World Transactions on Engineering and Technology Education *© 2012 WIETE*
Vol.10, No.2, 2012

A cooperative development system for an interactive introductory programming course

Luka Fürst & Viljan Mahnič

University of Ljubljana
Ljubljana, Slovenia

ABSTRACT: In this article, the authors present a system for the cooperative development of computer programs that was created for the laboratory sessions of an introductory programming course at the University of Ljubljana in Slovenia. The system has relieved students of the tedious task of retyping programs developed by the teaching assistant, and has enabled them to cooperate with the teaching assistant in solving programming problems. The authors have, thus, made the laboratory sessions more efficient and interactive, and have brought them closer to the spirit of active learning approaches.

INTRODUCTION

Introductory university-level programming courses tend to suffer from comparatively high failure rates [1-3]. Various reasons for this problem have been identified [3-6]. Jenkins argues that programming, unlike most subjects at the pre-university level, involves a hierarchy of skills (such as the syntax-semantics-structure-style hierarchy) and a multitude of processes (a given problem specification has to be first translated to an algorithm or a recipe and, then, to code) [4]. Both learning and teaching programming are, therefore, considered difficult, especially for students who do not study computer science as their major subject and, thus, often lack the internal motivation to learn programming [7].

Several approaches have been proposed to alleviate the problem of teaching and learning programming. To make programming concepts and algorithms more tangible and, thereby, easier to grasp, some teachers have employed advanced visualisation tools [8] or multi-sensory approaches [9]. Some courses teach programming by providing students with *attractive* problems [10]. For instance, students might learn programming through developing computer games [11] or manipulating media files [7][12].

In problem-based and project-based approaches, students, typically organised in groups, acquire knowledge by working on challenging but possibly ill-structured problems or projects [13-15]. Active learning approaches emphasise the student's responsibility in gaining his or her knowledge [16-18]. While traditional, teacher-centred approaches view the teacher as the sole purveyor of knowledge and the student as its passive receiver, active learning approaches regard the student as an active participant in the learning process and the teacher as the *mere* facilitator of learning. Active learning approaches are consistent with the constructivist learning theory [19].

In this article, the authors focus on the introductory programming course called *Basic Programming*, taught in the Faculty of Computer and Information Science (UL-FCIS) at the University of Ljubljana, Slovenia, as a compulsory part of the so-called *University Programme Computer and Information Science* [20]. In the 2009/10 academic year, UL-FCIS adopted the Bologna reform to ensure greater compatibility of its programmes and courses with other European universities [18][21][22]. Among other things, the Bologna reform calls for a more active, student-centred way of learning. To enable the students to assume a greater role in the learning process, the authors refreshed the laboratory sessions of the course Basic Programming in the academic year 2010/11. While retaining the overall form of the course, they made the laboratory sessions more interactive and, thus, closer to the spirit of active learning approaches and the Bologna reform.

Many modern approaches to teaching programming take advantage of state-of-the-art technology to encourage greater participation of students in the learning process. The laboratory sessions of this course were refreshed by the help of a cooperative development system called *Assistant's Assistant*. While this system was developed primarily to save some valuable time during the laboratory session blocks, its most important advantage is to facilitate cooperation between the teaching assistant (TA) and the students in acquiring programming skills.

In the rest of this article, the authors first describe the course Basic Programming at UL-FCIS and, then, the system *Assistant's Assistant*.

THE COURSE BASIC PROGRAMMING AT UL-FCIS

The course Basic Programming covers the fundamentals of object-oriented programming in Java, the language of choice at many universities throughout the world [23-26]. The syllabus of the course covers control structures, arrays, classes and objects, inheritance and fundamentals of computer graphics. The course is taught in the first semester of the first year. It comprises 15 weeks of formal lectures and 13 weeks of laboratory sessions. The lectures are conducted in a teacher-centred fashion. In each block of lectures, consisting of three consecutive hours per week (throughout this article, an *hour* actually refers to an academic hour, which lasts 45 minutes), the lecturer introduces a programming subject and presents several basic examples to the entire body of approximately 250 students simultaneously. The laboratory sessions are held in groups of 25-30 students. Each block of laboratory sessions is supervised by two TAs.

Before the Bologna reform, the laboratory sessions were conducted in blocks of three hours per week. The students played a fairly passive role. In each block, one of the two TAs first presented a programming problem and, then, gradually developed its solution (a Java program) on his or her computer. The TA's computer screen was projected onto the projection surface so that the students could observe the development of the program and simultaneously retype it on their own computers. After the TA finished developing the program, the students had to compile and run it and, if time allowed, to modify it in some minor way. In some laboratory session blocks, several smaller problems were solved instead of one larger, but the students remained more or less passive receivers of knowledge throughout the course. The other TA (i.e. the one who did not present and solve programming problems) helped the students in testing the programs, answered their questions, etc.

Such a way of conducting laboratory sessions suffered from several disadvantages. During the sessions, the students were occupied with retyping programs that were being developed by a TA, rather than with building their own solutions. Furthermore, the tedious process of retyping made it much harder for the students to think about the program while it was being developed. Retyping was also the source of many typographical errors. Those errors were usually easy to correct, but they still required considerable time, particularly with less skilled students. In addition, programming novices often needed help in spotting and correcting typographical errors, causing further delays. The role of the second TA, therefore, was often reduced to that of correcting typographical errors.

These problems became even more acute in the year 2009/10, when UL-FCIS adopted the Bologna reform. To enable the students to do more work at home and, thus, to assume a greater responsibility for acquiring their knowledge, the reform reduced the length of laboratory sessions from three to two hours per week, while the quantity of material covered in the course Basic Programming was not reduced. Two-hour laboratory session blocks did not allow any time for the students to *play around* with the developed programs, which effectively transformed the laboratory sessions into a pure teacher-centred pedagogical process. This change was particularly unfavourable in light of the fact that the Bologna reform promotes active, student-centred learning. The reform, therefore, prompted the authors to reconsider the design of laboratory sessions.

A possible way of conducting laboratory sessions would be to let the students develop their own solutions to the problem(s) presented by a TA at the beginning of each block. In such a setup, the responsibilities of both TAs would include giving hints to the students, answering their questions, etc, but not developing or presenting solutions on their own. However, a similar idea had already been tried out some years ago, but it did not produce the desired results. The problem is that many students have little or no background in programming and problem-solving techniques and, thus, some form of guidance from the TAs nevertheless seems to be appropriate. The authors, therefore, decided that the laboratory sessions have to retain a generally TA-driven character, but they should be refreshed according to the following guidelines:

- The unproductive and time-consuming chore of retyping has to be automated so that the students could follow the process of solving a programming problem without any distractions.
- The development of a solution to a programming problem should become an interactive process. The TAs and the students should cooperate in constructing a solution.

These two directives led to the development of *Assistant's Assistant*, a cooperative development system outlined in the next section.

THE SYSTEM *ASSISTANT'S ASSISTANT*

The system *Assistant's Assistant* (available at http://ltpo.fri.uni-lj.si/as2/ and at SourceForge: http://sourceforge.net/projects/as2/) can serve as an on-line file transfer system, as a tool for developing programs cooperatively, and as a Java development environment. The authors present each of these functionalities in a separate subsection.

Assistant's Assistant as an On-line File Transfer System

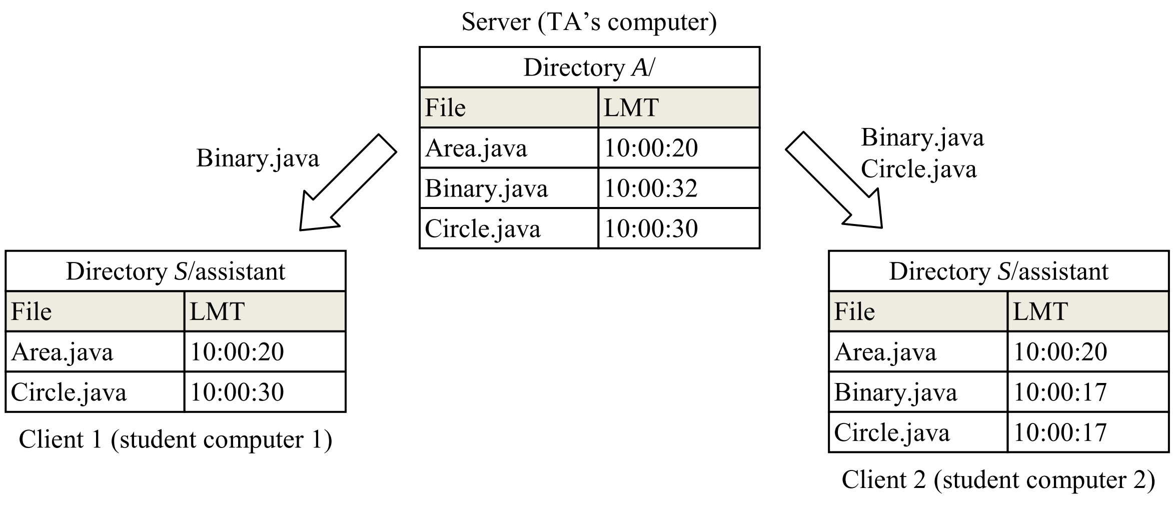
The system *Assistant's Assistant* is essentially a client-server Java application that automatically copies Java source files, at regular intervals, from a designated directory on the TA's computer to each of the students' computers. Conceptually, the system comprises a server and an arbitrary number of clients. The server runs on the TA's computer. Each student computer runs its own client instance.

When the TA initiates the server on his or her computer, he or she specifies the directory whose contents will be regularly copied to each of the students' computers. For the sake of conciseness, let us call this directory *A*. After the TA has started the server, each student initiates a client instance, specifying a directory *S* as a parameter. The client automatically creates the directory *S/assistant*, into which the files from the TA's directory *A* will be copied, and the directory *S/me*, which will be used to store the student's own solutions to programming problems. As long as the client and the server are both active, the contents of the client's directory *S/assistant* are updated automatically with Java source files from the server's directory *A*.

At regular intervals (every *N* seconds, with *N* = 3 in the current implementation), each client sends a synchronisation request to the server. A synchronisation request is a description of the current contents of the directory *S/assistant*. For each Java source file currently contained in the directory *S/assistant*, the synchronisation request specifies its name and the time of its most recent modification (last-modification time - LMT). When the server receives a synchronisation request from a client, it compares the contents of its directory *A* with the data specified in the synchronisation request.

The server, then, sends to the client all files that are present in the directory *A* but not in the synchronisation request (those files exist on the server but not yet on the client), as well as all files for which the LMT in the directory *A* is more recent than the LMT specified in the synchronisation request (those files exist on both the server and the client, but the server holds more recent versions of them than the client does).

The client saves all files received from the server into the directory *S/assistant*. This process is illustrated in Figure 1. Note that the server deals with each client separately. Each client is driven by its own timer, which, by firing every *N* seconds, regularly *reminds* the client to send a new synchronisation request to the server.



Directory *A*/	
File	LMT
Area.java	10:00:20
Binary.java	10:00:32
Circle.java	10:00:30

Directory *S*/assistant	
File	LMT
Area.java	10:00:20
Circle.java	10:00:30

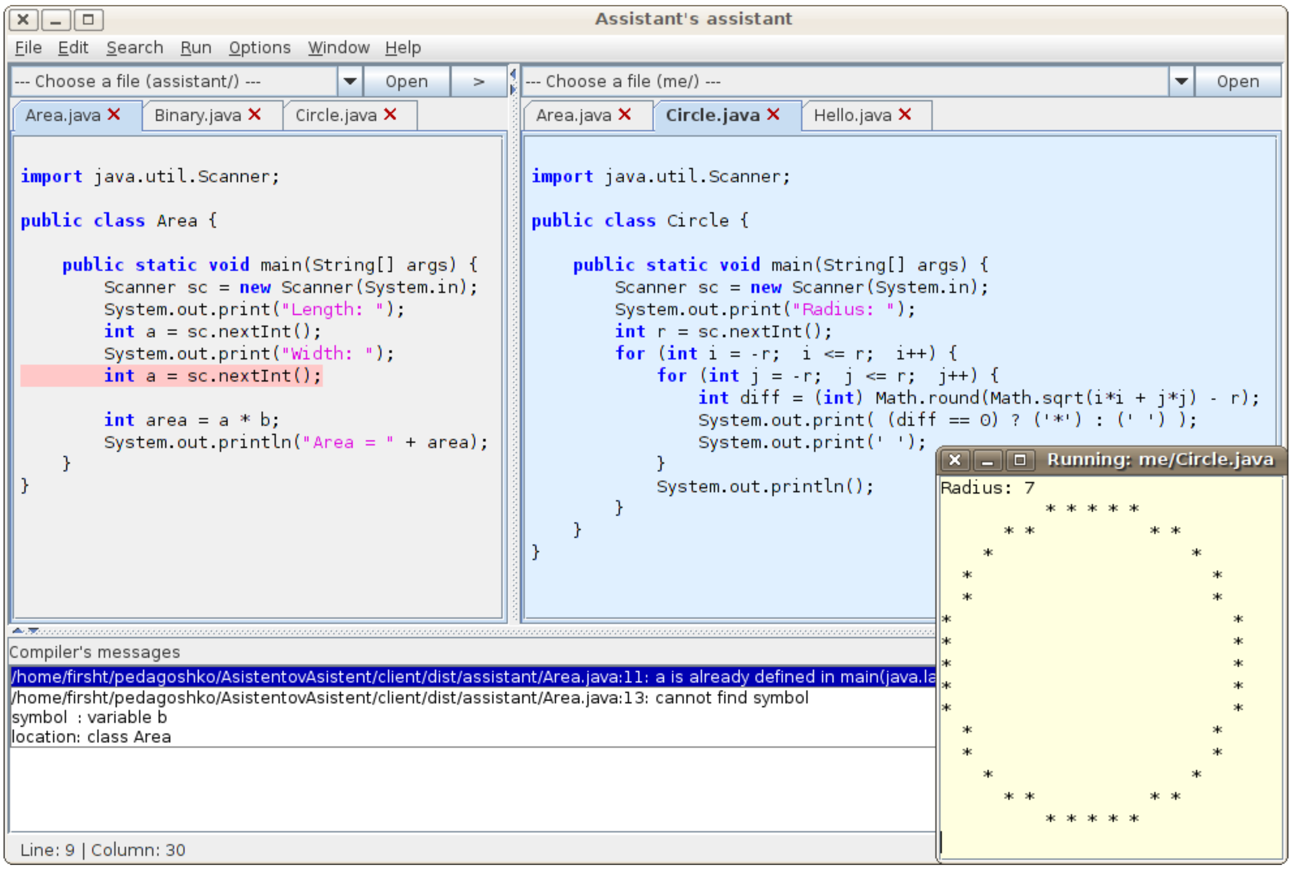
Directory *S*/assistant	
File	LMT
Area.java	10:00:20
Binary.java	10:00:17
Circle.java	10:00:17

Figure 1: The server responds to the synchronisation requests of individual clients by sending them the most recent version of its files.

The client executes a screenshot in a graphical user interface (GUI), which is shown in Figure 2. The main panel of the GUI is divided into two sub-panels. In the left-hand sub-panel, students can view (but not edit) the contents of the directory *S/assistant*. In the right-hand sub-panel, they can view and edit the contents of the directory *S/me*.

In the right-hand sub-panel, they can view and edit the contents of the directory *S/me*. The contents of the left-hand sub-panel are refreshed automatically, without any user intervention. After each synchronisation event (i.e. every *N* seconds), the files currently open in the left-hand sub-panel are automatically updated.

If the TA saves his or her file(s) frequently enough (or simply turns on the *autosave* option, available in many modern editors), the students will be able to observe the development of the program in real time on their computer screens, with no action required on their part.


Assistant's assistant
File Edit Search Run Options Window Help
--- Choose a file (assistant/) ---
Open
>
--- Choose a file (me/) ---
Open
Area.java
Binary.java
Circle.java
Area.java
Circle.java
Hello.java

```
import java.util.Scanner;

public class Area {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Length: ");
        int a = sc.nextInt();
        System.out.print("Width: ");
        int a = sc.nextInt();

        int area = a * b;
        System.out.println("Area = " + area);
    }
}
```

```
import java.util.Scanner;

public class Circle {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Radius: ");
        int r = sc.nextInt();
        for (int i = -r;  i <= r;  i++) {
            for (int j = -r;  j <= r;  j++) {
                int diff = (int) Math.round(Math.sqrt(i*i + j*j) - r);
                System.out.print( (diff == 0) ? ('*') : (' ') );
                System.out.print(' ');
            }
            System.out.println();
        }
    }
}
```

Compiler's messages

```
/home/firsht/pedagoshko/AsistentovAsistent/client/dist/assistant/Area.java:11: a is already defined in main(java.la
/home/firsht/pedagoshko/AsistentovAsistent/client/dist/assistant/Area.java:13: cannot find symbol
symbol  : variable b
location: class Area
```

Line: 9 | Column: 30

Running: me/Circle.java

```
Radius: 7
          * * * * *
      * *           * *
    *                   *
  *                       *
  *                       *
*                           *
*                           *
*                           *
*                           *
*                           *
  *                       *
  *                       *
    *                   *
      * *           * *
          * * * * *
```

Figure 2: A screenshot of the client GUI.

Assistant's Assistant as a Cooperative Development Tool

The button labelled '>' in the top-right corner of the left-hand sub-panel copies the currently active file in the left-hand sub-panel (in Figure 2, this is *S/assistant/Area.java*) to the directory *S/me* and opens the copy in the right-hand sub-panel. This button makes it possible to develop programs cooperatively. For instance, the TA might write a skeleton of a program (or a partial solution), place it into his or her directory *A*, and ask the students to complete it. The skeleton is automatically copied to the directory *S/assistant* of each student. After a student opens the skeleton in the left-hand sub-panel and presses the button '>', the skeleton is copied to the directory *S/me*. The copy is immediately opened in the right-hand sub-panel, where it can be edited. After the students have completed the program, the TA might do the same in his or her own way by editing the skeleton. The students' left-hand sub-panels (and corresponding files in their directories *S/assistant*) are automatically updated with the TA's solution. Each student can then compare the TA's solution with his or her own simply by comparing the contents of the left-hand sub-panel with those of the right-hand sub-panel.

Assistant's Assistant as a Java Development Environment

The left-hand sub-panel of the GUI serves as a container of Java viewers, and the right-hand one as a container of Java editors. Besides the standard editing features, such as cut/copy/paste, undo/redo, find/replace, etc, the editors offer features such as syntax highlighting, brace matching, increasing and decreasing indentation, etc. Furthermore, Java source files in the directories *S/assistant* and *S/me* can be compiled and executed directly from within the GUI. The compiler's output is displayed in a separate sub-panel. When the user clicks on a compilation error or warning, the GUI highlights the corresponding line in the corresponding source file. If a file compiles without errors, it can be executed. In the situation presented in Figure 2, the file *Area.java* in the left-hand sub-panel (i.e. the file *S/assistant/Area.java*) has been compiled, with compilation errors shown in the bottom sub-panel, and the file Circle.java in the right-hand sub-panel (i.e. the file *S/me/Circle.java*) has been successfully executed, with the output shown in a separate window in the bottom-right part of the screenshot.

The client can be initiated even if the server is not running. In this case, the left-hand sub-panel can be ignored or hidden, and the GUI behaves as a pure Java development environment.

DISCUSSION

The system *Assistant's Assistant* enabled the authors to make the laboratory sessions of the course Basic Programming more efficient and interactive. Prior to the introduction of *Assistant's Assistant*, i.e. before the year 2010/11, much time was spent on correcting trivial typographical errors resulting from manual retyping of code from the projection surface. Furthermore, some students could not keep up with the TA's typing pace, so the TA and the other students often had to wait for them to catch up. The authors estimate that, on average, at least 15 minutes of a 90-minute block were spent in this fashion.

Using *Assistant's Assistant*, the authors saved both on time and on the *energy* of the students, who can now follow the process of solving a problem instead of being occupied with retyping. The two TAs who supervise the sessions were freed from the unpleasant obligation of correcting typographical errors and can now both help the students in solving problems, check their solutions and comment on them, etc.

The resources saved by *Assistant's Assistant* were used to adopt a cooperative approach to solving problems. Some problems or some parts of a larger problem were solved by a TA and others by the students. The students were given enough time to solve their share of (sub) problems. After the time limit for a (sub) problem assigned to the students expired, the TA quickly developed his or her own solution to that (sub) problem so that each student could compare the TA's solution with his or her own. The system *Assistant's Assistant* saved enough time to make such cooperative programming possible. While the number of programming problems solved in the year 2010/11 was approximately the same as the year before, almost all problems were solved either entirely by the students or cooperatively. In the year 2009/10, by contrast, almost all of these problems were solved entirely by the TA, while the students were usually only able to retype and test them.

It might be worthwhile to compare the system *Assistant's Assistant* with some other approaches that may appear to achieve the same goal. Some form of cooperation between the TA and the students in solving a programming problem can be established simply by preparing a partial solution to the problem in advance and making it accessible on some Web server at the beginning of a laboratory session block. While this approach saves the students from retyping, it fails to give them the opportunity to observe the development of the partial solution. The system *Assistant's Assistant*, on the other hand, enables the students to follow the process of building programs in *real time*.

Viable alternatives to *Assistant's Assistant* include web-based collaborative editing systems, such as *collabedit* (http://collabedit.com/). The main advantage of *Assistant's Assistant* over such systems is its two-panel design, which makes it easy for a student to compare the TA's solution to a problem with his or her own. Another advantage of *Assistant's Assistant* is its built-in way to compile and execute both the TA's and the student's programs.

Last but not least, the system *Assistant's Assistant* can itself motivate the students to learn and practice programming. By creating and using a course-related programming product, the instructors directly demonstrate the value of programming to the students. The source code of *Assistant's Assistant* is freely available, inviting the students to browse it or experiment with it.

CONCLUSIONS

In this article, the authors outlined a cooperative development system called *Assistant's Assistant* that has enabled the authors to create a more interactive learning environment during the laboratory sessions of an introductory programming course at the University of Ljubljana, Slovenia. The system was primarily created to relieve the students from the time-consuming and error-prone task of retyping programs from the projection surface and to enable them to follow the development of programs on their computer screens. The two-panel design of the client part of the system, as well as the time saved by automating file transfer, enabled the TA and the students to cooperate in developing solutions to programming problems. The system was met with many positive responses from the students.

The system *Assistant's Assistant* can be enhanced in several ways. One possible idea is to provide support for other common programming languages, such as Python or C/C++. Another possible improvement is to enable continuous file transfer in the direction from individual students to the TA, which would enable the TA to monitor and compare the students' work during laboratory sessions.

REFERENCES

1. Bennedsen, J. and Caspersen, M., Failure rates in introductory programming. *SIGCSE Bulletin*, 39, **2**, 32-36 (2007).
2. Corney, M., Teague, D. and Thomas, R., Engaging students in programming. *Proc. 12th Australasian Conf. on Computing Education*, Darlinghurst, Australia, 63-72 (2010).
3. Hawi, N., Causal attributions of success and failure made by undergraduate students in an introductory-level computer programming course. *Computers & Educ.*, 54, **4**, 1127-1136 (2010).
4. Jenkins, T., On the difficulty of learning to program. *Proc. 3rd Annual LTSN-ICS Conf.*, Loughborough, UK, 65-71 (2002).
5. Milne, I. and Rowe, G., Difficulties in learning and teaching programming - views of students and tutors. *Educ. and Infor. Technologies*, 7, **1**, 55-66 (2002).

6. Huet, I., Pacheco, O., Tavares, J. and Weir, G., New challenges in teaching introductory programming courses: a case study. *Proc. 34th Annual Frontiers in Educ. Conf.*, Savannah, Georgia, USA, T2H/5-T2H/9, 1 (2004).
7. Guzdial, M., A media computation course for non-majors. *Proc. 8th Conf. on Innovation and Technol. in Computer Science Educ.*, New York, NY, USA, 104-108 (2003).
8. Hundhausen, C. and Brown, J., What you see is what you code: a *live* algorithm development and visualization environment for novice learners. *J. of Visual Languages and Computing*, 18, **1**, 22-47 (2007).
9. Katai, Z., Multi-sensory method for teaching-learning recursion. *Computer Applications in Engng. Educ.*, 19, **2**, 234-243 (2011).
10. Djordjevic, M., Java projects motivated by student interests. *Proc. 13th Conf. on Innovation and Technol. in Computer Science Educ.*, New York, NY, USA, 321-321 (2008).
11. Leutenegger, S. and Edgington, J., A games first approach to teaching introductory programming. *Proc. 38th SIGCSE Technical Symposium on Computer Science Educ.*, New York, NY, USA, 115-118 (2007).
12. Moor, S., Music in MATLAB: a series of programming challenges for an introductory course. *Computer Applications in Engng. Educ.*, 18, **1**, 67-76 (2010).
13. Kinnunen, P. and Malmi, L., Problems in Problem-Based Learning - experiences, analysis and lessons learned on an introductory programming course. *Informatics in Educ.*, 4, **2**, 193-214 (2005).
14. Davenport, D., Experience using a project-based approach in an introductory programming course. *IEEE Transactions on Educ.*, 43, **4**, 443-448 (2000).
15. Fink, F.K., Problem-Based Learning in engineering education: a catalyst for regional industrial development. *World Transactions on Engng. and Technol. Educ.*, 1, **1**, 29-32 (2002).
16. Barak, M., Harward, J., Kocur, G. and Lerman, S., Transforming an introductory programming course: from lectures to active learning via wireless laptops. *J. of Science Educ. and Technol.*, 16, **4**, 325-336 (2007).
17. Moura, I. and van Hattum-Janssen, N., Teaching a CS introductory course: an active approach. *Computers & Educ.*, 56, **2**, 475-483 (2011).
18. Fernandez, C., Diez, D., Zarraonandia, T. and Torres, J., A student-centered introductory programming course: the cost of applying Bologna principles to computer engineering education. *Inter. J. of Engng. Educ.*, 27, **1**, 14-23 (2011).
19. von Glasersfeld, E., *Radical Constructivism: a Way of Knowing and Learning.* London: Falmer Press (1995).
20. Mahnič, V. and Gams, M., Some experiences in teaching introductory programming at the faculty level. *World Transactions on Engng. and Technol. Educ.*, 2, **3**, 441-444 (2003).
21. Wächter, B., The Bologna Process: developments and prospects. *European J. of Educ.*, 39, **3**, 265-273 (2004).
22. Zieliński, W., The application of the Bologna Declaration in Polish technical universities. *World Transactions on Engng. and Technol. Educ.*, 1, **1**, 137-140 (2002).
23. Richards, B., Experiences incorporating Java into the introductory sequence. *J. of Computing in Small Colleges*, 19, **2**, 247-253 (2003).
24. de Raadt, M., Watson, R. and Toleman, M., Introductory programming: what's happening today and will there be any students to teach tomorrow? *Proc. 6th Conf. on Australasian Computing Educ.*, Darlinghurst, Australia, 277-282 (2004).
25. Benander, A., Benander, B. and Sang, J., Factors related to the difficulty of learning to program in Java - an empirical study of non-novice programmers. *Information and Software Technol.*, 46, **2**, 99-107 (2004).
26. Anik, Z. and Baykoç, O., Comparison of the most popular object-oriented software languages and criterions for introductory programming courses with analytic network process: a pilot study. *Computer Applications in Engng. Educ.*, 19, **1**, 89-96 (2011).