Teaching Advanced Computing Concepts in Java: A Constructivism-based Approach

Kleanthis C. THRAMBOULIDIS
Electrical & Computer Engineering
University of Patras,
26500 Patras, Greece
thrambo@ee.upatras.gr

ABSTRACT
The belief that concurrent programming concepts are too difficult for the average programmer is one of the most important reasons for not covering concurrent programming in introductory computing courses, although much needed last years. In this paper, an approach, used to teach advanced computing concepts and especially concurrency and exception handling in introductory computing courses, is presented. The proposed approach, which is greatly influenced by constructivism, exploits the novice programmer’s existing knowledge from real-life. Multi-entity systems from every-day life are adopted and techniques used to solve real-life problems are examined and exploited in order to build the required conceptual framework on existing knowledge. The mechanisms of Java that implement these techniques are introduced and used extensively. Students found the course extremely challenging while the pass-fail ratio improved considerably.

Keywords
Teaching concurrent programming, teaching Java, constructivism-based approach, OO conceptual framework, algorithm of Dekker.

INTRODUCTION
Introductory computer courses traditionally cover neither concurrent programming nor even exception handling in a formal manner. This occurs, according to Bustard (1990), for two main reasons: the first being the lack of suitable implementation languages and the second the belief that concurrent programming concepts are too difficult for the average programmer. However, language support is no longer a problem. Java, as well as other modern Object-Oriented (OO) programming languages support the representation of concurrency and provide the appropriate mechanisms to enable the average programmer to handle it. Moreover, many educators, as for example Gelertner (1988) and Hansen (1997), argue that the latter argument i.e. that the concurrency concept is too difficult for the average programmer, is unfounded. In addition, ACM in its Curriculum 91 recommendations (ACM, 1991) advocated the introduction of distributed and parallel programming constructs into the undergraduate study curriculum.

Many researchers have already reported their positive experience from teaching concurrent programming in introductory computer courses, as for example Feldman (1997) and Kolikant (2001). Moreover, much work has been devoted to developing supporting environments (Exton, 2000; Feldman, 1990; Perski and Ben, 1998; Shene, 2002; Carr, Fang, Jozwowski, Mayo, and Shene, 2002) as well as teaching methods (Kolikant, 2001; Resnik, 1996). Ben-Ari and Kolikant (1999) reported their empirical research that carried out to study students’ conceptions and attributes and their positive experience from teaching concurrent and distributed computing to high school students. They found that both students’ work methods and conceptions evolve during the course enabling students to successfully develop algorithms and prove their correctness. At the same time, some researchers claim that the study of concurrent programming concepts constitutes an essential first step to a better understanding of programming in general.

Issues in science education, mainly constructivism, have been considered, in an attempt to improve the effectiveness of our introductory OO programming course. The constructivism-based approach that was utilized to update the course (Thramboulidis, 2003a), facilitates students in exploiting their real-life experience and building on it the conceptual framework of the OO paradigm. The use of concepts from every-day life has a positive impact on students’ ability to learn, since they understand that they are familiar with the basic concepts that constitute the OO approach.

This paper describes the use of a constructivism-based approach to teaching advanced programming concepts, such as concurrency and exception handling, which are also covered by the updated introductory course. The same real-life system
used to introduce the basic OO concepts is utilized to guide students in exploiting their existing knowledge model emanating from real-life and building on it the conceptual framework for both topics. The adopted real-life example is based on "Goody’s,” Greece’s most popular fast-food restaurant chain. 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 behavior. The “Goody’s” example is utilized to make evident to students that they are already familiar with the basic concepts of event handling, garbage collection, exception handling, and concurrency and that this experience comes from every-day life. The Olga Square Goody’s (OSG), an instance of Goody’s that was selected and used throughout the example, consists of many entities (objects) operating independently of one another, which, however, occasionally need to synchronize, communicate, or use shared resources. Concepts identified by the terms: busy-waiting, deadlock, starvation, mutual exclusion, etc., are already known to our students from every-day life. It is only the terms used to refer to as well as the mechanisms used to implement them in the programming domain that must be introduced.

The remainder of this paper first briefly presents the application of constructivism in computer science education and describes our constructivism-based approach to teaching the OO programming paradigm. Then it presents and discusses the approach used to teaching concurrent programming as well as exception handling. Next, it evaluates and discusses the proposed approach and finally the last section gives a summary of the paper.

BACKGROUND WORK

Constructivism in Computer Science Education

Constructivism, which is one of the fundamental ideas in education (Glasersfeld, 1989; Confrey, 1990; Greenco, Collins and Resnick, 1996; Kafai and Resnick, 1996; Matthews, 1998), is a philosophy of learning which claims that students construct knowledge by reflecting on their experiences, 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 non-productive. 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, and 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 (Ben-Ari, 2002). Matthews (2002) notes that the influence of constructivism has extended beyond the research and scholarly community and has had an impact on national curricular documents and national education statements such as the recently released US National Science Education Standards and New Zealand National Science Curriculum. Curricula in Spain, UK, Israel, Canada and Australia have also been influenced by constructivism in varying degree. Matthews also highlights the high hopes that are expressed for constructivism from proponents in science education, such as Mintzes and Wandersee (1998), who argue that constructivism “can serve as an alternative to the hunches, guesses, and folklore that have guided our profession for over 100 years.”

Ben-Ari (2002) surveys the theory of constructivism and its application in the context of computer science education, and shows how this theory can provide the theoretical basis required for debating issues and evaluating proposals. Ben-Ari argues that even though constructivism has been extremely influential in science and mathematics education, it has had less impact on computer science education. A number of notable exceptions to this are given and discussed by Boyle (2000) to illustrate the potential of constructivism in computer science. Logo and the CORE approach provide, according to Boyle, constructivism inspired approaches for the development of learning environments for programming. However, even though many computer science educators have been influenced by constructivism, their results have only been explicitly discussed in published works during the last few years.

Hadjerrouit (1999), in one of these works, proposes a constructivism approach for a course in object-oriented design and programming and claims that a re-examination of the teaching methods is recommended in order to promote the object-oriented approach at introductory level. He claims that the traditional approach does not focus on building strong links between the three object-oriented knowledge types that he proposes, namely: object-oriented concepts knowledge, object-oriented language knowledge and problem-specific knowledge. Hadjerrouit claims that object-oriented knowledge must be actively constructed by students and not passively transmitted by teachers, while program development must be guided by object-oriented concepts and not by language technicalities. Duggings (2002) applied constructivism to redesign an Object-Oriented Analysis and Design course to maximize its active learning aspects. Students, according to Duggings, utilize problem solving and critical thinking skills based on their prior knowledge and experience and integrate these with the new
The proposed approach consists of three stages. In the first stage, the problem is presented using a real-life case study and for the conceptual model of the OO programming paradigm. However, we were faced with the commonly referred to and now well-known problem of paradigm shift, i.e., the switch in paradigm from procedural to OO. It was found that students tried to build the concept maps of the new paradigm on structures created during the pre-existing procedural paradigm.

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 this knowledge already exists in students and emanates from real-life. The “Goody’s example” was devised to establish an alternative and more appropriate knowledge base, which when refined, would create the OO knowledge. This example is used during the first segment of the course to guide students in exploiting their prior knowledge emanating from real-life and building on it the conceptual framework of the OO paradigm. Students are forced to focus on the underlying concepts rather than on the idiosyncratic features of Java. Borstler (1999) claims that during the first weeks of the course, which are very critical, there are many aspects that must be solved to help students acquire OO thinking, apart from working in a particular OO programming language. Only when students have become familiar with the principles of the paradigm, they are able to take their attention away from the language’s definitions and focus on the essentials of the programming paradigm. This is why the first segment of the updated course addresses the conceptual framework of the OO paradigm. Simplified versions of UML use-cases, class diagrams, object interaction and scenario diagrams are used to create draft models of the “Goody’s example” to highlight the structure and behavior of the system.

In the second segment of the course, focus is shifted on the constructs of the Java environment that are necessary to satisfy the requirements of the system developer. Students are asked to adopt a “Lego construction” approach that is to first focus on the basics of using and integrating existing components and to build new ones at a later step. A set of carefully developed assignments constitute the heart of this segment of the course. Students are asked to implement their own reverse polynotic notation graphical calculator following a well-defined step-by-step development process (Thramboulidis, 20003b). The remainder of the course addresses the topics of event handling, garbage collection, exception handling and concurrency. The “Goody’s example” is utilized and an approach greatly influenced by the one used for the introduction of the basic OO concepts is used for all the above topics. Figure 1 depicts some of the slides used in the course.

A CONSTRUCTIVISM BASED APPROACH TO TEACHING CONCURRENT PROGRAMMING

Nowadays, concurrency has moved from operating systems to applications and the need for concurrent programming has moved from the small number of system programmers to the large community of application programmers. Application and embedded system programmers must be able to apply concurrent programming effectively and reliably, in order to confront the increasing number of systems that are better constructed as concurrent ones. Many software systems are simulations of real-life systems and concurrent programming is useful and, in many cases, the necessary solution because it provides a more natural mapping of real-life objects to the system’s components. The resulting representation that is more natural than the sequential one is easier to code, debug and maintain. There is an urgent need to include concurrent programming in introductory computing curricula.

Basics of Concurrent Programming

During the early development phases of our introductory course the 4 attempts provided by Ben-Ari (1990) to solve the mutual exclusion problem were presented to our students. During this time it was found that students were not able to create a solid conceptual model for concurrent programming. Only a small number of them had a clear understanding of Dekker’s algorithm and the concept of semaphore. To address this situation, a constructivism-based approach similar to the one used for the conceptual model of the OO programming was developed. It was found that much better results were obtained in improving the students’ conceptual models when the OSG case study was utilized.

The proposed approach consists of three stages. In the first stage, the problem is presented using a real-life case study and the techniques used in real-life to solve the problem are examined. The mechanisms of Java that implement the basic concepts are introduced during the second stage. The third stage is composed of a set of examples and assignments that students may
execute either in the lab or in their own time. Although, the above approach is quite similar to the one described by Ben and Kolikant (1999), more emphasis is given to students’ knowledge emanating from every-day life.

The “Goody’s example” is considered and the following situation is described. Helen and Nick are instances of the employee class in the OSG. Helen has been assigned the task of performing the process required to serve the clients’ orders that include at least one croissant. On the other hand, Nick serves the clients’ orders that include at least one cup of coffee. It is assumed that there is no client order including both croissant and cup of coffee and that there is only one microwave oven (MWO) for use by both employees. Students are asked to provide abstract descriptions of the processes executed by Helen and Nick in response to the assigned requests. Figure 2 presents example algorithms, given by students, describing the processes that are executed by Helen and Nick in response to the client orders. The resources required for the execution of the “croissant-order” are different from those required for the execution of the “coffee-order”, except from the MWO that has to be used in both types of orders.

```
(a) Coffee-order processing
get the order from the client
warm the water
make the coffee
prepare the rest order
deliver the order

(b) Croissant-order processing
get the order from the client
select croissant
warm the croissant
prepare the rest order
deliver the order
```

Figure 2. Algorithm descriptions for Nick’s and Helen’s activities

The concept of interleaving is also presented to students at this time. Students are asked to provide possible interleaving scenarios for the above case study. The majority of students work on the interleaving at the level of abstraction provided by the algorithm descriptions of Figure 2. However, a small number of them consider the interleaving at a lower level granularity description of the action “warm <thing>” that is provided in Figure 3.
warm <thing>
- put the <thing> in the microwave oven
- set the timer
- start the microwave oven
- wait until the timer expires
- take the <thing> out

Figure 3. Detailed description of the activity “warm <thing>”

Students understand that, in order for the system to operate properly, it must be guaranteed that Nick and Helen are not allowed to simultaneously execute (i.e. interleave) their subsequences of actions related to the use of the MWO, since the MWO is a resource that must be acquired for exclusive use. These actions have to be explicitly defined for each process and constitute what it is called “critical-section” of the process. The order in which the critical-sections are executed is not important, but one critical-section must be completed before the other commences. It is clear that resource management is required to ensure the exclusive use of the MWO, i.e. the system’s good operation. Students can easily understand the need for a specific set of actions that are required to be carried out before and after the critical-section to ensure the exclusive use of the MWO. Actions required before the critical-section are called pre-protocol, while those after it are called post-protocol. Figure 4 shows an abstract representation of the response of Nick and Helen to their orders. This representation includes a solution to the mutual exclusion problem and is very well understood by students.

pre-protocol
critical-section
post-protocol
remaining-section

Figure 4. Abstract representation of the solution to the mutual exclusion problem

Next, it is assumed that the manufacturer of the MWO has not confronted the mutual exclusion problem and mechanisms required to solve it are requested. Four solutions are presented and discussed in class to demonstrate possible pathological behaviors in concurrent programming. A detailed description of the proposed solutions is given in (Thramboulidis, 2000) and an on-line version in http://seg.ee.upatras.gr/OOCourse.

In the first solution that is shown in Figure 5 a cubbyhole is used to write in it the name of the entity (employee) that is using or is able to use the MWO. The following scenario is assumed for each employee. The employee has to open the cubbyhole and check if his name is written in it, i.e., if it is his turn to acquire the resource. If this is true, the resource is acquired and the actions defined as critical-section are executed. After the execution of the critical-section the employee is required to write in the cubbyhole the name of the other employee.

Figure 5. First solution to the mutual exclusion problem of the OSG example

Students are requested to describe in Java pseudo-code the response of the employee class to the coffee or croissant order according to the proposed scenario. Figure 6 presents the prepareOrder method of the employee class in Java pseudo-code. Some students model the cubbyhole in a better way providing to the cubbyhole class methods for check and update. Pre-protocol, critical-section and post-protocol are clearly identified in this description.

During class discussion, students are asked to decide whether this algorithm is correct i.e. whether the system reliably performs its services. The two kinds of correctness properties, i.e. safety, which deals with getting the right answer and
liveness, which deals with the rate of progress of a process (Sethi, 1996) have to be introduced at this point. The solution is discussed to resolve that Nick and Helen can not be in their critical-section simultaneously, which means that mutual exclusion, the most common safety property, is satisfied. The terms deadlock-free and livelock-free as important properties for liveness have to be introduced at this point of the course since these concepts have no equivalent in sequential programming. However, students have already a clear understanding of these concepts from every-day life.

```java
prepareOrder( ){
    while(commonCub.turn == this.otherEmpl);  //pre-protocol
    this.useMWO();                             //critical-section
    commonCub.turn = this.otherEmpl;          //post-protocol
    this.prepareRemOrder();                   //remaining-section
}
```

Figure 6. First solution to the mutual exclusion problem of the OSG example

Students are next requested to describe the behavior of the system in the following two special cases:

(a) Helen is an efficient employee while Nick is leisurely, and
(b) Nick suddenly leaves the OSG and never comes back.

The first case is used to highlight the influence of the proposed algorithm to the system’s performance, while the latter is used to show that the proposed solution is not deadlock-free.

Next, in an attempt to provide a more reliable solution for the OSG system, the use of two cubbyholes, one for each employee instance is proposed, as shown in Figure 7. Each cubbyhole contains a flag which shows that the corresponding employee instance is using or is going to use the MWO. Nick’s behavior is modified so as to first check Helen’s cubbyhole. If the flag in Helen’s cubbyhole is raised, Nick continues to check it until the flag goes down. If the flag is down, Nick raises the flag of his cubbyhole and proceeds to his critical-section. He has to turn down the flag in his cubbyhole when he exits his critical region (post-protocol). The Java-like pseudo-code for this behavior is given in figure 8.

```java
prepareOrder( ){
    while(this.otherEmpl.cub.cs);
    this.cub.cs = true;
    this.useMWO();                             //critical-section
    this.cub.cs = false;
    this.prepareRemOrder();                   //remaining-section
}
```

Figure 7. Second solution to the mutual exclusion problem of the OSG example

The Java-like pseudo-code for this behavior is given in figure 8.

Students are asked to check the proposed solution by dramatizing several scenarios of the algorithm. One of them plays the role of Nick, another plays the role of Helen, and a third student plays the role of the scheduler. It is important to highlight the role of the scheduler who arbitrarily interleaves the actions of Nick with those of Helen. Students conclude that in most cases
the algorithm works but sometimes it fails. Always, in every class, some students find that there is a scenario that allows both Nick and Helen to execute their critical-section at the same time i.e. to violate mutual exclusion. Students are already accustomed with such situations from every day life but now they use the proposed term to refer to it. As Ben and Kolikant (1999) claim, students have to understand that “a correct algorithm must give the correct output not only for every input, but also for every possible scenario on those inputs. They have to understand that having the program computing the required result in one scenario does not ensure that all scenarios will produce the same result. Correctness is a theoretical issue rather than a lab assignment.” The existence of even one scenario that violates mutual exclusion is enough to abandon the proposed solution. This is why a new solution with slightly modified behavior for Nick is proposed. According to it, Nick is requested to first raise the flag in his cubbyhole, in order to indicate his intention to proceed in his critical region and, then, check the flag in Helen’s cubbyhole. Students have to check if the proposed solution satisfies the requirement of mutual exclusion. They identify the existence of at least one scenario that leads the system to deadlock. This leads them to the definition of an informal method of proving that an algorithm is wrong. It is enough to identify a falsifying scenario.

The Algorithm of Dekker

One more solution, with a slightly different behavior for Nick, is proposed and discussed in class to demonstrate livelock, i.e. the case where actions of pre-protocol are performed by both Nick and Helen but neither of them enters its critical-section. Students now have the required background to examine the final solution that represents the algorithm of Dekker. A third cubbyhole, called turn, to be used by both employees is utilized as shown in Figure 9 to present the solution. Nick is requested to first raise the flag in his cubbyhole, in order to indicate his intention to proceed in his critical region and, then, check the flag of Helen’s cubbyhole. If the flag in Helen’s cubbyhole is down, Nick proceeds to his critical-section, otherwise he checks the common cubbyhole. If it is his turn to persist, he periodically checks Helen’s cubbyhole, otherwise he has to lower the flag in his cubbyhole to allow Helen to proceed and avoid livelock.

![Figure 9](image-url)

It was very exiting to see that the majority of students have no difficulties in understanding this complex algorithm. However, most of them had difficulties in their attempts to describe using Java-like pseudo-code the prepareOrder method an example version of which is given in Figure 10. Even more, students are disappointed thinking that this is the way they should cope with concurrency. But this proved to be a very good starting point for our dentist example to follow.

The Dentist Problem: Semaphore, the Redhead Secretary

In order to introduce the concept of semaphore, a solution is requested from students to our dentist problem, which includes one dentist and 12 weaklings. Weaklings are very busy with a lot of activities. However, from time to time they have to visit the dentist, who may see only one weakling at a time.

Students are asked to come up with a solution to this problem using the already presented material. They soon realize that it is impossible to apply Dekker’s algorithm and that it is impossible to write a correct algorithm with the tools they have at their disposal. However, they already know that their dentist has solved the problem many years ago and that, s/he does not know anything about the algorithm of Dekker. The dentist uses a secretary to guarantee the system’s correctness i.e. solve the mutual exclusion problem and guarantee the system’s liveness. Each weakling that needs to be seen by the dentist has to go into the waiting room i.e. the secretary’s room, and ask the secretary for permission to go into the service room, i.e. the dentist’s room. The task of the secretary is very simple; s/he has to open the drawer and check the flag that represents the availability of the dentist. If the flag is raised, which means that the dentist is available, permission to go in is given and the flag is lowered. Otherwise, if the flag is already down, which means that the dentist is not available, the weakling is informed.
to wait for the dentist to become available. Even more, since it is very annoying for the weaklings in the waiting room to keep
asking if it is possible to go in, the weaklings are allowed to take a cat-nap until it is their turn to be seen by the dentist. This
last modification in behavior was proved an excellent example that helps students understand clearly the issue of busy-
waiting. The weakling after having been seen by the dentist has to notify the secretary while leaving. The secretary either wakes up
the next weakling if at least one is waiting, or raises the flag in the drawer if there is no weakling waiting. A graphical Java
application is utilized to demonstrate the described operation of the dentist system in class.

Students are asked to express the behavior of weaklings in Java pseudo-code. They are subsequently told that Dijkstra’s
semaphore is nothing more than the redhead secretary of our dentist example. The behavior of the secretary in the case where
more weaklings are waiting is examined, and starvation as a possible situation is considered. The following situations are also
examined:

(a) Two or more weaklings concurrently request permission to go in.
(b) A weakling that has just been seen notifies the secretary about it, while at the same time another weakling is asking
for permission to go in.

A service room with more than one dentist is considered and examined, to introduce the concept of the general-semaphore. In
this case the scenario that more than one weaklings can simultaneously notify the secretary that they have just been seen must
be considered.

At the next stage, students are asked to solve in class the MWO problem of OSG using the mechanism of semaphore. In
Figure 11 a description for the prepareOrder method that was given by students is shown. The strength of the new mechanism
becomes very clear to them.

```
prepareOrder( )
{
    this.cub.cs = true;
    while(this.otherEmpl.cub.cs)
        if(commonCub.turn = = otherEmpl) {
            this.cub.cs = False;
            while(commonCub.turn = = otherEmpl);
            this.cub.cs = True;
        }
    this.useMWO();                       //critical-section

    secretay.turn = otherEmpl;
    this.cub.cs = False;
    this.prepareRemOrder();             //remaining-section
}
```

Figure 10. Pseudo-code for the prepareOrder method when applying the algorithm of Dekker

Students are next asked to consider the case of a weakling that is going into the dentist’s room without first acquiring
permission from the secretary, as well as the case of a weakling leaving the dentist’s room without notifying the secretary. It
is easy for students to identify that our system is susceptible and its liveness properties are not well conserved in such cases.
A solution to this problem is requested in class. Some students propose forcing every weakling to follow the “rules of the
game”. A “guard” is introduced into the waiting room to enforce the rules. This is the way the concept of monitor is
introduced as a more reliable mechanism to solve the mutual exclusion problem.

Students are next asked to identify real world systems that involve concurrency and study the way that correctness is
obtained. They have to describe these systems using the recently introduced technical terms. A number of systems are also
given to students and they are asked to come up with solutions. It is essential for students to recognize where mutual
exclusion is needed and to control it accordingly. Since the language mechanisms are not known yet, students are forced to
work at a conceptual level to provide the algorithms. It is necessary for them to get accustomed to working on this level, before considering the actual implementation.

After the introduction of the conceptual model of concurrent programming, where emphasis is given to the underlying concepts involved, the terminology used, and the studying of a number of problems, the basic mechanisms of Java that implement this conceptual model are introduced. Students are ready to be exposed to the details of Java’s concurrency mechanism. Finally, a set of examples and assignments allow students to practice the new concepts.

TEACHING EXCEPTION HANDLING

For the presentation of the basic concepts of exception handling the following scenario from the “Goody’s example” is considered. Helen has just accepted the message “a sandwich with cheese and ham please.” Helen carries out the actions required to prepare the sandwich. The response of Helen to this event is predictable since she was given instructions on how to handle it.

Students are asked to consider the case in which there is no cheese slices available. Even though this is an unusual event, i.e. an exceptional case, there is still a possibility, even small, for this to occur. The term exception is introduced to refer to the exceptional case and students are asked to describe the behavior of Helen to this exception. It is clear to all of them that if Helen was not informed on how to identify and handle this exception, the system will toggle to a state of undefined behavior. It is next assumed that Helen is informed as to how to identify this exception and the following two cases are considered:

(a) Helen has the knowledge to carry out a set of actions to handle this special event, and
(b) Helen has no knowledge of the actions required to cope with this special event.

In the first case Helen identifies the exception, catches the exception, and executes the set of actions required to cope with it. There is no need to notify the client or her supervisor about the exception. In the latter case, Helen only has the knowledge required to identify this special event but since her supervisor has not informed her how to handle it, she only has to pass the required information to another entity to cope with it. This entity should catch this information, the exception object, and execute the required actions.

Both cases are examined, and terms such as exception, exception type, identify the exception, create an exception, throw the exception, cope with the exception, etc. are introduced and discussed in class with students. Questions that are given to students include the following:

- Which entity should Helen notify about the exception?
- Is it her supervisor or is it the entity that has assigned the job, or another entity of the system?
- Does Helen continue her job after the identification of the exception or does she have to wait for the exception to be handled by someone else?

These questions make clear to our students that exception handling is a complex problem with many parameters.

Students are next asked to describe systems from every day life where exception handling is present. They have to identify the exception handling mechanisms and the policies used. They are subsequently asked to consider the procedural programs they have developed so far and identify similar situations. The question is “how have you as a programmer been handling the exceptions so far?”

After the introduction of the conceptual framework of exception handling, students are ready to be exposed to the details of Java’s exception handling mechanism. Finally, a set of assignments from the ones described in (Thramboulidis, 2003b) allow students to gain hands-on experience of exception handling.

EVALUATION

Students liked the approach and they found it extremely challenging. They also liked the fact that case studies are taken from real-life and felt that the course contributes to their understanding of computerized systems. At the end of the course students have understood the need for design. They learn to recognize when and how concurrency can be used in the solution of a software design problem. Some of them are becoming proficient in writing simple concurrent programs in the Java programming language.

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 structures they had learned while using the MWO problem. They were able to identify the need for concurrency, to identify critical-sections and
to use semaphores and monitors to provide solutions to mutual exclusion problems. Students were also able to utilize the syntactic elements and the concepts of Java in concurrency in order to express their solutions using JDK or the development environment of their choice.

The approach described in this paper has been used as the major vehicle for teaching advanced programming concepts to: (a) novices, (b) students already exposed to procedural programming and C, and (c) postgraduate students experienced in other programming languages. With regard to the acceptability of the approach, even though an extended assessment has not been conducted, students of all categories accepted the new approach with a lot of positive remarks.

To better understand the students’ perspective on the approach, a questionnaire was given to them in courses of the last two of the above categories. The questionnaires were given during the semester and just after the completion of the part that deals with the introduction of exception handling and concurrent programming. Unfortunately since there is no formal course-evaluation process established at department level at the moment students are not accustomed to this process and a small number of questionnaires were returned fully completed. Some 96% of the students found the approach “very useful” overall, while the others found it “mostly useful”, with 0% selecting “useless” i.e. the low mark. The dentist problem proved, according to the questionnaire an excellent problem to present and discuss the concept of semaphore and terms such as busy-waiting. However, about 58% of the class suggested that it would be better if more lecture time was allocated to the part of the course that dealt with concurrent programming and event handling.

Another quite interesting result is that showing how useful the approach has been in helping students to understand and properly use the quite complex event handling mechanism of Java. In the question “How useful was the approach 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 who reported “useless”) understood and used the mechanism effectively as it was corroborated in the final assignments and projects. This can be explained by the fact that students had not been exposed to another approach used to introduce this mechanism so as to be able to compare with. Four students who stated they had a “good” knowledge of Java prior to taking the course were impressed by the way that event handling, exception handling and concurrent programming were simplified.

It must be noted that the evaluated courses are elective with students being high achievers in programming or having decided to become better in this topic. The whole course has been experimentally evaluated by the author and the results of this evaluation were very encouraging (Thramboulidis, 2003a). There was a general understanding that the course, compared with the first phase of its development, has vastly improved.

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

SUMMARY

A constructivism-based approach to teaching the object-oriented programming paradigm was developed. A real-life system was adopted to introduce its conceptual framework. A similar approach was utilized to teach advanced programming concepts such as garbage collection, event and exception handling and concurrent programming. The use of students’ existing knowledge as an anchor on which to build the conceptual framework for these topics proved effective. The use of the OSG case study has resulted in much better results in improving the students’ conceptual models. It was quite interesting to see that the majority of students had no difficulty in understanding even Dekker’s complex algorithm. The use of the redhead secretary metaphor proved very effective in understanding the concept of semaphore and terms such as busy-waiting, starvation, etc. In general, it seems that our students have improved their ability to solve problems using OO techniques, apply event handling, exception handling and concurrency and to implement their designs effectively using Java language and Java API by the time they finished the updated course. The software designed and implemented by them is easier to code, debug and extend.

The first results of this approach are very promising. It was found that students’ conceptions evolved during the course to the point that they were able to confront by the end of the course the requirements imposed by today’s demanding applications concerning the issues of concurrency, exception and event handling. Students found the course extremely challenging and the pass-fail ratio improved considerably.

ACKNOWLEDGEMENTS

The author is grateful to the anonymous referees for the numerous remarks and suggestions that have lead to significant improvements of this paper. This work would not have been possible without his students. Discussions with them in class and in the laboratory were the main source of inspiration.
REFERENCES


