PUBLISHED BY

INTECH
open science | open minds

World's largest Science,
Technology & Medicine
Open Access book publisher

3,300+
OPEN ACCESS BOOKS

107,000+
INTERNATIONAL
AUTHORS AND EDITORS

114+ MILLION
DOWNLOADS

BOOKS
DELIVERED TO
151 COUNTRIES

AUTHORS AMONG
TOP 1%
MOST CITED SCIENTIST

12.2%
AUTHORS AND EDITORS
FROM TOP 500 UNIVERSITIES

WEB OF SCIENCE™

Selection of our books indexed in the
Book Citation Index in Web of Science™
Core Collection (BKCI)

Chapter from the book *Advanced Learning*
Downloaded from: http://www.intechopen.com/books/advanced-learning

Interested in publishing with InTechOpen?
Contact us at book.department@intechopen.com

Guidelines for Designing and Teaching an Effective Object-Oriented Design and Programming Course

Stelios Xinogalos
University of Macedonia
Greece

1. Introduction

Teaching Object-Oriented Programming (OOP) to novices is without doubt one of the most challenging topics in Computer Science Education. Extended research has been carried out, which focuses on:

- proposing and/or evaluating teaching approaches (Bennedsen & Caspersen, 2004; Brusilovsky et al., 1997; Chang et al., 2001; Nevison & Wells, 2003; Proulx et al., 2002), such as "objects-first", "model-first", and using coding patterns,
- devising educational programming environments and tools (Allen et al., 2002; Buck & Stucki, 2000; Cooper et al., 2000; Kölling et al., 2003; Sanders & Dorn, 2003; Xinogalos et al., 2006a), and
- studying students' understanding and misconceptions of object-oriented programming (Carter. & Fowler, 1998; Fleury, 2000; Fleury, 2001; Holland et al., 1997; Ragonis & Ben-Ari, 2005a; Truong & Bancroft, 2004; Xinogalos et al., 2006b).

In most cases this research has been carried out with the form of a pilot/empirical study or an experiment in the context of an OOP course. There is little evidence of evaluating an OOP course as a whole: teaching approach(es) used; sequence of concepts presented and organization of lessons; programming environment(s) and educational tools used; students' difficulties and misconceptions recorded and didactical situations/activities for dealing with them. Consequently, the problem for those struggling to design an effective OOP course for their students remains.

In this chapter we present the long-term evaluation and refinement of an OO Design and Programming course, which was initially exclusively based on the well-known environment of BlueJ, the book "Objects First with Java: A practical introduction using BlueJ" (Barnes & Kölling, 2004), the established guidelines for teaching with BlueJ (Kölling & Rosenberg, 2001) and the results of the research regarding the teaching of OOP. Although, the teaching approach adopted is not an empirical one, it was our belief that it should definitely be evaluated by recording students' achievements and studying their conceptions. This evaluation gave us invaluable insights into students' difficulties and helped us reform the course with increasingly positive results the last four years.

In the second section of the chapter we review the research regarding the teaching of OOP to novices that guided the design of our course. Specifically, we review the most important teaching approaches that have been proposed, as well as the educational programming environments and programming microworlds that have been devised.
In the third section we present the most important results, referring to the design of the course, from the long-term evaluation and refinement of the "Object-Oriented Design and Programming" course that is being taught for the last four years at a Technology Management Department. First, we describe the placement and the aims of the course, which played a central role in its design process. Next, we present the results of three distinct versions of the course.
In the fourth section we present some guidelines for designing and/or teaching an effective OOP course. These results are based on the rigorous four-year evaluation and refinements of the concrete undergraduate "Object-Oriented Design and Programming" course described, but are presented in such a way that can be exploited by CSE educators in various contexts. Specifically, we present:

- the proposed refined sequence of lessons. The fundamental OO software development tasks and programming concepts taught in each lesson, the tools used for teaching them and the time allocated for each lesson are presented.
- the advantages of three distinct, but complementary for our purpose, environments: objectKarel, BlueJ and JCreator that represent a programming microworld, an educational IDE, and a more professional programming environment respectively. Emphasis is given on presenting guidelines for making good use of the features of BlueJ and avoiding the underlying pitfalls of this or similar environments that give the chance of direct manipulation of objects.
- guidelines for teaching each one of the proposed lessons. The way the new concepts are presented, the most common difficulties and misconceptions which were recorded in our four-year evaluation of the course, as well as specific didactical interventions are presented.

Finally, we present our future plans that focus on developing even more appropriate educational material (i.e. examples, activities, assignments).

2. Literature Review on the Teaching of OOP

2.1 Teaching approaches

Teaching OOP to novices is widely known to be quite problematic. As the authors of the ACM Curricula report acknowledge, the objects-first strategy creates added difficulties to both the teaching and learning of programming (Chang et al., 2001). Various teaching approaches have been proposed for supporting the objects-first strategy and making the teaching and learning of OOP concepts easier and more effective. The most important are described briefly in the following paragraphs.
Presenting especially designed examples for avoiding common difficulties. Holland et al. (1997) suggest the presentation of examples and the assignment of exercises especially designed for avoiding misconceptions that have been recorded and cannot be easily shifted once acquired by students. For example, in order to avoid the misconception that "object" and "class" are the same concept, teachers should use examples where several instances of each class are used.

Using Graphical User Interfaces. Proulx, Raab, & Rasala (2002) state that teaching should begin with the concept of objects, which should have a noticeable behavior and should interact with other objects. The familiarization of students with the main concepts of OOP is accomplished through a series of labs and lectures that are based on using GUIs developed by Proulx et al. (2002) in Java. Although some researchers (Kölling et al., 2003) are opposite with the use of GUIs as a medium of teaching OO concepts, Proulx et al. consider their use necessary. In each one of the 4 labs described (Proulx et al., 2002) a different GUI is used with the aim of: (1) exploring the behavior of objects without making any reference to the source code (1st lab); (2) observing the changes of member data of the objects as a consequence of responding to actions that invoke member functions, and learning the syntax of such calls (2nd lab) ; (3) studying part of the source code that implements the GUI that is used, and making small changes (3rd lab); (4) extending an existing class (4th lab).
Using Case Studies. Nevison & Wells (2003) believe that teaching OOP should begin with teaching objects, as most of the instructors do. However, in order to present interacting objects the use of difficult/complex examples is acquired. Nevison and Wells suggest the use of case studies that give the chance to present complex examples at the beginning of an introductory programming course, as well as the ability to adopt an object-oriented approach to solving problems. The choice of a case study must be done carefully, which means that the complexity of the system that the case study describes favors the presentation of object-oriented concepts and the implementation of simple programs that can be used in an introductory programming course. Also, the case studies that are going to be used must allow the gradual presentation of concepts (one concept each time) through the development of a series of programs of increasing complexity.
Using Educational Programming Environments. Empirical studies and experience has shown that one of the most important sources of difficulties for novice programmers is the use of professional programming environments for their introduction to programming. This ascertainment has led many researchers in implementing educational programming environments. Educational programming environments give emphasis on the didactic needs of students rather than the professional needs of programmers. Kölling et al. (2003) in particular state that teaching OOP is difficult due to the lack of appropriate tools and teaching experience for the specific programming paradigm. The most important problems of the environments that are used for teaching OOP are: (1) the environment is not object-oriented; (2) the environment is complex; (3) the environment focuses on the development of GUIs and not on object-orientation.
In order to avoid these problems Kölling et al. (2003) developed the BlueJ environment. In contrast with the rest educational programming environments BlueJ is accompanied with a textbook (Barnes & Kölling, 2004; 2006) and an established set of guidelines for teaching (Kölling & Rosenberg, 2001). The approach proposed is an "objects-first", iterative (important concepts are taught first and often), **project-driven approach**. Students begin with a predefined set of classes, create objects and invoke the available methods, in order to study the objects' behavior. The predefined classes are used for presenting the syntax of Java. Next, students extend existing classes by implementing or adding their own methods. The next step is the definition of classes by students in the context of an existing project. Finally, students are separated into groups and implement an application.
Using Programming Microworlds. Programming microworlds are a special kind of educational programming environments. What differentiates them is the fact that they use

an educational programming language or a subset of a conventional language. Programming microworlds are designed and implemented entirely for didactic purposes and as a consequence they share some common characteristics (Brusilovsky et al., 1997): (1) they are, usually, based on existing physical metaphors and use an educational programming language; (2) they constitute integrated programming environments that are characterized by usability and incorporate various forms of educational technology for supporting students, such as software visualization technologies; (3) the problems solved are, usually, everyday problems. Teaching with programming microworlds focuses on concepts, and not on the syntax of the language. However, in some cases the programming language used constitutes a subset of a conventional programming language, and learning its syntax is part of the didactical process.

Using a model-first approach and coding patterns. The model-first approach considers conceptual modeling - which refers to the use of programming constructs for describing concepts, structures and phenomena-, as the defining characteristic of object orientation. Bennedsen & Caspersen (2004, pp. 478) summarize nicely the idea of this approach: "*Introduction of the different language constructs are subordinate to the needs for implementing a given concept in the conceptual framework. After introducing a concept from the conceptual framework a corresponding coding pattern is introduced; a coding pattern is a guideline for the translation from UML to code of an element from the conceptual framework.*"

2.2 Educational Programming Environments

Numerous educational programming environments have been developed for supporting students in their introduction to OOP. However, most of them have not been thoroughly evaluated and their usage maturity is small, with the exception of BlueJ (Georgantaki & Retalis, 2007). A brief comparative presentation of various educational programming environments based on the evaluation approach suggested by McIver (2002) is provided in (Georgantaki & Retalis, 2007). In the next paragraphs we briefly present five such environments taking into account the graphical interface, interactivity, visualization, compilation features, and generally the support provided for teaching the OOP paradigm.

BlueJ (http://www.bluej.org). The main window of BlueJ (Kölling et al., 2003) demonstrates in a visual way classes and objects, as well as the application' s structure in the form of a simplified UML diagram. The UML diagram used presents the names of the classes and their inter-relations. Students can use the interactive interface of BlueJ in order to construct objects, invoke their methods and inspect their state. The source code of each class is presented in a separate window, using limited highlihting of source code elements. The editor supports line numbering, indenting, and bracket matching. Syntax errors are presented in the bottom of the class' s window one at a time, while the line of the error is automatically higligthed. The error messages are explanatory enough, providing the possible reason that caused the syntax error. BlueJ incorporates a visual debugger that allows the student to insert breakpoints and then execute the program step by step. When the pop-up menu of a class or an object is used in order to construct an object or call a method respectively, the window of the debugger appears automatically. Information about threads, call sequence, static/instance/local variables is presented.

DrJava (http://drjava.org). DrJava (Allen et al., 2002) has a simple interface based on a "Read-Evaluate-Print-Loop" (REPL). In its single window students can develop, test and debug a program in an interactive, incremental way. The editor supports brace matching,

syntax highlighting and automatic indenting. Students can type Java expressions and statements and see immediately the results, without the need for the main method. In this way, the idea that each individual unit of a program should be separately tested is reinforced. In the case of compilation errors, students can click on an error in order to highlight the corresponding line of the source code. DrJava incorporates a debugger, which is activated from a menu choice. Then the student can insert breakpoints and use the commands in the Debug menu to step into/out/over the execution of a method.

Ginipad (http://www.mokabyte.it/ginipad/english.htm). The editor of Ginipad uses colour for highlighting elements of the source code, as well as the function of code auto-completion. Classes, methods and fields are presented in a separate pane in a tree form, updated automatically, when the student edits the source code. Clicking on an element of the tree results in highlighting the corresponding code fragment in the editor. Ginipad supports interaction with error messages, but it does not incorporate neither a debugger nor the function of step-by-step execution.

JGrasp (http://www.end.auburn.edu/grasp/). The editor of JGrasp supports highlighting of source code elements using colors, as well as numbering of source code lines. What is more important is the generation of Control Structure Diagrams (CSD) intended to depict control structures, control paths and the overall structure of each program unit. JGrasp supports students in developing programs with templates for defining classes, methods and control structures. The description of the first compilation error, as well as the line of the source code that most likely caused it are both highlighted. JGrasp provides a visual debugger that allows students to set breakpoints and then execute the program one statement at a time. It also generates UML class diagrams for the classes of the current project. Furthermore, students can create instances from any class in the UML class diagram or Java. The instances of the classes are placed in an object workbench, as is the case for BlueJ too, and students can invoke their methods in isolation without having to write a main method. Students can also invoke class or static methods directly from the class diagram.

JCreator LE (http://www.jcreator.com/index.htm). The JCreator LE is the freeware version of the JCreator IDE used for learning purposes. The environment presents in different panes all the files of a project in tree form and an overview of the current class's structure in an expandable tree. It allows the addition of new classes to the project through templates and dialog boxes, as well as the addition of new methods to a class with the help of dialog boxes. JCreator LE gives the option of viewing line numbers in the selection margin, instant color syntax highlighting, auto-indent, word completion and code folding which allows the user to hide parts of the code. It does not have a tool for debugging and as a result does not support step-by-step execution. The version JCreator Pro (shareware) contains a debugger, step-by-step execution and visualization features.

2.3 Programming Microworlds

The most well known microworld is Logo, which was created by Seymour Papert. Although it was not designed specifically for the teaching of programming it appeared to be an effective tool for supporting it. However, the majority of the educational tools that have been developed in the context of this approach are based on the robots Karel (Pattis et al., 1995) and Karel++ (Bergin et al., 1997), which are used for introducing novices to procedural and object-oriented programming respectively. The metaphor used is that of a world of robots, that are assigned various tasks in a world comprised of crisscrossing horizontal

streets and vertical avenues, wall sections (that represent obstacles) and plastic cones that emit a beep noise (beepers). For example, a robot might be assigned to harvest a field of beepers, to escape from a labyrinth and so on. Representative programming microworlds are:

JKarelRobot (http://math.otterbein.edu/JkarelRobot). JKarelRobot (Buck & Stucki, 2000) is based on robot Karel and supports three languages: Pascal, Java and Lisp. JKarelRobot uses Control Structure Diagrams (CSD) for presenting programs and gives the chance to develop flow diagrams for existing programs. The programs are executed step by step either forward or backwards. The restriction of this simulator, regarding object-orientation, is the fact that students can create just one robot. This might lead students to the recorded misconception that refers to the object-class conflation.

Jeroo (http://info.nwmissouri.edu/~sanders/Jeroo/Jeroo.html). Jeroo (Sanders & Dorn, 2003) uses as a metaphor a kind of an Australian kangaroo and a programming language related to C++ and Java. The user interface consists of a single window, while the environment supports the implementation of recursive methods. However, Jeroo has the following restrictions: there is only one class; students can create up to four robots; inheritance is not supported; students can extend the Jeroo class with void methods, but not with predicates.

objectKarel (http://csis.pace.edu/~bergin/temp/findkarel.html). objectKarel (Xinogalos et al., 2006a) is based on Karel++ and uses a programming language related to C++ and Java. objectKarel incorporates a series of e-lessons and hands-on activities for motivating students through the use of concepts before they are asked to implement them. Programs are developed with a structure editor, which has the form of a menu that is automatically updated with the names of new methods, and dialog boxes. Students can run, trace with a predefined speed or execute programs step by step (only forward). Also, objectKarel incorporates the technology of explanatory visualization, which means that students are presented with explanatory messages for the semantics of the statement being executed. Furthermore, it incorporates the ability of recording students' actions during program development (recordability). Each compiled version and the compiler output is recorded and presented with the form of a two level tree.

Karel J. Robot (http:// csis.pace.edu/~bergin/KarelJava2ed/Karel++JavaEdition.html). Karel J. Robot is a Java-based descendant of Karel++ that has been recently developed.

Alice (http://www.alice.org). Alice (Cooper et al., 2000) is a successful microworld, which is used for building virtual worlds with 3D objects. Alice is based on the Python (www.python.org) language and has an object-oriented flavor. Students create virtual worlds that are populated by 3D objects, such as animals and vehicles. Students control the appearance and the behaviour of objects by writing simple scripts – invoking methods and then they watch the animation.

3. The long-term evaluation of the OO Design and Programming Course

3.1 Description of the course

The "Object-Oriented Design and Programming" course described in this chapter is a third semester compulsory course at a Technology Management Department. The duration of the semester is normally 13 weeks and the course consists of a weekly two-hour lecture and two-hour lab. Prior to this course, students are introduced to "Computer Programming"

with C. This means that basic programming concepts, such as variables, constants, conditional statements, loops and functions, are considered already known. However, passing the "Computer Programming" course is not a prerequisite for attending the "Object-Oriented Design and Programming" course.
The goals of the "Object-Oriented Design and Programming" course can be summarized as follows: (1) focus on the fundamental OO software development tasks and programming concepts rather than the teaching of Java; (2) learn to read class documentation and use Java standard library classes; (3) learn to distinguish, use and extend elements of a given class (fields, constructors, accessor and mutator methods); (4) designing simple applications using the OO technique; (5) implementing programs.

3.2 Evaluation of a course based on an educational IDE

The course was taught for the first time the academic year 2005-06. Based on the aims of the course and the literature regarding the teaching of OOP we organized the course based on the programming environment BlueJ and the book "Objects First with Java: A practical introduction using BlueJ" (Barnes & Kölling, 2004).
The teaching approach of Barnes and Kölling seemed to fit perfectly with the main aim of the course, which is the comprehension of fundamental OO software development tasks and programming concepts rather than the teaching of Java. Specifically, both the environment and the book, and consequently our course, shared the following well-known guidelines for teaching Object Orientation with Java (Kölling & Rosenberg, 2001): (1) Use an Objects first approach; (2) Don't start with blank screen; (3) Read code; (4) Use "large" projects; (5) Don't start with main; (6) Don't use "hello world"; (7) Show program structure; (8) Be careful with the user interface.
Heavily based on BlueJ, the accompanying text book and the guidelines mentioned above eleven two-hour lectures and eleven two-hour labs were organized with the following content: (1) Objects and classes; (2) Understanding class definitions; (3) Object interaction; (4) Flexible and fixed size object collections; (5) Using class libraries (i.e. HashMap, HashSet); (6) Testing and Debugging; (7) Designing classes; (8) Improving structure with inheritance; (9) Polymorphism and overriding; (10) Abstract classes and interfaces; (11) Static methods – the main method.
A vast amount of data was collected during the lessons from questionnaires, programming assignments carried out during lab sessions or at home, the final exams and conversations during lectures and labs. The rigorous analysis of the data identified several difficulties, which were categorized and subcategorized as follows (Xinogalos et al., 2006b):
Category 1 – "typical" difficulties encountered independently of the programming paradigm.
Category 2 – difficulties attributed to the special characteristics of OOP:
Subcategory 2.1 – constructors
Subcategory 2.2 – object instantiation
Subcategory 2.3 – "set" methods (mutators)
Subcategory 2.4 – "get" methods (accessors)
Subcategory 2.5 – method calling
Subcategory 2.6 – access modifiers
Subcategory 2.7 – object collections
Subcategory 2.8 – inheritance

Subcategory 2.9 – abstract classes/methods and interfaces

Despite these difficulties, students managed to carry out their assignments and comprehend basic OOP concepts. Furthermore, several difficulties and misconceptions recorded by Ragonis and Ben-Ari (2005a) in the context of a teaching based on BlueJ were not recorded in our course. This resulted in our decision to continue teaching the course with the same approach. However, it became clear that some adjustments should be made, since some difficulties were attributed to specific features of the course and BlueJ:

Feature 1 – emphasis on visualization and direct manipulation techniques incorporated in BlueJ. Several difficulties regarding "object instantiation" (subcategory 2.2) and "method calling" (subcategory 2.5) were attributed to this feature of BlueJ, such as: (1) omitting the type of the variable that keeps a reference to the object being instantiated due to extensive use of BlueJ' s pop-up menu for constructing objects; (2) calling a non-void method as a void method due to extensive use of BlueJ' s interface for invoking methods – even in the case of a non-void method the returned value is presented in a window and students do not handle it (i.e. assign it to a variable, use it in an expression). Ragonis & Ben-Ari (2005b) have also identified difficulties regarding the dynamic aspects of OOP that can be attributed to the extended use of the features of BlueJ.

Feature 2 – emphasis on existing projects. Students become accustomed to working with existing projects that have to be extended and face difficulties when they have to put everything together from scratch. Students face difficulties even with "set" and "get" methods (subcategories 2.3 and 2.4) and "constructors" (subcategory 2.1), since these code fragments are usually provided in the project and, although they see them, they rarely have to implement them. In the case of object collections (ArrayLists) students' difficulties are much more severe. Although students seem to comprehend the concept of object collections, they find it difficult, if not impossible, to manipulate flexible size collections for grouping objects (Xinogalos et al., 2006b).

Feature 3 – postponing the main method. Postponing the main method for the end of the course means that students have to use exclusively the interface of BlueJ for testing code, which leads in the difficulties mentioned above (see Feature 1).

3.3 Reformation of the course with the combined use of an educational and a professional IDE

Based on the results of evaluating the course the first year it was offered to students we re-designed it. The main adjustments that were made can be summarized as follows (Xinogalos et al., 2007):

- Students used the interactive GUI of BlueJ for creating objects and calling their methods in the first three lessons. In the 4th lesson students were taught the main method (11th lesson in the 1st version of the course) and started constructing objects and calling their methods by writing code.
- Students used the features of BlueJ with more caution. Students continued to use the direct manipulation features of BlueJ for experimenting with existing projects and debugging their own after the 4th lesson, but they were always asked to achieve the desired result by providing source code too. Furthermore, the environment of JCreator was presented to students and they had the choice of selecting the environment that fitted better to their needs.
- Students started to develop simple projects from scratch much earlier in the semester.

- We decided to omit the lesson about debugging (6th lesson in the 1st version of the course) and devote two lessons on object collections (ArrayLists), which turned out to be one of the most difficult concepts for students. Basic debugging techniques were presented in the context of one of the first labs and students were encouraged to study the corresponding chapter.
- Special didactical situations and assignments were designed based on the results of the assessment.

The evaluation of the re-designed course gave better results than the 1st version of the course. The evaluation showed that (Xinogalos et al., 2007):

- Some of the difficulties that were attributed to the emphasis given on the features of BlueJ (Feature 1) in combination with the late use of main (Feature 3) were eliminated. For example, in the 1st version of the course we found out that "some students do not declare the type of the variable that keeps a reference to the object they instantiate" (Xinogalos et al., 2006b). This difficulty was not recorded in the redesigned course. Furthermore, some difficulties regarding the dynamic aspects of OOP were significantly reduced, such as calling a non-void method from main as a void method.
- Some difficulties attributed to the emphasis given on existing projects were addressed satisfactorily. For example, developing projects from scratch early in the semester helped students face more effectively various difficulties, such us leaving the body of a constructor empty (subcategory 2.1), omitting the type of an object variable (subcategory 2.2), and directly accessing private fields outside their class instead of using "get" methods (subcategory 2.4, 2.6).

Although the re-designed course gave better results, is it obvious that several difficulties continued to exist. The third year of teaching the course no changes were made on its main features. The only change was a refinement of the assignments and the activities carried out at labs, based on the 2-year evaluation and experience of teaching the course.

3.4 Gradual exposure to OOP concepts with a microworld, an educational and a professional IDE

Although the re-designed course was taught with positive results for two academic years it is clear that students still faced various difficulties and there was still room for improvement. It is our belief that a major source for various students' difficulties is a flawed comprehension of OOP concepts. This is more intense when we have to deal with more advanced concepts, such as inheritance, polymorphism and overriding. This belief combined with the advantages of microworlds for introducing students to programming concepts (Brusilovsky et al., 1997) led us in studying students' conceptual grasp of OOP concepts in a course with BlueJ (the first course described in this paper) and a teaching with objectKarel. This study was based on a common test taken at the end of both teachings. The most important finding of this study was that the students taught with objectKarel were found to have a significantly better conceptual grasp of basic OOP concepts than the students taught with BlueJ (Xinogalos, 2008a).

The results of this study strengthened heavily our intention to devote the first two lessons (lectures and labs) for introducing students to the most fundamental OOP concepts with the programming microworld objectKarel (Xinogalos et al., 2006a). objectKarel uses a physical metaphor, that is robots carrying out various tasks on a restricted world. The environment incorporates a structure editor for developing programs, program animation, explanatory

visualization features, and enhanced error reporting. The introduction to OOP with objectKarel aimed at:

- presenting in a clear and concise way the most fundamental OOP concepts.
- concentrating exclusively on the concepts, leaving aside the obstacles of learning the syntax of a programming language and struggling with incomprehensible error messages.
- using the technology of program animation and explanatory visualization for supporting students in comprehending the semantics of constructs and concepts and revealing misconceptions before they are established.

So, the 4th year (2008-09) of teaching the course the first two lectures and labs were based on objectKarel. In the 1st lesson the concepts object, class, message/method, object instantiation and inheritance were presented. The 2nd lesson was devoted to multilevel inheritance, polymorphism and overriding. In the 3rd lesson students were taught class definitions (2nd lesson in the 1st version of the course, the 1st lesson was no longer needed).

Another change in the course was the sequence of activities and assignments used for comprehending ArrayLists. Carefully didactic activities/situations and assignments were devised: (1) comprehending the structure of an ArrayList by presenting simple programs and the corresponding object diagrams, and then having students draw object diagrams in order to simulate basic ArrayList operations; (2) filling in blanks that represent error prone elements in an excerpt of code, in order to think about specific concepts and constructs more consciously and comprehend them; (3) developing projects that use ArrayLists from scratch.

The evaluation of the course based on the combined use of the programming microworld objectKarel, the educational programming environment BlueJ and the professional programming environment JCreator gave even better results that the previous version of the course. Specifically, we found that (Xinogalos & Satratzemi, 2009):

- Students' active participation and achievements in the written exams improve, as the course evolves.
- The results regarding specific students' difficulties show a gradual improvement for most of the subcategories of difficulties, and in few circumstances slight variations.
- The interventions on the teaching of ArrayLists had positive results. Better results were recorded regarding basic operations on ArrayList collections: iterating an ArrayList collection, retrieving the objects stored in an ArrayList, accessing the private fields of the retrieved objects using "get" methods.
- The use of objectKarel for introducing students, at the very beginning of the course, to the most fundamental OOP concepts turned out to be a good decision. The main concern regarding the use of objectKarel was whether the knowledge acquired in its context would be transferred afterwards to Java. In a questionnaire about the use of the three environments, 69% of the students stated that did not face any difficulty in associating the concepts taught in objectKarel with the corresponding ones presented afterwards with the use of Java.
- Furthermore, more students participated in the lectures and the labs, completed the assignments and gained confidence in their ability to program, in comparison with previous years. In the questionnaire mentioned above, 91%of the students stated that the introduction to the main concepts of OOP with objectKarel helped them comprehend these concepts.

4. Guidelines for Designing and Teaching an OOP Course

4.1 Outline of the course

Based on the placement of the course (section 3.1) and its four-year rigorous evaluation and reformation, we have ended up to the series of lessons presented in Table 1. The course is based on the combined usage of three distinct but complementary for the goals of the course programming environments: objectKarel, BlueJ and JCreator. The didactical material is based on the material incorporated in the programming microworld objectKarel and the textbook "Objects First with Java: A practical introduction using BlueJ" by Barnes and Kölling (2006). In section 4.2 we present the rationale of using each one of the three programming environments, and in section 4.3 we present the most common difficulties and misconceptions as they were recorded in our four-year evaluation of the course, as well as specific didactical interventions, activities and/or assignments for dealing with them.

Lesson	Environment	Content
1	objectKarel	*Objects, classes, inheritance*: object, construction & initialization of an object, classes, messages/methods, attributes and behavior of an object, inheritance, superclass, subclass, inheritance hierarchy, UML class diagram
2	objectKarel	*Multilevel inheritance, polymorphism and overriding*
3	BlueJ	*Class definitions:* fields, constructors, accessor and mutator methods, return statements, parameters (formal, actual), variable scope/lifetime, conditional statements
4	BlueJ	*Object interaction*: abstraction, modularization, objects creating objects, multiple constructors (overloading), class diagram, object diagram, primitive and object types, internal/external method call
5	BlueJ, JCreator	*Static methods*: instance vs. static/class methods, the main method, executing without BlueJ, byte code, Java Virtual Machine
6, 7	BlueJ, JCreator	*Grouping objects in collections*: flexible size collections (ArrayList), fixed size collections (array), generic classes, iterators, loops (while, for, for-each)
8	BlueJ, JCreator	*Using class libraries*: Java standard class library, reading documentation, interface vs. implementation of a class, exploring and using classes (Random, HashMap, HashSet), access modifiers (public, private), information hiding, class/static variables
9	BlueJ, JCreator	*Designing classes*: coupling, cohesion, encapsulation, code duplication, responsibility-driven design, code reuse
10	BlueJ, JCreator	*Improving structure with inheritance*: inheritance, superclass, subclass, inheritance hierarchy, superclass constructor, subtyping, substitution, autoboxing
11	BlueJ, JCreator	*Polymorphism, overriding*: static/dynamic type, dynamic method lookup, super (in methods), protected access
12	BlueJ, JCreator	*Abstract classes and interfaces*

Table 1. Course outline.

4.2 Usage of programming environments in the course

The choice of a programming environment for introducing novices to OOP is crucial for the success of a course. However, this choice is not an easy one. Professional programming environments are not suitable for such an introduction, since they have too many options, use terminology that novices ignore, incorporate debugging capabilities that cannot be easily utilized by novices, and generally are designed to fulfill the needs of professionals. Programming microworlds and educational programming environments seem to be a better choice. Programming microworlds are, usually, used when the goal is presenting OOP concepts and not a specific OOP language. Educational programming environments, on the other hand, are used when the goal is presenting OOP concepts using a conventional language, usually Java. Needless to say, no matter what sort of educational tool is utilized for the introduction to OOP, the transition to a professional programming environment is necessary. The four-year evaluation and reformation of the course, as well as experience in teaching programming in other contexts, has shown that more than one tools are necessary in order to teach cognitive demanding topics, such as programming. The rigorous evaluation of the course established the following guidelines, regarding the choice and combined usage of complementary programming environments that fulfill different goals of the course:

- *Use a programming microworld for an initial, short presentation of fundamental OOP concepts.* The microworld used should, ideally, use an attractive and simple for students metaphor and GUI, and incorporate various forms of educational technology, so as to concentrate on the comprehension of concepts and not on syntax details. An ideal environment of this kind is objectKarel, which: (1) uses a simple metaphor of a world of robots that has features of video games; (2) incorporates hands-on activities, based on direct manipulation, program animation and explanatory visualization, for familiarizing students with concepts before they are asked to implement them; (3) includes a structure editor that guides students through the process of developing programs and eliminates the need to learn the syntactic details of the language; (4) supports an easy compilation process, interaction with compilation errors that describe the source of the error in natural language; (5) uses program animation and explanatory visualization in order to help students comprehend the semantics of OOP concepts and control structures. The most important reasons and at the same time advantages of using objectKarel, or a similar microworld, are the following: (1) fundamental OOP concepts are presented in a clear and concise way: objects are no more an abstract concept, but instead they are depicted entities that are manipulated by students; (2) students gain confidence in their ability to program and are not frustrated from the complexity of the real thing.
- *Use an educational programming environment for connecting the OO concepts presented in the microworld with their Java implementation.* We believe that the transition from the microworld to an educational programming environment will be much smoother than a transition to a professional programming environment. An appropriate educational programming environment will help students deal with difficulties located in: the syntax; the complexity of the programming environment; the lack of a structure editor; and most importantly in the conceptual change that is located in students' familiarization with developing programs that have a visual appeal. The environment selected must have a simple GUI that supports easy compilation, explanatory error

messages, debugging and visualization features. Visualization of classes and objects, interactivity and direct manipulation of classes and objects is extremely important. Students must be able to create objects and invoke their methods interactively without the need of writing a main method. The most well-known environment of this kind is BlueJ. However, such environments must be used with caution. The interaction and direct manipulation techniques provided by environments of this kind must be used for familiarizing students with OOP, but students should not rely on them for too long. Extended use of these features leads to misconceptions, especially regarding the dynamic aspects of OOP.

- *Use a professional programming environment for preparing the professionals of tomorrow.* It is inevitable that students will have to use a professional programming environment at some time. When this is accomplished in the context of an undergraduate course students gain satisfaction and confidence on their knowledge. Of course, this shift from the educational programming environment must be done at the right time. And the right time is when students feel confident enough for this transition. Since this time may vary for each student, we decided to present the professional programming environment to students at the fifth lesson. Students are guided through an activity in the process of writing, compiling, debugging and executing a project taking advantage of main functionalities of such a professional programming environment. In order to encourage students to use such an environment we give emphasis on features that provide help to them, such as: (1) the tree presentation of the classes of a project and the data/function members of a class that make navigation and exploration of a project easier; (2) the auto-completion ability; and (3) the enhanced highlighting of source code elements with colour. Furthermore, we provide them with a brief manual describing in steps the processes of writing, compiling, debugging and executing a project.

4.3 Guidelines for teaching the proposed lessons

Each one of the lessons consists, as we have already mentioned, of one two-hour lecture and one two-our lab. The teaching approach is heavily based on the approach of the textbook accompanying BlueJ (Barnes & Kölling, 2006):

- *"Objects-first"*: students create objects and call methods from the first lessons.
- *Iterative approach*: all the main concepts are presented early and are revisited and examined in more depth as the lessons advance.
- *Focus on OOP concepts and not on the syntax details of Java.*
- *The lessons are organized based on the fundamental tasks of developing an OO application and not the constructs of Java.*
- *Project-driven approach.*

Also, elements of the "*model-first*" approach (Bennedsen & Caspersen, 2004) are applied. The typical procedure for presenting new concepts is the following : (1) a problem is described that requires the creation of a model of some existing, usually, model of a system; (2) basic concepts of the problem's domain and their relations are presented; (3) a cognitive model of the system is constructed and the parts from which the model is structured – and consequently the objects that comprise the domain of the problem- are recognized; (4) the cognitive model is presented with the form of a simplified UML diagram showing all the classes and their relations, (5) the classes are implemented, or their implementation is presented, introducing new concepts and constructs; (6) design and code patterns are

presented for application in similar situations. In the next sections we present important information for each lesson: how are the new concepts presented; what difficulties have been recorded; and what didactical interventions have been devised for dealing with these difficulties. The projects used for presenting the new concepts are the projects that come with BlueJ (Barnes & Kölling, 2006). The name of each project and its chapter are referenced in parentheses.

4.3.1 Lesson 1: objects, classes and inheritance in objectKarel

The goal of the 1st lesson is the presentation of the most fundamental OOP concepts with objectKarel: objects, classes and inheritance. All the necessary didactical material (theory, activities) is incorporated in the environment itself. The activities incorporated in objectKarel play a central role in the lessons based on objectKarel, and both the lecture and the lab should – if possible – be carried out at the lab.

Students construct and initialize a robot (object) of the basic model (class) using a dialog box that guides them. Emphasis is given on the fact that a robot's state is defined by specific data values, such as its location (street and avenue). Although students do not declare instance variables (fields) it is explained that the initial state of a robot is stored in special variables, called instance variables, which are updated during program execution. Then, students click on buttons labelled with the names of the commands (messages/methods) recognized by a robot of the basic model in order for the robot to carry out a specific task and watch: (1) how the robot responds to each one of the available messages; (2) how the execution of methods alters the state of the robot (the values of fields are presented); (3) what is the syntax of the actions performed interactively in the language of robots. Although, students do not – as we have mentioned – declare fields, emphasis must be given by the instructor on the fact that: execution of methods can alter the state of an object, which is defined by the values of its fields; the behaviour of an object may alter substantially based on its state (Holland et al., 1997).

The presentation of the concepts "class and object" is followed by the presentation of a simple program that is solved using a robot of the basic class and the primitive methods defined in it. Despite the fact that the problem is conceptually simple, its solution consists of a large number of statements and students find it difficult to implement, debug and extend it. In the context of this situation students are presented with the ability to extend the capabilities of the basic class of robots by defining a new class that inherits the instance variables and methods of the basic class and extends it with the definition of new methods. The solution of the problem with a new class that takes advantage of inheritance is presented. This way, students comprehend the advantages of inheritance. Program animation and software visualization is used for executing programs in order to comprehend program flow, as well as to make clear that work in methods is done by message passing (Holland et al., 1997).

The simple metaphor and the hands-on activities of objectKarel give the chance to present the concepts in a clear and concise way, making it difficult for common misconceptions to arise. However, the instructor must carefully select assignments. For example, at least one assignment must involve the construction of multiple objects of a single class, so as to avoid the "object-class conflation" (Holland et al., 1997). Moreover, students can be given the source code of a program that uses an object of the basic class and duplicates code, and be asked to refactor the program using inheritance in order to improve it. Although, students find duplicate code easier than class reuse (Fleury, 2000), such an assignment helps them comprehend the true value of inheritance.

4.3.2 Lesson 2: multilevel inheritance, polymorphism and overriding in objectKarel

The same didactic rationale described in the 1st lesson is used: (1) students are presented with problems that can not be solved efficiently with the already known concepts, come to a dead end, and consider as a natural consequence the adoption of new concepts; (2) the hands-on activities incorporated in objectKarel are used.

In contrast with the concepts presented in the first lesson, the concepts polymorphism and overriding cause difficulties in some students. Students do not find if difficult to understand that we can have methods with the same name in different classes as Fleury (2000) has found, but find it difficult to distinguish between polymorphism and overriding.

In order to help students clarify the taught concepts, the instructor must devise a series of carefully designed assignments with increasing complexity. The assignments must involve the definition of classes that implement a multilevel inheritance hierarchy taking advantage of polymorphism and overriding. UML class diagrams must be utilized. At the beginning students can be given the description of classes and their relations, and next they can be given just the description of a problem and analyse it on their own.

4.3.3 Lesson 3: class definitions

In the 3rd lesson a simple class definition in Java is presented using the environment of BlueJ. The class simulates a ticket machine for train tickets (project ticket-machine, chapter 2). Students use the visualization and direct manipulation techniques of BlueJ in order to create multiple objects with different ticket values and invoke their methods. The *inspect* function is presented and students are encouraged to use it in order to inspect objects' state during method invocation.

This lesson is extremely important for two reasons: (1) the interaction and direct manipulation techniques of BlueJ are presented and students must learn to use them correctly and not excessively; (2) the connection between the OOP concepts presented in objectKarel and their implementation in Java must be made.

Several difficulties were recorded the first year of teaching the course and were gradually dealt with, due to special didactical interventions. Some of these difficulties are characterized as "typical" difficulties, which are difficulties widely known to instructors and independent of the OOP paradigm: they refer to parameters, return types and values of methods. Other important difficulties that instructors must have in mind are the following (categorized accordingly to the classification presented in section 3.2):

Subcategory 2.1 - constructors:

- Several students have difficulty in defining multiple constructors in a class (Carter & Fowler, 1998), and define just one constructor or give the constructors wrong names.
- The use of `this` is not easy for students. However, in cases where its use in a constructor was needed, students did not shadow instance variables as recorded by (Truong, 2004), but instead chose to change the name of the parameter so as to avoid the use of `this`.
- Students face difficulties in initializing the fields in a constructor. The most common errors are: assigning the value of an undefined identifier and not the value of the parameter; assigning incorrect constant values.

Subcategory 2.3 – "set" methods (mutators)

- Declaring as return type of a "set" method the type of the parameter (instead of `void`).

Subcategory 2.4 – "get" methods (accessors)

- Some students access directly private fields instead of using a "get" method, even when this is not allowed (accessing a private field outside its class).

Subcategory 2.6 – access modifiers

- Some students do not apply their knowledge regarding access modifiers in the context of reading or writing code, even though they seem to understand their meaning. This leads to errors, such as accessing directly a private field outside its class.

Also, some difficulties and misconceptions that come to surface later, usually the fifth lesson when students use the main method for testing a class instead of the GUI features of BlueJ, stem from the wrong usage of BlueJ's features in the first lessons. The most important difficulties and misconceptions that must be taken into account by instructors are (Xinogalos et al., 2007):

Subcategory 2.2 – object instantiation:

- Some students do not declare the type of the variable that keeps a reference to the object they instantiate, as a side effect of using the BlueJ's pop-up menu for constructing objects. The instructor must emphasize from the very beginning that for each object constructed, a reference is kept in a variable of the appropriate type. All this information is provided in the dialog box used for naming the instance and initializing it and students must learn to use it consiously and not mechanically. The instructor should also present the statement in Java for accomplishing the same result.

Subcategory 2.5 – method calling:

- Several students call non-void methods as void methods. This behavior is attributed to BlueJ's misleading way of executing methods using the pop-up menu: invoking a non-void method results in showing the return value in a dialog box. The instructor must emphasize the fact that the value returned when calling a non-void method must be appropriately manipulated by the program, for example the value can be printed in the terminal using a `System.out.println` statement, used in a condition or an expression and so on. The misconception that "calling a non-void method results in printing automatically the returned value" must be avoided from the very beginning.

The instructor should also take into account the difficulties regarding the dynamic aspects of OOP, which are attributed by Ragonis & Ben-Ari (2005b) to the extended use of BlueJ's features.

In order for the students to comprehend the concepts presented in this lesson and face their difficulties, they are asked to extract information from the ticket machine class regarding its fields, constructors and methods (name, return type and value, parameters, role) and fill in tables using paper and pencil. Next, students can extend the class. One of the first extensions must be adding a second constructor to the class, in order to detect potential difficulties with multiple constructors. Finally, the instructor must make sure that students have comprehended the relation between the OOP concepts presented in objectKarel and their implementation in Java. This could be done by asking students to define in Java a class simulating the basic model of robots, leaving out – of course – graphical representation issues.

4.3.4 Lesson 4: object interaction

Students are presented with the concepts of abstraction, modularization and "object interaction", using an implementation of a digital clock display (project clock-display, chapter 3). The project consists of two classes: (1) the `NumberDisplay` class that represents

a two-digit number display incrementing by one and rolling over to zero when it reaches a given limit; (2) the `ClockDisplay` class that contains two `NumberDisplay` object fields, one for the hours and one for the minutes.
Students face various difficulties when the fields of a class are not of a primitive but of an object type (Xinogalos et al., 2006b):
Subcategory 2.2 – object instantiation:

- students are not sure where and how they should instantiate the objects that the fields will refer to, as well as
- how to manipulate this type of field

Subcategory 2.5 – method calling:

- the fact that they have to use dot notation for calling methods for a field, which is an object itself, seems to confuse them
- in a class where both internal and external method calls should be used, some students use in both cases an external method call.

In this lesson it is necessary to use object diagrams in order to help students form a conceptual model of an object that "includes" other objects. As a first activity, students study the implementation of the two classes used for the display of the digital clock and: fill in tables regarding the fields, constructors and methods of the classes; draw the object diagram at the moment that the a digital clock is created; write down the line-numbers in the source code of the `ClockDisplay` class where we have an internal and external method call (providing the name of the object too in the later case).
Next, students implement their first class from scratch. The class includes primitive type fields, two constructors, accessor and mutator methods. The definition of object type fields is avoided for the moment.

4.3.5 Lesson 5: the main method

The goal of this lesson is the presentation of the main method and execution without using the visualization and direct manipulation features of BlueJ. This topic is not adequately covered in the textbook of BlueJ and the instructor must prepare notes with the following content: instance vs. class/static methods, the syntax and role of the main method, byte code, execution of a Java application, Java Virtual Machine. An example of a class containing a static method is presented and the environment of BlueJ is used in order to make clear the difference between instance and static methods: students right-click on the class containing the static method and realize that they can invoke the static method, without creating an instance of the class first. Students implement, with guidance, a main method for a project they have already worked with, for example the ticket-machine project. They write code for accomplishing the same result they had previously accomplished by interacting with the environment of BlueJ. The difficulties that arise are many and some of them are related, as we have mentioned in section 4.3.3, to the way the features of BlueJ were used in the previous lesson. The professional programming environment of JCreator is presented to students. Students are guided through the process of creating a new project, adding existing java files of a BlueJ project, adding a new class defining a main method using the auto completion feature, navigating through the classes of the project and the elements of each class using their tree representation, compiling, debugging and executing the project with JCreator. BlueJ is still used for presenting new concepts with the use of projects, as well as for exploring projects interactively. However, students are free to choose

the environment that suits better to their needs and preferences when carrying out their assignments.

The difficulties that arise are many and some of them are related, as we have mentioned in section 4.3.3, to the way the features of BlueJ were used in the previous lesson. However, several difficulties which were attributed to the way the BlueJ environment was used in the first version of the course were not recorded in the following versions of the course. For example, "forgetting to declare the type of the variable that keeps a reference to the object being instantiated" was recorded only in the first version of the course (Xinogalos et al., 2006b; 2007). The most important difficulties that instructors must take into account and try to face are (Xinogalos et al., 2006b; 2007; 2009):

Subcategory 2.1, 2.2 – constructors, object instantiation:

- Some students confuse the definition of a constructor with its invocation, and use formal parameters instead of arguments when calling the constructor.
- Some students use as arguments in the invocation of a constructor the fields that are going to be updated.
- The type of object variables is missing.

Subcategory 2.3, 2.5 – "set" methods (mutators), method calling:

- The name of the field being updated is used as argument in "set" methods.

Category 1 – typical difficulties & Subcategory 2.5 – method calling:

- Missing arguments in method calls.
- Missing ( ) in methods without arguments.
- Calling a non-void method as void.
- A method is called without an instance.

Subcategory 2.4, 2.6 – "get" methods (accessors), access modifiers:

- Private fields are accessed directly outside their class instead of using a "get" method.

Special didactical situations must be devised by the instructor in order to help students face their difficulties. During the evolution of the course we found out that the following ones help:

- Students are provided with a presentation which utilizes animation in order to present processes that are either done automatically and "silently" by the BlueJ system - such as the declaration of an object variable during object instantiation - , or are overlooked – such as using the value returned by a non-void method. Screenshots of the dialog boxes that appear during the interaction of the user with the system are presented together with explanations of the processes that take place automatically and their form using Java syntax.
- Examples and assignments that are based on already known projects are used. Students are asked to write a main method for accomplishing the same result accomplished in previous lessons by interacting with the system of BlueJ.
- A program containing common errors is presented. Students are asked to track down the errors and then compile the program in order to check their answers. The corresponding error messages, which are not always comprehended by students, are presented and explained, so as to help students in utilizing error messages for tracking down and correcting the errors and not "blindly" changing the source code and waiting for the errors to disappear as if by magic.

4.3.6 Lesson 6: grouping objects in collections (ArrayList)

The use of collections for grouping objects is very common in object oriented applications, and even small-scale programs developed by undergraduate students. One of the most popular Java collections is the ArrayList collection, which represents a flexible-sized collection. The advantages of ArrayList collections are many and researchers state that it should be introduced first and emphasized over Arrays because it is *a "better structure for representing contiguous lists, both conceptually and in terms of implementation*" (Jacobson & Thornton, 2004) and "*it gives students better abstractions that are more general and will serve them in years to come*" (Ventura et al., 2004). So, the 6th lesson is devoted to the popular flexible size collection of ArrayList and the 7th lesson to the fixed size collection of array.

For the presentation of ArrayList collections a simple electronic notebook project is used (project notebook, chapter 4). Object diagrams are used for supporting students in understanding the structure of an ArrayList. Students perform various functions – add objects, remove objects, iterate the collection and print its objects' info - using the GUI of BlueJ, while they inspect the state of the ArrayList object using BlueJ's inspect function. Manipulating an ArrayList is not easy for students. Although students seem to comprehend the concept of object collections, they find it difficult to use flexible size collections for grouping objects (*Subcategory 2.7 – object collections*) (Xinogalos et al. 2006b, Xinogalos et al., 2008b):

- Students face difficulties in defining methods that return an ArrayList object and then calling it from main(): (i) other return types are used instead of ArrayList; (ii) the return statement is missing; (iii) the method is called from main as a void method.
- Some students do not use the built-in add method for adding objects to an ArrayList or use it incorrectly. For example, add is called for each field of the object being stored to the ArrayList separately and not for the object as an entity.
- Students face difficulties in iterating and retrieving the objects stored in an ArrayList: (i) a while loop is used but objects are not retrieved; (ii) the ArrayList is not iterated; (iii) the retrieved object is not assigned to a variable; (iv) students find it more difficult to use an iterator for iterating an ArrayList instead of an index and the built-in get method.
- Students access directly the private fields of the retrieved objects, sometimes even without an instance.
- Several students can not manipulate an ArrayList at all.

During the four years of teaching the course we have tested various activities for supporting students in dealing with their difficulties. The evaluation of the course led us to the following conclusions:

A first activity should focus on comprehending the structure of an ArrayList. As Kölling and Rosenberg (2001) state in one of their well-known guidelines for teaching object orientation with Java "*visualising class structure is crucial for students to develop an understanding of the important concepts*" (guideline 7, pp. 35). In correspondence to this statement, we believe that visualizing an ArrayList object structure is of great importance for students to comprehend the concept of ArrayList. Object diagrams seem to help students. Also, the visualization abilities of BlueJ seem to help some students, but these visualizations are not as effective for collections of objects (like ArrayLists) as for standalone objects, since they consist of object inspection windows that are not connected in any way. However, the students should not just "read" object diagrams. As Bergin (2000) states in one

of his pedagogical patterns "*Students can learn to read programs earlier than they can learn to write them. But, they should not be permitted to be overly passive in their reading*" (pattern "fill in the blanks"). So, we believe that students must not just "read" object diagrams but 'write' them too. In the context of their first activity students can be given a problem specification and the corresponding code (including a main method) and asked to draw in paper the object diagram. Students can also be asked to simulate basic ArrayList operations - such as iterating and processing their objects, adding new ones, removing objects – making the necessary changes to the object diagram and answering carefully designed multiple choice and open type questions. The correspondence of object diagrams with the visualizations provided by BlueJ should be presented and students should be encouraged to use them.
The next activity can be an exercise requiring filling in blanks. As Bergin (2000) states in his pedagogical pattern under the name "fill in the blanks": "*Students can often learn a complex topic by building several small parts of a larger artifact*". This pedagogical pattern has been adopted by Kölling and Rosenberg (2001) too as a guideline under the name "Don't start with a blank screen" (guideline 2). However, in our case, we believe that the "blanks" should not represent whole methods but specific error prone elements in an excerpt of code. A paper with the problem specification and the source code with blanks can be given to students in order to fill it in. When interviewed after an activity like this, students stated that the activity helped them "*track the points that cause them difficulty and make specific questions*", "*focus on issues that cause them great difficulty*" and "*manage to write some source code that otherwise could not be accomplished*" (Xinogalos et al., 2008b).
Next, students can implement programs from scratch.

4.3.7 Lesson 7: grouping objects in collections (ArrayList vs. Array)

This lesson is devoted to array collections and their comparison with ArrayList collections. In order to present arrays and their differences with ArrayLists we have implemented the notebook project of the previous lesson using arrays. This way, students comprehend better the similarities and differences of the two structures. Students are assigned a project that is implemented using both ArrayLists and arrays.

4.3.8 Lesson 8: using class libraries

In this lesson students are introduced to reading documentation and using library classes in general, as well as specific library classes in particular. This is accomplished in the context of exploring and improving a primitive implementation of an Eliza-like dialog program based on text used to provide technical support to customers of a software company (tech-support project, chapter 5). Students learn to read class documentation and are shown how various classes are used for improving the technical support system: the `String` class is used for improving the processing of user's input (ignoring letters' case, removing spaces and so on in order to process users' requests and respond); the `Random` class is used for adding random behavior in the system's responses; the `HashMap` class is used for associating keywords present in users' questions with related responses; the `HashSet` class is used for tokenizing users' input to a set of words. BlueJ's ability to generate the interface of a class (in the editor window) is presented and students are encouraged to use it for exploring the classes of the projects incorporated in BlueJ. Also, the way that javadoc and basic key symbols, present in all BlueJ's projects, are briefly presented. However, due to time

limitations students, although encouraged, are not asked to write class documentation.The first lab activity is about locating the `String` class in Java' standard class library, study its documentation and track down specific information: define the role of specific methods; search for methods that carry out specific tasks; recognize the information provided by the signature of methods, such as their return type and parameters. Finally, students use the methods of the class for implementing specific tasks that represent fundamental functions on strings. Next, students develop projects using class libraries. For example, a simple phone book is implemented utilizing the `HashMap` class.
Students comprehend easily the structure of Java's standard class library, the concept of packages, the way the documentation is read and used. The only obstacles are the language and the terminology used for describing methods. Some times students guess the role of a method from its name – after all students realize that using meaningful identifier names is important!

4.3.9 Lesson 9: designing classes

Presenting principles of good class design or/and recognizing bad class design is not an easy task. Just presenting theoretically the relevant concepts will not have a great impact on students' knowledge on designing classes. It must be clarified that an application that performs the intended task is not necessarily well-designed. Problems come to surface when the application has to be extended. An extension that would require trivial effort in a well-designed application might require extended changes in a badly-designed application. The best way to present this to students is to use an application with badly-designed classes that from the point of users works perfectly. Trying to make small extensions will bring to surface the underlying problems. An excellent project for this purpose is the world-of-zuul project (chapter 7) of BlueJ's textbook. This project is an interactive, text-based adventure game, which is highly extendable. The most interesting extension for students is providing a GUI for the game, a topic that is not covered in the course. Various extensions for the game are considered that give the chance to present clearly several concepts regarding class design: code duplication, responsibility-driven design, coupling, cohesion and refactoring. Students implement at the lab some of the extensions discussed in the lecture for the world-of-zuul game. Unfortunately, it is difficult for students to apply all the principles of good class design presented to them, since this means that a project of a considerable size must be implemented. Taking into account the time limitations of the course this is not easy to be accomplished. However, students should definitely be assigned a project that involves designing the classes on their own utilizing UML class diagrams.

4.3.10 Lesson 10: improving structure with inheritance

Inheritance is one of the most fundamental OOP concepts and is presented from the first lesson with the use of objectKarel. In the 10th lesson inheritance is revisited more formally using Java for implementing it. The 4th year of teaching the course we were pleasantly surprised to see how easy students comprehended the concept of inheritance. The project used for presenting the concepts related to inheritance is a database of CDs and DVDs (dome project, chapter 8). The project consists of three classes: CD, DVD and Database containing two ArrayList fields for grouping CDs and DVDs. In its first version the project does not use inheritance. Students easily realize the existence of duplicate code in all three

classes. When asked how the project can be improved students propose the use of inheritance, which was clearly presented in objectKarel. The way that inheritance is implemented in Java is presented and a new version of the project that uses inheritance is presented and discussed. Polymorphism is also introduced with the definition of a print method in both subclasses, and the superclass for printing the values each class's fields. When students execute the program they are confronted with a problem: just the values of each class's fields are printed and not the ones inherited. This problem is faced in the next lesson.

Using objectKarel for introducing students to the fundamental OOP concepts provides great help to students. Some of the difficulties recorded in the previous years were not recorded in the 4th year of teaching the course, while others were reduced. However, the most common difficulties that instructors must have in mind, in order to deal with them, are (Xinogalos et al., 2006b):

Subcategory 2.1, 2.8 – constructors, inheritance:

- In the constructor of a subclass students do not invoke the constructor of the superclass (Truong & Bancroft, 2004) and as a consequence they do not use parameters and/or assign values to the attributes/fields inherited from the superclass.

Subcategory 2.8 – inheritance:

- In cases where multilevel inheritance is implemented, students - when asked to define the messages that an object of a subclass responds to – go up only one level at the class hierarchy.
- Some students face difficulties with subtyping. We observed that when students are confronted with a statement that declares a variable and stores a reference to an object, most of them recall the rules of subtyping and give correct answers to relevant questions. On the other hand, when students are confronted with the declaration of a variable and asked what types of objects can be referenced by this variable, several students do not think about subtyping and mention only the type of the class.
- Some students fail to realize the way that multiple inheritance is implemented in Java and believe that each concrete class can extend directly more than one class.

An activity that helps students comprehend what happens when the constructor of a subclass is executed is using the debugger and executing the project presented in the lecture step by step. This helps student understand: how the constructor of the subclass invokes the constructor of the superclass; how are the inherited fields initialized; that they have to provide in the subclass constructor invocation arguments for the superclass constructor too. This activity also prepares students for the introduction of some concepts introduced at the next lesson: static and dynamic type, dynamic method lookup.

4.3.11 Lesson 11: polymorphism and overriding

Having as a starting point the problem with the print method in the project of the previous lesson and the observations made during the step-by-step execution activity described above, the concepts of static/dynamic type and dynamic method lookup are introduced. It is clarified that the static type is used during compilation and the dynamic type during execution for dynamic method lookup. Students having in mind the concept of overriding presented at the second lesson with objectKarel comprehend without difficulty that the problem is solved by calling in the print method of the subclasses the overridden print method of the superclass. What is different in comparison with objectKarel is the syntax for

calling an overridden method in subclasses. With the help of objectKarel polymorphism and overriding are more easily comprehended by novices. The difficulties recorded in the first version of the course were significantly reduced:
Subcategory 2.8 – inheritance:

- In a "set" method of a subclass that overrides a "set" method of a superclass with the purpose of assigning values to all fields (including those inherited), students omit the call to the overridden "set" method of the superclass and just assign values to the fields of the subclass.
- Students include in a subclass method a call to the overridden method of the superclass, but they do not use the corresponding parameters in the overriding method.
- Some students can not differentiate between polymorphism and overriding.

4.3.12 Lesson 12: abstract classes and interfaces

Abstract classes and interfaces are presented using a classic predator-pray simulation (foxes-and-rabbits, chapter 10). The initial version of the project does not take advantage of inheritance. The project consists of several classes, but conversation focuses on three classes: the `Fox` class used to represent foxes; the `Rabbit` class used to represent rabbits; and the `Simulator` class which controls the simulation. This project is used not just for presenting the new concepts, but for reinforcing the already taught inheritance concepts and good class design principles as well. Students are guided through the process of creating an `Animal` superclass extended by the subclasses `Fox` and `Rabbit`; using one collection for grouping both kinds of the simulation's actors (foxes and rabbits) instead of two; substituting the `hunt` method of foxes and the `run` method of rabbits with a polymorphic `act` method implementing the basic behavior of each actor in the corresponding subclass; presenting the need to add an abstract `act` method to the `Animal` superclass in order for the project to compile (`act` is called for the objects-actors grouped in a collection and retrieved in a variable with static type `Animal`); an `Actor` interface is added so as to make possible the usage of other kinds of actors besides animals.
The evaluation of the course has shown that students do not find it difficult to implement abstract classes and interfaces, but instead to understand their true meaning and when they should declare a class as concrete, abstract or interface. Next, we present some misconceptions that were recorded in the first course (Xinogalos et al., 2006b):

- Some students have the misconception that they can create objects not only from concrete classes, but from abstract classes and interfaces too.
- Some students believe that a class can implement just one interface.
- Some students believe that it is not necessary for a concrete class to implement an abstract method declared in its abstract superclass.
- Some students believe that an abstract class declares only abstract methods, while others believe that both abstract classes and interfaces declare only abstract methods.

5. Conclusions

The long-term evaluation and reformation of the "Object Oriented Design and Programming" course presented in this chapter has established some guidelines that can be utilized by researchers and instructors for designing and teaching an effective course. These

guidelines do not refer, as usually, to specific aspects of the course. The guidelines presented cover all the aspects of the course: teaching approaches utilized; sequence of concepts and organization of lessons; usage of the well-designed projects accompanying BlueJ and its textbook; programming environments used, their role in achieving the goals of the course and potential pitfalls; potential difficulties and misconceptions; didactical interventions for dealing with recorded difficulties.

Besides the specific guidelines for designing and teaching an effective OO Design and Programming course, the four-year evaluation and reformation of the course has also drawn some more general conclusions regarding course design and effective teaching of demanding subjects:

- *Continuous evaluation of teachings is necessary in order to reach valid conclusions and make improvements*. Although, the teaching approach adopted in the first version of the course was not an empirical one, it was our belief that it should definitely be evaluated by recording students' achievements and studying their conceptions. The decision to take on a long-term evaluation of the course proved to be right. This evaluation gave us invaluable insights into students' difficulties and helped us reform the course with increasingly positive results the last four years.
- *More than one tool is needed in order to support demanding cognitive areas, such as programming*. The usage of multiple tools for supporting the teaching of cognitively demanding subjects must not be avoided. In the OOP course described the use of three distinct but complementary programming environments helped in achieving better results without causing any cognitive overload.

With the belief that continuous evaluation of teachings is necessary in order to reach valid conclusions and make improvements we intend to continue the long-term evaluation of the course. The results of this never-ending evaluation along with the results of studies that are made available to the teaching community, the gathered experience and the new technology enhanced tools that come to surface will support us in developing even more appropriate educational material (i.e. examples, activities, assignments) and move to reformations for improving the teaching of OO Design and Programming. Our main goal for improving the course focuses on devising new material for supporting students in deeply comprehending the principles of good class design and the techniques for improving an application's structure and designing truly OO applications.

6. References

Allen, E.; Cartwright R., Stoler B. (2002). DrJava: a lightweight pedagogic environment for Java, ACM SIGCSE Bulletin, Vol. 34, Issue 1, pp. 137-141.

Barnes, D. & Kölling, M. (2004). Objects First with Java: A practical introduction using BlueJ, Prentice Hall/Pearson Education, Harlow, England.

Barnes, D. & Kölling, M. (2006). Objects First with Java: A practical introduction using BlueJ, 3rd edition, Prentice Hall /Pearson Education, Harlow, England.

Bennedsen, J. & Caspersen, M. (2004). Programming in Context – A Model-First Approach to CS1, Proceedings of SIGCSE '04, pp. 477-481.

Bergin, J.; Stehlik, M.; Roberts, J. & Pattis, R. (1997). Karel++ - A Gentle Introduction to the Art of Object-Oriented Programming, John Wiley & Sons.

Bergin, J. Fourteen pedagogical patterns, http://csis.pace.edu/~bergin/PedPat1.3.html (Accessed June 2008).

Brusilovsky, P.; Calabrese, E.; Hvorecky, E.; Kouchnirenko, A. & Miller, P. (1997). Mini-languages: A Way to Learn Programming Principles, Education and Information Technologies, 2(1), pp. 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), pp. 16-20.

Carter, J. & Fowler, A. (1998). Object Oriented Students? SIGCSE Bulletin, Vol. 28, No. 3, 271.

Cooper, S. ; Dann, W. & Paush, R. (2000). Alice: A 3-D Tool for Introductory Programming Concepts, Journal of Computing in Small Colleges, 15(5), pp. 108-117.

Chang, C.; Denning, P.J.; Cross, J.H.; Engel, G.; Roberts, E.; Shackelford, R.; Sloan, R.; Carver, D.; Eckhouse, R.; King, W.; Lau, F.; Mengel, S.; Srimani, P.; Austing, R.; Cover, C.F.; Davies, G.; McGettric, A.; Schneider, C. Michael & Wolz, U. (2001). Computing Curricula 2001. ACM Journal of Educational Resources in Computing, 1(3), Article #1, 240 pages.

Fleury, A. (2001). Encapsulation and reuse as viewed by java students, ACM SIGCSE Bulletin, Vol. 33, Issue 1, pp. 189-193.

Fleury, A. (2000). Programming in Java: student-constructed rules, ACM SIGCSE Bulletin, Vol. 32, Issue 1, pp. 197-201.

Georgantaki, S. & Retalis, S. (2007). Using Educational Tools for Teaching Object Oriented Design and Programming, Journal of Information Technology Impact, Vol. 7, Number 2, pp. 111-130.

Holland, S.; Griffiths, R. & Woodman, M. (1997). Avoiding object misconceptions, ACM SIGCSE Bulletin, Vol. 29, No. 1, pp. 131-134.

Jacobson, N. & Thornton, A. (2004). It is Time to Emphasize ArrayLists over Arrays in Java-Based First Programming Courses, ACM SIGCSE Bulletin, 36(4), pp. 88-92.

Kölling, M. & Rosenberg, J. (2001). Guidelines for Teaching Object Orientation with Java, ACM SIGCSE Bulletin, Vol. 33 Issue 3, pp. 33-36.

Kölling, M.; Quig, B.; Patterson, A. & Rosenberg, J. (2003). The BlueJ system and its pedagogy, Journal of Computer Science Education, 13(4), pp. 249-268.

Nevison, C. & Wells, B. (2003). Teaching Objects Early and Design Patterns in Java Using Case Studies, ACM SIGCSE Bulletin, 35(3), pp. 94-98.

Pattis, R. E.; Roberts, J. & Stehlik, M. (1995). Karel - The Robot, A Gentle Introduction to the Art of Programming, 2nd edn., New York: John Wiley & Sons.

Proulx, V. ; Raab, R. & Rasala, R. (2002). Objects from the Beginning – With GUIs, ACM SIGCSE Bulletin, 34(3), pp. 65-69.

Ragonis, N. & Ben-Ari, M. (2005a). A long-Term Investigation of the Comprehension of OOP Concepts by Novices, Computer Science Education, Vol. 15, No. 3, pp. 203-221.

Ragonis, N. & Ben-Ari, M. (2005b). On Understanding the Statics and Dynamics of Object-Oriented Programs, ACM SIGCSE Bulletin, 37(1), pp. 226-230.

Sanders, D. & Dorn, B. (2003). Jeroo: A Tool for Introducing Object-Oriented Programming, ACM SIGCSE Bulletin, 35 (1), pp.201-204.

Topor, R. Common (Java) programming errors.
http://www.cit.gu.edu.au/~rwt/p2.02.1/errors.html. (last access May 2006).

Truong, N. & P. Roe P. Bancroft (2004). Static Analysis of Students Java Programs, Proceedings of the 6th Australian Computing Education Conference, pp. 317-325.

Ventura, P. ; Egert, C. & Decker, A. (2004). Ancestor Worship in CS1: On the Primacy of Arrays, Proceedings of the OOPSLA '04 Conference, pp. 68-72.

Xinogalos, S.; Satratzemi, M. & Dagdilelis, V. (2006a). An introduction to object-oriented programming with a didactic microworld: objectKarel, Computers & Education, Vol. 47, Issue 2, September 2006, pp. 148-171.

Xinogalos, S.; Sartatzemi, M. & Dagdilelis, V. (2006b). Studying Students' Difficulties in an OOP Course based on BlueJ, Proceedings of IASTED International Conference on Computers and Advanced Technology in Education, pp. 82-87, October 2006, Lima, Peru, Acta Press.

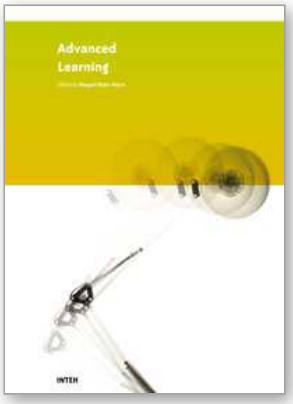
Xinogalos, S.; Satratzemi, M.; Dagdilelis, V. & Evangelidis, G. (2007). Re-designing an OOP based on BlueJ, Proceedings of the 7th IEEE International Conference on Advanced Learning Technologies, pp. 660-664, July 2007, Niigata, Japan, IEEE Computer Society Press.

Xinogalos, S. (2008a). Studying Students' Conceptual Grasp of OOP Concepts in Two Interactive Programming Environments, Springer Communications in Computer and Information Science, Vol. 19, pp. 578-585.

Xinogalos, S.; Satratzemi, M. & Dagdilelis, V. (2008b). An analysis of students' difficulties with ArrayList object collections and proposals for supporting the learning process, Proceedings of the 8th IEEE International Conference on Advanced Learning Technologies, pp. 180-182, July 2008, Santander, Cantabria, Spain, IEEE Computer Society Press.

Xinogalos, S. & Satratzemi, M. (2009). A Long-Term Evaluation and Reformation of an Object Oriented Design and Programming Course, Proceedings of the 9th IEEE International Conference on Advanced Learning Technologies, pp. 64-66, July 2009, Riga, Latvia, IEEE Computer Society Press.

INTECH



Advanced Learning

Edited by Raquel Hijn-Neira

ISBN 978-953-307-010-0
Hard cover, 444 pages
Publisher InTech
Published online 01, October, 2009
Published in print edition October, 2009

The education industry has obviously been influenced by the Internet revolution. Teaching and learning methods have changed significantly since the coming of the Web and it is very likely they will keep evolving many years to come thanks to it. A good example of this changing reality is the spectacular development of e-Learning. In a more particular way, the Web 2.0 has offered to the teaching industry a set of tools and practices that are modifying the learning systems and knowledge transmission methods. Teachers and students can use these tools in a variety of ways aimed to the general purpose of promoting collaborative work. The editor would like to thank the authors, who have committed so much effort to the publication of this work. She is sure that this volume will certainly be of great help for students, teachers and researchers. This was, at least, the main aim of the authors.

How to reference
In order to correctly reference this scholarly work, feel free to copy and paste the following:

Stelios Xinogalos (2009). Guidelines for Designing and Teaching an Effective Object-Oriented Design and Programming Course, Advanced Learning, Raquel Hijn-Neira (Ed.), ISBN: 978-953-307-010-0, InTech, Available from: http://www.intechopen.com/books/advanced-learning/guidelines-for-designing-and-teaching-an-effective-object-oriented-design-and-programming-course

INTECH
open science | open minds

InTech Europe
University Campus STeP Ri
Slavka Krautzeka 83/A
51000 Rijeka, Croatia
Phone: +385 (51) 770 447
Fax: +385 (51) 686 166
www.intechopen.com

InTech China
Unit 405, Office Block, Hotel Equatorial Shanghai
No.65, Yan An Road (West), Shanghai, 200040, China
中国上海市延安西路65号上海国际贵都大饭店办公楼405单元
Phone: +86-21-62489820
Fax: +86-21-62489821