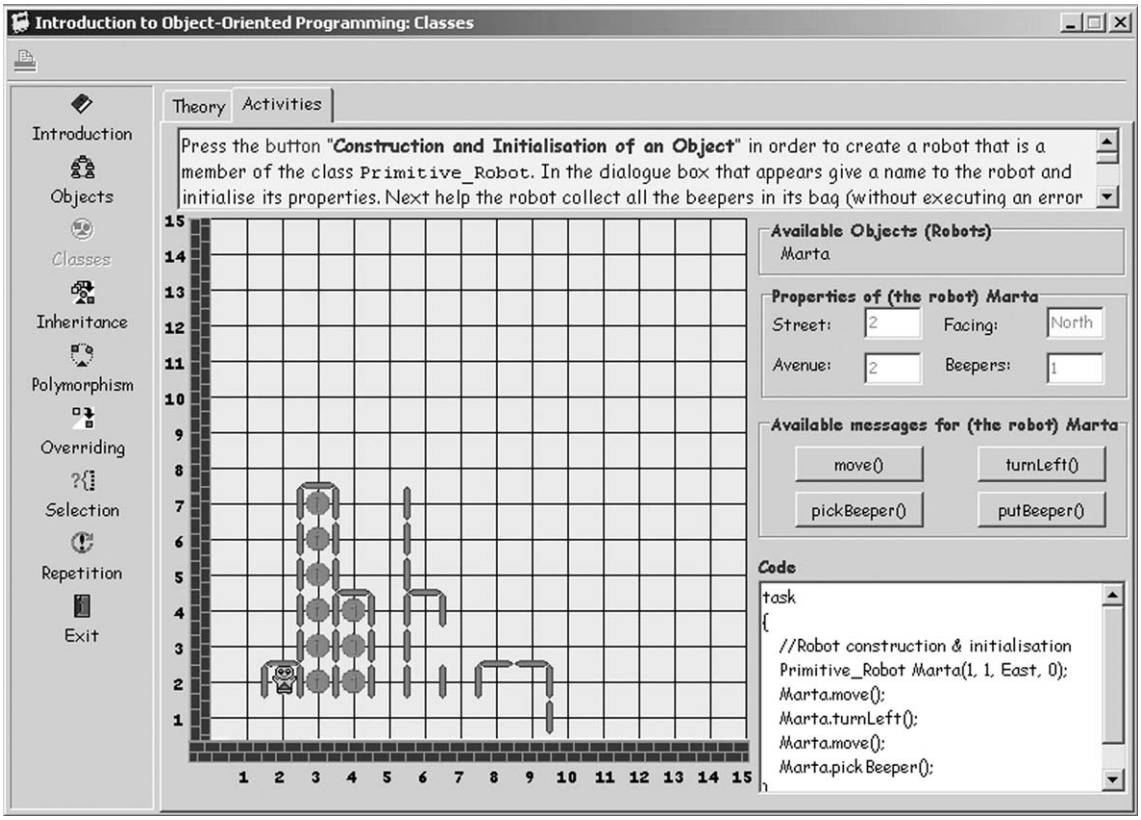

Fig. 6. The activity of the lesson "*Objects–Classes*".

chooses to send a message to the robot. In this activity, the student is also given the form of the main program.

After the end of the activities and in the context of the first lesson, students were asked to solve two exercises using the structure editor of the programming environment. Students designed, compiled and executed their programs in groups in order to become familiar with: (i) the construction of objects, (ii) sending messages to the objects, (iii) sequential structure, (iv) the use of the step-by-step method of program execution, since this is the only way for the students to check whether the robot is taking the desired route. The main aim is to present the advantages of this type of program execution as well as to convince students to use it later on with other more conventional programming environments. Another aim is for the students to become familiar with the development of programs with the help of the structure editor of *objectKarel*.

5.1.1. Results

The students found both the programming environment and the exercises interesting. All students became familiar with the programming environment very quickly. Furthermore, as can be seen

from the consecutive versions of their programs, they do not seem to have had any difficulties with the concepts of object and class. The majority of the programs (72%) did not have any syntactical errors whatsoever; while in the remaining (28%) only one syntactical error had been made. In contrast, under half of the programs had logical errors (39%) and errors of execution (44%). *Logical errors even when they do not lead to errors in execution were immediately located* since the students made changes at the point of source code where the error exists. By directly locating the logical errors it allows us to conclude that the *visualization* of a program's execution reveals the various processes, which take place in the notional machine in a way, which is completely understood by the students. In addition, we believe that significant help was provided to students by the *explanatory* visualization, as well as the fact that run-time errors are indicated by changing the color of the background and the characters of the corresponding source code and simultaneously the cause of the error is explained. Lastly, concerning the strategy of the solution, the vast majority of students developed the whole program and then went on to check it (reference to the solution strategy is made in the following lesson).

5.2. Lesson 2 – inheritance

The aim of the second lesson was for the students to comprehend the notion of inheritance, the advantages of creating new classes and re-using/modifying existing classes, as well as to familiarize the students with the statement of new classes and the defining of methods.

The second lesson consisted of exercises, one of which is indicatively presented below.

Create a new class called ***AugmentedRobot****, whose robots will be able to respond to the messages:*

- ***turnRight***: turning right at 90°
- ***turnAround***: making an 180° turn

Following, create four robots, as seen in Fig. 7, and send to each one of them the message turnRight.

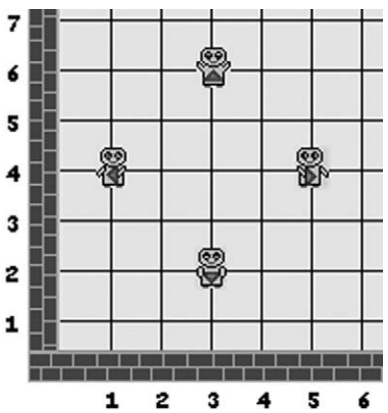

Fig. 7. A sample exercise of the lesson "Inheritance".

The objectives of the exercise were: (i) to ascertain whether Holland, Griffiths, and Woodman's (1997) conclusion, who report that some students tend to become confused between classes and their instances (objects) is verified; or whether Carter and Fowler's (1998) conclusion is verified, who state that the distinction between objects and classes does not cause problems for the students; and (ii) the created class will be used in the following lessons as a parent of new classes so that the students better comprehend the concept of inheritance and the advantages of creating code (classes) that can be reused.

5.2.1. Results

The results attained from the first lesson that were related to *locating logical errors, the correction of all categories of errors* and the **strategic solution** were likewise verified in the second lesson. Regarding errors, half the programs did not contain any errors or contained only one error, which was corrected directly. As was expected, there were more semantic errors than in the first lesson, however, the frequency with which these appeared was very low. The misconception that class and object were identical concepts (Holland et al., 1997) was confirmed for only one group of students. Furthermore, in three out of the 10 groups, we ascertained that the process of developing algorithms, even for simple problems, was particularly time consuming.

5.3. Lesson 3 – polymorphism and overriding

The objective of this lesson was for the students to comprehend the concepts of polymorphism and overriding. These concepts were explained indirectly through the activities of the corresponding units. The third lesson involved exercises whose main objective was to determine the degree of students' comprehension of the notions of polymorphism and overriding as well as the difficulties that they came up against. Fleury (2000) after taping and studying the interviews of students who took an introductory programming lesson based on Java reached the conclusion that *one in three students believe that methods of different classes can have the same name only if they have different signatures*. It is therefore likely that in *objectKarel*, where the methods do not have parameters, for the following misconception to appear: *methods with the same name cannot exist, even if these belong to different classes*. However, we believe that this misconception will not be present in *objectKarel*, since the step-by-step execution of the programs in the context of the activities will help students to realize in the most obvious and persuasive manner, that the compiler can differentiate between methods with the same name based on the class which they belong to.

5.3.1. Results

All students understood that when one method, which exists in different classes, is called (by the same name) then the class of the object determines which of those methods is executed. The students had greater difficulty in understanding the concepts of overriding. 50% of students fully understood the concepts of polymorphism and overriding, while 20% did not understand the concept of overriding or confused the two concepts.

In regards to Fleury's comment that " *. . .many students find duplicate code easier to understand than class reuse*", (2001: 191) we wish to highlight that within the context of the particular lesson a

significant number of students initially repeated the same source code twice in one method instead of summoning it twice, but in the end most corrected the program. Therefore, it seems that:

The step execution of a program and visualization help students to understand the final outcome of the execution of one group of commands; to ascertain the existence of blocks of code which are repeated; and most importantly, to understand and utilize the functionality of the methods.
Some students (three out of the 10 groups) had difficulty in developing a program, which includes more than one class, and/or in understanding and using existing classes.
For a better understanding of the concepts of polymorphism and overriding, two lessons rather than one are needed and naturally students should be given more exercises to practice on.

5.4. Lesson 4 – control structures – selection

In the fourth lesson the concept of selection and the commands *if, if/then* as well as the class *Robot* and the corresponding methods stated in that class were presented with the unit "Selection" (Fig. 8) of the e-lessons. The proposed activity is explained below:

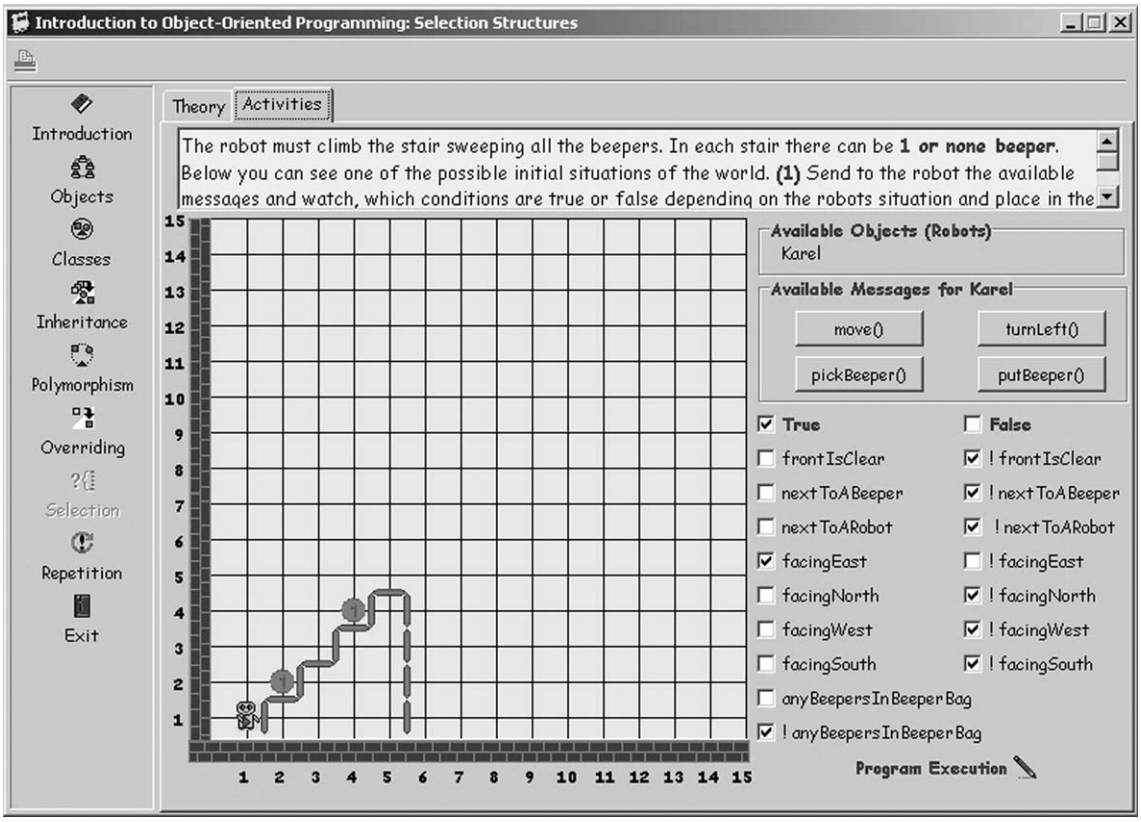

Fig. 8. The activity of the lesson "*Selection*".

The robot must climb the stairs sweeping all the beepers. On each step there can be 1 or no beepers. Below you can see one of the possible initial situations of the world. (1) Send to the robot the available messages and watch, which conditions are true or false depending on the robots situation and place in the world. (2) Press the button "Program Execution" and study/execute the program that solves the problem for all the possible initial situations.

Through the activity in the unit "Selection" the novice programmer understands:

- The importance of the selection structures as well as the flow of the execution of a program which includes those selection structures. Students were recommended to place particular emphasis on the explanations, which appear, at the bottom of the main window (explanatory visualization) in order to deal with difficulties in comprehending the meaning of the commands and the flow of execution.
- The meaning of the methods (predicates) of the class Robot which will enable them to use these methods correctly in the ensuing programs they will develop.
- To become familiar with the negative form of the predicates, which causes more difficulties to novice programmers as opposed to the positive form.

5.4.1. Results

Even though, the literature seems to show that students have difficulty in understanding the syntax and the semantics of the control structures, as well as various misconceptions, in our research, a large majority of students (70%) developed a correct program.

Our hypothesis that the functional error messages, the step-by-step execution and the explanatory visualization would help students overcome many of their difficulties and misconceptions was verified. It is obvious from the presentation of each groups' problem-solving paths that most students had to deal with a variety of difficulties, many of which were related to their misconceptions. From the systematic study of students' consecutive versions of their programs it was seen that without the support provided by the programming environment (functional error messages, step execution, explanatory visualization, access to theory) most groups would not have been able to solve the exercise or they would have come up against very serious problems.

As was expected, almost all the groups made syntactical errors, which mainly had to do with having omitted the syntactic signs which determine the range of the blocks of *then* and *else* of the selection structures.

The vast majority of students corrected the syntactical errors in their first attempt. However, we observed that when the programming environment reports syntax errors students immediately compile the program two or more times without having made any attempt to correct it. As a matter of fact it seems that students do not even read the error messages. We believe that the most prominent reason for this behavior is that students' adopted this attitude during their prior exposure to the incomprehensible and sometimes misleading error messages of professional programming environments. Several times the students do not treat the error messages as inevitable and it is the repetition of the process of compiling – reporting of errors that forces them to comprehend beyond any doubt the meaning of the message and consequently to correct the error – in their first attempt, as we have already mentioned. Below, we give a representative example:

Compilation	Time	Action
1st	17:10:18	Error reported: <exact line number>: The end symbol '}' of the statements block of the main program is missing.
2nd	17:10:22	Students have not made any change.
3rd	17:10:24	Students have not made any change.
4th	17:10:27	Students correct the error.

Finally, regarding error messages, we must mention that there was a number of students that seemed to become tense with the report of errors, that is they find it difficult to correct the errors guided by the messages.

Concerning the strategy of solution, in contrast to the previous lessons, one in two groups developed their program in stages. The change in the strategy of solution in this particular lesson can be attributed to:

- difficulty in developing algorithms which require the use of selection structures;
- difficulty in implementing an algorithm in a programming language which is based on selection structures;
- students' low self-esteem concerning the correctness of the program they develop.

5.5. Lesson 5 – repetition

In the fifth lesson the concept of repetitive structure was presented and didactic situations were created in order for the difference between the structures *repeat and selection* to become clear. The students were asked to solve two exercises with the aim:

- To familiarize students with the repetitive structures *while* and *loop*.
- To examine to what extent the students come up against difficulties related to the three procedures involved in the creation of a loop (Rogalski & Samurcay, 1990) and which had been highlighted during the teaching of programming:
  - *Initialization of variables*: In the particular programming language, even though the variables are not explicitly stated, students have to correctly initialize the robot's properties (variables), which will execute the loop.
  - *Loop invariant update*: Updating the variables – the properties of a robot in the specific programming language – is done by the execution of the methods and is not explicitly expressed by the students (e.g. with an assignment statement). We assume that the activities and the exercises of the previous lessons have made it quite clear that the robots' properties change through the execution of the methods and that the students will use the more suitable methods on the body of the loop in order to have the correct update on a robot's properties.
  - *Test*: This procedure refers to the assessment of the condition.

5.5.1. Results

The students came up against fewer difficulties in the syntax of repetitive structures in contrast to the selection structures. Generally, all errors were significantly reduced from all categories, while all the students developed correct programs. The findings from the consecutive versions of the programs of the four groups which had difficulty show that a significant number of students:

- Confuse the structures *if* and *while*.
- Have difficulty in developing programs which require the use of both the structures *if* and *while*.
- Also, it became obvious that without the support of the programming environment these groups would not have managed to solve the problem or they would have had far greater difficulty. This conclusion was drawn from students' replies in the final questionnaire regarding the usage of *objectKarel* features (see Table 2).

5.6. Lesson 6: evaluation of students' knowledge – evaluation of objectKarel

In the sixth lesson, the students:

- Checked and debugged an existing program which consisted of errors from all categories.
- Responded (individually) to open type questions in order for us to ascertain their level of understanding of the basic concepts of the OOP example.
- Lastly, students were given a questionnaire with both open and closed type questions in order to evaluate the programming environment and the course. This questionnaire was completed outside of the lesson. In particular, the aim of the questionnaire was:
  - to evaluate the usability of the programming environment;
  - to evaluate the specific course as well as the educational material;
  - to locate problems and to receive students' suggestions on how they think the programming environment can be improved; especially their views on the program's syntax, the course in general and the didactic material
  - to locate the functions of the programming environment which helped the students to understand the concepts of objects-first programming.

5.6.1. Results

Although the task of understanding and debugging an existing program is quite difficult and time-consuming, the students in our study did not report any difficulty and did not ask any questions concerning the use of the programming environment's structure editor.

The results of our research show that the task of understanding and debugging an existing program seems to be a tedious process for the beginners. When the students were asked to debug an incorrect program, some of them expressed their preference for rewriting the program from scratch instead of debugging the incorrect one. This was done from the very begging of the task. The preference of these students to rewrite the program might be attributed to the excessive cognitive effort required by debugging. Another plausible account might be the fact that the automations provided by the structure editor during program development possibly hinder debugging. Unfortunately, our data do not allow us to answer this question with certainty.

An analysis of the questionnaires given to students before and after the course is summarized in Tables 1 and 2.

The students' replies on the questionnaire (see Table 2) showed that all the features of the programming environment helped them. The most popular features of *objectKarel* are its structure editor and animation system.

The e-lessons with theory and activities substantially helped both the learning and teaching process. 79% of students responded that teaching the lessons mainly with the use of the programming environment rather than the blackboard and chalk helped them, while 21% answered that they did not know if this helped or not. The same percentage of students (79%) stated that they had had to refer to the theory or to the activities incorporated in the programming environment during the problem-solving process. Furthermore, the students evaluated the structure and the quality of the educational material used in this particular course positively (grading it 4.5 out of 5).

Table 1
Students' difficulties before and after the lessons/course

Difficulties	Before (%)	After (%)
The implementation of an algorithm	79	37
The syntactic rules	26	5
The syntax of conditional structures	32	5
The semantics of conditional structures	32	26
The syntax of repetitive structures	37	5
The semantics of repetitive structures	37	21
The choice of the appropriate structures	52	16
The usage of the program editor	58	5
Understanding the error messages of compilation	32	10

Table 2
Students' usage of *objectKarel* features

Feature	Help provided according to the students	(%)
Incorporated lessons	Teaching process	79
	Problem solving	
Structure editor	Program development	100
	Program editing	95
Program animation	Program debugging	100
	Comprehension of the program's semantics and flow of control	
Explanatory visualization	Deeper understanding of how a program functions	68
	Better understanding of the commands' semantics	
	Debugging	
Friendly error messages	Analytical explanation of syntax errors	90
	Simple language	

All the students stated that the *structure editor* helped them in developing their programs. In the question about the difficulties that the structure editor helped them to overcome they included the following: "I did not have to memorize the commands" (50%), "avoidance of syntax errors" (28%), "guidance during program development" (28%) and "easier understanding of concepts" (11%).

Step by step execution was used by all the students for debugging their programs and understanding flow of control and semantics.

Explanatory visualization proved useful too, since it helped 68% of the students to "understand at a deeper level how a program, as well as isolated commands function" and to "detect errors easier".

90% of the students stated that the friendly error messages, or to use their exact words the "analytical explanation of syntax errors" and the use of simple language", helped them to understand the source of the corresponding errors and to correct them.

Finally, the average score for the eight criteria of usability (Nielsen, 1994) is 4.4 out of 5.

6. Conclusion

The environment *objectKarel* as can be seen from its use is an effective tool for the teaching of OOP to novice programmers. We believe that this is due to the fact that it was constructed bearing in mind the difficulties and the errors of novice programmers, which have been recorded in other related research.

ObjectKarel is an effective teaching tool precisely because it is based on simplicity. In other words, we made a concerted effort to omit details which were unnecessary for the concepts being taught as we believed this would enable students to intuitively form a correct model for the concepts program, program execution, object, class, methods and inheritance. We regard as details the variables where in *objectKarel*, the abstract concept of the state of a variable was replaced by the concept of the situation of the world in which the robots live, as well as the situation of the robots themselves. Students initialize the values of the four properties that define the situation of a robot (street and avenue number, direction and number of beepers in bag) in the context of its construction. During program execution the values of these properties (variables) change through the execution of methods. No explicit reference to the concept of variables is made. In this case the state of the robots' and their world is more realistic than the use of values, which are stored in memory positions.

Furthermore, particular attention was given to the design of the interface and generally to the functions incorporated into the environment, in order to reduce the sources of difficulties that arise in the commercial programming environments. The use of a structure editor and its implementation with dialog frames and menus helped the students in the development and correction of their programs, and as was ascertained from the lessons, the syntactical errors were limited. The possibilities of program execution help in the understanding of the semantics of each command as well as the debugging of logical errors. In addition, by incorporating the explanatory presentation of wrong messages, an attempt was made to reduce the difficulties which the error messages of more common programming environments create in beginners, where the error messages are coded and often are viewed as being "misleading" by novices.

In order to record data related to how novice programmers work while using a particular environment, as well as collect information on the difficulties that arise with OOP, the function of recordability was incorporated into *objectKarel*. This function, as is evidenced from the analysis of the results of each lesson, is particularly useful for studying novice programmers' behavior in developing programs and particularly in verifying (or not) hypotheses that relate to the difficulties as well as the wrong perceptions of these difficulties that novices have.

Of course, it must be stated that the language Karel++ is limited and the environment ObjecKarel was designed to be used for an initial, short introduction to OOP. Needless to say, the transition to an actual programming environment is necessary as a follow up. We suspect that the shift from the *objectKarel* environment to a "real" programming language will not be that easy. The difficulties are located: in the syntax; in the programming environment; in the lack of a structure editor; and in conceptual change. Conceptual change can be regarded as the biggest difficulty of all since the students are used to developing programs that have a visual appeal. These difficulties, however, can be successfully dealt with by using an educational programming environment such as BlueJ.

References

Bergin, J., Stehlik, M., Roberts, J., & Pattis, R. (1997). *Karel++ – a gentle introduction to the art of object-oriented programming* (2nd ed.). New York: Wiley.

Bergin, J., Stehlik, M., Roberts, J., & Pattis, R. Karel J. Robot a gentle introduction to the art of object oriented programming in Java. Unpublished manuscript. Available [April 25, 2004] from: http://csis.pace.edu/~bergin/KarelJava2ed/Karel++JavaEdition.html.

Birch, M., Boroni, C., Goosey, F., Patton, S., Poole, D., & Pratt, C., et al. (1995). DYNALAB: a dynamic computer science laboratory infrastructure featuring program animation. *ACM SIGSCE Bulletin, 27*(1), 29–33.

Brusilovsky, P. (1993). Program visualization as a debugging tool for novices. In *Proceedings of the ACM INTERACT'93 and CHI'93 conference companion on human factors in computing systems* (pp. 29–30).

Brusilovsky, P., Calabrese, E., Hvorecky, J., Kouchnirenko, A., & Miller, P. (1997). Mini-languages: a way to learn programming principles. *International Journal of Education and Information Technologies, 2*, 65–83.

Buck, D., & Stucki, D. J. (2000). JKarelRobot: a case study in supporting levels of cognitive development in the computer science curriculum. *ACM SIGCSE Bulletin, 33*(1), 16–20.

Carter, J., & Fowler, A. (1998). Object oriented students?. *ACM SIGCSE Bulletin, 30*(3), 271.

Chang, C., Denning, P. J., Cross, J. H., Engel, G., Roberts, E., & Shackelford, R., et al. (2001). Computing curricula 2001. *ACM Journal of Educational Resources in Computing, 1*(3), Article #1, 240pp.

Cooper, S., Dann, W., & Paush, R. (2000). Alice: a 3-D tool for introductory programming concepts. *Journal of Computing in Small Colleges, 15*(5), 108–117.

Du Boulay (1989). Some difficulties of learning to program. In E. Soloway, & J. Sprohrer (Eds.), *Studying the novice programmer*. New Jersey: Lawrence Erlbaum Associates.

Fleury, A. E. (2000). Programming in Java: student-constructed rules. *ACM SIGCSE Bulletin, 32*(1), 197–201.

Fleury, A. E. (2001). Encapsualtion and reuse as viewed by java students. *ACM SIGCSE Bulletin, 33*(1), 189–193.

Holland, S., Griffiths, R., & Woodman, M. (1997). Avoiding object misconceptions. *ACM SIGCSE Bulletin, 29*(1), 131–134.

Kolling, M., & Rosenberg, J. (2001). Guidelines for teaching object orientation with Java. *ACM SIGCSE Bulletin, 33*(3), 33–36.

Kolling, M., Quig, B., Patterson, A., & Rosenberg, J. (2003). The BlueJ system and its pedagogy. *Journal of Computer Science Education, Special Issue on Learning and Teaching Object Technology, 13*(4), 249–268.

Miller, P., Pane, J., Meter, G., & Vorthmann, S. (1994). Evolution of novice programming environments: the Structure Editors of Carnegie Mellon University. *Journal of Interactive Learning Environments, 4*(2), 140–158.

Milne, I., & Rowe, G. (2002). Difficulties in learning and teaching programming – views of students and tutors. *International Journal of Education and Information Technologies, 7*(1), 55–66.

Nielsen, J. (1994). Enhancing the explanatory power of usability heuristics. In *Proceedings of CHI '94* (pp. 152–158). ACM Press.

Rogalski, J., & Samurcay, R. (1990). Acquisition of programming knowledge and skills. In J. Hoc, T. Green, R. Samurcay, & D. Gilmore (Eds.), *Psychology of programming*. London: Academic Press.

Ruckert, M., & Halpern, R. (1993). Educational C. *ACM SIGSCE Bulletin, 25*(1), 6–9.

Sanders, D., & Dorn, B. (2003). Jeroo: a tool for introducing object-oriented programming. *ACM SIGCSE Bulletin, 35*(1), 201–204.

Sleeman, D., Putman, R., Baxter, J., & Kuspa, L. (1988). An introductory Pascal class: a case study of students' errors. In R. Mayer (Ed.), *Teaching and learning computer programming*. New Jersey: Lawrence Erlbaum Associates.

Xinogalos, S. (2002). Educational technology: a didactic microworld for an introduction to object-oriented programming. Ph.D. Thesis, Dept. of Applied Informatics, University of Macedonia (in Greek).