An Experience on Ada Programming Using On-line Judging

Francisco J. Montoya-Dato, José Luis Fernández-Alemán, and Ginés García-Mateos

Department of Informatics and Systems,
University of Murcia, 30100 Espinardo, Murcia, Spain
{fmontoya,aleman,ginesgm}@um.es

Abstract. Ada has proved to be one of the best languages to learn computer programming. Nevertheless, learning to program is difficult and when it is combined with lack of motivation by the students, dropout rates can reach up to 70%. In order to face up to this problem, we have developed a first-year course for computing majors on programming based on two key ideas: supplementing the final exam with a series of activities in a continuous evaluation context; and making those activities more appealing to the students. In particular, some of the activities are designed as on-line Ada programming competitions; they are carried out by using a web-based automatic evaluation system, the on-line judge. Human instructors remain essential to assess the quality of the code. To ensure the authorship of the programs, a source-code plagiarism detection environment is used. Experimental results show the effectiveness of the proposed approach. The dropout rate decreased from 61% in the autumn semester 2007 to 48% in the autumn semester 2008.

Key words: Programming, e-learning, assessment.

1 Introduction

Currently, educational tendencies are centered in the student's learning rather than the instructor's teaching. A clear example of this trend is the European Space of Higher Education. One of the aims of the European Union countries is to develop new teaching methodologies based on the student's learning process. The purpose is to create independent, reflective and life-long learners. The new methods should stimulate students interest and offer appealing material, fair assessment and appropriate feedback.

This paper describes an innovative experience with a first-year course on programming which is supplemented with some activities of e-learning. A web-based automatic judging system called Mooshak [1] has been adapted to receive and evaluate programs in Ada. Previous experience on programming competitions for secondary and higher education shows the viability of the proposal and a high capacity to generate motivation and enthusiasm among students. The approach described is highly complementary with other learning techniques and methods.

The rest of the paper is organized as follows. Section 2 presents a review of related work. Section 3 briefly describes the fundamentals of on-line judging. Then, we introduce in Section 4 the methodological approach of the proposal. Section 5 offers the main results of the e-learning experience applied to 107 students in a first-year course for computer programming majors. In Section 6, we discuss the results achieved by employing this new methodology. The last section presents some concluding remarks.

2 Related Work

In the literature, most authors reach the same conclusion: learning to program is difficult [2]. For example, some studies point out that it takes approximately ten years to transform a novice into an expert programmer [3]. A large number of techniques and methods have been proposed to improve students' comprehension in computer programming courses [2]. E-learning activities constitute a viable and promising supplement in programming pedagogy. Particularly, on-line judging systems have already been applied in this discipline.

Guerreiro and Georgouli [4, 5] propose an e-learning educational strategy in first-year programming courses. They adopt Mooshak automatic judging system for grading lab assignments and for self-assessment purposes. Automatic evaluation accounts for about 30% of the final mark. This approach provides important benefits in a CS1 course. A well thought out set of test cases prevents wrong programs sent by students from passing test runs. As a consequence, students must be much more rigorous in developing their programs. Likewise, students obtain immediate feedback from Mooshak. Another advantage of their proposal is the objectivity of the evaluation. Moreover, the authors consider that teachers can save time and work if an automatic judging system is used. Nevertheless, important concepts such as robustness and legibility are manually graded by the instructors.

In order to address this issue, Bowring [6] proposes a new paradigm for programming competitions where the quality rather than the fast completion of the programs is evaluated. Both technical and artistic merit are taken into account as judging criteria. According to the author, technical quality refers to how well submissions meet the stated requirements, whereas artistic quality is related to the organization of the code, the readability of the code and its documentation, and the readability of other artifacts such as output files.

Our novel contribution resides in the use of the on-line judging system in Ada programs. Our approach complements the traditional "final exam evaluation" with a series of activities, many of them using Mooshak. Four important benefits are obtained: (i) students are very motivated to take part in the proposed activities; (ii) the work of the students is evaluated along the course, rather than just in a single final exam; (iii) the workload of instructors is reduced since many compilation and runtime errors are detected by the on-line judge; and (iv) students receive feedback on their submissions during the process of acceptance

by the on-line judge, and can ask questions to the human judges, which promotes both independent learning and reflective thinking.

3 On-line Judging

An on-line judging system is an automatic tool which is able to evaluate the correctness of computer programs, based on a predefined set of pairs input/output. We are using Mooshak 1.4 [1], which is free and publicly available.

Mooshak has a web-based interface, which is different for the students, teachers, guest users and the system administrator (see Figure 1). For example, a user (student) can access the description of the problems, the list of submissions sent by all users, the ranking of the best students, and the questions asked and answered. In contrast, a judge (teacher) can see and analyze the submissions sent, rejudge submissions, answer questions, and view statistics of system's usage.

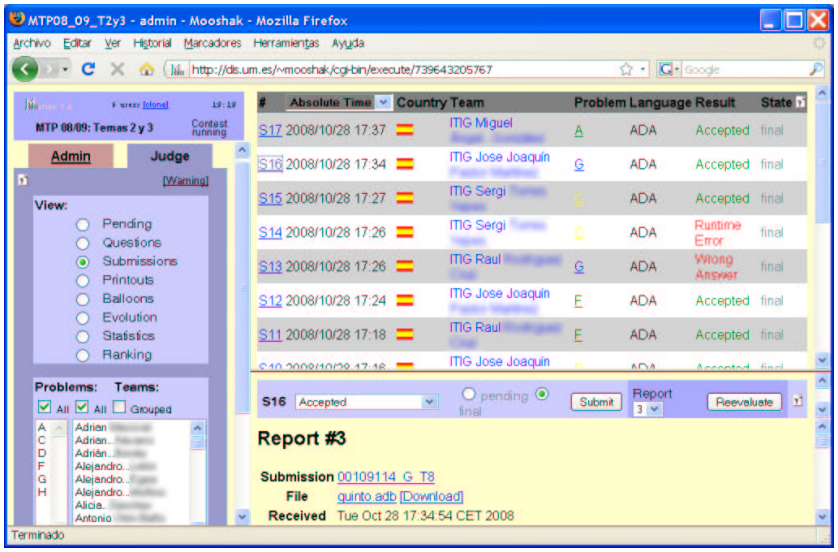


Fig. 1. Sample view for a judge (teacher) of Mooshak.

The on-line judge works as follows:

- A set of problem descriptions is available in the students' web interface. These descriptions present problems related to the theoretical concepts studied in class. Each description contains a statement of the problem, a precise specification of the input of the program and the expected output, along with some sample input/output pairs.
- The students tackle each problem in their own computers, by writing a program which efficiently produces the expected outputs. When they have tested

their implementation enough, they submit the solution to the judge using their interface.
- The on-line judge receives the source code, compiles the program, and executes it using the predefined sets of secret input cases. Then, Mooshak analyzes the output of the program (comparing it to the expected output) and sends a response to the student which indicates whether the program is correct or not.
- Statistical information is accessible both for teachers and for students. In particular, a ranking of the students sorted by the number of problems solved is given. The system also includes tools to send comments about any problem and ask questions to the teachers.

4 A Programming Methodology

We will introduce here the methodology that we follow in our CS1 programming course. First, the educational context in which this activity takes place is described. Then, we will justify the choice of Ada as the first programming language for our students. Finally, we will see the kind of problems that we find suitable to be judged by Mooshak, and how it is a convenient tool to help in the evaluation of the solutions provided by students.

4.1 Our programming course in CS1

Our programming course in CS1 is called *Methodology and Technology of Programming* (MTP), and it extends along the whole academic year. It is organized as three hours of classroom lessons and two hours of laboratory practices per week. For this course we have chosen the imperative paradigm, as the concepts of encapsulation and data hiding in implementations of abstract data types provide an adequate way for the study of data abstraction whose path naturally leads to the concept of class. For a more detailed discussion on this topic, the reader may see [7] and [8]. Object Oriented Programming (OOP) is studied later with the introduction of *classes* in the programming courses in CS2, and more in depth in a programming course utterly dedicated to OOP in CS3.

The MTP course consists of two differentiated parts, which take about half of the course each:

- A first part in which students concentrate their attention on the design of elemental iterative algorithms. In this part, a great emphasis is made in methodological aspects, like the *loop invariant* based design and the mathematical definition of inductive relations that allow programmers to obtain the most significant sentences of the iterative construction, like initialization and loop body. Some other basic concepts like subprogram decomposition, algorithmic schemes, scope and visibility rules, and scalar and structured data types (arrays and records) are also studied simultaneously.
  Contents of this part are based on Anna Gram Group's works, most of them gathered in [9] (in French). These works have been collected and extended

by the authors and can be found in their book [10] (in Spanish). Most of the problems studied in this part are related to the data type *sequence*, which is a container data type that only allows sequential access to its elements. Particularities of this data type make it specially suitable to be taken as a base for most iterative algorithms design problems.
- A second and last part in which some other topics are studied: recursive design, dynamic memory management, elemental data structures (linear structures, and binary trees), abstract data types (ADT), generics, and an introduction to efficiency. ADT's and generics are introduced by means of algebraic specifications, in order to clearly distinguish an ADT, which is an abstract and purely mathematical concept by itself, from its implementation, which is a computational and more concrete entity.

4.2 Why Ada?

In our opinion, Ada perfectly matches the required conditions for a programming language that should simultaneously be: (1) the most appropriate to be the very first students' programming language; and (2) an adequate framework to clearly illustrate the course contents, as described above. Most of these advantages are related to the early error detection that this programming language provides by itself.

For the first condition, we find the following advantages in Ada:

- In contrast to some other more popular languages, there exist several international standards for the Ada programming language, supported by institutions like ANSI and ISO.
- Ada is a very strongly typed language. This feature allows the programmer to trust in the compilation process to catch some of the errors derived from mixing different magnitudes in expressions. This type checking includes also parameter control in subprogram calls and instances of generics. Ada includes also the concept of *subtype*, which allows domain checking but relaxing incompatibilities that otherwise would exist among different types.
- Its syntax is very clear and well structured. Any construct beginning has its own terminator. All sentences end with semicolon, that does not act as a separator but as a terminator.
- A strict access to control loop variables is imposed in `for` iterations. Also, this control variable is implicitly declared in the iteration and it exists only during the loop execution. This prevents ambiguity problems in the semantic of the `for` iteration related to the final value of this variable upon loop termination.
- Ada provides mechanisms to adequately handle exceptions.
- It has suitable compilers freely available for academic institutions. In particular, we highly appreciate and acknowledge the excellent and free support that the company AdaCore offers for their Ada GNAT based compiler to all universities enrolled in their Academic Program. This software is currently available for a wide variety of platforms, and also includes the GPS

integrated development environment. There exist also some other friendly environments based on the GNAT free Ada compiler that may adequately be used as a basis for CS1 programming practices.
– Ada provides a good basis for further courses on parallel/concurrent programming, OOP, hardware description languages (HDL) and software engineering that could also be taught using Ada as base programming language.

And for the second condition, we find the following advantages. Most of them are closely related to the above ones:

– Taking in mind that this is a CS1 programming course, details related to compilation, execution and debugging should be as easy as possible, so that students may concentrate their efforts in studying and learning topics related to the course itself. GPS provides a very friendly environment to adequately cover with easiness all stages in any project development.
– A rich repertoire of control structures (*while*, *repeat-until*, *loop-exit-end* and *for* loops) can be illustrated in Ada.
– Ada features in modular programming make easy to implement extensions to deal with concepts (like the data type *sequence*) that are not usually included in common programming languages. In particular, we have created packages for students to be able to use these extensions in their Ada programs in a transparent way and in almost the same way they do in the algorithmic notation we use in our classroom lessons to teach these concepts.
– The structure of Ada allows an easy top-down or bottom-up program design, where a subprogram may be decomposed into some others in a hierarchical fashion. Ada also closely keeps track of how a subprogram makes use of its own parameters depending on their kind (`in`, `out`, or `in out`) and forces all parameters of any `function` to be only of `in` kind, respecting in this way the theoretical concept of function as an operator whose invocation should never modify the computational process state.
– The previously mentioned structure not only affects to an independent program or compilation unit, but also to the relation among all compilation units that a whole project consists of.
– Ada *packages* provide an excellent mechanism for opaque data types encapsulation and information hiding. Though this is not an exclusive feature of opaque data types, the possibility of declaring some of these types as `limited` prevents problems of misbehavior of default comparisons among expressions and, in the case of assignments, aliasing of complex data structures which could lead to data corruption.
– Ada perfectly supports generics since its first Ada'83 standard. This support allows us to define generic packages for data types like *sequence*, which due to its own nature of container data type is clearly a generic type.
– Mechanisms for data handling through `access` data types provide adequate methods for dynamic memory management. They are also designed in such a way that some hazardous situations like side effects due to aliasing of static and automatic variables are prevented by default. These variables may still

be handled by `access` data types, but the programmer should be aware of this fact and must explicitly declare these variables as `aliased`.

4.3 Proposed exercises

We will present here a taxonomy of the different kinds of exercises that can be proposed to students using Mooshak. First, we identify the pedagogical principles that have guided our efforts. Second, we will see how to overcome some of the problems that arise at this point when using the Mooshak on-line judging system.

The assignments proposed in our course are designed to cover the cognitive domain of Bloom's Taxonomy [11]. Bloom's cognitive domain involves knowledge and development of intellectual skills. There are six categories, of different degrees of difficulty. The correspondence between these categories and some of our educational activities is shown in Table 1.

Mooshak is a tool flexible enough to be used with any programming language, provided that its corresponding compiler is available and installed on the server. The compilation and execution processes can be done by means of a simple script that does not necessarily require interactive human intervention, like it could be the case of languages that require the use of GUIs environments in any of these two stages. Fortunately, Ada falls into the category of usable languages.

There is another aspect to take into account: when submitting a solution for any problem proposed in Mooshak, the submission process consists of uploading a *single* file. This fact imposes a severe constraint to the problems we should propose to fully cover all the topics involved in our course. How could we manage to propose problems whose solution is not a program from which an executable could be obtained, but just a compilation unit like a package or a generic subprogram? In the case of packages, there exists an additional problem: a package consists of *two* different files, whereas Mooshak on-line judging system only allows to upload one.

The solution we finally decided to adopt for these cases is as follows:

1. Students should upload one single zip archive which contains all the files required by the problem. It is part of students' responsibility that the names of these files and also the interface of their compilation units match the specifications given in the problem description.
2. As a first step, the content of this zip file is extracted by the script that performs the judging process. These files are compiled and object modules are obtained from them if no compilation errors are found.
3. Then, a testing program that is designed to test students' solution (and that is not known by them, of course), is compiled and linked against the object modules obtained in the previous step. If everything went well, an executable program should be obtained in this step.
4. This last executable program is the one that will be run and judged.

Testing programs are designed to make a test as much exhaustive as possible of all features that the compilation units provided by students should offer. Mutation testing [12] was used to ensure the quality of the test cases.

Category	Educational activities
Knowledge: Recall data.	Memorize concepts such as type, variable, constant, function, procedure, algorithm, algorithmic scheme.
Comprehension: Understand the meaning of instructions and problems. State a problem in one's own words.	Translate an algorithm written in pseudocode into a programming language. Write a program from a known formula or algorithm (e.g. greatest common divisor, factorial, Fibonacci sequence). Choose the correct program from a list to solve a given problem. Fill an incomplete algorithmic scheme according to certain sequential access model. Create a problem with the format of the judge: problem description, source code to solve it, input cases, and expected outputs.
Application: Use a concept in a new situation or unprompted use of an abstraction.	Four sequential access models are introduced using the sequence data type. Apply these control models to new data structures such as arrays, lists and trees. Parameterize a data structure such as a stack, queue or tree to build a generic data type. Generalize a numerical sorting algorithm to any data type with an order relation defined.
Analysis: Separate materials or concepts into component parts so that its organizational structure may be understood. Distinguish between facts and inferences.	Divide and conquer, stepwise refinement, recursion, inductive reasoning are techniques used to tackle the complexity of an algorithmic problem. The use of these techniques implies performing both analysis and synthesis.
Synthesis: Build a structure or pattern from diverse elements. Put parts together to form a whole, with emphasis on creating a new meaning or structure.	Implement iterative schemes starting from four pieces of code: initialization, termination condition, treatment of the current element, ending treatment. Several schemes can be built by using four sequential access models. Students have to achieve a tuned solution.
Evaluation: Make judgments about the value of ideas.	Choose between linear search and binary search and justify the response. Calculate the algorithmic complexity and select the most efficient sorting algorithm in a certain context.

Table 1. Educational activities in the cognitive domain of Bloom's Taxonomy.

Bearing in mind the above strategy to automatically grade the different kind of exercises, we can group problems into four main categories:

Single problems: We include in this category problems whose solution is just a compilation unit from which an executable program may be obtained as a result. This is the kind of problems that are usually proposed in programming contests. Some of these problems should be solved by using some of the extensions mentioned above (like the data type *sequence*), so these compilation units needed should be present in the system in order to correctly compile and link the source code uploaded by the students.

We can distinguish here two kinds of problems, depending on how programs should get their input data. First kind are problems whose data is taken from the standard input, as it is usually done in programming contests. We have also another kind of problems, where the input consists of a single line containing the name of the file where the input should be taken from. We use this kind of input for the case of problems related to the data type sequence, where data should be loaded from the file to the sequence before proceeding.

Compilation unit development problems: These are problems whose solution is not a main program but a compilation unit (package, subprogram, generic, etcetera). As mentioned above, for this kind of problems the main program is already uploaded in the judging server and is compiled and linked against the compilation unit(s) provided by the students. This testing program should be designed in such a way that it checks that:

- All elements implemented in the compilation unit(s) (types, subprograms, constants, exceptions, etcetera) match the names provided in the problem description.
- Compilation unit(s) subprograms return correct results and show the right behaviour as specified by the problem description. These tests should be performed for a wide variety of different subprogram input data. It should also be checked that the name, type and kind of any subprogram parameters are those specified by the problem description.
- Exceptions defined in the compilation unit(s) are raised in the cases, and only in the cases, specified by the problem description. Exceptional situations should be provoked and the corresponding exceptions properly handled in order to perform an adequate test of the expected unit(s) behaviour (for example, popping out from an empty stack, etcetera).

Whole project problems: In this kind of problems, students should provide all modules that the project consists of. The only constraint imposed for these cases is the name of the main unit where the executable program should be obtained from. Solutions provided for this kind of problems are judged as usual. These problems may be thought as a continuation to the previous ones: first, compilation units are tested and judged separately, and then the project is judged as a whole.

Judge problems: The students have to create a problem with the format of the judge: problem description, source code to solve it, input cases, and expected outputs.

Table 2 shows the Mooshak's activities organized in the course. Notice that the level of difficulty (Bloom's level) of the activities is gradually increasing. All of them are to be done individually. The problems are graded from 1 to 5 according to their degree of difficulty and are freely chosen by students. The instructor notifies the results to the students in a personal interview. The activities are voluntary and are not required as part of the regular assignments.

Mooshak's activities evaluation accounts for 20% of the final mark. Since most work is not done in the presence of the teacher, a tricky concern is to guarantee the originality and authorship of the programs submitted by the students. Some strategies are applied to reduce the risk of plagiarism and to detect it:

Activity	Type	Language	# p.	Bloom
Sequentiation	SP	Ada	6	K, C
Selection	SP	Ada	5	K, C
Iteration	SP	Ada	8	C, Ap
Schemes	CUDP	Ada	12	An, S
Generics	CUDP, WPP	Ada	2	Ap
Packages	CUDP, WPP	Ada	2	Ap
ADT	SP	Maude	7	An, S
Dynamic Memory	WPP	Ada	8	Ap, An, S
Recursion	SP	Ada	10	An, S
Sorting	SP, CUDP, WPP	Ada	11	Ap, An, S, E
Miscellany	JP	Ada	6	E

Table 2. Description of the activities proposed in Mooshak. "Type": Single Problems (SP), Compilation Unit Development Problems (CUDP), Whole Project Problems (WPP) and Judge Problems (JP);"# p.": number of problems existing in the judge; "Bloom": category covered in the cognitive domain of Bloom's Taxonomy, knowledge (K), comprehension (C), application (Ap), analysis (An), synthesis (S) and evaluation (E).

- There are many aspects of programming that are not so easy to automatically evaluate: computational complexity, design and organization of the code, programming style, robustness, legibility, etcetera. For this reason, all activities include a compulsory interview with a teacher, where students have to explain their submissions and answer some questions. Nevertheless, some of these quality factors could be automatically assessed using software quality assurance tools. These tools will be considered in a future work.
- The formula $\sum_{c=1}^{NC} \sum_{p=1}^{NP_c} \frac{0.1 \times (TNS_c - NSSP_{c,p})}{TNS_c}$, is used to grade the Mooshak's activities performed by each student. NP_c is the number of problems proposed in the contest c and NC is the number of contests organized. A student is considered as a contestant in the contest c if he has a Mooshak account in this contest. $NSSP_{c,p}$ is the number of contestants that solved the problem p of the contest c, and TNS_c is the total number of contestants in the contest c. Note that the score of each accepted submission is in inverse proportion to the number of accepted submissions. Therefore, we think that this formula is an effective deterrent measure against plagiarism lovers.
- Students will have to demonstrate their knowledge on the topics by an individual written exam.
- For the activities done in Mooshak, we use a plagiarism detection system developed by Cebrian *et al.* [13]. Thanks to Mooshak, all the submissions are available in judge's server, so the plagiarism detector can be easily applied. In our case, this plagiarism detector reported three possible cases of copy. Nevertheless, after manual inspection of the programs and an individual interview with students, plagiarism was ruled out.

5 Evaluation of the Method

The approach proposed here has been used effectively in an introductory computer programming course at the University of Murcia (Spain). In this section, detailed information about the experiment designed and conducted during the autumn semester of 2008 is provided. The aim was to assess the application of the programming learning method proposed in this paper.

5.1 Participants and Background

As mentioned in Section 4, the experience described here was applied to a first-year course for computer programming majors. MTP has a load of 12 ECTS (European Credits Transfer System) and has been traditionally organized in a *monolithic* form: weekly lectures, laboratory sessions, and a final exam for each semester. The first exam consisted of between 3 and 4 algorithmic problems about basic procedural programming constructs of sequence, selection and iteration, inductive reasoning and loop patterns. The second exam consisted of between 5 and 7 problems about recursive design, dynamic memory management, linear structures, binary trees, efficiency and algebraic specifications to represent abstract data types. Grading was manually done by the instructors according to criteria such as correctness, efficiency, robustness, extendibility and legibility.

In autumn of 2008, the 107 students enrolled in MTP were involved in the new learning method. Though participation was voluntary in Mooshak's activities, most students actively participated in the proposed activities.

In previous years, the main problem observed in this course was a low motivation and participation of the students in class, that resulted in a high dropout rate. In the last five years, around 70% of enrolled students dropped out, as shown in Table 3. With the aim of reversing this trend, we decided to adopt a new learning paradigm based on a continuous evaluation organization, with activities that are appealing and motivating for all students.

5.2 Results of the On-line Judge

Statistical data related to the programming learning method described in this paper was gathered during the autumn semester of 2008. Up to 92 of the 107 enrolled students (86%) participated in some activity related to the on-line judge; 60 of them (56%) solved and 62 (58%) tried to solve at least one problem. In total, the on-line judge received 2512 submissions, i.e., Ada programs and packages, with an average of 27.3 submissions per student. The on-line judge classified around 1019 of these as "accepted" (40.6%), and 359 as "wrong answer" (14.3%). More information on the classification of the submissions, and the percentages per unit of knowledge is shown in Table 4. The average number of submissions per student until getting the program accepted is 2.2. Nevertheless, many students found the solution to the problems at the first attempt (mode is 1). The highest number of programs that a student submitted to get an "accepted" was 21.

Year	2003/04		2004/05		2005/06		2006/07		2007/08	
Duration	Ann.	Aut.	Ann.	Aut.	Ann.	Aut.	Ann.	Aut.	Ann.	Aut.
Language	Modula-2		Modula-2		Modula-2		ADA		ADA	
Pass rate	14	18	17	20	12	14	9	13	16	21
	11%	14%	16%	19%	11%	12%	10%	14%	13%	17%
Failure rate	25	28	21	27	18	24	18	30	20	27
	19%	21%	20%	26%	16%	22%	19%	33%	17%	22%
Dropout rate	92	85	67	58	82	74	65	49	85	73
	70%	65%	64%	55%	73%	66%	71%	53%	70%	61%
Total of students	131	131	105	105	112	112	92	92	121	121

Table 3. Pass, failure and dropout rates of MTP in previous years. Ann.: Annual; Aut.: Autumn semester.

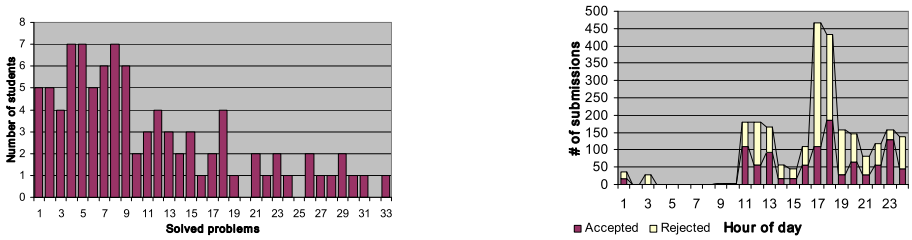
Activity	# subm.	A	PE	WA	RE	CE
Sequentiation	276	111 (40%)	51 (18%)	41 (14%)	59 (21%)	14 (5%)
Selection	280	122 (43%)	28 (10%)	74(26%)	44 (15%)	12 (4%)
Iteration	718	299 (41%)	145(20%)	87 (12%)	155(21%)	32 (4%)
Schemes	1039	409 (39%)	220 (21%)	135 (12%)	200 (19%)	75 (7%)
Generics	193	78(40%)	35 (18%)	22 (11%)	51 (26%)	7(3%)
Total	2512	1019 (40%)	479 (19%)	359 (14%)	509 (20%)	140 (5%)

Table 4. Detail of the classification of the submissions by knowledge unit. "# subm.": Number of submissions. A: Accepted. PE: Presentation error. WA: Wrong answer. RE: Runtime Error. CE: Compile Time Error.

Figure 2(a) shows a histogram of the number of problems solved per student. This value covers a range from 1 to 33. The mean number of solved problems per student is 11, with a standard deviation of 8.2, and with three modes of 4, 5 and 8. It is also interesting to analyze when the students work. Figure 2(b) represents the number of accepted and rejected submissions in each hour of the day. The minima are located at 2 and 6 am (0 submissions) and the maximum at 4 pm (467 submissions). On the other hand, the students achieved the highest acceptance rates at 10 pm and 10 am with 82% and 66%, while they had the lowest acceptance rate at 6 pm with 17%. Most activity takes place from 8 am to 8 pm, when computer laboratories are open to the students. However, submissions done by the students outside these hours represent a total of 24%.

6 Discussion

The results obtained after the application of our programming learning method during the autumn semester 2008 are very promising. We have observed a significant increase in the pass rate, from 17% (13% in whole academic year) in 2007 to 21% in 2008. However, the most striking fact is the dramatic decline in the dropout rate, from 61% (70% in whole academic year) in 2007 to 48%



(a) Number of problems solved by the students.

(b) Submissions in each hour of the day.

Fig. 2. Statistical information on Mooshak.

in 2008. The new methodology encourages students to make them get back on pace again.

The advantages that we find in our approach, considering the kind of problems that we propose to students in the on-line judge, can be summarized in the following points:

- In single problems, the importance of making a methodological and systematic program development in order to minimize errors becomes evident. It frequently happens that students, most times due to an excess of self-confidence, omit the necessary steps to analyze the problem and to design the solution. In most cases, this kind of hurried development results in a program that apparently works correctly, but there exist some cases in which it fails. When the validation stage is made locally in the own student's computer, the causes of these errors are usually quite evident and the proper program modifications quite easy too. This creates in the students the false illusion that those methodological issues are not important, as the consequences of disregarding them can be easily and quickly overcome.
  On the contrary, in the case of Mooshak on-line judging system, input test cases are unknown by students and the only feedback that they get is a laconic "wrong answer" message, which forces them to make a more in depth and methodological review of their programs in order to discover the error. This situation is similar to the real case, in which all program input data cases are obviously a priori unknown and the only information that programmers will get is that their programs fail *sometimes*. As a result, the error will be rather difficult to isolate just by analyzing the program behaviour from the input/output point of view.
- In compilation unit development problems, the student becomes aware of the importance of strictly following the specification of the unit interface. The slightest variation in an identifier name, number/type/kind of parameters in a subprogram, etc. may result in a testing program compilation error. On the other hand, unit behaviour should be as expected: subprograms re-

sults should be correct, exceptions should be raised in the right and only in the right cases, etcetera. Otherwise, testing program execution would not generate the expected output or even would result in a run-time error.
- In whole project problems, the student becomes responsible of correctly organizing all the submitted code. In particular, it is possible that some of the project compilation units were independently judged in previous problems, and more than one version were accepted by the on-line judging system. The student should decide at this point which is the version that he/she considers the best one to be included in the final project.
- In judge problems, students have to make judgments about the interest, difficulty and complexity of a problem. The creativity of the students to produce original and relevant problems is evaluated.

On the other hand, we think the proposed organization of the course successfully meets most important pedagogical principles [14]:

Motivation. The public ranking plays a fundamental role in motivating students to solve more problems, faster and more efficiently. If students get an "accepted", a rise in ranking means students get an incentive to continue tackling other algorithmic problems.

Active learning. When students solve the proposed problems, they are involved in, and conscious of, their own learning process in order to achieve a real and long-lasting learning.

Continuous learning. The new methodology has a crucial advantage: the students work along the course, and not just some weeks before the final exam.

Autonomous work. Students can work in the laboratories, where they have help from the teachers. However, students mostly work at home, and ask questions to the teachers by using Mooshak.

Feedback of the learning process. The web system provided feedback to help students to correct many errors of their programs, thus avoiding assistants spend much effort figuring out the causes of the failure, as happens in a traditional evaluation. The judge is accessible 24-hours a day and the feedback is instantaneous. From the point of view of the teachers, information is also comprehensive and immediate; they can analyze the difficulty of the problems, the evolution of the students, identify the best students, etcetera.

Finally, regarding our experience on these e-learning activities, we advocate the use of on-line judging systems as a support for the teacher in the task of evaluating students' know-how. In any case, it will always be also necessary the teacher's criterion to determine the degree of correctness of the submitted code.

7 Conclusions

We have presented in this paper an innovative experience on computer science education using Ada. In general, the results of our experiment are excellent. We

have shown that on-line judging systems can be used to make the activities of a programming course more interesting.

The approach improves self-assessment skills and encourages students to work independently. The public ranking and other statistical data provided by Mooshak, promote competitiveness and offer appealing material to the students. The assessment is fair and objective, and students are able to gain additional feedback from the human judges. The approach contributes to build a strong foundation for the student's life-long learning. Students get themselves more involved into their own learning process.

References

1. Leal, J.P., Silva, F.M.A.: Mooshak: a web-based multi-site programming contest system. Softw., Pract. Exper. **33** (2003) 567–581
2. Robins, A., Rountree, J., Rountree, N.: Learning and teaching programming: A review and discussion. Computer Science Education **13** (2003) 137–172
3. Winslow, L.E.: Programming pedagogy—a psychological overview. SIGCSE Bull. **28** (1996) 17–22
4. Guerreiro, P., Georgouli, K.: Enhancing elementary programming courses using e-learning with a competitive attitude. Int. Journal of Internet Education (2008)
5. Guerreiro, P., Georgouli, K.: Combating anonymousness in populous CS1 and CS2 courses. In: Proc. ITICSE 2006. (2006) 8–12
6. Bowring, J.F.: A new paradigm for programming competitions. In: SIGCSE '08: Proceedings of the 39th SIGCSE technical symposium on Computer science education, New York, NY, USA, ACM (2008) 87–91
7. Bruce, K.B.: Controversy on how to teach CS 1: a discussion on the SIGCSE-members mailing list. SIGCSE Bulletin **36** (2004) 29–34
8. Reges, S.: Back to basics in CS1 and CS2. In: SIGCSE. (2006) 293–297
9. Peyrin, J., Scholl, P.: Schemas Algorithmiques Fondamentaux. Sequences et Iteration (in French). Masson, Paris (1988)
10. García-Molina, J., Montoya-Dato, F., Fernández-Alemán, J., Majado-Rosales, M.: Una Introducción a la Programación. Un Enfoque Algorítmico (in Spanish). Thomson (2005)
11. Bloom, B., Furst, E., Hill, W., Krathwohl, D.: Taxonomy of Educational Objectives: Handbook I, The Cognitive Domain. Addison-Wesley (1956)
12. Woodward, M.R.: Mutation testing—its origins and evolution. Information and Software Technology **35** (1993) 163–169
13. Cebrian, M., Alfonseca, M., Ortega, A.: Towards the validation of plagiarism detection tools by means of grammar evolution (in press). IEEE Transactions on Evolutionary Computation (2008)
14. Vrasidas, C.: Issues of pedagogy and design in e-learning systems. In: SAC '04: Proceedings of the 2004 ACM symposium on Applied computing, New York, NY, USA, ACM (2004) 911–915