Teaching Advanced Programming Concepts in Introductory Computing Courses: A Constructivism Based Approach

Author:

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

Abstract *— Teaching object-oriented programming in introductory computer courses is still an area not well understood by instructors and teachers. A new approach, quite different from the one used to teach the procedural paradigm, is required. We have developed and used for several years a teaching approach that is greatly influenced by constructivism, which stresses the importance of prior knowledge on top of which new knowledge is built. A real-life system was adopted, to exploit the prior knowledge that students have from every-day 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 a design-first approach. We also recognized the need imposed by the complexity and the event driven nature of today's applications, for improved techniques and mechanisms concerning exception handling, garbage collection and concurrency. In this paper we describe the constructivism-based approach that we use to teach exception handling and concurrency in introductory computer courses as well as our experience from using this approach. The first results of this approach are very promising. We 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 exception handling and concurrency. Students found the course extremely challenging and the pass-fail ratio was improved considerably.*

Index Terms *— Constructivism in education, teaching exception handling, teaching concurrency, semaphore.*

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. It 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. In [16] we presented our constructivism-based approach that we have developed in the context of this course. Applications of constructivism in education lie in creating learning environments or curricula that match students' understanding, fostering further growth and development of mind. The proposed approach facilitates students in exploiting their real-world experience and building on it the conceptual framework of the object-oriented paradigm.

In this paper we describe the use of the same approach to teach advanced programming concepts such as exception handling and concurrency, which have been also addressed by the updated course. We utilize the same real-life system used to introduce the OO concepts, to guide our students in exploiting their existing knowledge model emanating from real-life and building on it the conceptual framework for both topics. Our real-life example is based on "Goody's," Greece's most popular fast-food restaurant chain. 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 behavior. We utilize the "Goody's example" to make evident to our students that they are already familiar with the basic concepts of exception handling, garbage collection and concurrency and that this experience comes from every-day life. The Olga Square Goody's (OSG), i.e. the specific instance of Goody's that is used in our example, consists of many entities (objects) operating independently of one another, needing occasionally 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. Students have a good understanding of these concepts. Only the terms used to refer to as well as the mechanisms used to implement these concepts in the programming domain must be introduced.

Traditionally, introductory computer courses do not cover concurrent programming not even exception handling in a formal manner. This avoidance of a concurrent program representation has occurred according to Bustard for two main

reasons [5]. The first was the lack of suitable implementation language for the application concerned; the second was the belief that the concurrency concept is too difficult for the average programmer. However, language support is no more a problem. Java as well as other modern programming object-oriented languages, supports the representation of concurrency. It provides the appropriate mechanisms for the average programmer to handle concurrency. Moreover, many educators, as for example Gelertner and Hansen, argue that the latter argument i.e. that the concurrency concept is too difficult for the average programmer, is unfounded [10][11]. Also ACM in its Curriculum 91 recommendations advocated the introduction of distributed and parallel programming constructs into the undergraduate study curriculum. Many software systems are simulations of real-life systems, and concurrent programming is useful, in many cases the "must" solution, because it provides a more natural mapping of the real-life objects to system's components. The resulting representation that is more natural than the sequential one is easier to code, debug and maintain.

Nowadays, concurrency has moved from operating systems outward to applications and the need for concurrent programming has moved from the small number of systems programmers to the much larger community of applications programmers. Application programmers should apply concurrent programming effectively and reliably in order to confront the increasing number of systems that are best coded as concurrent ones. Many researchers have already reported their positive experience from teaching concurrent programming in introductory computer courses [8] [12], and much work has been devoted to developing supporting environments [6] [7] [9] [13] [15] as well as teaching methods [12] [14]. Ben-Ari and D. Kolikant in [2] reported their positive experience from teaching concurrent and distributed computing to high school students and their empirical research that was done to study students' conceptions and attributes. They found that both conceptions and students' work methods evolve during the course to the point that they were able to successfully develop algorithms and prove their correctness. Even more some researchers claim that the study of concurrent programming concepts will become an essential first step in better understanding programming in general.

The remainder of this paper is organized as follows. In the next section, we briefly refer to our constructivism-based approach to teach the object oriented programming paradigm. In section 3, the approach used to teach concurrent programming is presented and discussed in detail. In section 4 we describe our approach to teach exception handling. Finally in the last section we summarize the paper.

BACKGROUND WORK: OUR CONSTRUCTIVISM-BASED APPROACH TO TEACH OO

In this section we briefly present the outline of our constructivism-based approach that we use to teach object oriented programming in introductory computer courses. The focus of the corresponding course is on teaching the basic concepts of the OO programming paradigm rather than a specific programming language. A detail description of this work is given in [16].

When planning the course in 1996, we selected Java as the language to be used for the introduction of the OO programming paradigm and we adopted the traditional approach, which is followed by the majority of textbooks on object-oriented programming. However, 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. We found that students tried to build the concept maps of the new paradigm on structures created during the pre-existing procedural paradigm. 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-life. 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. This example is 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.

The proposed approach ensures that students should focus on the underlying concepts rather than on the idiosyncratic features of the language. Borstler in [4] 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, besides working in a particular OO programming language. Only when students have become familiar to the principles of the paradigm they can take their attention away from the language's definitions and focus on the programming assignments. This is why the first segment of the updated course addresses the conceptual framework of the OO paradigm. The "Goody's example" is adopted, and simplified versions of use-cases, class diagrams, object interaction, and scenario diagrams are used to create draft models to highlight the structure and behavior 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.

We next, adopt a "Lego construction" approach, and ask students to first focus on the basics of integrating existing components and later on building new ones. In the second segment of the course we focus on the constructs of the Java environment that are necessary to satisfy the requirements of the system developer. A set of carefully developed assignments, constitute the heart of this segment of the course. Students are asked to implement their own reverse polish notation graphical

calculator following a well-defined step-by-step development process [18]. The remainder of the course addresses the topics of exception handling and concurrency. We use for both topics the "Goody's example" and almost the same approach with the one used for the introduction of the basic OO concepts.

TEACHING THE BASICS OF CONCURRENT PROGRAMMING

In this section we describe how we teach the basics of concurrent programming. During the early phases of the course development we were presenting to our students the 4 solutions provided by Ben in [1]. However, much better results were obtained in improving the students' conceptual models when the OSG case study was utilized. Our updated approach includes three stages. In the first stage, the problem is presented using a real-world case study and the techniques used to solve the problem are examined; the mechanisms of Java that implements 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 lab or at their own time. Although, the approach used is quite similar to the one described in [2], 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, in the OSG, has been assigned the task of performing the process required to serve the client requests that include at least one cup of coffee. Nick on the other hand serves the client requests that include at least one croissant. We assume that there is no client request including both croissant and cup of coffee. We ask from students to provide abstract descriptions of the processes executed by Helen and Nick. In Figure 1(a) we present example algorithms describing the processes that are executed by Helen and Nick. 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 microwave oven (MWO) that has to be used during the preparation of both orders. Since the MWO is a resource that must be acquired for exclusive use by each process, resource management is required to ensure the system's good operation. The concept of interleaving is presented to students at this time and they are asked to provide possible interleaving scenarios for our case study. The majority of students work for the interleaving in the level of abstraction provided by the algorithms of figure 1(a), but some of them consider the interleaving in a lower level granularity description of the action "warm something" that is provided in figure 1(b). The concept of critical region is well understood by students as well as the abstract representation of the mutual exclusion problem. Students see the need for a specific set of actions to ensure the exclusive use of the microwave oven.

We next assume that the implementer of the MWO has not confronted the problem and we are looking for the mechanisms required to solve it. A cubbyhole was used as is shown in figure 2 and the name of the process that is using or is able to use the MWO is written in it. The following protocol is assumed. Each process has to open the cubbyhole, and check if its name is written in it, i.e., if is its turn to acquire the resource. If this is true, the resource is acquired and the critical region is executed. At the end of the critical region the process is required to write in the cubbyhole the name of the other process.

During class discussion, students are asked to decide whether this algorithm is correct or not i.e. whether the system performs reliably its services. The problem is discussed to resolve if the requirements of mutual exclusion are satisfied and correctness is obtained. The terms deadlock-free and starvation free for correctness properties 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. Students are next requested to describe the behavior of the system in the following two special cases:

a) Mary is a whippy employee while Nick is a leisurely one, and
b) Nick suddenly walks out from the OSG and never comes back.

The first case is used to highlight the influence of the proposed algorithm to system's performance, while the later case is used as an example of deadlock.

Next, in our attempt to provide a more reliable solution for the OSG system, we propose the introduction of two cubbyholes, one for each employee. Each cubbyhole contains a flag that shows that the corresponding process is using or is going to use the microwave oven. We modify Nick's behavior so as to first check the Helen's cubbyhole and next, if the flag is down, to raise the flag of its cubbyhole and proceed to his critical region. If the flag in Helen's cubbyhole is raised, Nick continues to check it until the flag is going down. We also request from Nick to turn down the flag in his cubbyhole when he exits his critical region. Students are requested 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 that arbitrarily interleaves the actions of Nick with those of Helen. Students conclude that in most of the cases the algorithm works but sometimes it does not. Always, in any class, some students find that there is a scenario that violates 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 claim in [2], 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 we proceed and propose a new solution with slightly modified behavior for Nick. We request from Nick first to raise the flag of his cubbyhole to indicate his intention to proceed in his critical region and next to check the flag of Helen's cubbyhole. We request from students to check if the proposed solution satisfies the requirement of mutual exclusion. They identify the existence of one at least scenario that leads the system in deadlock. This leads students to the definition of an informal method of proving that an algorithm is wrong. They only have to identify a falsifying scenario.

We next propose a slightly different behavior for Nick and we ask students to identify scenarios that lead Nick and Mary in starvation. We finally proceed to the discussion of the solution presented in figure 3 that is the algorithm of Dekker. We were very exited to see that the majority of our students have no difficulty in understanding this complex algorithm. However, they had difficulties in their attempt to describe the algorithm using C-like pseudo code. Even more, thinking that this is the way they should cope with concurrency disappoints them. But this is a good starting point for our dentist example to follow. A detailed description of the proposed solutions is given in [17] and an on-line version in http://seg.ee.upatras.gr/OOCourse.

The dentist problem: Semaphore-the redhead secretary

We next ask from students to provide a solution to the dentist problem. Our dentist problem includes one dentist and 12 weaklings. Each weakling is very busy with a lot of activities. However, there is a need from time to time to visit the dentist. The dentist may serve only one weakling at any time. Students are asked to express a solution to this problem using the already presented material. Students see that with the tools they have in their disposal, it is impossible to write a correct algorithm; they realize that it is impossible for them to apply Dekker's algorithm. However, they already know that their dentist has solved the problem many years ago and even more he/she does not know anything about the algorithm of Dekker. He uses a secretary to solve the mutual exclusion problem and guaranty the system's correctness. Each weakling that needs to be served by the dentist has to go in the secretary's room and ask from the secretary the permission to pass in the dentist room. The task of the secretary is very simple; the permission to pass is given if the dentist is available, otherwise the weakling has to wait in the sofa for the dentist to be available. Our redhead secretary has to open the cubbyhole to check for the flag that represents the availability of the dentist. If the flag is raised (the dentist is available) the permission to pass is given and the flag is taken down; otherwise if the flag is down (the dentist is not available), the weakling is informed to wait in sofa for the dentist to be available. Even more since it is very annoying both for the weakling and for our secretary to always be asked from the weakling if it is possible to pass, the weakling is allowed to take a cat-nap until his/her turn to be served by the dentist should come. This was proved an excellent example for students to very well understand the issue of busy waiting. The weakling has to notify the secretary when leaves the dentist room. The secretary either wakes up the next weakling if one exists; or raises the flag in the cubbyhole if there is no weakling waiting in sofa.

Students are next asked to express the solution for the dentist problem in pseudo code. They next are told that Dijkstra's semaphore is nothing more than the redhead secretary of our dentist example. The dentist problem was proved an excellent problem to present and discuss the concept of semaphore and terms such as busy waiting, etc. The behavior of our secretary in the case where more weaklings are waiting in sofa is examined. Starvation is a possible situation and must be considered. We have also to consider the behavior of the secretary in the following situations:

- Two or more weaklings concurrently request permission to pass.
- A weakling that has just been served notifies the secretary about it while at the same time another weakling is asking for a permission to pass.
- More than one, weaklings notify the secretary that they have just been served (if possible).

The service room with more than one dentist is introduced and examined, to introduce the concept of general-semaphore. Then in a next stage students are asked in classroom to solve the MWO problem of OSG using the mechanism of semaphore. The strength of the new mechanism is very clear to them.

We next ask students to consider the case of a weakling that is passing in the dentist room without first acquiring the permission from the secretary, as well as the case of a weakling leaving the dentist room without notifying the secretary. It is easy for them to identify that our system is susceptible and its liveliness properties are not well conserved. A solution to this problem is requested in classroom. Some students propose to enforce every weakling to follow the rules of the game. A "guard" is introduced in the secretary room to enforce every weakling to follow 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 are requested to describe their behavior using the recently introduced technical terms. A number of systems are given to students and solutions are required for them. 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 in the conceptual level to provide the algorithms. This is what they should learn to do, before considering the actual implementation.

It is clear that during this part of the course emphasis is given to the terminology used and the underlying concepts involved. After the introduction of the conceptual model of concurrent programming and the confrontation 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.

Students liked the course. They found it extremely challenging. They also liked the fact that the case studies are 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 become proficient in writing simple concurrent programs in Java programming language.

TEACHING EXCEPTION HANDLING

For the presentation of the basic concepts of exception handling we consider the following scenario from the "Goody's example." Helen has just accepted the message "a toast with cheese and ham please." The response of Helen to this event is well known since she was given instructions on how to handle it. Helen makes the actions required to prepare the toast. However, lets assume that there is no cheese chop available. Even thought this is an unusual event, there is still a possibility, even small, for this to occur. This event is an exceptional case; it is an exception. We ask from students to consider the behavior of Helen to this situation. 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. We next assume that Helen was informed on how to identify this exception and we consider the following two cases:

(a) Helen has the knowledge to make 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, we say that Helen identifies the exception, catches the exception, and makes 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 later case, Helen has only the knowledge required to identify the special event but since her supervisor did not informed her (for any reason) on how to cope with it, she only has to pass the required information to another entity to cope with this special event. This entity should catch this information (the exception object) and make the required actions.

We discuss both cases in class with students and we introduce in the discussion terms such as exception, exception type, identify the exception, create an exception, throw the exception, cope with the exception, etc. Questions that are given to students include the following ones:

- What is the entity that Helen should notify about the exception?
- Is it her supervisor or is it the entity that has assigned the job, or a special entity of the system?
- Does Helen continue her job after the exception?

These questions make clear to our students that the 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 are asked to identify the exception handling mechanisms and the policies used. They are next asked to consider their procedural programs they have developed so far and identify similar situations. The question is "how did you as a programmer were 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 examples and an assignment from the ones described in [18] allow students to gain hands-on experience on exception handling.

CONCLUSIONS

We have developed a constructivism-based approach to teach the object-oriented programming paradigm. A real-life system was adopted to introduce its conceptual framework. We are using the same approach to teach advanced programming

concepts such as exception handling and concurrency. The use of students' existing knowledge as an anchor on which to built the conceptual framework for both topics was proved effective. Much better results in improving the students' conceptual models were obtained when we introduced the use of the OSG case study. It was quite interesting to see that the majority of our students have no difficulty in understanding even Deker's complex algorithm. The use of the redhead secretary metaphor proved very effective in understanding semaphore's concepts and terms such as busy waiting, etc. The selection of Java as our programming environment in teaching object-orientation proved successful for both exception handling and concurrency. In general it seems that our students when they finish the new course have improve their ability to solve problems using OO techniques, apply exception handling and concurrency and to implement the design effectively using the Java language and the Java API. The designed and implemented by them software is easier to code, debug and extend. This is very important as a first step towards teaching how to develop software that should be reliable and less costly to maintain over its application lifetime.

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.

REFERENCES

[1] Ben, Ari, M., *Principles of Concurrent Programming*, Prentice Hall International, 1982.

[2] Ben, Ari, M., Kolikant, David, Y., "Thinking Parallel: The Process of Learning Concurrency," *Proceedings of the Fourth Annual SIGSCE/SIGCUE Conference in Computer Sceince Education*, ACM, Cracow, Poland, 1999, pp. 13—16.

[3] Ben, Ari, M. *Principles of Concurrent and Distributed Programming*, Prentice-Hall International, 1990.

[4] Borstler, J. and A. Fernandez, "Quest for Effective Classroom Examples," *OOPSLA'99 Workshop Report*, Sweden. 1999.

[5] Bustard, David, W., "Concepts of Concurrent Programming," *SEI-CM-24*, Carnegie Mellon University, Software Engineering Institute, April 1990.

[6] Carr Steve, Fang Changpeng, Jozwowski Tim, Mayo Jean and Shene Ching-Kuang, "A Communication Library to Support Concurrent Programming Courses," *Proceedings of SICCSE '02*, February-March 2002, Covington, Kentacky, USA, pp 360-364.

[7] Exton, Chris, "Elucidate: A Tool To Aid Comprehension of Concurrent Object Oriented Execution," *Proceedings of Fifth SIGCSE/SIGCUE Conference on Innovation and Technology in Computer Science Education*. Helsinki, Finland, 2000, pp. 33-36.

[8] Feldman, Michael, B., "Concurrent Programming CAN be Introduced into the Lower-Level Undergraduate Curriculum," *2nd Annual Conference on Integrating Technology into Computer Science Education*, Uppsala, Sweden, June 1997.

[9] Feldman, Michael, B., "Language and System Support for Concurrent Programming," *SEI-CM-25*, Carnegie Mellon University, Software Engineering Institute, April 1990.

[10] Gelertner, D., "Parallel Programming: Experiences with Applications, Languages and Systems," *SIGPLAN Notices 23*, 9 (Sept. 1988) ACM/Sigplan PPEALS.

[11] Hansen, Brinch, P., *The Architecture of Concurrent Programs.* Englewood Cliffs, N.J.: Prentice Hall, 1977.

[12] Kolikant, David, Y., "Gardeners and Cinema Tickets: High School Students' Preconceptions of Concurrency," *Computer Science Education*, 2001, Vol. 11, No. 3, pp 221-245

[13] Perski, Y., and Ben-Ari, M. "Re-engineering a Concurrency Simulator," *Proceedings of the Third SIGCSE/SIGCUE Conference on Integrating Technology into Computer Science Education*, Dublin, Ireland, 1998, pp. 185-188.

[14] Resnick, Michael, "Beyond the Centralized Mindset," *Journal of the Learning Sciences*, vol. 5, no. 1 1996, pp.1-22.

[15] Shene, Ching-Kuang, "ThreadMentor: A System for Teaching Multithreaded Programming," *Proceedings of ITiCSE'02*, June 24-26, 2002, Aarhus, Denmark. Pp. 229

[16] Thramboulidis, Kleanthis, C., "Teaching Programming in Introductory Computing Courses," *Journal of Informatics Education and Research*, 2003, (in press).

[17] Thramboulidis, Kleanthis, C., *Teaching Advanced Programming Techniques*, (in Greek) University of Patras Press, Patras Greece 2000.

[18] Thramboulidis, Kleanthis, C., "A Sequence of Assignments to Teach Object-Oriented Programming," *ACS/IEEE International Conference on Computer Systems and Applications, Workshop on Practice and Experience with Java Programming in Education*, July 18, Tunis, Tunisia 2003.

FIGURES AND TABLES

FIGURE. 1
ALGORITHM DESCRIPTIONS FOR NICK'S AND HELEN'S PROCESSES.

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

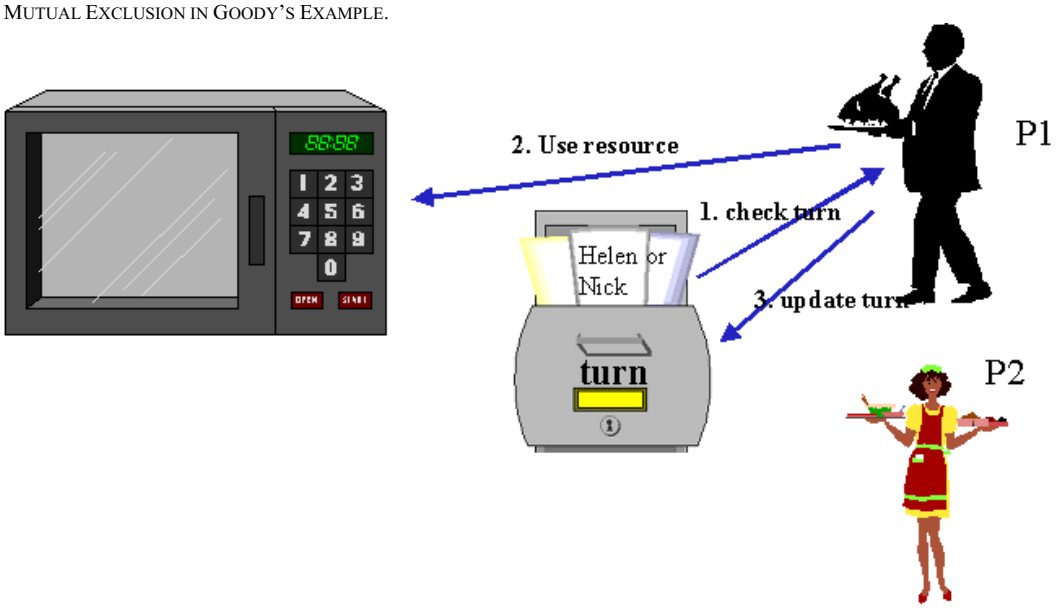
Coffee-order processing
get the coffee-order from the client
(warm the water)
make the coffee
prepare the rest order
deliver the order

(a)

warm <something>
put the thing in the microwave oven
set the timer
start the microwave oven
wait until the timer expires
put the thing out

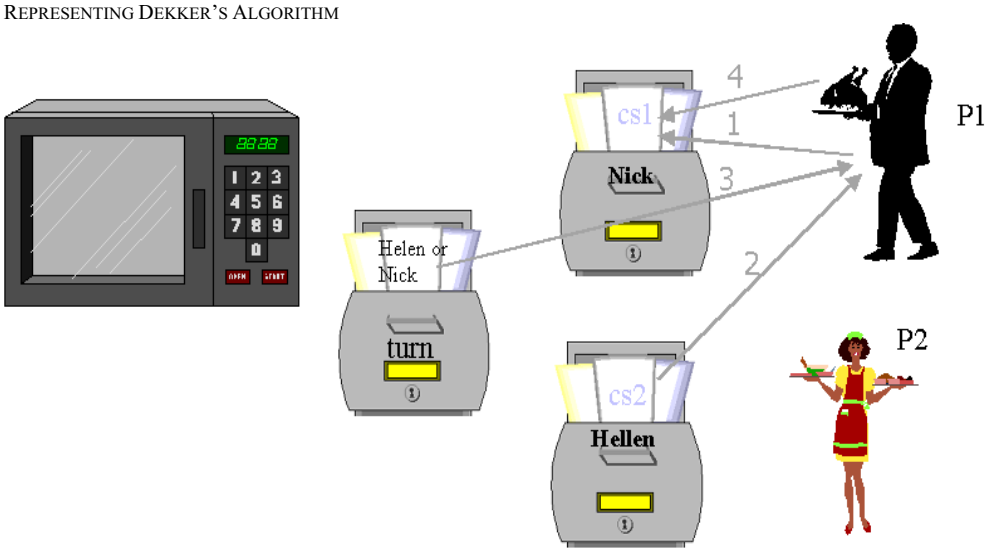
(b)

FIGURE. 2
MUTUAL EXCLUSION IN GOODY'S EXAMPLE.




```
P1()
{
while(TRUE) {
    while(turn = = 2);
    critical_section1();
    turn = 2;
    rem_section1();
    }
}
```

FIGURE. 3
REPRESENTING DEKKER'S ALGORITHM




```
P1( )
{
while(TRUE) {
    cs1 = TRUE;
    while(cs2)
        if(turn = = 2) {
            cs1 = FALSE;
            while(turn= =2);
             cs1 = TRUE;
            }
    critical_section1( );
    turn = 2;
    cs1 = FALSE;
    rem_section1( );
    }
}
```