A Constructivism-Based Approach to Teach Object-Oriented Programming

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

ABSTRACT
We have developed a course in object-oriented programming for students already familiar with the procedural programming paradigm. To address persistent difficulties mainly imposed by the already known paradigm, we adopted the approach of constructivism, which stresses the importance of prior knowledge on top of which new knowledge is built. A real-world system was adopted, to exploit the prior knowledge that students have from everyday life. This perspective guided us in making a shift in focus from the algorithm-centered view to the software-engineering-centered view and more precisely to the design-first approach. We found that an informal use of use-cases, class diagrams, and object interaction diagrams facilitates students in exploiting their real-world experience and building on it the conceptual framework of the object-oriented paradigm. A set of assignments based on these findings was developed and used over the last 2 years. The first results of this approach are very encouraging.

INTRODUCTION
We offer a one-semester course in Object-Oriented (OO) programming to undergraduate students, who have already studied, during their first programming course, procedural programming using C language. The focus of the course is on teaching the basic concepts of the OO programming paradigm rather than a specific programming language. The course is comprised of both lectures and lab activities and is taken by all Electrical and Computer Engineering students. It is a key course in our computer-engineering curriculum that includes about 9 courses to prepare students for software development. Three of the courses are obligatory and are taken by all students, while the rest are taken mainly by students who follow the software option of the curriculum.

The course has been developed over the past 6 years and it has gone through three major phases of development. In each phase, the feedback received from students was assessed and utilized to improve the students’ understanding of the fundamental OO concepts. During the first 2 years that constitute the first phase of the course we followed the traditional approach adopted by the majority of textbooks on OO programming. During this time, we found that students had persistent difficulties mainly imposed by the already known procedural paradigm. This problem that is recognized by many educators (Eckstein, 1997; Sheetz et al., 1997) is responsible to a large extent, for changing first-year courses in many universities to OO programming. However, since we still believe that Electrical and Computer Engineers must have a good knowledge of procedural programming and C language, we had to invent new ways to overcome the problems emanating from the student’s prior knowledge on procedural programming. 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 does not work well (Borstler et al.,
A “new pedagogy to teach objects well” is required. In fact, as Bergin (2000) reports, this is one of the most important and perhaps the most controversial issue under discussion in many educational meetings.

We considered issues in science education, mainly constructivism, which is one of the fundamental ideas in education (Glasersfeld, 1989; Confrey, 1990; Greenco, Collins & Resnick, 1996). Constructivism is a philosophy of learning, which claims that students by reflecting on their experiences, construct knowledge rather than merely receiving and storing the knowledge transmitted by the teacher. From the viewpoint of constructivism, knowledge cannot be judged as correct or incorrect, but as productive or nonproductive. Thus, teaching is the process of presenting ideas that sensibly and consistently explain problematic aspects of the learner’s world. Pieces of the knowledge system that do not function this way do not last. The knowledge system consists, according to diSessa (1988), of a large number of interrelated and context-specific components organized in such a way that each component, as part of the system, affects other components and is affected by them. Moreover, according to Schoenfeld, Smith & Arcavi (1993), learning is the process of strengthening and weakening the connections between knowledge elements rather than replacing elements. Applications of constructivism in education lie in creating learning environments or curricula that match students’ understanding, fostering further growth and development of mind. Kolikant (2001), for example, anchored his work on concurrency in constructivism and found that students’ existing knowledge plays a major role in their performance.

Motivated by constructivism and in our attempt to: (a) avoid the persistent difficulties imposed by the already-known procedural paradigm, and (b) find a new pedagogy to teach objects well, we devised the “Goody’s example” to establish an alternative and more appropriate knowledge base, which may be refined in order to create the OO knowledge. Our real-world example was based on “Goody’s”, the 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 behaviour.

A lot of work was devoted to reorganizing the course. The first segment of the updated course has the clear objective to establish the conceptual framework of the OO paradigm. We utilize the “Goody’s example” to guide our students in exploiting their prior knowledge emanating from real life and building on it the conceptual framework of the OO paradigm. During the second segment we focus on mapping the concepts of the OO framework into a formal programming language. The constructs and the mechanisms of the Java programming language, which implement the basic concepts of the OO paradigm, are introduced. The Reverse Polish Notation (RPN) calculator was adopted as a case study for this segment of the course and an object-oriented realization, by means of language features, is required from students. A specific set of examples and assignments were developed to complement the practice sessions of this segment. Topics including exception handling and concurrency are covered in following segments.

This paper is organized as follows. In the next section, we consider the refinement process of the course and present problems encountered during the first and second development phases of the course. In particular, we discuss the problem of deciding what to teach in the course and present the set of actions taken during the first and second major refinements of the course. We next give the main outline of the updated course. Special attention is given to the “Goody’s example” as well as to the reverse
polish notation calculator case study. Finally, we discuss the approach and conclude the paper.

**COURSE DEVELOPMENT PHASES**

This course has gone through the three phases of development shown in Table I. In this section we refer to these phases, discuss the problems we encountered, and describe the actions taken in our attempt to improve the course.

### Table I

**Course Development Phases**

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1998–2000</td>
<td>2nd version based on the object-first approach. Use of real-world experience through the “Goody's example,” Introduction of class diagrams. RPN calculator case study. Exception handling and concurrency are considered as new topics.</td>
</tr>
<tr>
<td>2000–2002</td>
<td>3rd version based on a design-first approach and the “computation as interaction” model. Use of informal versions of use cases, object interaction diagrams and class diagrams. Establish the conceptual framework through the “Goody's example.” Utilize the “hit them twice” strategy and in class laboratory.</td>
</tr>
</tbody>
</table>

**Preparing the first version of the Course**

When planning the course in 1996, our first dilemma was the selection of the language to be used for the introduction of the new programming paradigm. Due to our Software Engineering background and our experience from teaching object-oriented analysis and design for several years, we abandoned, from the beginning, the use of C++ as a language to introduce the object-oriented programming paradigm. The hybrid nature of the language and the large number of low-level constructs that must be mastered, were among the main reasons for this decision. Instead of C++ we selected Java, even though the language: a) was still in its early stages, b) was little used by institutions and universities, and c) there was a need for more and better libraries and extensive curricular material (Bergin, 1996). We received a lot of criticism regarding this decision since C++ was at the time and until recently, the most commonly used language for teaching the OO programming paradigm. Nowadays it is widely accepted that C++ adds a huge amount of complexity introduced by the nature of the language itself. This complexity makes concentration on the fundamental concepts of object-orientation very difficult. Even though a lot of ways are suggested to cope with C++ pitfalls in education (Berman et al., 1994; Headington, 1995; Horstmann, 2001), it is still difficult to concentrate on the OO concepts while at the same time having to confront all these issues.

For the course realization, we adopted the traditional approach, which is followed by the majority of textbooks on object-oriented programming. This approach, which is the same as that used in the preceding procedural programming course, is based on the von Neumann serial computation model. According to this approach we began “with a question and described the answer in terms of the question.” We defined programming as “the process of writing down the sequence of calculations required to get from a particular instance of the question to the corresponding instance of the answer,” and computation as “the process of executing those steps, i.e., the algorithm,
to deduce the answer to a particular question” as is very well described in Stein (1996).

**Moving from traditional to a constructivism object-first approach**

During the first 2 years of the course realization, we followed the same approach with that used in the procedural-programming course. We were facing the commonly referred to and now well-known problem of paradigm shift, i.e., the switch in paradigm from procedural to OO programming. Bergin & Winder (2000) performed an empirical case study to demonstrate that OO is a real paradigm shift rather than just a packaging mechanism for procedural programming and this is why a change in mental model in the practitioners is required. We found that students tried to build the concept maps of the new paradigm on structures created during the pre-existing procedural paradigm. McCloskey (1983) reports an analogous problem working with physics students. He suggests that instruction process should replace one theory with the other. However, this proved to be a very difficult task for the majority of our students. Moreover, according to Schoenfeld et al. (1993) replacing one idea with another is not a simple task. Smith (1993) believes that learning is not a process of replacing an incorrect component of knowledge with a correct one, but it is accomplished by the refinement of existing knowledge in order to create new knowledge.

Motivated by the above results and in our attempt to exploit the benefits of constructivism, which stresses the importance of prior knowledge on which new knowledge is built, we decided to look for this prior knowledge. We found that this knowledge already exists in our students and emanates from real-world life. We devised the “Goody’s example” and we use it through the first segment of the course to introduce the conceptual framework of the OO paradigm.

During this 1st major refinement we adopted an object-first approach. Numerous researchers (Arnow and Weiss, 1998; Bruce, Danyluk & Murtagh 2001a) have adopted the object-first approach and reported encouraging results. To improve the effectiveness of our object-first approach, we adopted: (a) a restricted use of UML class diagrams, and (b) an extensive use of the BlueJ Integrated Development Environment (IDE). 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).

**Moving into a design-first approach**

Teaching during the second phase of course implementation, we discovered that the problem of paradigm shift did not disappear. Later on, we found that a brief introduction to use-cases, class diagrams and object interaction or scenario diagrams, helped students exploit their real-world experience and use it to build their new knowledge of the object-oriented programming paradigm. We decided to introduce, during the first segment of the course, the above analysis and design artifacts through the “Goody’s example,” and utilize them, to create draft models for the systems of our examples for the rest of the course. After we modified this first segment, we 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. We encouraged, 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. Moreover, this discovery guided us in making a shift in focus from the algorithm-centered view to the software-engineering-centered view and more precisely to the design-first approach. We redefined programming as the process of building communities of interacting entities. During this process the main tasks of the programmer are to define: a) the ways the community entities interact and b) what goes on inside each entity of the community.

We found that, although the first iteration of the “Goody’s example” was necessary to provide background, it was not fully effective until a second iteration was used to really get the concepts across. After students try their hand at OO development, it is the appropriate time to present the initial part of the course again. This second iteration, which should not be identical to the first one, helps students who have already grappled with the day-to-day challenges of object-oriented software construction to internalise the concepts. Actually students found the application of the “hit them twice” strategy, as it is called in Meyer (1996), very efficacious towards an in-depth understanding of the basic concepts of the OO paradigm.

To address the new direction taken by the course, a major re-writing of examples, exercises, and laboratory assignments was required. Moreover, we found that although concurrency and exception handling are considered to be new terms for students, they are concepts existing in the “Goody’s example” and well understood by our students. We decided to expand our course in this direction with an introduction to both concepts. The selection of Java as our programming environment in teaching object-orientation proved successful for this direction too. However, the RPN calculator did not constitute a problem, which would properly bring out the issues related to concurrency. A set of small case studies is currently used to address concurrency issues, while at the same time a project is running to produce a more coherent assignment. This assignment should not necessarily be complex but it should originate from an authentic problem of interest to students.

Finally, we found that dedicating one of the three hours of lectures per week to discussing and preparing the laboratory assignment greatly enhances the effectiveness of the course. Students need to be 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. This in-class-laboratory is used to discuss and justify design decisions and approaches used in the laboratory assignment. After the laboratory, they are given 3 to 4 days to prepare their final report.

CONTENT OF THE UPDATED COURSE

The outline of the course

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, makes evident 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, object interaction, and scenario diagrams in a very informal UML notation, were adopted during the second major refinement and 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, we teach the above material without reference to any programming language.
After having introduced the conceptual framework, it is the appropriate time to define in the classroom with students the developer’s expectations from an environment that should allow the construction of software systems according to the object-oriented approach. We adopted a “Lego construction” approach and we ask students 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 below:

1. Provide a library of predefined object-types.
2. Provide means to exploit predefined components.
3. Send messages to objects.
4. Create instances of an object-type.
5. Define an object-type, i.e., the structure and the behaviour of its instances.
6. Define the interface as well as the implementation part of objects.
7. Define an object as a composition of its components.
8. Define an object as a specialization of another object.

This is the best time to start the second segment of the course in which we focus on the basic constructs of the Java environment that are necessary to satisfy the above requirements of the system developer. 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. We selected the RPN calculator as a case study for this segment and require each student to implement their own calculator following a well-defined step-by-step development process, which runs after the introduction of the language constructs. Computation is discussed in terms of community of entities rather than in terms of flow of control, as described in Stein (1998). Object Interaction Diagrams (OIDs) are utilized appropriately to this direction. A draft design is given and students are guided through a step-by-step approach in developing their own calculator using JDK or BlueJ.

In the third segment of the course we present again the initial part of the course to help students better internalise the OO concepts. 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. However, since the material needed to support this new approach was not available, we decided to proceed to the edition of our book in 1998 for use as the main textbook on the course. We recognized that we were teaching to our students a view of computation that would no longer correspond to the requirements of modern applications, the ones students use in everyday life.

The “Goody’s example”

As noted previously, all our students are already familiar with “Goody’s.” They have all used its services and have an understanding of its structure and behaviour. They know that they use the word “Goody’s” to refer to a specific type of restaurant and that this word is not enough to identify the specific restaurant located in “Olga Square.” They understand that both “Goody’s” and “Olga Square Goody’s” (OSG) are identifiers of entities (objects), but the first one is used to identify a conceptual object while the second one identifies a specific real-world object, whose structure and behaviour is specified by the first one. Students have a clear understanding of the
concept of type and instance and they are only required to learn the terms used to refer to them. They all know that in order to eat a hamburger they must find an instance of “Goody’s” and send a message to it. They already know that it is preferable to use the service of an existing instance, if one is available, rather than build their own.

Students have a good understanding of the composition of this type of restaurant. This understanding however is not for the entire object but only for the front-end of it, i.e., the part with which they have to interact. The back-end of “Goody’s” is actually hidden from its clients; only some components of the front-end can interact with some components of the back-end. Students also understand the reasons for such packaging and controlled visibility. It is a well-organized restaurant that provides reliable services within time and budget. And this is the reason they prefer using this type of objects.

The following paragraph is from the main textbook used in the course: “Helen has just accepted an order from Chris, a client requesting a hamburger and a cheese pie. She sends an oral message to Nick to prepare a hamburger, but she has to properly set the time button of the microwave oven and then press the start button to heat the cheese pie. Nick, in the back-end, uses the services of the toaster to prepare the hamburger. The dishwasher sends the message “need water” to the object tap, in a different way than it sends the message “washing finished” to Nick. … We can see that the objects that compose our system collaborate to provide the environment with the required services or to respond to accepted messages” (Thramboulidis, 1999, p. 277). Students are requested to identify classes and instances from the above text, as well as from other texts that describe the creation and operation of the OSG. They have to represent them using the UML artifacts for class and instance. Aggregation and generalization-specialization are concepts already known, so students are requested to create a draft class diagram using the related UML artifacts to represent them. They are next requested to describe their interaction with the OSG in order to obtain the required service, i.e., get a hamburger and a glass of beer. Students are asked to describe the whole interaction in natural language and next they are guided to express it using a simplified notation for OIDs (Thramboulidis, 1995). Figure 1 shows a sample OID that was created by one of our students. Similar OIDs are used in the context of the “Goody’s example” to introduce the constructs required to create OIDs. We have represented “Chris” in this OID to make clear that the user of the system is another entity in the community of interacting objects, and this entity has its own structure and behaviour. However, we do not consider the definition of its structure and behaviour since it is an entity that is outside of the context of our system.

The “Goody’s example” is used to introduce most of the concepts of the OO programming paradigm. We also use it for exception handling and concurrency. The following paragraph is from the section that introduces concepts such as interface, implementation and information hiding. “For example, Helen in order to use the microwave oven, only needs to have access to the timer, start, and open-door buttons. We say that these buttons constitute the ‘interface’ of the microwave oven. Nick uses only this “interface” and there is no need to know the internals of the microwave oven. We say that the microwave oven hides its internals, i.e., its implementation, and this is its constructor’s decision. However, its constructor has access to its implementation and of course the same should be true for the technician who is going to repair it. We have just cited an application of the information hiding principle. All objects, simple or composite, are constructed in such a way that they expose only those items that are involved in the process of getting the messages from the
environment or exporting their responses to it. These items must be visible and known to their clients.”

![Object Interaction Diagram](image)

**Figure 1.** A sample object interaction diagram created by one of our students.

We must note that during the time the “Goody’s example” is being studied, material covered in class shows how to expand upon simple concepts inherent in this example and apply them to the more realistic examples that would follow and would be used throughout the course and in the following weeks in the laboratory. Students are requested to use the OO concepts as a means to describe systems from domains they already know. This way, they practice on the Booch (1994) statements “objects do things and we ask them to perform what they do by sending them messages” and “we view the world as a set of autonomous agents that collaborate to perform some higher level behaviour.”

**Helen’s model**

Most of the objects that were used to create the OSG were ready to use: the toaster, the dishwasher, the cash register, and so on. However, Helen had to be “programmed” on how to respond to accepted messages. The model of Helen’s behaviour that is given in Figure 2 reflects the natural human model. It abstractly represents the behavior of any active entity of the real world, which can accept messages from its environment and respond to them. Such entities in the “Goody’s example” are Helen, Nick, the microwave oven, the toaster, the cash register and so on. Messages originate from entities called clients, when they require the services provided by the receiving entities called servers. However, it is fundamental to make clear to our students that most of this abstract model is already implemented by the development environment and they only have to define its last line, i.e., the entity’s behaviour to accepted messages.
Keep your eyes and ears alert.
while working,
accept next message,
process the message according to its type.

Figure 2. Helen’s abstract model.

The “Reverse Polish Notation Calculator” case study

After having introduced the basic constructs that are required by a development environment to support the implementation of systems according to the OO conceptual framework, we are ready to proceed to a realistic example that will allow students to apply and refine their OO knowledge. The problem given to students is to develop a calculator that follows the reverse Polish (postfix) notation and supports the operators +, -, *, /, and =. The use of a stack for the development of the calculator is mandatory.

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. During the first assignment students are given the expression 24 - 12 * 14 - 3 = and are asked to calculate its value using the RPN calculator. They are asked to describe in natural language the interaction between the user and the program and in a subsequent step to draw an OID, such as the one drawn in the OSG example, to represent this interaction. To allow students to practice and understand the calculator’s behaviour we give them a RPN calculator program with a GUI interface. Students are then guided in building a draft class diagram as they did in the “Goody’s example.” The sample class diagram of Figure 3 is given to them at the end of this assignment and is discussed in depth so as to form the base for the subsequent assignments.

Figure 3. Draft class diagram of the reverse Polish notation calculator.

We then gradually move, with the next assignments, to the implementation by requesting the development of a program that calculates the value of the following expression 12.0 24.0 +=. We request students to use the Double and Stack classes of the standard Java library and the BlueJ environment. As a first task students must describe in natural 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. This is the time to explain that the main method is compliant with the object-oriented approach although many textbooks and researchers consider that this is not true. 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. The concept of meta-class and the static modifier are introduced with the use of simple examples.

In the third assignment students are asked to define the Operand class so as to support for the definition of the expression 12 24 + = the following sequence of keystrokes:

\[1< \textless 3 \textless \texttt{Backspace} \textless 2 \textless \texttt{Enter} \textless 3 \textless 4 \textless \texttt{CE} \textless 2 \textless 4 \textless \texttt{Enter} \textless \texttt{+} \textless \texttt{=}\]

After they have defined the Operand class they are asked to develop a program to test its behavior.

In subsequent assignments students are given the calculatorGUI class in bytecodes and they are asked to implement their own first version of the RPN calculator. Later in assignments 8, 9 and 10 we expose students to most of the underline event-handling model of Java. Bruce, Danyluk & Murtagh (2001b), claim that event-driven programming facilitates the learning process of many programming topics. However they controvert the feasibility of Java’s mechanism for beginners and provide a library with facilities to simplify the handling of mouse events.

In following assignments students are given a version of the calculatorGUI class in source code with limited functionality and they are asked to add more functionality. Following on from this, they have to develop: a GUI to display the stack status, expand the calculator’s functionality with memory and other features, as well as to practice with inheritance, polymorphism, and exception handling. An outline of the RPN calculator assignments is given in Appendix A and an online version is available at http://seg.ee.upatras.gr/OOCourse.

EVALUATION

It is difficult to make a formal assessment of the new approach. We are currently studying assessment tools and processes used from other schools. Much work has been devoted to this. Scoles (2000), for example, describes a formal course evaluation process that has been successfully implemented on the Web. At the moment there is no formal course-evaluation process established at the department level. The existing one has many weaknesses and could not be used for the continuous quality improvement of the curriculum. However, the course results in terms of pass/fail ratio and success ratio were promising. There was a considerable improvement in the course when the constructivism object-first approach was adopted. The overall results for the last three years are summarized in Table II. Since courses in Greek universities have a high drop out rate for reasons that are beyond the scope of this paper, we have included in the table the number of students that actually attended the course.

<table>
<thead>
<tr>
<th></th>
<th>number of students</th>
<th>pass</th>
<th>fail</th>
<th>success rate</th>
<th>pass/fail ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spring 2002</td>
<td>84</td>
<td>65</td>
<td>19</td>
<td>77%</td>
<td>3.4</td>
</tr>
<tr>
<td>Spring 2001</td>
<td>85</td>
<td>63</td>
<td>22</td>
<td>74%</td>
<td>2.9</td>
</tr>
<tr>
<td>Spring 2000</td>
<td>90</td>
<td>61</td>
<td>29</td>
<td>68%</td>
<td>2.1</td>
</tr>
</tbody>
</table>

The organization of the course also seemed to work very well. In general the course was well received by students most of who were enthused by the new approach comparing it with the one of the preceding procedural programming course. Some of
the students from other universities who had been on traditional Java courses described the approach as “extremely exciting and much more effective”.

The use of concepts from the “Goody’s example” has a positive impact on students’ ability to learn since they understand and are familiar with the basic concepts that constitute the OO approach. It was exhilarating to see students, during the RPN case study, looking for and identifying objects such as calculator, calculatorGUI, operand, operator, adder, multiplier, divider and stack, instead of a single calculator-engine that was the case in the first version of the course. Students appear now to be familiar with the concept that systems are composed of objects that collaborate to obtain the system’s required behaviour.

Regarding the development environment, the selection of Java was well received by students. Only 3 students in the last year were negative and should prefer C++, which they were already using. However, they realized by the end of the course that they were writing mainly procedural code even using C++. Java was proved to be an easy to learn language. We were able to introduce more about programming, design, and problem solving than we could with C++. We found that Java supports the set of constructs that embody the major technical advances in programming. These advances are what we should teach to our students along with the basic modelling principles. However, the disadvantages of the language must be properly addressed. We found, for example, that it is not only difficult to introduce the event-handling mechanism of Java, during the early stages of the course, but it also has little to add to the understanding of basic OO concepts. We adopted BlueJ to help students interact with their applications during the first development phases. This allowed us to delay the introduction of interface technologies, whether text based, GUI, or applet, until a more appropriate point in the course. Moreover, we decided to provide students with the bytecode version of the calculatorGUI class, in early stages of the RPN case study. The calculatorGUI class in bytecodes, not only simplifies the development at this stage, but also gives students the satisfaction that they can write real programs similar to the highly interactive ones they use everyday. It also allows students to: (a) visualize the behaviour of their code, and (b) gain hands-on experience on event handling.

After the course, students appear to have a good and deep knowledge of the basic concepts of the new paradigm, as well as a greater understanding of Java as an environment to implement the OO conceptual framework. Both of these provide a solid point of reference for other OO environments that students may come across in the future. Actually, there is already a positive feedback from courses such as Algorithms and Software Engineering.

CONCLUSIONS
The shift from procedural to object-oriented paradigm is a difficult task. We faced the problems emanating from such a transition and we were guided to a fundamental change in the way we teach introductory object-oriented programming. Our approach was motivated by the theoretical viewpoint of constructivism that stresses the importance of the continuity of knowledge growth, based on existing knowledge. From that viewpoint, experience from every day life is used to anchor new knowledge in constructing software systems. We devised the “Goody’s example” and utilized it to create the conceptual framework of the OO paradigm. Java was adopted to introduce the constructs necessary for the realization of this framework. The idea of computation as interactions between communities of entities that constitute the system.
was adopted, versus computation as calculation that characterizes the von Neumann serial programming. We gradually moved from the traditional to the object-first approach and finally to the design-first approach. The results are so encouraging that we even wonder if such an approach should be considered as a more effective way to teach the procedural programming paradigm to novices in the first computer science course. We hope that sharing our approach, which focuses on the construction of the knowledge regarding the OO programming paradigm, should be useful for those who are currently teaching an OO programming course or those who are planning to teach such a course in the feature. Moreover, many of the concepts and guidelines presented in this paper may be expanded and applied to procedural programming courses too.

ACKNOWLEDGEMENTS

The earliest ideas for the described approach stemmed from the time I was writing the book “Programming Languages II: Object-Oriented programming” during 1997 for the Hellenic Open University. This work would not have been possible without my students. Discussions with them in classroom and in the laboratory were the main source of inspiration. Moreover, some of them have made useful contributions in the preparation of the course. I would also like to thank Nikos Karacapilidis and Vasilis Paliouras for commenting on early versions of this article.

REFERENCES


## APPENDIX A

Outline of the RPN calculator assignments

<table>
<thead>
<tr>
<th>Assignment Number</th>
<th>Material given</th>
<th>Brief description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A working RPN calculator program</td>
<td>Understand the system. Use OIDs to represent the user-system interaction. Use OIDs to represent the interaction of system’s components to obtain a system service. Object identification. Class diagram.</td>
</tr>
<tr>
<td>2</td>
<td>Java standard library</td>
<td>Utilize components of the standard library. Build their own components. Main method explained.</td>
</tr>
<tr>
<td>3</td>
<td>-</td>
<td>Class responsibilities. Test the behaviour of components.</td>
</tr>
<tr>
<td>4</td>
<td>CalculatorGUI class in bytecodes</td>
<td>Using third party classes. Using GUIs.</td>
</tr>
<tr>
<td>5</td>
<td>Class Operand in source code</td>
<td>Examine alternative designs on the complete method of Operand class.</td>
</tr>
<tr>
<td>6</td>
<td>-</td>
<td>Work with abstract classes, inheritance and polymorphism.</td>
</tr>
<tr>
<td>7</td>
<td>A limited functionality CalculatorGUI source code</td>
<td>Add extra functionality i.e. add memory support.</td>
</tr>
<tr>
<td>8</td>
<td>-</td>
<td>Hands on GUI development. Display stack status.</td>
</tr>
<tr>
<td>9</td>
<td>A sample calculatorGUI source code</td>
<td>Modifying code to increase modularity and reusability.</td>
</tr>
<tr>
<td>10</td>
<td>source code of the operatorButtonHandler(char ch) method</td>
<td>Avoid using the switch statement.</td>
</tr>
<tr>
<td>11</td>
<td>Sample source code of a working RPN calculator</td>
<td>Finding bugs. Extending system’s behaviour.</td>
</tr>
<tr>
<td>12</td>
<td>Sample source code of a working RPN calculator</td>
<td>Hands on exception handling on the RPN calculator.</td>
</tr>
</tbody>
</table>