Automatic Tools for Diagnosis and Feedback in a Project Based Learning Course

Rubén San-Segundo, Juan M. Montero, Javier Macías-Guarasa, Ricardo Córdoba and Javier Ferreiros
Department of Electronics Engineering. ETSI Telecomunicación (Universidad Politécnica de Madrid)
28040-Madrid – SPAIN { lapiz , juancho, macias, cordoba, jfl}@die.upm.es

Abstract - **In this paper, we present advanced diagnosis and feedback tools to improve student software quality. After several years of quantitative analysis of the relationship between the assigned grades and certain software features, we have been able to characterize high-quality assembly software.**

With these results, we have defined new learning objectives after an instructors' consensus, and we have developed a set of automatic tools that help to supervise how well the objectives have been achieved and to feed this information back to the students along the course. We have successfully used these analysis tools in a new course, with a considerable improvement in software quality factors. In the 2003-2004 academic year, there were 54,7% more subroutines per program, with 48,7% fewer lines per subroutine and an increase of 43,6% in the use of the more complex addressing capabilities. This improvement in quality had a positive impact on students' surveys.

Index Terms - automatic estimation of software quality, Project Based Learning, automatic evaluation tools.

INTRODUCTION

The PBL technique has been successful in both university [1] and pre-university courses [2]. In university teaching it has been applied to a great variety of disciplines: law, medicine [3]; but most applications has been in technical and engineering courses [4] [5]. A comparison to the traditional ways of teaching reveals a greater degree of learning in the case of the PBL technique [6]. The difference is greater when new technologies support this technique [7]. PBL allows increasing student involvement in the learning process, obtaining better results in terms of knowledge and habits acquired by the students. With this technique, they must face a multidisciplinary project aimed at developing new capabilities that complete their instruction to better face the work in a company. Some of these additional capabilities are team interaction, self-learning, assumption of responsibilities, resources management and time planning.

This technique also has several implementation problems: a greater effort in management and coordination, and a more complex and difficult evaluation process. In massive courses (around 200 teams of 2 students), it is very difficult to supervise and feed back to the students continuously. This fact can cause that both students and instructors may focus on the functionality of the project, setting aside other non-functional quality aspects. On the other hand, there are several instructors in our course (7-10) and they must evaluate a disjoint set of students; increasing the risk of a discrepancy in their evaluation criteria. In order to carry out a good supervision and evaluation in massive PBL courses, it is necessary to use automatic tools that help instructors to control and supervise the student evolution and to analyze the evaluation process.

The development of automatic tools for continuously monitoring the evolution of the students in a PBL course is a field of teaching innovation with an increasing interest. The main reason for investing effort in this development is to be able to increase the quality of the learning process without increasing the workload of the instructors, especially when the courses are massive and they are based on projects [4] and when one tries to evaluate both the final result of the process and the associated teamwork [8].

In the last years, there have been several works to develop automatic tools for supervising, feeding back and evaluating student work [9] [10] [11]. Generally, these tools are applied to software assignments and to circuit simulations. In these cases, it is possible to verify the software or hardware functionality in a fully automatic way, using test vectors or case vectors (a set of inputs and their corresponding correct outputs). If we want to develop similar tools for PBL courses, we find the following problems:

- In our engineering projects, planned for a semester, students try to develop a complete communication or control system of medium complexity, including interface modules: sensors, keyboards, screens... In this case, it is necessary an eyewitness verification of the functionality that is very difficult to automate.
- Secondly, the project is not fully specified. In PBL, the target is to foster student initiative and creativity. Because of this, the final systems exhibit important functionality differences from team to team; therefore, it is very difficult to carry out an automatic verification based on standard test vectors. In our course, the students define a relevant portion of the functionality, that can account for more than 15% of the total score.
- Finally, an evaluation process based on test vectors focuses on the functionality or response time, leaving out aspects such as the structure of the developed system, the management of available resources or the scalability of the proposed solution.

On the other hand, when developing an automatic tool for supporting the evaluation, it is necessary to analyze the grades assigned by the instructors in order to fine-tune the tool. This article describes a set of analysis and diagnostic tools for PBL courses that allow:

- To control and supervise the students' process, helping the instructors to feedback about wrong decisions or implementation errors. All these actions can have a very short response time and very low demand of time.
- To support the instructor in the evaluation process by means of quantitative measurements.

DESCRIPTION AND CONTEXT OF THE COURSE

We used the developed tools in the LSED course (Laboratory of Digital Electronic Systems) in the Department of Electronic Engineering at the Telecommunication Engineering School, of the Technical University of Madrid (UPM). This Department is also responsible for several courses focused on the design of electronic systems based on microprocessors: a theoretical course in the 3rd academic year (5th semester) of Telecommunication Engineering (SEDG: digital electronic systems), another laboratory within the same academic year (LCEL: Laboratory of Electronic Circuits), and two optional ones in 5th year on the Electronics specialty (ISEL: Electronic Systems Engineering, and LSEL: Laboratory of Electronic Systems). In the first and second semester, the students must pass a course and a laboratory on standard programming issues, based on Java.

The LSED laboratory is closely related to SEDG because SEDG is previous and it is focused on the same microprocessor (Motorola 68000) and a common set of peripheral devices. Both courses try to make a balance between a reasonable workload and highly formative contents.

LSED is a laboratory with about 400 students attending every year. The students, grouped in teams of two, have to design, build, test and document a complete microprocessor-based system (both HW and SW). The starting point is a description of the system to be implemented (about 30-40 pages) that includes:

- the functional requirements of the system: the scope, a general description and the use cases,
- part of the system analysis: a modular description of the system and a detailed description of the main subsystems,
- some guidelines for the implementation of the system and subsystems: including a proposal of the basic software architecture. This architecture establishes a distribution of tasks among the main process and the sub-processes, making a special emphasis on the use of interrupt routines,
- a tentative planning to help students on how to organize the different laboratory sessions in order to achieve the objectives in a professional-like environment

The students must complete the analysis of the system (the initial specification is always incomplete) and they must carry out the design, implementation, testing and documentation. The target system changes every year and the students must develop a completely functional prototype with the associated documentation. Some of the specifications are open to the student creativity. In order to reach the maximum grade, the students must implement optional functionality improvements on the basic proposed system, accounting for more than 15% of the total score. Some of these improvements are suggested in the assignment document (but they are not thoroughly described) and some of them are fully analyzed and designed by the students. This measure has been very effective for fostering student initiative: our experience shows that more than 80% of the teams provide some new functionality to the basic system we propose.

We carry out the evaluation of each student in two steps:

- The first one is the evaluation of intermediate reports during the semester. These reports help instructors to verify the evolution and originality of the work.
- The second step is the final evaluation based on the complete documentation of the system and an oral examination. The instructors must verify that the prototype follows the specifications and must make individualized questions to verify the authorship of the work, to determine the capacity of each student to explain the obtained results, etc. Other factors that we evaluate are: the quality of the technical writing, the skills for oral communication, teamwork capabilities, etc.

At the end of the evaluation process, the instructors must fill in a detailed evaluation sheet. The global evaluation is ranged in a 0-100 scale including many evaluation items with smaller scales (0-3, 0-5, etc). This granularity of these evaluation criteria increases the objectivity of the evaluation process.

In LSED we teach the students not only the microprocessor capabilities and some practical implementation issues, but we also teach a systemic point of view, involving multidisciplinary knowledge. Microprocessors and programming are the tools to build systems that include communications, control, telemetry, user interfaces, etc. An important point covered by this laboratory is the management of real time components. The proposed learning approach is the use of routines executed in periodic interruptions, that complicates the debugging of the system and the development of the prototype. To help students, the initial description provides some recommendations on how to face the problems of real time programming: concurrence and resource sharing.

Typically, the system proposed is a simplified version (both economically and in terms of time demand) of a consumer electronics device. For example, in 2003 the proposed system was a talking calculator based on the MC68000 and in 2004, we proposed a wireless chat with an infrared link.

I. Web management tools

Management tools can carry out the following actions:

- To manage the enrolment of the students and the assignment of laboratory slots per week (time schedule), because it is necessary to have a list of students and teams (for automating the monitoring) and several e-mail distribution lists (for electronic tutoring and for answering Frequently Asked Questions).

- To link the examination of each team to an available instructor on a certain date.
- To provide the students with additional services: extra-slot booking or electronic access to grades.
- To make the final anonymous survey through the web: in order to evaluate the competence of each instructor and some aspects of the course.

II. Data acquisition tools

These basic management tools are not enough and need the complement of a web tool to obtain monitoring data from students' programs at certain dates:

- To verify the attendance of students to the laboratory in their pre-assigned slots or in extra ones.
- To estimate their degree of achievement of the functional objectives according to the time schedule proposed by the instructors.
- To collect partial electronic deliverables that contain the software developed until that date. Typically, in a semester, students must carry out four or five partial deliveries and a final one, with three purposes: early detection of bad programming habits or errors (before it is too late to fix them); to deter students from plagiarism: a case of partial or full software copy is easier to detect by analyzing the history and the coherence of the deliveries; and the evaluation of the final software delivery is one the key points in the final grading (in addition to the oral examination, the final written report and the monitoring data). By the use of automatic analysis tools, one can get a great deal of measurements to help the instructors to assign an almost objective grade for a rather complex project.

Currently, the system is mainly based on freeware:

- Linux operating system, kernel version 2.4.18
- PHP3 interpreter, for dynamic content generation
- MySQL v. 3.23.49, for database clients and server
- HTTP and HTTPS protocols, using an Apache server v. 1.3.26
- Support software in C, bash, bison, flex and perl

The whole set of tools makes it possible to have an objective snapshot of the course at a certain date, without increasing the workload of the instructors.

III. Collected data

Up to now, we have collected the partial and final deliveries and the final grades from two academic years or semesters.

In the 2002-2003 academic year, the proposed system was a talking calculator based on a MC68000 microprocessor. The system was able to add, to substract and to multiply numbers typed on a matrix keyboard. It was able to read out the operators and the operands through a DAC and a loudspeaker, as the user presses the keys (without losing keystrokes or degrading voice quality).

In the 2003-2004 academic year, the students had to implement a chat system based on an infrared link and a MC68000. Through the matrix keyboard, the user types a new message in a several keystrokes-per-symbol fashion (as in mobile SMS phones); the message is serially transmitted using a simple protocol with one bit for start, one for stop and one for parity.

SOFTWARE QUALITY ANALYSIS AND AUTOMATIC TOOLS

It is not easy to make a precise definition of software quality, although experimented professionals are able to classify software programs in terms of quality and they are able to estimate it reliably. To avoid the difficulties of a explicit formal definition, one can use the final grades assigned by the instructors as a source of expert knowledge. This way, quality analysis is a particular case of a more general problem: statistical feature analysis and pattern matching. A classifier comprises:

- A feature extraction phase: aimed at comprising a program into a set of measurable characteristics. These feature vectors characterize the programs and allow comparing them to each other or comparing them to a high-quality reference. Therefore, as different programs will have different feature values, we could distinguish the good ones from the bad ones automatically.
- A set of programs already evaluated by an expert: their feature vectors can be the reference patterns for comparison. This set, usually called the training database, can be useful for estimating the parameters of the pattern comparison distance (training of the classifier): the more relevant features for evaluation must have a greater weight in the comparison.
- An evaluation phase: using the feature vectors of the training database and the distance formula previously obtained, one can estimate the quality of a new program by means of a sequence of pattern comparisons to the database vectors.

I. Relationship between features and software quality

There are a great deal of quantifiable features that could be related to software quality. In the training phase, we must gather a great set of characteristics and we must estimate their relevance according to the evaluation of the laboratory instructors. In this study, we have analyzed up to 48 basic features of assembly programs:

- The use of CPU resources: such as the data and address registers, the set of microprocessor instructions, the number of different addressing modes that were used by the students…
- Data structures used by the programmer: the number of declared variables, the number of constants, tables or messages…
- Structural characteristics: such as the number and the average length of the subroutines (or the interrupt service routine), the average number of exit and entry points in a routine, the average and the maximal length of a jump…
- Comments inserted in the code: the number of line comments, block comments, etc.

Using the data collected in the 2002-2003 academic year, we studied the Pearson correlation between the feature values and the final numerical grades (Table 1).

TABLE I
RELEVANCE OF THE MAIN FEATURES IN 2002-2003

Main features	Correlation with grades
Number of addressing modes	0.19
Number of instructions	0.53
Number of complex data structures	0.19
Number of subroutines	0.48
Number of exit points per subroutine	-0.15
Number of interlaced subroutines	-0.26
Mean length of jumps	-0.32
Number of commented lines	-0.18
Number of lines of code	0.55

The results in Table 1 deserve a qualitative discussion:

- Complex addressing modes: we intuitively considered that the use of the more complex addressing modes (such as the indirect or indexed ones), could reveal a higher or lower quality in a program. The reason would be that these addressing modes ease the access to complex data structures such as tables or lists and these modes are related to the use of elegant algorithms based on arrays or lists. For many students, indexing is the most complex addressing mode because it involves a simultaneous use of two registers and several sizes of operands (the size of data and the size of each register). Only the best students are able to use it fluently, whereas the other students prefer to avoid it. As a result, the remaining addressing modes are associated to lower quality systems. The use of more addressing modes in the same program is a cue of mastery and it correlates positively with grading.
- Use of other CPU resources: the best students are able to design and implement the more complex systems and seem to use a greater variety of instructions and registers; the students with fewer programming abilities seem to use always the same resources, the ones that make them feel comfortable.
- Data structures: the most relevant feature related to data structures is the use of the more complex ones: arrays (that allows more compact and smarter algorithms), messages (warning or error messages are linked to a better user interface)
- Use of subroutines: as we expected, programs with more subroutines are better programs in general, and the excessive length of one subroutine reveals a flaw in the design (the routine should have been split into several smaller ones). Generally speaking, the students with less subroutines develop almost basic programs with fewer functionality extensions and receive lower grades (without a good set of subroutines, it is very hard to implement an improvement that could add some new functionality to the basic specifications). This explains why the programs with more lines of code are better graded: they characterize programs with more functions. If one analyzes only the basic systems, then this feature is negatively correlated to grades: the simpler programs with more lines are worse than the more compact ones; however, an analysis of the programs with more functions reveals that this feature is irrelevant for them. The structure of the subroutines is also relevant: they should be non-interlaced (non-overlapped) and with just one exit point as the correlations suggest.
- Conditional and unconditional jumps: the use of jumps is related to the use of loops and if-then-else structures; the more complex programs (with more functions implemented) have more jumps, but the jumps are shorter because all of them are local and limited by the average size of the subroutines.
- Comments: in the 2002-2003 academic year, instructors did not assign better grades to the more commented programs, probably because the longer programs had a lower percentage of lines with comments: the students had concentrated their effort on the creative task of adding new functions, without paying the same attention to keeping the number of comments at the same high level.

In addition to this, we must take into account that a certain project proposal can bias the use of some features:

- In 2002-2003, the use of the post-increment and pre-decrement addressing modes was dependent on the development of a specific improvement, and it was positively correlated to the total grade. If we analyze the correlation for those students without any improvement, that feature is irrelevant.
- For the use of the indexing mode or the number of symbolic constants, the problem is just the opposite: it is not relevant for the general student but relevant for the students without improvements.
- If we analyze separately the students with improvements and the students that developed a basic system, some features lose discriminating power under this classification, because they are especially useful for the identification of these two classes, but these features are not so good for intra-class discrimination. For instance, the total number of lines of code or the number of jumps are generally positive (the more lines of code or jumps, the higher the grade), but they are negative for the basic systems. The number of modes or the length of the longest jump are not relevant for the best students, but they are good predictor features for the worst students.

II. Automatic tools for quality analysis

As a first approach, we have designed a simple effective linear classifier based on a vector of weights obtained from the available training data (from human graders). This vector is multiplied by the vector of feature values in a scalar way in order to obtain an estimation of the grade that the students should deserve if they were evaluated by an instructor.

To try to minimize the influence of the small differences between the proposed systems (that change every year), we must normalize the feature values. In PBL, it is convenient to change the specification of the project each year in order to avoid plagiarism from previous-year students, but then the conditions have changed from training to testing. If we assume that the average student is quite similar from year to year, we must avoid the use of absolute values that could be dependent on the proposed system. This way, we are able to predict a relative grade: whether a student is in the top 10% or the bottom 10% students, or so on.

As a by-side product, we can use the tool in the grade revision process: we can show the objective measurements to those students that think their grades are not fair. The comparison between their vector values and the mean vector provides the student with a more objective view of their work, minimizing the controversy.

The students have a second chance to pass the course in September. They have to develop almost the same system, but then one of the proposed improvements is compulsory to pass. This similarity of contents makes the learnt classifier very useful, because it is perfectly tuned for the evaluation of such systems.

MEASURES ADOPTED FOR THE FOLLOWING COURSE

The results in 2002-2003 suggested important warnings about the poor quality of the software developed by students. Because of this, we decided to make an important effort to address these faults.

Although we do not have all the data from 2001-2002 or previous years, the available data and the opinion of the instructors involved in both years reveal a certain improvement in some features (the number of subroutines or their average length). Nevertheless, most of the values of these important characteristics are poor and should be greatly improved (average longest subroutine, the number of exit points per subroutine, the number of commented lines...).

I. Measures to improve the quality of the students' software

Instructors must avoid students to focus just on functional aspects of the system under development, without making an adequate emphasis on stylistic aspects that also define the quality of the system. According to our experience, it is not enough to devote part of the grade to these aspects, because they are apparently secondary for the students, so they are confident on passing the final exam just because the system is working and they are the authentic authors.

If the instructors lack quantitative automatic tools to estimate quality and they cannot show automatic figures to the students in an intermediate revision, students focus mainly in short-term functional aspects. However, if automatic tools are available and instructors are able to show to students that this part of the grade is not only based on subjective appreciations of the instructors (that seem hard to obtain), but also on objective measurements (obtained almost-effortless), they will pay attention to the evaluated quality factors.

The intermediate feedback must be carefully carried out. If the information is given too early, it can be based on an insufficient set of data and normalization can be misleading. The initial phase of a project is the most irregular one, because some students have a faster learning curve than others or they spend more time at the beginning (in order to avoid the stress of the final dates). On the other hand, if we provide feedback too late, it is difficult to fix some bad habits of the students.

The frequency of these monitoring tasks is important too. Instructors must feed back to students at least once during the semester, but feeding back continuously can be negative because students can focus excessively on quality aspects or can try to fool the program in a trial-and-error strategy.

This feedback must not be abstract but specific. For instance, one must not say "your subroutines are too long" but "your subroutines are 81 lines long on average while your fellows average only 51". One should use precise assertions such as "you have a very long subroutine named Receiver that is 215 lines long and more than 50 lines is not advisable for a subroutine because it is longer than the typical size of the screen". This way the student perceive that their specific code have been automatically analyzed, and so will be for the final grading. When instructors comments how to improve the quality of the software, the students will learn how to suppress specific flaws of their programs.

Although this is the main strategy for improvement, it needs some previous documentation:

- the assignment document must clearly state the evaluation criteria: the students must know that a 20% of the grade will depend on how well they perform in quality-related features;
- regularly, we must send explanatory information (based on e-mail or web) on these aspects in order to fix the general concepts or some rules of quality;
- intermediate software deliveries must be compulsory: these deliveries provide the data for the automatic analysis and monitoring, and they are also useful to deter software plagiarism.

TABLE II
COMPARISON OF THE MAIN FEATURES FOR THE LAST TWO YEARS

Main features	Improvement over 2002-2003
Complex addressing modes	72%
Number of different instructions	-2%
Number of complex data structures	26%
Number of symbolic constants	64%
Number of subroutines	54%
Number of exit points	23%
Mean subroutine length	49%
Number of interlaced subroutines	74%
Length of the longest subroutine	69%
Number of jumps	3,4%
Number of commented lines	29%

ANALYSIS OF THE 2003-2004 ACADEMIC YEAR

I. Software quality improvements

In Table 2, we show the improvement of the average value of the main software features from 2002-2003 to 2003-2004. The number of lines of code is quite similar (a 5% increase, from 436.5 instructions in 2003 to 460.8 instructions in 2004), so the assignment in 2003-2004 is comparable to the 2002-2003 assignment in terms of global software complexity.

From these results, we can conclude that:

- There has been a significant improvement in the student software quality, due to the new automatic tools and strategies. This improvement has never been reached in any year before. Although we have increased the maximum grade assignable just to software quality (as we did in previous years), and we have written a more specific documentation about it (this strategy was not new), the real cause for this improvement has been the availability of automatic tools. These tools assist the instructors in objective quality analysis and now the students know that this important factor can be automatically measured.

- Some concepts that were difficult for the students in 2002-2003, in 2003-2004 were perfectly assimilated by the students and they should not be used as quality-discriminating features. They are still important in terms of software quality, but they are useless for prediction in 2003-2004. Nevertheless, as students change from year to year, we must repeat the successful strategy in order to preserve the advances.

II. Evaluation of the student opinion

On 2002-2003 we began making a great emphasis on software quality and we developed the first release of the tools. We improved the results of the previous year, but the improvement was not satisfactory. Only when the full methodology was used, we were able to reduce the difficulty of the course significantly while increasing its interest, worthiness (from 2.44 to 3.22) and global evaluation (from 6.48 to 7.17 in 0-10 scale). The new specific comments on software quality have helped students develop programs in an easier way, and this fact has greatly influenced the good results of the surveys.

CONCLUSIONS

As Project-Based Learning (PBL) needs a great deal of supervision, advanced diagnosis and feedback tools are presented and evaluated in this paper. Significant improvements in student software quality are shown, especially in non-functional aspects. The tools are the result of a thorough study of the relationship between the numerical grades and certain software features.

After this study, we have defined new learning objectives after an instructors' consensus, and we have developed a set of automatic tools that help to supervise the degree of achievement of each objective and to feedback this information to the students along the course. We have successfully used these analysis tools in a new course, with a considerable improvement of software quality factors. In 2003-2004 there were 54,7% more subroutines per program, with 48,7% fewer lines per subroutine and an increase of 43,6% in the use of the more complex addressing capabilities.

We can distinguish three types of parameters according to their relevance in software quality prediction: irrelevant features (their variance has never correlated to the variance of the grades), relevant saturated features (they have been good at discriminating in previous courses, but not now) and relevant unsaturated features (they are still important features for the discrimination of good and bad pieces of software).

Finally, the students' opinion about the course has been improved for all the questions considered.

ACKNOWLEDGMENT

We want to thank José D. Romeral (in memoriam) and the members of the Departamento de Ingeniería Electrónica at ETSIT-UPM, for their continuous effort to offer high-quality education to our students and for all the fruitful comments and suggestions that made this work possible.

REFERENCES

[1] Solomon, G. (2003). Project-Based Learning: a Primer. *Technology and Learning*. Volume 23(6), pp. 20-27.

[2] Chard, S.C. (1992). The Project Approach: A Practical Guide for Teachers. Edmonton, Alberta: University of Alberta Printing Services.

[3] Vernon, D. T. A. & Blake, R. L. (1993). Does problem-based learning work? A meta-analysis of evaluation research. *Academic Medicine*. Volume 68(7), pp. 550-563.

[4] Hedley, M. (1998). An undergraduate microcontroller systems laboratory. *IEEE Transactions on Education*. Volume 41(4), pp. 345-345.

[5] Ambrose, S.A. & Amon, C.H. (1997) Systematic design of a first-year mechanical engineering course at Carnegie-Mellon University. *Journal of Engineering Education*. Volume 86, pp. 173-182.

[6] Ryser, G. R, Beeler, J. E., & McKenzie, C. M. (1995). Effects of a Computer-Supported Intentional Learning Environment (CSILE) on students' self-concept, self-regulatory behavior, and critical thinking ability. *Journal of Educational Computing Research*. Volume 13(4), pp. 375-385.

[7] Coley, R. J., Cradler, J., & Engel, P. K. (1996). Computers and classrooms: The status of technology in U.S. schools (Policy information report). Princeton, NJ: Educational Testing Service.

[8] Barros, M. & Verdejo, M. (2000). Analysing student interaction processes in order to improve collaboration. The DEGREE approach. *International Journal of Artificial Intelligence in Education*. Volume 11, pp. 221-241.

[9] Cheang, B., Kurnia, A., Lim, A. & Oon, W.C. (2003). On automated grading of programming assignments in an academic institution. *Computers & Education*. Volume 41, pp 121-131.

[10] Korhonen, A., Malmi, L., Nikander, J., Tenhunen, P. (2003). Interaction and Feedback in Automatically Assessed Algorithm Simulation Exercises. *Journal of Information Technology Education*. Volume 2, pp. 241-255.

[11] Baillie-de Byl, P. (2004). An Online Assistant for Remote, Distributed Critiquing of Electronically Submitted Assessment. *Educational Technology & Society*. Volume 7. pp 29-41.