# Automated Assessment in a Programming Tools Course

José Luis Fernández Alemán

***Abstract*—Automated assessment systems can be useful for both students and instructors. Ranking and immediate feedback can have a strongly positive effect on student learning. This paper presents an experience using automatic assessment in a programming tools course. The proposal aims at extending the traditional use of an online judging system with a series of assignments related to programming tools. Some empirical results on how students use an automated assessment system in a CS2 course are presented. Research suggested that automated assessment systems promoted the students' interest and produced statistically significant differences in the scores between experimental and control groups.**

***Index Terms*—Active learning, competitive learning, e-learning, programming tools, Web tool.**

## I. Introduction

Automated assessment systems for student programming assignments are in their third generation [1]. These systems offer important benefits: immediate feedback, objectivity and consistency of the evaluation, and a substantial saving of time in the evaluation of the assignments. However, quality aspects such as robustness and legibility must be manually graded by the instructors [2]. Some studies have also reported [3], [4] that ranking generates motivation and enthusiasm among students, although there are indications that this may be culturally dependent [5].

Traditionally, an automated assessment system is used to evaluate the correctness of computer programs based on a predefined set of input/output pairs [3], [4], [6]. The novel contribution of this paper resides in how to apply the online judging system. First, the traditional strategy is taken one step further by extending the use of an automated assessment system called *Mooshak* [7] with a series of activities related to programming tools. Second, some quality factors are automatically graded by using a *static corrector*, which is an external program invoked before Mooshak's correction to classify the program's source code.

This paper is structured as follows. Section II presents a review of the relevant literature. Section III describes the course context and the methodological approach of the proposal. Section IV includes the research questions and offers the main results of the e-learning experience applied to 46 students in a second-year course for computer programming tools students. The main findings of the paper are presented in Section V. Finally, Section VI discusses some concluding remarks.

Manuscript received June 14, 2010; revised October 12, 2010; accepted November 30, 2010. Date of publication December 23, 2010; date of current version November 02, 2011.

The author is with the Department of Informatics and Systems, Facultad de Informática, University of Murcia, Campus de Espinardo, Murcia 30071, Spain (e-mail: aleman@um.es).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

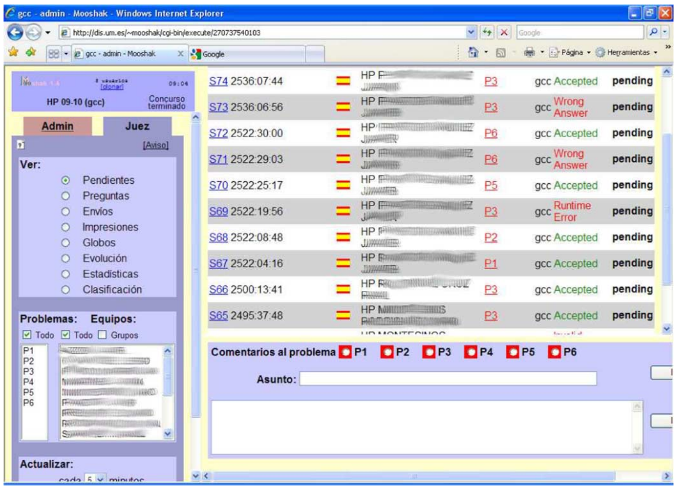
Digital Object Identifier 10.1109/TE.2010.2098442

## II. Literature Review and Background

Computer-based assessment systems allow students to solve programming exercises and to submit their solutions. Robo-Prof [8] and Automated System for the Assessment of Programming (ASAP) [9] are two Java-oriented assessment systems. RoboProf presents programming problems within a Web browser and is presented in a series of levels to give the student a view of progress, thus allowing him or her to move onto the next programming activity. The automatic programming assessment element of this tool is integrated with a multiple-choice questions system. The ASAP project is a system that automatically assesses programs and provides access to learning materials and tools. An XML document is generated, containing comments on the programs assessed, a description of tests applied, and a final grade. ASAP fits into an abstract framework that is known as the e-learning framework (or ELF). The framework is intended to guide the construction and development of interoperable and reusable software components that can be combined together to meet the requirements of a particular education institution.

Three of the most important Web-based tools that support automatic assessment for several languages are CourseMarker, BOSS, and Mooshak. CourseMarker [10] is a Web-based assessment system for evaluating Java programming assignments and *diagrams*. Assignment assessment is done by analyzing the program across a number of criteria: typographic, lexical structure, presence of particular features, program complexity, and execution efficiency. CourseMarker also provides administration facilities, statistics reports, and a rich marking interface, allowing students to have their program graded at frequent intervals prior to submission. The BOSS system [11] is a Web-based tool to facilitate the online submission and processing of programming assignments. BOSS provides an extensible set of simple program metrics, such as number of comments and percentage of methods declared abstract. It supports online marking and provides an interface for delivering feedback. BOSS incorporates plagiarism detection software and uses a client–server architecture with separate clients for students and for authorized staff for security reasons. The first version of BOSS tested assignments written in the C language. A redesign of BOSS was required to test Java software automatically.

The Web-based automated judging system used in the e-learning course on programming tools presented in this paper was *Mooshak*, a free third-generation assessment system, which



| | | | | | | |
|---|---|---|---|---|---|---|
| S74 | 2536:07:44 | HP [redacted] | P3 | gcc | Accepted | pending |
| S73 | 2536:06:56 | HP [redacted] | P3 | gcc | Wrong Answer | pending |
| S72 | 2522:30:00 | HP [redacted] | P6 | gcc | Accepted | pending |
| S71 | 2522:29:03 | HP [redacted] | P6 | gcc | Wrong Answer | pending |
| S70 | 2522:25:17 | HP [redacted] | P5 | gcc | Accepted | pending |
| S69 | 2522:19:56 | HP [redacted] | P3 | gcc | Runtime Error | pending |
| S68 | 2522:08:48 | HP [redacted] | P2 | gcc | Accepted | pending |
| S67 | 2522:04:16 | HP [redacted] | P1 | gcc | Accepted | pending |
| S66 | 2500:13:41 | HP [redacted] | P3 | gcc | Accepted | pending |
| S65 | 2495:37:48 | HP [redacted] | P3 | gcc | Accepted | pending |

Fig. 1. Sample view for a system administrator of Mooshak.

was originally created to manage online programming contests. Mooshak has already been used in programming courses. Guerreiro and Georgouli [12], [13] propose an e-learning educational strategy in a first-year programming course. They adopted the Mooshak automatic judging system for grading lab assignments and for self-assessment purposes. Mooshak was also successfully employed in CS1 using Ada [4], and CS2 using C++ and Maude [3].

Mooshak has a Web-based interface, which appears differently to the students, teachers, guest users, and the system administrator (Fig. 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, analyze, and mark the submissions sent, rejudge submissions, answer questions, and view system use statistics.

The online 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 on their own computers by writing a program that efficiently produces the expected outputs. When they have tested their implementation enough, they submit the solution to the judge using their interface.
- The online 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 that indicates whether the program is correct or not.
- Statistical information is accessible both for teachers and for students. In particular, a ranking is given of the students, sorted by the number of problems solved. The system also includes tools to send comments and to ask questions about any problem to the teachers.

Mooshak was chosen for its flexibility in the deployment of a testing system that may, depending on the programming tool employed in the online activities, involve preprocessing or post-processing of the student's submissions. All of the aforementioned tools are language-dependent, and to the best of the authors' knowledge, none of them can support the automatic assessment of programming tool assignments.

## III. Methodological Issues

The methodology followed in a CS2 programming tools course is introduced in this section. First, the educational context in which this activity takes place is described. Then, the kinds of problems suitable to be judged by *Mooshak* are presented. Finally, this section shows how *Mooshak* can be used to help in the evaluation of the solutions provided by students.

### A. *Programming Tools Course*

Mooshak has been used in a first-term programming tools course at the University of Murcia, Murcia, Spain. *Programming Tools* (PT) is organized as 2 h of classroom lessons and 2 h of laboratory practical per week. For this course, C language was chosen, an imperative paradigm programming language, since object-oriented programming (OOP) is studied later with the introduction of *classes* in a data structures course in CS2, and more in depth in a programming course completely dedicated to OOP in CS3. Nevertheless, some tools can be used as a support for programming languages of other programming paradigms. The PT course consists of two distinct parts, each taking about half of the course.

- A first theoretical part addresses testing, debugging, and software deployment and management issues, in which students concentrate their attention on the methodological aspects. In this part, a great emphasis is made in the basic testing techniques, both white-box and black-box testing. Techniques such as *basic path testing*, *branch testing*, *loop testing*, *data-flow testing*, *mutation testing*, *equivalence partitioning*, and *boundary value analysis* are studied and illustrated by examples. This course continues with an introduction on how to debug programs by *brute force*, *induction*, *deduction*, and *backtracking*, as well as a *divide-and-conquer* approach to make it easier to debug. Finally, other more advanced topics like version management and automatic deployment are also studied. Most of the contents of this part are based on [14]–[16].
- A second practical part introduces programming tools supporting and extending the topics taught in classroom lessons. Table I shows the tools used to illustrate each topic. All these tools have become popular in the open-source software world and are released under the GNU General Public License.

The assignments set in this course cover different learning objectives. In terms of the cognitive domain of Bloom's Taxonomy [17], the correspondence between the cognitive learning levels and some of the educational activities is shown in Table II. All students of PT had successfully completed an introductory course called *Operating Systems*, and therefore

TABLE I
TOOLS AND TOPICS COVERED IN PT

| **Tools** | **Topics covered** |
|---|---|
| **gcc** | Compilation, static analysis of programs, statically and dynamically linked programs, creation and use of static and shared libraries, and support for other tools. |
| **gcov, gprof** | Testing: code coverage and profiling. |
| **gdb** | Debugging by tracing the execution of programs. Post-mortem debugging with core file. |
| **doxygen** | Documentation. |
| **assert, nana** | Assertions and logging. |
| **make** | Deployment of software systems. |
| **efence** | Detection of dynamic memory errors. |
| **cvs** | Version control systems. |
| **autoconf** | Generating configure scripts. |
| **automake** | Generating makefiles compliant with the GNU Standards. |
| **libtool** | Creation and deployment of static and shared libraries. |

TABLE II
EDUCATIONAL ACTIVITIES IN THE COGNITIVE DOMAIN OF BLOOM'S TAXONOMY

| **Category and educational activities** |
|---|
| **Knowledge**. Memorize concepts such as black box testing, white box testing, debugging, static and shared library, software deployment, version control and son on. |
| **Comprehension**. Understand the testing and debugging techniques. For example, comprehend the data flow testing strategies. |
| **Application**. Use the programming tools in a new context. For example, add a remote CVS repository or generate *doxygen* documentation in a new format. |
| **Analysis**. Identify and separate the components of an application to prepare its deployment, for example, by means of a *makefile* file. |
| **Synthesis**. Build new artifacts using the tools. For example, build new static and shared libraries. |
| **Evaluation**. Choose the most suitable tool for solving a problem. Compare static linking with dynamic linking. |

had enough background to write shell scripts, which are needed to use the online judge, as seen later.

## *B. Teaching Intervention: An E-Learning Approach*

*Mooshak* is a tool flexible enough to be used with any language, provided that its corresponding compiler or interpreter is available and installed on the server. Note that when submitting a solution for any problem set in *Mooshak*, the submission process consists of uploading a file. This imposes a severe constraint on the problems that should be set in order to fully cover all the topics involved in PT. Particular difficulties were found in the problems whose solution is not a program from which an executable file could be obtained, but just a compilation unit like a library, a *gdb* script, a *makefile* file, or a *shell* script that performs a certain task. The solution adopted in each case depends on the features of each tool. Generally, the compilation and execution processes are done by a simple script. *Mooshak* was adapted to accept and evaluate programming tool assignments set in PT.

- **gcc**: A submission is a shell script. Both the input and output files can vary, depending on the compilation stage involved in the problem.
- **gcov, gprof**: A submission is a shell script that gets the name of the source file from the standard input and produces, for example, a listing of the source code along with frequency of execution for each line (*gcov*).
- **gdb**: A submission is a *gdb* script that contains commands to debug a binary file. The code source is given to the student in the description of the problem. A shell script is executed by *Mooshak* in which *gdb* reads the binary file and executes the *gdb* commands from the file submitted by the student. The output is a file in text format that is compared to the expected output.
- **assert**: In this case, *Mooshak* is used as in programming contests in which a submission is a program. The expected output is a message reporting an assertion violation.
- **make**: A submission is a *makefile* file containing a list of rules that indicate how to execute a series of commands in order to build a target file from source files. The names of the source files are stored in *make* variables. These variables are set in the command line that invokes *make* in a shell script executed by *Mooshak*. Finally, the shell script lists the target files created to be compared to the expected output.
- **efence**: A submission is a shell script that gets the source file name from the standard input and produces a text containing the exact instruction that causes the dynamic memory error (e.g., write or read access to freed memory).
- **Libraries**: A submission is a shell script that gets the library name from the standard input and builds a library in a directory. Then, *Mooshak* executes a shell script designed to test students' solutions. A testing program is compiled and linked against the library previously obtained. If everything goes well, a binary file is obtained, and it is this last executable program that will be run and judged.
- **cvs**: A submission is a shell script that gets the repository name from the standard input and executes *cvs* commands according to the set problem. Since the history of source files is recorded, the submissions can be easily judged by checking the *cvs* repository.

Note that all commands included in the shell scripts must be silent to avoid their interfering in the judgment phase. The level of difficulty (Bloom's level) of the activities is gradually increasing.

Most of the students' submissions are automatically assessed by a static corrector. An external program is invoked before Mooshak's correction to classify the script's source code according to some quality factors such as efficiency and legibility. For example, efficiency is measured in terms of the number of shell variables used, the number of levels in nested loop constructs, and the number of shell commands written. Legibility is measured by metrics such as the number of noncomment lines of code and the number of lines longer than a given threshold of characters. Other quality factors, such as robustness and reusability, are manually graded by the instructor.

## *C. Authorship Identification*

Since most work is not done in the presence of the tutor, a tricky concern is to guarantee the originality and authorship of the programs submitted by the students. The following strategies are applied to reduce the risk of plagiarism and to detect it should it occur.

- All activities include a compulsory interview with a teacher, where students have to explain their submissions and answer some questions.
- The formula $\sum_{c=1}^{\mathrm{NC}} \sum_{p=1}^{\mathrm{NP}_c} (0.1 \times (\mathrm{TNS}_c - \mathrm{NSSP}_{c,p}) / \mathrm{TNS}_c)$ is used to grade the Mooshak activities performed

by each student. $\mathrm{NP}_c$ is the number of problems set in the contest $c$, and NC is the number of contests organized. A student is considered as a contestant in the contest $c$ if s/he has a Mooshak account in this contest. $\mathrm{NSSP}_{c,p}$ is the number of contestants who solved problem $p$ of the contest $c$ correctly, and $\mathrm{TNS}_c$ is the total number of contestants in the contest $c$. Note that the larger the number of students who solve the problem $p$ of the contest $c$ ($\mathrm{NSSP}_{c,p}$), the lower is the score for the problem $p$ ($0.1 \times (\mathrm{TNS}_c - \mathrm{NSSP}_{c,p})$). Therefore, if a contestant shares a correctly solved problem with other contestants, a score penalty is incurred by the contestant. This formula is an effective deterrent against plagiarists.

- A useful measure to prevent students from copying from each other is to use a plagiarism detection tool. A plagiarism detection system developed by Cebrian *et al.* [18] is used. Nevertheless, assignments are freshly set for each course to avoid interyear plagiarism.

## IV. Empirical Study

This section gives detailed information on the experiment designed and conducted during the Fall term of 2009.

### A. Aim

The primary aim of the experiment was to compare the effects of an automated assessment system to those of conventional face-to-face classroom teaching, on the acquisition and retention of testing, debugging, deployment, and versioning skills and knowledge. The hypothesis was the following.

1) There would be a positive difference in the testing, debugging, deployment and versioning skills performance scores of students taught using Mooshak when compared to those taught using conventional methods.

### B. Participants

A total of 46 students of mixed gender, age, and educational background gave their verbal consent to participate in the study. A formal sample size calculation was not performed due to a lack of information from previous studies. However, efforts were made to recruit the largest possible sample.

### C. Design

The study employed a randomized controlled design. After recruitment, 46 programming students were randomly divided into two groups. An experimental group of 23 participants had gained experience in programming tools using *Mooshak*. A control group of 23 subjects received the same training, but without using an automated assessment tool.

Data were collected at four points during the study. Baseline data, generated from a demographic questionnaire, were collected from all participants immediately prior to the teaching intervention. During the intervention period, online activities were performed by the experimental group students and automatically assessed by *Mooshak*, whereas homework assignments were performed by the control group students and manually graded by the instructor. A testing, debugging, deployment, and versioning skills and knowledge test was conducted 10 weeks later. Finally, a questionnaire was administered to evaluate the students' experience of the use of Mooshak as an e-learning tool.

### D. Outcome Instruments and Measures

The test tasks were organized into four blocks. Any programming tool introduced during the course could be used. The exam was conducted in two 3-h sessions, distributed over two days. It asked students to perform the following tasks.

- **Testing**: Two testing problems that addressed both white-box and black-box techniques were set. All testing techniques studied were asked: *basic path testing*, *branch testing*, *loop testing*, *data-flow testing*, *equivalence partitioning*, and *boundary value analysis*. Target tools: *gcc*, *gprof*, and *gcov*.
- **Debugging**: Three debugging problems were presented: one consisting of debugging a C program into which some faults had been injected, another related to a memory leak, and the last being a post-mortem analysis of a dump. Students had to choose the most suitable technique for solving each problem: *backtracking*, *induction*, or *deduction*. Moreover, the programs were lengthy enough to call for a *divide-and-conquer* approach. Target tools: *gcc*, *gdb*, *efence*, and *assert*.
- **Deployment**: A deployment problem was set in which a *makefile* file had to be written. Students had to provide rules to build and install libraries, generate documentation with *doxygen*, and create object and executable files. Target tools: *gcc*, *make*, and *doxygen*.
- **Versioning**: A total of six problems related to software versioning was set. The assignments illustrated *cvs* commands to get the status of files; add, update, and remove files; merge branches; get a full log of all the changes made to a file; and display the differences between revisions of files. To carry out this activity, a repository had previously been created on the server. Target tool: *cvs*.

The automated assessment tool was not used for this exam. Participants were assessed by an examiner who used a numerical system to determine scores; a mark was allocated to each of the four blocks. The 12 problems of the in-class, closed-book written test covered various course content areas. Therefore, using internal consistency to check reliability of the instrument was not applicable. However, the degree of reliability of the test scores over time (intraobserver reliability [19]) were determined by means of the median Cohen's $\kappa$ and the median percentage of agreement.

## V. Results

The 46 students who completed the baseline questionnaire had diverse ages, gender, educational backgrounds, and grade averages in the previous academic year. Participants were aged between 17 and 27 years, with most belonging to the 18-year age group, with a mean age of 20.4 $\pm$ 0.7 years and grade point average of 7.3 (out of 10) $\pm$0.5. Most participants were male ($94\%, n = 43$). Two educational backgrounds were represented: Baccalaureate ($85\%, n = 39$) and Senior Technician ($15\%, n = 7$). All participants reported that they had never used an automated assessment system before. Chi-square analysis revealed no significant differences ($p = < 0.05$) when partici-

TABLE III
DETAIL OF THE CLASSIFICATION OF THE SUBMISSIONS BY KNOWLEDGE UNIT. "#p": NUMBER OF PROBLEMS. "#s": NUMBER OF SUBMISSIONS. A: ACCEPTED. WA: WRONG ANSWER. RE: RUNTIME ERROR. SP: NUMBER OF SUBMISSIONS PER PROBLEM

| Tool | #p | #s | A | WA | RE | SP |
|---|---|---|---|---|---|---|
| **gcc** | 7 | 140 | 55(39.3%) | 39(27.8%) | 46(32.9%) | 20.0 |
| **gcov** | 5 | 76 | 31(40.7%) | 20(25.3%) | 25(32.8%) | 15.2 |
| **gprof** | 5 | 74 | 32(43.2%) | 19(22.3%) | 23(31.1%) | 14.8 |
| **gdb** | 7 | 117 | 56(47.8%) | 27(24.7%) | 34(31.5%) | 16.7 |
| **doxygen** | 4 | 76 | 35(46.0%) | 17(22.3%) | 24(28.2%) | 19.0 |
| **assert** | 6 | 92 | 45(48.9%) | 21(23.5%) | 26(29.2%) | 15.3 |
| **make** | 8 | 126 | 59(46.8%) | 32(25.3%) | 35(27.7%) | 15.7 |
| **efence** | 7 | 96 | 44(45.8%) | 25(25.2%) | 27(28.1%) | 13.7 |
| **libraries** | 5 | 92 | 40(43.4%) | 24(25.0%) | 28(30.4%) | 18.4 |
| **cvs** | 7 | 146 | 68(46.5%) | 35(32.4%) | 43(29.4%) | 20.8 |
| **All** | 61 | 1035 | 465(44.9%) | 259(25.0%) | 311(30.1%) | 16.9 |

pant characteristics were compared between the study groups at baseline and at the 10-week follow-up.

## A. *Results of the Online Judge*

A total of 20 students (83%) solved, and 21 (87%) tried to solve, at least one problem. In total, the online judge received 1035 submissions, with an average of 45 submissions per student. The online judge classified around 465 of these as "accepted" (44.9%), 259 as "wrong answer" (25.0%), and 311 as "runtime error" (30.1%). The reader can observe the high value of the "runtime error" submissions, which provides evidence of the difficulty of submitting a well-written script. The percentage of "runtime error" submissions is even higher than the percentage of "wrong answer" submissions. This difference can be attributed to the fact that a number of bugs are fixed when a "runtime error" script is changed and resubmitted, thus sometimes avoiding a "wrong answer" result. More information on the classification of the submissions and the percentages per tool are shown in Table III. The highest number of programs that a student submitted to get an "accepted" was 15.

All problems were to be done individually. They were graded from 1 to 5 according to their degree of difficulty. They were freely chosen by students, and there was no minimum number of problems necessary to pass the activity. The students' outcomes in Mooshak were public. The *Mooshak*'s activities assessment for the experimental group (and the homework assignments assessment for the control group) accounted for 35% of the final mark.

## B. *Skills and Knowledge Test Scores*

Fig. 2 shows the results of the test of programming tools skills and knowledge as two overlapping histograms. This study used one independent variable (learning method) and four dependent variables: the ability to test (TEST), debug (DEBUG), deploy (DEPLOY), and version (VERSION) a software system. Students were scored from 0 to 10. It can be noticed that the experimental group had higher scores in all tests. Means, standard deviations, modes, and medians for both groups are summarized in Table IV. Notice that the values were considerably higher for the experimental group than for the control group. This showed that the online activities encourage students to get much more involved in certain parts of the syllabus, especially those that are

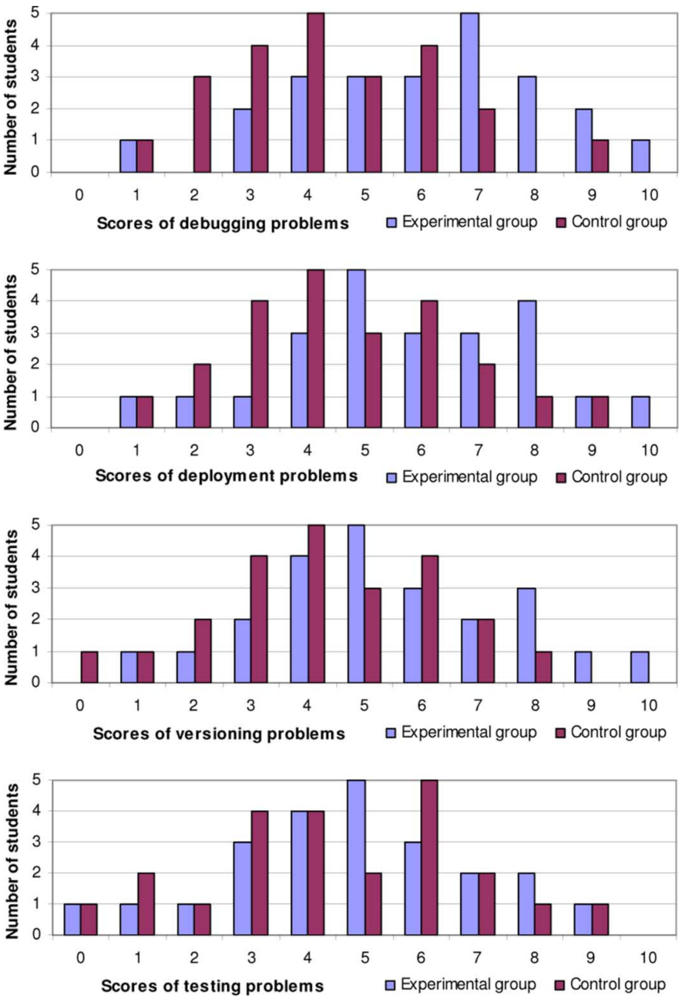


Fig. 2. Score distribution.

TABLE IV
STATISTICAL RESULTS OF THE ACTIVITIES PROPOSED IN *MOOSHAK*. "Exp.": EXPERIMENTAL. "Ctr.": CONTROL. "MN": MEAN. "SD": STANDARD DEVIATION. "MD": MODE. "MDN": MEDIAN

| **Statistics** | **MN** | | **SD** | | **MD** | | **MDN** | |
|---|---|---|---|---|---|---|---|---|
| **Group** | Exp. | Ctr. | Exp. | Ctr. | Exp. | Ctr. | Exp. | Ctr. |
| TEST | 4.78 | 4.47 | 2.23 | 2.31 | 5 | 6 | 5 | 4 |
| DEBUG | 6.04 | 4.39 | 2.22 | 1.94 | 7 | 4 | 6 | 4 |
| DEPLOY | 5.78 | 4.65 | 2.23 | 2.01 | 5 | 4 | 6 | 4 |
| VERSION | 5.43 | 4.26 | 2.25 | 2.00 | 5 | 4 | 5 | 4 |

really exciting for them [20]. As noted in [21], if you want the students to understand something, you have to get them to think about it.

With the usual 95% confidence interval, a $t$-test for independent samples determined that the results for debugging $(t(44) = 2.67, p = 0.005)$, deployment $(t(44) = 1.89, p = 0.03)$, and versioning $(t(44) = 1.86, p = 0.03)$ problems are statistically significant, thus supporting the *hypothesis 1*. However, for the block of problems related to testing, the $p$-value is 0.32 and, therefore, the difference between the two means is not statistically significantly different from zero at the 5% level of significance. The explanation for this can be found in the fact that the *deep* learning required in the testing problems where application, analysis, and synthesis are tested at the higher levels of Bloom's Taxonomy was not exercised in the online activities proposed. Notice that the programming tools *gcc*, *gcov*, and *gprof* are *instruments* to test code coverage in programs, but they do not help students in applying and reflecting on white box testing techniques.

Finally, the relationship between the results obtained in *Mooshak* and the final exam scores of the experimental group students was investigated using Pearson's correlation coefficient. An important positive correlation (0.85) between the number of accepted submissions and the marks of the students in PT was observed.

### *C. Students' Attitude*

The subjects of the experimental group were asked to fill in a questionnaire about their experience with Mooshak. This contained nine questions to be answered by selecting numbers on a five-score Likert-type scale of 0–4. A total of 20 experimental group students completed the survey (about 87%). The Mooshak activities were positively evaluated by the students (mean: 3.6; SD: 1.2). They thought that the competitive nature of the contests led them to solve more problems, faster and more efficiently (mean: 3.5; SD: 1.08). In relation to the anonymity of participants, students would prefer to participate publicly (mean: 3.8; SD: 1.19). This is particularly curious in the light of other studies [22], which claim that students prefer anonymous assessment. It is likely that students were influenced by the existence of an online ranking.

### *D. Limitations*

The positive perception of the Mooshak system may be attributed to the "Hawthorne effect" [23], a special case of novelty effects. Furthermore, as the data collection sessions were in addition to scheduled classes, the timing of the data gathering should have been taken into account when designing this study. Another limitation of the experiment is the lack of a formal sample size calculation, which may have influenced the findings. Therefore, findings from this study must be interpreted with caution, and future research is recommended with a larger sample to investigate the effect of an automated assessment system on the acquisition and retention of testing, debugging, deployment, and versioning skills and knowledge. Despite these limitations, the strength of this study lies in the use of a rigorous trial design.

## VI. Conclusion

In this paper, an automated assessment approach applied to a programming tools course has been presented. In a controlled study, the scores of the experimental group showed that students gained a more solid grasp of concepts related to debugging, deployment, and versioning. This clearly supports the idea that automatic assessment promotes students' ability also to learn programming tools.

## References

[1] C. Douce, D. Livingstone, and J. Orwell, "Automatic test-based assessment of programming: A review," *J. Educ. Resources Comput.*, vol. 5, no. 3, pp. 1–13, 2005.

[2] J. F. Bowring, "A new paradigm for programming competitions," in *Proc. 39th Tech. Symp. Comput. Sci. Educ.*, 2008, pp. 87–91.

[3] G. García-Mateos and J. L. Fernández-Alemán, "A course on algorithms and data structures using on-line judging," in *Proc. 14th Innov. Technol. Comput. Sci. Educ.*, 2009, pp. 45–49.

[4] F. J. Montoya-Dato, J. L. Fernández-Alemán, and G. García-Mateos, "An experience on ada programming using on-line judging," in *Proc. 14th Int. Conf. Rel. Softw. Tech., Ada–Europe*, 2009, pp. 75–89.

[5] P. Brown, "Setting the stage for the culturally adaptive agent," in *Proc. AAAI Fall Symp. Socially Intell. Agents*, 1997, pp. 93–97.

[6] J. English and T. Rosenthal, "Evaluating students' programs using automated assessment: A case study," in *Proc. 14th Innov. Technol. Comput. Sci. Educ.*, 2009, pp. 371–371.

[7] J. P. Leal and F. M. A. Silva, "Mooshak: A web-based multi-site programming contest system," *Softw., Pract. Exper.*, vol. 33, no. 6, pp. 567–581, 2003.

[8] C. Daly and J. Waldron, "Assessing the assessment of programming ability," in *Proc. 35th Tech. Symp. Comput. Sci. Educ.*, 2004, pp. 210–213.

[9] C. Douce, D. Livingstone, J. Orwell, S. Grindle, and J. S. Cobb, "A technical perspective on ASAP—Automated system for assessment of programming," in *Proc. 9th Int. Conf. Comput. Aided Assess.*, 2005, pp. 1–13.

[10] C. A. Higgins, G. Gray, P. Symeonidis, and A. Tsintsifas, "Automated assessment and experiences of teaching programming," *J. Educ. Resources Comput.*, vol. 5, no. 3, pp. 1–21, 2005.

[11] M. Joy, N. Griffiths, and R. Boyatt, "The boss online submission and assessment system," *J. Educ. Resources Comput.*, vol. 5, no. 3, pp. 1–28, 2005.

[12] P. Guerreiro and K. Georgouli, "Combating anonymousness in populous CS1 and CS2 courses," in *Proc. 11th Innov. Technol. Comput. Sci. Educ.*, 2006, pp. 8–12.

[13] P. Guerreiro and K. Georgouli, "Enhancing elementary programming courses using e-learning with a competitive attitude," *Int. J. Internet Educ.*, vol. 10, pp. 38–43, 2008.

[14] B. Beizer, *Software Testing Techniques*, 2nd ed. New York: Van Nostrand, 1990.

[15] G. J. Myers, C. Sandler, T. Badgett, and T. M. Thomas, *The Art of Software Testing*. Hoboken, NJ: Wiley, 2004.

[16] R. Pressman, *Software Engineering: A Practitioner's Approach*. New York: McGraw-Hill, 2009.

[17] B. Bloom, E. Furst, W. Hill, and D. Krathwohl, *Taxonomy of Educational Objectives: Handbook I, The Cognitive Domain*. Reading, MA: Addison-Wesley, 1956.

[18] M. Cebrián, M. Alfonseca, and A. Ortega, "Towards the validation of plagiarism detection tools by means of grammar evolution," *IEEE Trans. Evol. Comput.*, vol. 13, no. 3, pp. 477–485, Jun. 2009.

[19] T. Lang and M. Secic, *How to Report Statistics in Medicine: Annotated Guidelines for Authors*. Philadelphia, PA: American College of Physicians, 1997.

[20] J. Preston and B. Morrison, "Entertaining education—Using games-based and service-oriented learning to improve STEM education," *Trans. Edutainment*, vol. 3, pp. 70–81, 2009.

[21] M. Guzdial and E. Soloway, "Teaching the Nintendo generation to program," *Commun. ACM*, vol. 45, no. 4, pp. 17–21, 2002.

[22] L. M. Regueras, E. Verdú, M. F. Muñoz, M. A. Pérez, J. P. De Castro, and M. J. Verdú, "Effects of competitive e-learning tools on higher education students: A case study," *IEEE Trans. Educ.*, vol. 52, no. 2, pp. 279–285, May 2009.

[23] A. Bullock and O. Stallybrass, *The Fontana Dictionary of Modern Thought*. New York: Harper Collins, 1977.

**José Luis Fernández Alemán** received the B.Sc. (Hons.) and Ph.D. degrees in computer science from the University of Murcia, Murcia, Spain, in 1994 and 2002, respectively.

He is a Professor of programming and software quality with the Department of Computer Science and Systems, University of Murcia, Spain. He has published several papers in the areas of e-learning and software engineering, including journal articles in *Software and System Modeling*, the IEEE Transactions on Education, *Nursing Education Today*, and *Transactions on Edutainment*. He has contributed to many Spanish-funded research projects whose topics were related to software engineering. Currently, his main research interest is in computer-based learning and its application to the fields of computer science and nursing.