A Sequence of Assignments to Teach Object-Oriented Programming: a Constructivism Design-First Approach

Article *in* Informatics in Education · April 2003

DOI: 10.15388/infedu.2003.09 · Source: DBLP

CITATIONS
6

READS
1,801

1 author:

Kleanthis Thramboulidis
University of Patras
144 PUBLICATIONS **2,194** CITATIONS

SEE PROFILE

Informatics in Education, 2003, Vol. 2, No. 1, 103–122

A Sequence of Assignments to Teach Object-Oriented Programming: a Constructivism Design-First Approach

Kleanthis C. THRAMBOULIDIS
Electrical & Computer Engineering
University of Patras, Greece
e-mail: thrambo@ee.upatras.gr

Received: March 2003

Abstract. A constructivism-based approach to teach the object-oriented (OO) programming paradigm in introductory computer courses was developed and used for several years. A multi-entity system from every-day life was adopted, to exploit the novice programmer's existing knowledge and build on it the OO conceptual framework. A sequence of assignments has been designed and developed to allow students exposed to this approach to experiment with Java programming and see how the OO conceptual framework is implemented. In this paper, this sequence of assignments is presented, discussed and evaluated in the context of the defined approach. The set of assignments that is based on a software-engineering-centered view and more precisely on a design-first approach, comes with the description of the strategy and graded hints that lead students to the final solution. Although it was first implemented as supplementary material, it quickly became the core component of the course.

Key words: teaching OO programming, constructivism-based approach, design-first, informal use-cases, Java assignments, assignment-based learning.

1. Introduction

Object-Oriented (OO) programming has become a critical subject in most computer science curricula. However, as reported by many educators (Eckstein, 1997; Sheetz *et al.*, 1997; Kolling, 1999a; Borstler and Fernandez, 1999; Demuth *et al.*, 2000; Borstler *et al.*, 2002), teaching object-oriented programming is still difficult while the approach used for procedural programming does not work well. A "new pedagogy to teach objects well" is required, as reported by Bergin (2000).

In an attempt to address this problem and improve the effectiveness of our OO course, issues in science education, mainly constructivism, which is one of the fundamental ideas in education, have been considered (Glasersfeld, 1989; Confrey, 1990; Greenco *et al.*, 1996). Motivated by constructivism the "Goody's example" was devised and used during the first segment of the course, to guide our students in exploiting their prior knowledge emanating from real-life and building on it, the conceptual framework of the OO paradigm (Thramboulidis, 1998). The "Goody's example" is based on "Goody's",

Greece's most popular fast-food restaurant chain. All our students are already familiar with "Goody's" from every-day life. They have all used its services and have an understanding of its structure and functioning. The use of concepts from the "Goody's example" had a positive impact on students' ability to learn, since they understood that they were familiar with the basic concepts that constitute the OO approach. Later, it was found that an informal use of use-cases, class diagrams, and object interaction diagrams (OIDs) facilitates students in exploiting their real-world knowledge system and more effectively building on it, the conceptual framework of the OO paradigm. The results of these findings were utilized for the development of a design-first approach that has been successfully practiced for the last three years.

To address the new direction taken by the course, a major re-writing of examples, exercises, and laboratory assignments was required. A new set of assignments was designed and developed so as to provide the context in which the knowledge and facts acquired through the first segment of the course should be immediately used and understood. Although the set of assignments was developed to complement the practice sessions of the course, students found the assignments with the accompanying material very useful and more productive than lectures or even laboratory sessions. These findings are in conformity with those of Madden and Chambers (2002) who argue that respondents have a clear preference for learning through the medium of assignments and tutorials rather than lectures. Moreover Duke (2000) argues that problem-based learning is a core idea for the mastering of computer language. In response to these findings a lot of time and work was devoted to improving the assignments and making them the core component of the course. However, the development of the assignments was not the only thing that had to be done. Our own teaching strategy that required a long-term effort in several directions had to be created. The following directions as are reported by Hadjerrouit (1999) had to be addressed: (a) improve our understanding of students' prior knowledge, (b) refine the program of activities, (c) explore problem-solving skills, and (d) develop material and teaching aids to support constructivist learning. The development of better assessment and evaluation procedures are being worked on.

Many researchers have been influenced by constructivism in teaching OO (Fleury, 2000; Hadjerouit, 1999; Gray, 1998; Jones, 2002). For example, Hadjerouit (1999) describes a pedagogical framework rooted in the constructivist epistemology for teaching OO design and programming, and argues that traditional approaches to teaching OO design and programming do not focus on building strong links between the three main types of knowledge that are relevant to the OO approach. Fleury (2000) claims that constructivism suggests that designing appropriate laboratory experiences for the students is likely to be more effective than just talking to them. However, to our knowledge, none of the researchers: (a) adopts a design-first approach with use of informal UML diagrams, (b) exploits to the extent we do, the existing knowledge of students emanated from a real-life system, and (c) provides a coherent sequence of assignments strictly adapted to the design-first approach.

The remainder of this paper is organized as follows. In Section 2, the two major refinements of the course, are briefly described. The approaches used during this time to

teach the OO paradigm are reffered to and the main outline of the updated course is presented. In Section 3, the layout of the development methodology used in the context of the proposed approach, is given. In Section 4, the assignment sequence is described; each assignment is briefly described and its most interesting material is presented. Finally, the approach is evaluated and discussed and the paper is concluded.

2. Background Work: Moving to a Constructivism Design-First Approach

A course in OO programming for students already familiar with the procedural programming paradigm was developed. The course has been developed over the past 6 years and it has gone through three major phases of development (Thramboulidis, 2003a). In each phase, a different approach was used and the feedback received from students was assessed and utilized to improve the students' understanding of the fundamental OO concepts. In this section, these approaches and mainly the approach of the third phase are briefly referred to; the outline of the updated course is also presented.

The Traditional Approach

During the first 2 years that constitute the first phase of the course development, the traditional approach that is adopted by the majority of textbooks on OO programming was followed. This approach, which is the same as that used in the preceding procedural programming course, is based on the von Neumann serial computation model. During this time, it was found that students had persistent difficulties mainly resulting from the already known procedural paradigm. Students tried to build the concept maps of the new paradigm on the knowledge system constructed during the pre-existing procedural paradigm. The commonly referred to (Eckstein, 1997; Sheetz *et al.*, 1997) and now well-known problem of paradigm shift had to be faced. As demonstrated by an empirical case study performed by Bergin and Winder (2000) the OO is a real paradigm shift rather than just a packaging mechanism for procedural programming and this is why a change in the mental model of the programmers is required. Furthermore, even in the case where the OO paradigm is the first programming paradigm, many educators feel that is still hard to teach OO, since the approach used for procedural programming is not effective.

Motivated by the results of McCloskey (1983), Schoenfeld *et al.* (1993) and Smith (1993), and in an attempt to exploit the benefits of constructivism, which stresses the importance of prior knowledge on which new knowledge is built, it was decided to look for this prior knowledge. It was found that students already had this knowledge taken from real-life experience. The "Goody's example" was devised and used through the first segment of the course to introduce the conceptual framework of the OO paradigm (Thramboulidis, 1998).

The Object-First Approach

During the first major refinement we were guided to an object-first approach. Numerous researchers (Arnow and Weiss, 1998; Bruce *et al.*, 2001) have adopted the object-first approach and reported encouraging results. To improve the effectiveness of our

object-first approach, the following were adopted: (a) a restricted use of UML class diagrams, and (b) an extensive use of BlueJ (Kolling, 1999b). However, during the teaching in the second phase of course implementation, it was discovered that the problem of paradigm shift did not disappear. Later on, it was found that a brief introduction of use-cases and object interaction or scenario diagrams, helped students to exploit their real-life experience and to use it to build the knowledge system of the OO paradigm. It was decided to introduce, during the first segment of the course, the above analysis and design artefacts through the "Goody's example", and utilize them to create draft models for the systems of our examples for the remainder of the course.

A Constructivism Design-First Approach

After the above modifications, it was found that the introduction of the new course had nothing in common with the preceding procedural programming course. Our students started thinking in ways different from the von Neumann serial computation model. They were encouraged in this direction since "it exists in the world at large and is better matched to the requirements imposed by the majority of today's modern applications" as is reported by Stein (1998). Moreover, we were guided by this discovery to making a shift in focus from the algorithm-centered view to the software-engineering-centered view and more precisely to the design-first approach. Students are required to build communities of interacting entities just like the "Olga Square Goody's", a well known to our students instance of Goody's. During this process the main tasks of the programmer are, just like the creator of the "Olga Square Goody's", to define: a) ways in which the community entities interact and b) the responsibilities as well as the internals of each entity of the community.

Finally, it was found that although exception handling and concurrency were considered to be new terms for students, their basic concepts exist in "Goody's example" and are well understood by our students. It was decided to expand our course in this direction with an introduction to both topics. The selection of Java proved successful for this direction too.

The Final Course Outline

Since the focus of the course is on teaching the basic concepts of the OO programming paradigm rather than a specific programming language, the first segment of the updated course addresses the conceptual framework of the OO paradigm. The "Goody's example" that is used during this segment, clarifies to our students that they are already familiar with the basic concepts of the new paradigm and that this experience comes from every-day life. Simplified versions of use-cases, class diagrams, and OIDs in a very informal UML notation, are used to create draft models to highlight the structure and behaviour of the system. To help students create a conceptual framework independent of any particular programming language, the above material is taught without reference to any programming language.

After having introduced the conceptual framework, the developer's expectations from an environment that should allow the construction of software systems according to the OO approach are defined in classroom in co-operation with students. Adopting a "Lego

construction" approach, students are asked to first focus on the basics of integrating existing components and later on building new ones. A sample list of developer's expectations resulting from such an approach is given in (Thramboulidis, 2003a).

This is the best time to start the second segment of the course in which the basic constructs of the selected OO language that are necessary to satisfy the above requirements of the system developer are focused on. Since the programming language is considered as the medium through which the concepts are formally presented and learned through practice, any modern OO programming language can be used. Java, which has increasingly become the language of choice for teaching the OO programming paradigm to beginners was selected; see (Thramboulidis, 2003a) for a list of reasons. Tyma (1998) claims that Java "takes the best ideas developed over the past 10 years and incorporates them into one powerful and highly supported language". Even more a wide-range survey conducted by Madden and Chambers (2002) has shown that the majority of respondents found Java easier to learn and program than C, the most widely used language in introductory programming courses until recently.

Classes, instances, methods, constructors, means of utilizing the standard Java Library, data variables, and inheritance are covered. Since students are already familiar with C data types, operators, control statements, and functions, less than two hours are spent on covering Java-related issues. However, more time should be dedicated if students have not been exposed to C programming. Overwhelming students in lectures with Java's idiosyncratic features is best avoided and only the most basic elements, just enough to enable them to do the assignments are taught. The Reverse Polish Notation (RPN) calculator was selected as a case study for this segment and each student was required to implement their own calculator following a well-defined step-by-step development process, which runs parallel with the introduction of the language constructs. Students see example programs in lectures, have to work with the assignments in lab, or in their own time, and complete their own calculator before the end of the course. They are allowed to use JDK, BlueJ or other development environment. Some students work on projects. Although it is believed that the project must be an important component of the course, we are compelled by university's policies to consider it elective.

Finally, the remainder of the course addresses the topics of exception handling and concurrency. The "Goody's example" is utilized once again to create the basic conceptual knowledge for both topics (Thramboulidis, 2003b).

3. Our Design Methodology

Our approach has resulted in the definition of a methodology that is divided into 6 steps. Each step is actually a progress towards teaching and applying the main OO concepts. To make students understand and apply these concepts, the methodology attempts to fully explore the pre-existing knowledge of students, which was emanated from the "Goody's example", and use it to construct the OO conceptual model. The layout of the design methodology is as follows:

1. Identify the services provided by the system (use cases).
2. For each use-case:
   a. write a paragraph describing the interaction between the user and the system for the service to be accomplished;
   b. draw an OID to represent the described interaction.
3. For each use-case:
   a. try to identify the objects that the system must contain so as to satisfy the specific use-case description. If there are any objects that cannot be directly represented using existing types, (basic Java library, already defined types) define a class to specify the structure and behaviour of these objects;
   b. draw a detailed OID to show the objects of the system and the way they collaborate, to satisfy the specific use case description. This OID must be in compliance with the corresponding first level OID.
4. For each OID of the previous step identify the contribution of each object and define the corresponding methods.
5. Draw a simple UML-like class diagram to show the classes of the system and their relationships (aggregation and generalization/specialization).
6. For each class of the class diagram:
   a. give an implementation using Java;
   b. test the behaviour of the implemented object; and
   c. integrate the object into the program.

4. The Sequence of Assignments

The proposed assignments are used during the second segment of the course, which focuses on mapping the concepts of the OO framework into a formal programming language. This is what Harel (1991) defines as constructionism. “Constructionism is a synthesis of the constructivist theory of development psychology and the opportunities offered by technology to base education for science and mathematics on activities in which students work towards the construction of an intelligible entity rather than on the acquisition of knowledge and facts without a context in which they can be immediately used and understood”.

The RPN calculator was selected since our students knew it from the corresponding example of Kernighan (1988), which they have been taught during the preceding procedural programming course. However, this is not a prerequisite since the problem can be explained in relatively little class time. The use of a system that students have already developed with the procedural approach allows us to better highlight the differences between the two paradigms as well as to emphasize the strength of the OO paradigm.

Students have to follow a set of 12 assignments that guide them to proceed step-by-step in the development of their own GUI calculator. Each assignment builds on the

results of the previous one. Students are informed to proceed in the next assignment only after they have completed the current one. The assignment sequence was designed not only to illustrate concepts, but also to suggest ideas for investigation. For the assignment sequence to provide pleasure and satisfaction to our students it was decided to use graphical user interface from the very beginning. Even more, since a problem is incomplete without a solution, the assignments come with the description of the strategy and graded hints leading to the final solution. Hints are designed not only to help the student who is having difficulties, but also to offer alternative suggestions and discuss issues that arise. Another issue that was addressed during the development of the assignments is the fact that the audience may span from the complete novice, to programmers experienced in other languages, mainly procedural. Their individual needs for undertaking the assignments vary significantly from novices wishing a well defined step-by-step process with a lot of supporting material for each step (in some cases an indicative solution to the specific step must be given), to experienced programmers who may require only some hints and a good reference in order to proceed. An attempt to offer support for these two extremes together with various intermediate levels of experiences and skills, presented great difficulty in the design process of the assignment sequence and required a lot of major revision. However, the results are very encouraging since this approach allows the high achievers to experiment and learn independently using the vast amount of on-line material available for Java, without overloading or distracting the weaker students.

1st Assignment: Informal Use-Cases and Object Interaction Diagrams

After having completed the conceptual framework of the OO approach students are impelled to interact with a program that is implemented according to the OO approach. An RPN calculator program with a GUI similar to the one of Microsoft Windows OS is given to students to understand the calculator's behaviour. They are asked to interact with the RPN calculator using the reverse polish notation to find the value of the expression $24 - 12 * 14 - 3$. The purpose of this assignment is to convey the basic concepts introduced by the use of the "Goody's example" to a system that is going to be implemented with the OO approach. Students are asked to describe in every day language, as they did in the "Goody's example", the interaction between the user and the program in order to evaluate the above expression (i.e., to get the service). They are asked to identify objects that compose the RPN calculator and expand the above description with appropriate reference to these objects. They next have to graphically represent the above interaction by drawing an OID like the one they have drawn in "Goody's example". Through this process students are guided to identify the main objects that compose the RPN calculator and create a draft class diagram using the UML notation for class and aggregation. Students construct a class diagram like the one given in Fig. 1. The produced diagram is refitted at the end of this assignment and is discussed in depth so as to form the basis for the subsequent assignments.

During this assignment, program development is mainly guided by the OO conceptual framework and not by the way in which the specific language implements the OO approach. As Hadjerrouit argues in (1999), building the OO knowledge structures that are strongly linked to each other requires problem solving at a higher level than the code level. This is a message we want to pass to our students, through this assignment.

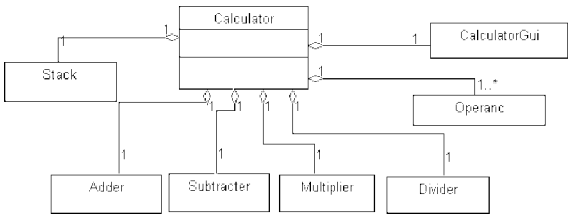

Fig. 1. Draft class diagram of the reverse Polish notation calculator.

2nd Assignment: Lego Construction Approach

With this assignment students are gradually moved to the implementation. Emphasis is given to the Lego-construction approach and students extend and refine their understanding on the topic of integrating pre-defined components. With this approach, students are able to make use of existing classes from the beginning, to create relatively sophisticated applications. Moreover by adopting the Lego-construction approach impressive graphical applications can be coded from the very beginning, since there is no need for students to know the details of the classes being used. It is only required to know the messages that can be sent to a class or instance and the way that the class or instance will respond to them. A set of classes has been specifically developed to support the Lego-construction approach. This approach enables some of the more difficult aspects of Java such as input, event handling and exception handling to be postponed until late in the course. To use `MyInput` class for example the following information is given:

```
public class MyInput {
  public static StringBuffer getString(String prompt) {...}
  public static double getDouble(String prompt) {...}
  }
```

Students are asked to develop a program that calculates the value of expressions like the following: $12.0\ 24.0+\ =$. Students are requested to use the `Double` and `Stack` classes of the standard Java library. It is suggested to them to first experiment with `Double` and `Stack` classes in the BlueJ environment. BlueJ, which is specifically designed to support teaching the OO programming paradigm in beginning Java-based courses, is used as an alternative to the Java Development Kit (JDK), in both examples and assignments to elicit students' difficulties that originate from the complexity of the Java environment (Kolling, 1999b).

After having experimented with BlueJ, students realize that they must write their own object to represent the program they are requested to develop. They are asked, as a first task, to describe in every day language the behaviour of the requested program to the event "run" that originates from the user and comes to the program through the operating system. It has already been explained in lectures that the *main* method is compliant with the object-oriented approach although many textbooks and researchers consider that this is not correct. The `main` method simply defines the behaviour of the object that is used to represent the program under development, to the message *run* that is coming from the user through the operating system.

3rd Assignment: Class Responsibilities

Before this assignment, we have already started teaching about methods. Instance methods were covered first with class methods and class data members following. By this time students are already familiar, through a set of examples, with the basics of class definition.

In this assignment students are asked to define the Operand class so as the following sequence of keystrokes results in the definition of the expression $12\ 24+\ =$:

<1><3><Backspace><2><Enter><3><4><CE><2><4><Enter><+><=>.

After they have defined the `Operand` class they are asked to develop a program to test its behaviour. The development and test of behaviour of `Adder` and `ResultPresenter` classes is also required. Students are guided to test the behaviour of components using BlueJ or writing small programs in JDK.

4th Assignment: Using GUIs

Today students are accustomed from every-day life with computer programs with flexible graphical interfaces. They have little interest in laboratory assignments that use the old-fashioned line-by-line text input and output. This is why the graphical user interface of the RPN (in bytecodes) is provided from the very beginning and students are requested to write a program demonstrating the integration of the `Operand` class they have defined in previous assignment with the `calculatorGUI` class. Students have by this time already grasped the fundamental idea of message passing and this allows them to integrate existing pre-defined object into a working program.

The requested program must allow the user to define the value of an `Operand` instance through the provided GUI. The use of digit buttons as well as the backspace and the reset buttons should be supported. Students are asked to provide the following functionality to the `Operand` class: when the ENTER button is pressed the already defined operand must be pushed in stack. The constructor of the `calculatorGUI` class was defined to accept a reference of type `Operand` as an argument. This reference is used by the `calculatorGUI` to communicate with the `Operand` instance of our students. The behaviour of the `Operand` instance to the messages initiated by the `calculatorGUI` is defined by the followings methods:

```
void addDigit(char ch);
void deleteLastDigit();
void reset();
void complete();
```

5th Assignment: Examine alternative designs

The purpose of this assignment is to discuss and evaluate alternative design solutions. Alternative designs for the *`Operand`* class are investigated. This provides an excellent motivation for novice programmers to consider the association of data representation with the complexity of the produced methods.

Students are asked to consider the following two alternatives for the representation of the Operand's value:

a) `StringBuffer val;`
b) `double val`.

They are asked to write for each representation the methods of `Operand`. The objective is to clarify that the processing algorithms are greatly influenced by the representation of the information. Using examples like this, students see how investing time in data representation can simplify the methods one should write. Students are next asked to consider the following alternatives for the implementation of `complete()`:

a) convert the data member `val` to `Double` and push it in stack, or
b) push in stack the `Operand` instance itself.

6th Assignment: Abstract Classes – Inheritance – Polymorphism

At this time students have reached the stage known as object-based programming. It is the proper time to take one step closer to our goal that is the OO programming. It is the time to introduce inheritance and polymorphism, and allow students through this assignment to practice with Java's constructs that implement these concepts.

Students are asked to examine the possibility of the following RPN calculator classes: `Adder`, `Subtracter`, `Multiplier`, `Divider`, and `ResultPresenter`, to be considered as specializations of the `Operator` class. Students are asked to use inheritance to represent in code the above association. Discussing what happens when an inherited method is redefined by a subclass leads naturally to a discussion of polymorphism. How polymorphism can be used to eliminate if and switch statements is highlighted. Students use the UML notation for gen/spec to create the class diagram of Fig. 2.

In following assignments while building the `CalculatorGui` class, students should have the chance to practice on inheritance and polymorphism building their own inheritance tree and defining polymorphic behaviour.

7th Assignment: Abstract Window Toolkit – Event Handling

This assignment is used to introduce the basics of Abstract Window Toolkit (awt). Students are requested to create the CalculatorGUI shown in Fig. 3. The main window of the calculator graphical interface is an instance of the `Frame` class of the awt package. In the first step only 3 standard awt objects are used: `Frame`, `Button` and `TextField`. To help students handle the complexity of these classes, a brief text describing data members

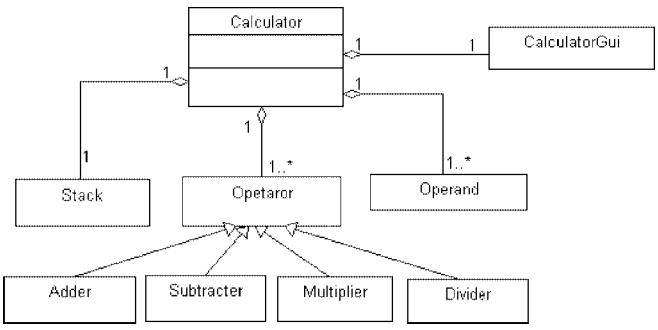

Fig. 2. Enhanced class diagram of the RPN calculator.

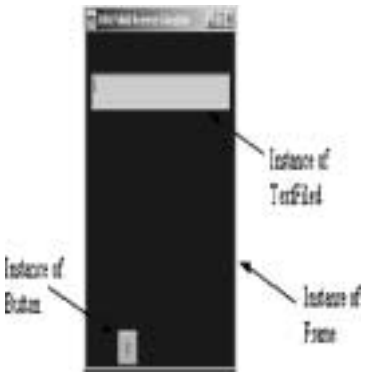

Fig. 3. The first version of the requested CalculatorGUI.

and methods that are used in the context of this action is given. For example students are guided to use the following methods of `Frame`:

- `setSize( )` // the method is inherited from `Component` to set the dimension of the window;
- `setFont ( )` // the method is inherited from `Container` to set font of the window;
- `toFront( )` // the method is inherited from `Window` to bring the window to the front.

It is very important not to overload students at this stage with too much detail from the awt but to emphasize only the basic concepts. Through this process students are encouraged to optionally study the definition of the above classes and discover useful data members and methods. This is also a good exercise for students to examine the way in which the designer of the Java basic library has assigned behaviour to the awt inheritance tree. Students enjoy navigating inside the Java library and discovering new methods to enhance their program. They discover the strength of reusability.

In the next step students are asked to assign behaviour to their Calculator window. They are asked to enhance their program to display the text "button 0 pressed" to the calculator's display when the user presses the button 0. This is the time to introduce the Java's event handling mechanism. OIDs are used to illustrate the basic concepts of event handling. The OIDs of Fig. 4(b) are given to students in order to understand that `Button` and `TextField` classes are used by the designer of the awt to implement the OIDs of Fig. 4(a) that represent the interaction between the user and the application's components. It is evident that the left-hand interactions are not implemented using the mechanism that is used for the interaction between the application's objects.

8th Assignment: Event Handling – Modifying Code to Increase Reusability

Students are given the coordinates of the remainder of the buttons of the calculator interface window and asked to complete the calculator GUI. They are next given hints to identify the class `DigitButton` to avoid the repetition of the code used to create and

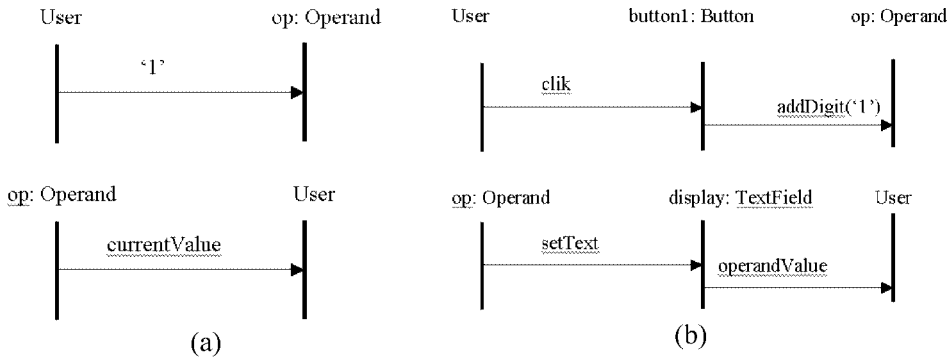

Fig. 4. Button and TextField are designed to support the interface between the user and the application's objects.

set-up each digit button as instance of `Button`. They are next guided to avoid the use of a specific handler for each digit button and define a handler that can act as listener for every digit button. They are expected to define `DigitButtonHandler` and `DigitButton` such as the (fragmentary) classes shown in Fig. 5.

During this time the interface construct is introduced and incorporated into assignments. Students have just experienced implementing the interaction between digit buttons and the `Operand` instance. They are given hints to implement the interaction between the Operator buttons and the corresponding `Operator` instances. They are expected, and the most of them provide an implementation similar to the one shown in Fig. 6.

```
class DigitButtonHandler implements ActionListener {
  public CalculatorGui calcGui;
  String label;
  public DigitButtonHandler(CalculatorGui calcGui, String label){
...}

  public void actionPerformed(ActionEvent pushingButton0){
    calcGui.display.setText("button " + label +" was pressed
                                    " + count +" times");
        }
    }

class MyDigitButton extends Button {
  String label;
  static CalculatorGui cg;

  public MyDigitButton(String label, int x, int y, int width,
                       int height){
    super(label);
    :
    addActionListener(new DigitButtonHandler(cg,label));
    cg.add(this);
    }
}
```

Fig. 5. DigitButton and DigitButtonHandler classes.

```
class Adder extends Operator implements ActionListener{
  public Adder(Stack st) {super(st); }
  public void actionPerformed(ActionEvent pushingButton){
     this.operate();}
  public void operate(){
     Double d = new Double(((Double)st.pop()).doubleValue() +
                  ((Double)st.pop()).doubleValue());
     st.push(d);
     }
  }
public class CalculatorGui extends Frame {
  :
  buttonAnd = new OperatorButton("+",195, 265, 35, 28);
  buttonAnd.addActionListener(Calc.ad);
  :
  }
```

Fig. 6. The Adder class implements the ActionListener interface.

Students can now identify that the operator's sub-classes have something in common. They all implement the `ActionListener` interface. They should modify their code to look like the one shown in Fig. 7. Finally they are asked to construct a UML class diagram to represent the structure of their calculatorGui.

9th Assignment: Increase Flexibility – Avoid Using Switch

A sample source code for the CalculatorGui is given to students. The code has been developed without application of basic OO design principles. Students are asked to do the following: (a) draw the class diagram of the given code, (b) identify imperfections and redesign the class diagram, and (c) modify the structure of the code to be in compliance with the new class diagram. Students notice that an OO design eliminates the use of the switch statement. After this assignment, students obtain a better understanding of the important role that the class diagram plays in program development.

10th Assignment: Hands on Java API

This assignment allows students to experiment with what they have learned about awt and event handling. They are asked to develop a window to visualize the status of the stack. Two or three instances of stack must be displayed for the user to have a better feeling of the operation of the RPN calculator. At this point students work without guidance. They are allowed to gain a deeper understanding of the development process and the way that tools and modelling techniques fit together.

```
abstract class Operator implements ActionListener{
  Stack st;
  public Operator(Stack st) {this.st = st; }
  public abstract void operate();
  public void actionPerformed(ActionEvent pushingButton){
     this.operate();}
  }

class Adder extends Operator { ...}
```

Fig. 7. Inheriting behaviour that was defined by the proper use of implements.

11th Assignment: Finding Bugs – Extending System's Behaviour

A sample source code for the RPN calculator is given and students are asked to first identify bugs and next add extra functionality. A lot of hints are given to guide students identify bugs and make the appropriate corrections. As Pea (1986) reports, emphasizing reading and debugging activities can be useful in bringing partial conceptions to light.

12th Assignment: Hands on Exception Handling

Students are given a set of scenarios in using an RPN calculator to force exceptions to be reported by the JVM. They are asked to modify code so as to handle the produced exceptions.

Through the set of assignments the idea that new concepts are been built on top of those already known is strongly emphasized. This is the reason why students after each assignment are given the full source code required by the assignment. Students use this code: (a) as a reference implementation to solve some difficulties they had with the specific assignment, and (b) as a reliable basis on which to start the next assignment. The aim is to make clear to students that effective programming is rarely about building a piece of code from scratch but is usually a mixture of reuse and intelligent adaptation of pre-existing code.

5. Evaluation

The whole course has been experimentally evaluated by the author in (Thramboulidis, 2003a). The results of this evaluation were very encouraging. There was a general understanding that the course, compared with the first phase of its development, has vastly improved.

The sequence of assignments has been used as the major vehicle for teaching the OO paradigm to (a) novices, (b) students already exposed to procedural programming and C, and (c) postgraduate students experienced in other programming languages. With regard to assignments' acceptability, even though an extended assessment has not been conducted, students of all categories accepted these with a lot of positive remarks. Preliminary results extracted from discussions with students, surveys, projects and course examinations, indicated that students were by the end of the course, able to apply during the development process the abstractions and structure they had learned while using the "Goody's example"; they were able to identify and describe use cases, identify objects and build their own informal OID's, and create informal class diagrams. Students were also able to utilize the syntactic elements and the concepts of Java, in order to implement their models using JDK, BlueJ or the development environment of their choice.

To better understand the students' perspective on the assignments, students of the last two categories were asked to fill out questionnaires. The questionnaires were given during the semester and just after the completion of the part that deals with the introduction of the OO programming paradigm. Unfortunately since there is no formal course-evaluation process established at the department level, at the moment, students are not accustomed to

this process and a small number of questionnaires were returned fully completed. Some 85% of the students found the assignments "very useful" in overall, while the others found them "mostly useful", with 0% selecting "useless", i.e., the low mark. Although 35–40% of students reported difficulties in working with assignments, no supplementary material except JDK API or any additional help other than the one provided during lecture time was required to complete the assignments. In more detail, regarding the difficulty of the assignment sequence the following results were obtained, on a scale from 1 ("no difficulty") to 5 ("very difficult"): 12.5% for 2, 50% for 3 and 37.5% for 4. This was better than expected since (a) about 30% of students have reported, prior to taking the course, that they had only a limited knowledge of programming in C, and (b) the introduction of the OO paradigm in the tested courses took only 4 weeks of the whole course. This is why about 58% of the class suggested that it should be better if more lecture time was allocated to this part of the course or if, at least, the deadline for the final report on assignments was extended. However, more than 70% fully covered the assignment sequence with the others having covered about more than the two-thirds of it. The approximate reported time that was spend for the assignments varied from the low of 12 hours for those rated their programming skills prior to taking the course as "strong", to 35 hours for those rated their programming skills as "weak". It must be noticed that the evaluated courses were elective with students being high achievers in programming or having decided to become better in this topic.

Another, quite interesting result is the one that shows how useful were the assignments in helping to understand and properly use the quite complex event handling mechanism of Java. In the question "How useful were the assignments in helping you learn Java's event handling?", even though about 35% reported "mostly useful" with the others stating "very useful" (only one student reported "useless"), almost all (even the one reported "useless") understood and used effectively the mechanism as was concluded by the final reports on assignments and projects. This can be explained by the fact that students have not been exposed to another approach used to introduce this mechanism so as to be able to compare with. Four students that have been reported "good" knowledge of Java, prior to taking the course, were impressed by the way that event handling was simplified.

Analogous results were extracted regarding the UML artefacts introduced in the course. Table 1 contains the results for the following question: "How useful were the "Goody's example" and the assignments in helping you understand the UML artefacts used in the course?" Finally, more than 85% of students found the "Goody's example" very effective in helping them learning the basic OO concepts.

Table 1

Questionnaire results (partial)

	Useless	**Mostly useful**	**Very useful**
Use-case	0%	32%	68%
Class diagram	7%	24%	59%
Object Interaction Diagram (OID)	0%	14%	86%

The proposed approach is also used in obligatory courses by the author as well as in other Greek universities to teach programming to novices, but no evaluation was conducted yet. However, an improvement in learning outcomes was reported along with a lot of positive comments from students and teaching staff.

6. Concluding Remarks

The development of this approach was greatly influenced by constructivism. It was found that novice programmers already know most of the concepts that constitute the conceptual frameworks of the OO paradigm, exception handling and concurrency, and this knowledge emanates from multi-entity systems of every-day life and human behaviour. The "Goody's example" that is based on a real-life system was devised and used as an anchor in building the conceptual framework of the above topics. However, it was found that the new course had nothing in common with the one following the traditional approach, only when the design-first approach was adopted and the assignments were brought up as the core component of the course. It was then, that the "Goody's example" proved an excellent means of eliminating or avoiding most of the misconceptions reported by Fleury in (2000) as student-constructed rules. The answers to these misconceptions were found in this well known from every-day life system. This analogical reasoning proved to be very successful. Students started thinking in ways different from the von Neumann serial computation model and this direction was encouraged. The assignments enforced this perspective and also allowed students to have a sense of satisfaction and achievement in creating an application with graphical user interface even of limited functionality from the very beginning.

The early versions of the assignments requested students to implement their own graphical GUI calculator. It was noticed that, even though use-cases, class diagrams and OID's had been presented during the "Goody's example", students did not follow a design-first approach but were immediately moving to the implementation, dealing with the idiosyncratic features of the language. Even more it was found that students had a lot of difficulties concerning the identification of objects and their collaboration, as well as some implementation issues such as the awt and the event handling mechanism of Java. These findings were translated into practical recommendations and the assignments were redesigned. New assignments were developed to practice on the underlying concepts using informal versions of UML artefacts and to force a design-first approach. Assignments were also developed for awt and event handling (assignments 7, 8 and 10). However, this proved a very difficult and time-consuming task since special attention was required not to overwhelm the novice programmer with the formalism of UML on one hand and the complexities of Java's awt and event handling on the other, as other approaches do. The result was that the assignments, in their final version, come with the description of the strategy and graded hints leading to the final solution. Hints that not only help the student having difficulties, but that also offer alternative suggestions and discuss issues that arise. Indicative solutions are also given for students to verify their work.

The development of the new course was a difficult and time-consuming job, but the results were worthwhile. While the important benefits are perhaps impossible to quantify, the immediate outcome is clear. Students when they finish the updated course have understood and appreciated the significance of the design-first approach. They are accustomed to the idea that writing code is not the first activity they have to do when they are asked to develop a software system. They have improved their ability, compared with previous versions of the course, to solve simple design problems using the OO concepts and to implement their draft designs effectively using the Java language and the Java API. The designed and implemented software is easier to understand, debug and extend.

In this paper it is strongly argued that the course should: (a) exploit the students' existing knowledge to build the OO conceptual framework and emphasize it using informal versions of selected UML artefacts, (b) introduce the language as a means of implementing the model and avoiding overwhelming the novice programmer with the idiosyncratic features of the specific language but rather focusing on the underlying concepts and language mechanisms, and (c) be assignment-based, with assignments accompanied with appropriate hints and helpful material to encourage active learning and student involvement in the learning process, and with lectures becoming more supportive rather than descriptive.

Acknowledgements

The earliest ideas for the described approach originated from the time the author was writing the book "Programming Languages II: Object-Oriented Programming" during 1997 for the Hellenic Open University. The work described in this paper would not have been possible without my students. Discussions with them in classroom and in the laboratory were the main source of inspiration to the author.

References

Arnow, D., and G. Weiss (1998). *Introduction to Programming Using Java: An Object-Oriented Approach*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA.

Bergin, J. (2000). *Teaching Objects with Elementary Patterns*. `http://csis.pace.edu/~bergin/patterns/ TeachingObjectsElemPat.html`

Bergin, J., and R. Winder (2000). *Understanding Object Oriented Programming*. `http://csis.pace.edu/~bergin/patterns/ppoop.html`

Borstler, J., and A. Fernandez (1999). *Quest for Effective Classroom Examples*. OOPSLA'99 Workshop Report, Sweden.

Borstler, J., T. Johansson and M. Nordstrom (2002). Teaching OO concepts – a case study using CRC-Cards and Bluej. In *32nd ASEE/IEEE Frontiers in Education Conference*. T2G-1, Boston, MA.

Bruce, B.K., A. Danyluk and T. Murtagh (2001). A library to support a graphics based object-first approach to CS 1. *ACM SIGCSE Bulletin*, **33** (1), 6–10.

Confrey, J. (1990). A review of the research on student conceptions in mathematics, science, and programming. In C.B. Cazden (Ed.), *Review of Research in Education*. DC, American Educational Research Association, Washington, **16**, pp. 3–56.

Demuth, B., H. Hussmann, L. Schmitz and S. Zschaler (2000). A framework-based approach to teaching OOT: aims, implementation, and experience. In S.A. Mengel and P.J. Knoke (Eds.), *Proceedings Thirteenth Conference on Software Engineering Education & Training*. IEEE Computer Society, Austin, Texas.

Duke, R., E. Salzman, J. Burmsteir, J. Poon and L. Murray (2000). Teaching programming to beginners – choosing the language is just the first step. *ACE 2000*, 12/00 Melbourne, ACM 2000, Australia.

Eckstein, J. (1997). A paradigm shift in teaching OOT. In *Proceedings of the Educators' Symposium at OOPSLA'97*. Atlanta, Georgia.

Glasersfeld, E. von (1989) Cognition, construction of knowledge and teaching. *Synthese*, **80** (1), 121–140.

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

Glasersfeld, E.V. (1995). A constructivist approach to teaching. In L.P. Steffe and J. Gale (Eds.), *Constructivism in Education*. Hillsdale, NJ, Lawrence Erlbaum Associates, pp. 3–16.

Greenco, J.G., A.M. Collins and L.B. Resnick (1996). Cognition and learning. In D. Berliner and R.C. Calfee (Eds.), *Handbook of Educational Psychology*. Simon & Schuster Macmillan, New York, pp. 15–46.

Gray, J., T. Boyle and C. Smith (1998). A constructivist learning environment implemented in Java. In *ItiCSE'98 Dublin*. ACM, Ireland.

Harel, I. (1991). *Children Designers: Interdisciplinary Constructions for Learning and Knowing Mathematics in a Computer-Rich School*. Norwood, NJ., Ablex Publishing.

Hadjerrouit, S. (1999). A constructivist approach to object-oriented design and programming. In *ITiCSE'99*, 6/99 Cracow, ACM, Poland.

Jones, A. (2002). Integration of ICT in an initial teacher training course: participants' views. In *7th World Conference on Computers in Education*, Copenhagen, Australian Computer Society, Inc.

Kernighan, B., and D. Ritchie (1988). *The C Programming Language.* Second edition, Prentice Hall International.

Kolling, M. (1999a). The problem of teaching object-oriented programming, Part 1: languages. *Journal of Object-Oriented Programming*, **11** (8), 8–15.

Kolling, M. (1999b). Teaching object-orientation with the BlueJ environment. *Journal of Object-Oriented Programming*, **12** (2), 14–23.

McCloskey, M. (1983). Naïve theories of motion. In D. Gertner and A. Stevens (Eds.), *Mental Models*. NJ, Lawrence Erlbaum Associates, Hillsdale, pp. 299–323.

Madden, M., and D. Chambers (2002). *Evaluation of Student Attitudes to Learning the Java Language. Principles and Practice of Programming in Java 2002*.

Pea, R. (1986). Language independent conceptual "Bugs" in Novice programming. *Journal of Educational Computing Research*, **2** (1), 25–35.

Schoenfeld, A.H., J.P. Smith and A.A. Arcavi (1993). Learning: the microgenetic analysis of one student's understanding of a complex subject matter domain. In R. Glaser (Ed.), *Advances in Instructional Psychology*, NJ: Erlbaum, Hillsdale, **4**, 55–175.

Sheetz, S., G. Irwin, D. Tegarden, J. Nelson and D. Monarchi (1997). Exploring the difficulties of learning object-oriented techniques. *Journal of Management Information Systems*, **14**, 103–131.

Smith, J., A. DiSessa and J. Roschelle (1993). Misconceptions reconceived: a constructivist analysis of knowledge in transition. *The Journal of the Learning Sciences*, **3** (2), 115–163.

Stein, L.A. (1998). What we've swept under the rug: radically rethinking CS1. *Computer Science Education*, **8** (2), 118–129.

Thramboulidis, K. (1998). *From Procedural to Object Oriented Programming*. Tziolas Publ. Company Inc, Thessaloniki, first edition (in Greek).

Thramboulidis, K. (2003a). A constructivism-based approach to teach object-oriented programming. *Journal of Informatics Education and Research*, **4** (2), 1–11.

Thramboulidis, K. (2003b). Teaching advanced programming concepts in introductory computing courses: a constructivism based approach. In *ICEE International Conference on Engineering Education*. Valencia, Spain.

Tyma, P. (1998). Why are we using Java again? *Communications of the ACM*, **41** (6), 38–42.

K. Thramboulidis is a research and teaching staff assistant professor at the university of Patras Greece. He has been using the object-technology since 1989 and has successfully applied it in many research and development projects. He is the designer of REDOM, an OO Language used in the airline domain, to define and on-line manipulate regulations in the resource (re)scheduling problem. He is currently working on CORFU, an object-oriented framework for the unified development of distributed control and automation systems. He is author of the following books (in Greek): *From C to Java*, *Procedural Programming – C*, *Object-Oriented Programming – Java*, published by TZIOLAS Publishing Inc.

Užduočių seka objektiniam programavimui mokyti: konstruktyvistinio metodo projektas

Kleanthis C. THRAMBOULIDIS

Konstruktyvistinis metodas objektiniam programavimui mokyti jau taikomas keletą metų. Norint išnaudoti pradedančiojo programuotojo žinias objektinio programavimo konceptualiai struktūrai sukurti buvo pasirinkta daugiapakopė sistema. Buvo suprojektuota ir parengta užduočių seka, kuri įgalintų studentus taikyti šį metodą naudojantis Java programavimo kalba ir leistų perprasti objektinio programavimo konceptualiuosius elementus. Šiame straipsnyje pateikiama užduočių seka, ji nagrinėjama ir įvertinama aprašyto metodo kontekste. Užduotys teikia pirmumą programinės įrangos inžinerijai, dar tiksliau, projektavimo darbams. Aprašoma strategija, kuri padeda studentams palaipsniui pasiekti sprendimą. Nors ši medžiaga buvo projektuojama kaip papildoma, tačiau ji greitai tapo kurso pagrindu.