
Pergamon

0360-1315(95)00056-9

Computers Educ. Vol. 25, No. 3, pp. 105–111, 1995

0360-1315/95 $9.50 + 0.00

AUTOMATIC SUBMISSION IN AN EVOLUTIONARY APPROACH TO COMPUTER SCIENCE TEACHING

MICHAEL LUCK and MIKE JOY
Department of Computer Science, University of Warwick, Coventry CV4 7AL, England

(Received 24 July 1995; accepted 8 August 1995)

Abstract—The teaching of programming languages will often involve students being assessed by means of programming assignments. The administration of such assessments is a complex and demanding task which is compounded by increasing numbers of students. Many institutions have attempted to address this problem by developing computer-assisted learning and assessment systems. However, in the rush to use technology, they have circumscribed the scope and severely restricted the educational experience to be gained from these courses. At Warwick, we have adopted an *evolutionary* approach to computer science teaching, in which students are exposed to an evolving pool of different tools and techniques from which they can choose according to their own needs and preferences. It is in this context that we have developed software for the automatic submission of assignments. This paper discusses this approach and describes how our automatic submission system fits in with it.

1. INTRODUCTION

The number of students following computer programming courses—either within a computing degree or as part of another degree course—is increasing. At the same time, university staff are under considerable pressure to deliver such courses using available resources with maximum efficiency. A major and critical part of a programming course consists of the programming assignments, typically in the form of programs or choices of programs which the students are required to write. However, the process of testing and marking assessed pieces of software is very time-consuming and can be unreliable if attempted manually with paper copies of programs. Sharing the work between several members of staff is one way of coping with the extra load generated by large student numbers, but it may lead to inconsistencies in marking.

Fortunately, much of the testing and marking process has the potential to be automated. At Warwick we have been developing software which will allow students to submit programming assignments on-line, and which will run those programs against test data. In contrast to other efforts at addressing these issues, this software has been developed in the context of a broad strategy which we will call an *evolutionary* approach to computer science teaching. Such a strategy avoids rigid locally-tailored solutions and instead provides for exposure to a wide range of available tools and methods. It is evolutionary because it is not prescriptive, but allows the system as a whole to grow and evolve with corresponding advances in tools and technology. This paper describes this approach and discusses how our automatic submission system fits in with it. First we discuss why an automatic submission system is necessary and our aims in developing such a system. Then we describe some related developments and outline why these are inappropriate. Finally, we discuss the solution adopted at Warwick which has been developed as part of a much larger picture, and we assess how the system fits in with and contributes to our evolutionary approach.

2. MOTIVATION FOR AUTOMATIC SUBMISSION AND ASSESSMENT

At Warwick, as at other universities, we have in the past relied on students handing in paper copies of programming assignments which are then marked by the appropriate lecturer. This method of assessing students' programming skills is flawed for several reasons.

- A decision must be reached for each submitted program as to the degree it satisfies the problem requirements. If the program is only available as hard copy it is difficult and time-consuming to decide whether or not it works. In addition, the final verdict may even be incorrect.
- It is only possible to demonstrate that a program has been tested on a relatively small set of test data, and it therefore cannot be tested on unanticipated data. Consequently students will often tailor their programs to the test data rather than construct more general programs.
- If a student is required to submit test output from a program, that output can easily be forged.
- The volume of paper required is high, resulting in time spent physically handling it, together with possible errors if documents are accidentally mis-filed.

In response to these problems, the processes of submission and assessment were analysed. It was concluded that an automatic submission system is motivated by three primary concerns.

- The need to be efficient. As student numbers rise, the difficulty of effectively and efficiently managing the manual processes of submitting and assessing asignments, and the corresponding workload placed on academic and clerical staff, increase significantly.
- The wish to demonstrate our commitment to using technology. The application of technology in the organization of computing-based courses provides a strong and immediate demonstration of the relevance and usefulness of the subject matter. This is especially true when concerned with the automation of those tasks that affect students in their learning environment on a regular basis. An automated submission system gives evidence of our own commitment to and enthusiasm for our subject in a real-world environment rather than the laboratory. We want to make use of technology to provide up-to-date facilities in a stimulating environment.
- The need to maintain and improve quality, and to recognize that innovative use of technology is only sensible if high quality results can be achieved. We aim to improve accuracy and consistency and reduce the time taken in assessing assignments. Students put a large amount of effort into writing their programs, and expect—and deserve—thorough and accurate marking of their assignments, coupled with rapid turnaround so that they will receive useful feedback.

3. REQUIREMENTS OF AN ON-LINE SUBMISSION SYSTEM

In considering these problems, we identified a number of issues which would need to be addressed if a feasible system were to be introduced, as one component amongst many in a broader strategy. These cover technological requirements, specific course requirements, and broader educational requirements, all in the context of our evolutionary approach to teaching computer science.

- The system must be easy to use—both for the students and for the lecturer setting and marking an assignment.
- Security is paramount—although we accept that complete security on a university computer network is probably unattainable, we needed to minimise any risks introduced by the system. These include students "hacking" into the system, students' programs accidentally or deliberately damaging the system, and the possibility of submitted documents becoming corrupted.
- The system must be sufficiently flexible to cope with different courses using different programming languages, both interpreted and compiled.
- The system should provide feedback to students, giving some indication of the performance of assessed programs.
- The system must not impact on the use of other tools and techniques in a significant way. Students should still be required to use existing compilers, editors, and other such utilities in working on assignments, so developing a strong familiarity with a range of available methods.

- The system must not constrain the ability of students to initiate and pursue their own learning goals or to develop their skills.

The last two requirements itemized above are particularly relevant to our evolutionary approach. It is important that students are exposed to a broad range of software tools that is encountered not just in an academic environment, but also in commercial and industrial settings. There is minimal educational value in providing software systems that demand significant effort in attaining a familiarity for regular use, but which do not develop skills that are transferable to the workplace. In addition, software systems which deny access or the need for access to widely available tools that are likely to be encountered in commerce and industry severely counteract any benefits to be gained.

By limiting the functionality of our automatic submission systems, we require students to use and gain familiarity with those software tools necessary to complete their assignments. Furthermore, by making a range of tools available but not prescribing the use of any particular tools, we encourage students to learn much more than is demanded by formal course requirements. As more and more students purchase their own personal computers (PCs), this takes on even greater significance. While many software tools and utilities are available for a variety of systems including PCs and larger systems, allowing students to work on their own machines if they choose, an encompassing centralized system is unlikely to be available without a prohibitive amount of effort. CLEM, for example, a hypertext-based system for learning programming [1], has so far been unavailable for home use due to a combination of cost and technical issues. Limiting functionality also improves academic portability [2]. Educational software should be able to cope with changes to the curriculum and to teaching style, avoiding fixed pedagogic content.

Finally, we must consider the problem of keeping pace with technology. This has proved to be a key issue for hardware in education, given the limited resources that are available. As technology progresses, machines purchased relatively recently quickly become outdated and must be replaced as funds allow. This tends to be much less frequent than is desirable. Fortunately, this problem has been largely confined to hardware while software, in education in particular where much is freely available, has been kept up-to-date. However, if this freely available software is replaced with an encompassing locally-developed system, then this immunity is likely to be lost. By contrast, the evolutionary approach allows the set of tools and utilities available to change over time with the technology, providing a continually evolving and current environment.

As a simple illustrative example, consider the case of text-formatters. For many years, a principal means for formatting documents in academic environments was a Unix utility called *nroff*. This is a very basic text justifier for line printers. As technology progressed, newer versions of the program were developed to cater for the new laser printers. At the same time, other developments were taking place. The growing PC market gave rise to several increasingly powerful word-processors, while TEX and LATEX [3] were developed for Unix systems, and established themselves in the education sector. There is now a vast range of tools available for all kinds of machine, and even tools which have huge user bases are being upgraded. For example, a new version of LATEX is shortly to be released. The point of this example is not to document the development of text-formatters, but to show how quickly and radically even standard utilities are improved and upgraded. It is vitally important that students are exposed to current technology, and are motivated to exploit that exposure. Educational software systems that constrain this motivation, either by removing the need to use certain technology or by limiting students to particular tools, must surely deny students important opportunities for learning and developing their skills.

4. RELATED DEVELOPMENTS

There have been other attempts to address some of these concerns, and we examined systems available from other institutions which might have assisted us. These include packages from Collier at Northern Arizona University, Kay at UCLA, Isaacson and Scott at the University of Northern Colorado [4], Reek at RIT [5], and a package called *Submit* developed by Cameron Shelley at Waterloo. All of these packages excited us, but were inappropriate due to reasons of security. We were especially concerned about opportunistic attempts by students to exploit loopholes in the

systems, given that the learning environment at Warwick is intended to encourage and stimulate experimentation.

One package in particular deserves some discussion because of its size and distinct approach. *Ceilidh*, a system developed by Steve Benford *et al.* at Nottingham [6] is a large system that contains many features including on-line exercises and teaching aids. Each course has several distinct components, all of which are accessible from *Ceilidh*. First, general course information such as lecture notes and assignment deadlines, specifications and solution outlines can be viewed or printed. Second, programs can be edited, compiled and tested, with details of the compilation process hidden to a greater or lesser extent from the student. Third, the programming assignments can be marked by the system, and the marks provided as feedback, with completed work being submitted and retained with mark details for further analysis. Finally, model solutions and test data can be made available to be viewed, run and tested.

While such a system is indeed a formidable achievement—and is gaining wider use [7]—in encapsulating in one monolithic structure nearly all of the possible forms of interaction that a student can have on a programming course, it sanitises the learning environment and denies students the variety of useful experience that they would otherwise have. It suffers from very many of the problems that our evolutionary approach is intended to avoid. There is a great temptation in using technology to develop such systems, yet we must be aware of the difficulties that they cause, and of the barriers that they can create in attempting to provide a rich, flexible and stimulating learning environment.

5. AUTOMATIC SUBMISSION AT WARWICK: *BOSS*

In response to these issues, we have developed our own system for automatic submission of assignments. The package we developed, together with Chris Box, is named *BOSS*, and contains a collection of programs which run under the Unix operating system. It is designed specifically for courses which have a large number of students attending, and which are assessed by means of programming exercises. Assessed work must be in a form which can be specified very precisely so that the output from students' programs can be compared with expected output; it is therefore not suitable for courses involving more generalized software design. Introductory programming courses in high-level computer languages are thus the typical target for *BOSS*.

The basic aims of the system are 2-fold: to assist the lecturer in marking assignments, and to provide a form of rapid feedback to students. In this respect, the functions of the system are limited. All development by students of their programs takes place outside the *BOSS* system. This includes the processes of editing, compilation, running and testing. The way in which any student chooses to go about these tasks, and the tools they use, are both issues for the student to decide. There are no constraints imposed by the *BOSS* system on these tasks. Only when the student has a program which they believe to be acceptable is it appropriate to use *BOSS* for obtaining feedback or for submission.

The individual component programs of *BOSS* are as follows.

5.1. The program **submit**

This program reads a student's program, and stores it so that the lecturer can at a later date test it and mark it. It is an easily used program which will conduct a dialogue with the student to ensure that the correct submission is made. Preliminary checks will be carried out on the submitted program, to ensure that it appears to be in the correct language (for instance). The identity of the student submitting the program is verified.

An "acknowledgement of receipt" is sent to the student by email; this contains a code which identifies the contents of their submission. A file only very slightly different (even by just one character) will generate a different code. Thus if a dispute arises, and it is claimed that a different file is assessed to that actually submitted, the code can be used to authenticate that file.

A student can also submit extra files (such as might contain documentation) which will also be available to the lecturer to mark.

5.2. *The program* run_tests

This program, which can only be run by a course tutor, will cause all submissions for a specified item of coursework to be run against a number of sets of data. Time and space limits are placed on the execution of a program so as to prevent a looping program from continuing unchecked, and other steps are taken to minimize the potential for a program to damage the system. The output from the student's program is checked against the expected output for each set of data, typically using a utility such as diff.

5.3. *The program* mark

This utility also can only be run by a course tutor. Initially the tutor is prompted to select one or more students. Each selected student's program is, in turn, made available to the tutor together with the output of run_tests on that program.

5.4. *The program* testsubmit

This program, which can be used by the students, will run the program which they are developing against one of the data sets on which it will eventually be tested, and under precisely the same conditions. Thus a student can check that their program will run correctly under the final testing environment. It is not a method for students exhaustively to test their program.

This program is important both for technical and for pedagogical reasons. Since the *BOSS* system runs under Unix, the Unix environment is crucial to the correct running of a program, and many utilities require Unix variables to be set correctly. In addition, programs may exist in several locations, and a given utility may have different versions. Many systems have, for example, two or more C compilers. So even if a student's program appears to the student to be working correctly, it is not always the case that it will work as expected when run by run_tests.

It is important to students as it provides a confidence hurdle which they can pass, by running their program on a well-chosen data set. They then have a reasonable expectation that their program is well on the way to completion.

5.5. *Discussion*

The *BOSS* system is a tool to allow students to submit assignments, and for those programs to be tested automatically. It is not an automated marking system. It is the responsibility of the individual lecturer to provide a marking scheme which takes account of the results produced by *BOSS* together with all other factors which may be regarded as important (such as program style, commenting, etc.).

Action that should be taken when a student's program does not pass one or more of the tests on which it is run, is again the lecturer's responsibility. It may be desirable to award marks for a partially working program—however *BOSS* does not address that problem. We do not aim to remove the instructor from the teaching loop, but instead simply to assist the instructor in achieving a quicker, more accurate and more consistent assessment of programming assignments. This is important, and should be made clear to students to avoid any misconceptions about the extent and scope of the automated system. It is our experience that students gain confidence from the system, but they are also uneasy about the possibility of its unlimited significance in the assessment process.

5.6. *Benefits*

The *BOSS* system has provided us with a number of benefits without compromising the general approach taken of maximizing exposure to standard tools and utilities.

Large numbers of students can be handled efficiently by the system, with security of assignment submission being assured. Programs submitted cannot be copied by other students, and the possibility of paper submissions being accidentally "lost" is removed.

Secretarial staff do not need to be employed at deadlines to collect assignments, making more efficient use of secretarial time, and the volume of paperwork involved is reduced to (almost) zero both for the lecturer and for administrative and secretarial staff.

The time needed to mark an assignment is reduced considerably, while the accuracy of marking,

and consequently the confidence enjoyed by the students in the marking process, is improved. In addition, consistency is improved, especially if more than one person is involved in the marking process.

6. *BOSS* IN THE EVOLUTIONARY APPROACH

BOSS has been developed in the context of our evolutionary approach, and though it runs under Unix, it was designed in a modular fashion using standard C and conforming to the emerging Unix standard known as POSIX (recently renamed PASC). In this respect, it maximizes portability across different systems, allowing great flexibility. It is also just one of several distinct components that comprise the complete set of tools available. The evolutionary approach is thus intended to stimulate and to encourage students to experiment and develop their computing skills, by placing as few restrictions as possible in the way of individual learning choices.

Course notes, exercises and example programs are available on-line through the use of a variety of standard tools and utilities including hypertext browsers such as Mosaic and Cello, and other information delivery systems such as Gopher [8]. These notes are also accessible by navigating the Unix filestore using generally applicable Unix commands [9]. Students can view these notes in any way they choose, but are encouraged to gain experience of the more sophisticated tools by virtue of their graphical interfaces and good previewing facilities.

Programming assignments will often also require accompanying documentation, and this can be prepared by using any of the variety of text-formatters, word-processors, previewers, spelling-checkers, and any other relevant software that is available either on university or on personal machines.

It should be noted that most documentation of software tools and utilities is provided in the form of Unix on-line manual pages rather than hard-copy paper documentation. This is intended to encourage students to explore and experiment with the system on-line, and so gain experience and familiarity.

In summary, students are provided with opportunities to use a variety of hardware platforms (including PCs and Unix) and software tools for completing their assignments. The only constraint is that they must at some point load their programs onto the central computers so that they can be submitted for assessment.

7. EXPERIENCE IN USE

The system we have running has so far been used on three courses, two involving Pascal and one which covered Unix Shell programming, and each attracting roughly 150 students. As it stands, the system is functioning well. There has been a generally favourable student response, and this is expected to improve once the culture of automatic submission has been established within the Department. In addition, lecturers and tutors have also found the system to be simple and easy to use, and marking times have been reduced significantly with a corresponding increase in consistency throughout.

We hope to extend the system to include extra facilities. We are currently designing an extension to `mark` which will perform automatic checks to indicate possible instances of plagiarism. Note that all future developments are designed to enhance the effectiveness of staff in providing a quick and effective service for students. No developments are intended to have a deleterious effect on the learning opportunities that are available and an important part of our teaching strategy.

The results of using the system in our evolutionary approach have been highly encouraging, and the significant beneficial effects of using the system have already been felt by students, academic staff and secretarial staff alike.

REFERENCES

1. Boyle T., Gray J., Wendl B. and Davies M., Taking the plunge with CLEM: the design and evaluation of a large scale cal system. *Computers Educ.* **22**, 19–26 (1994).

2. Thomas R., Durable low-cost educational software. *Computers Educ.* **22,** 65–72 (1994).
3. Lamport L., *LaTeX: A Document Preparation System.* Addison–Wesley, Wokingham (1986).
4. Isaacson P. C. and Scott T. A., Automating the execution of student programs. *ACM SIGCSE Bull.* **21** No. 2, 15–22 (1989).
5. Reek K. A., The try system—or—how to avoid testing student programs. *ACM SIGCSE Bull.* **21**, No. 1, 112–116 (1989).
6. Benford S. D., Burke K. E. and Foxley E., A system to teach programming in a quality controlled environment. *Software Qual. J.* 177–197 (1993).
7. Benford S. D., Burke K. E., Foxley E., Gutteridge N. H. and Mohd Zin A., Experience using the Ceilidh system. *Monitor* **4**, 32–35 (1993/94).
8. Gilster P., *Finding it on the Internet, Essential Guide to Archie, Veronica, Gopher, WAIS, WWW and Other Search Tools.* Wiley, New York (1994).
9. Kernighan B. W. and Pike R., *The UNIX Programming Environment.* Prentice–Hall, Englewood Cliffs, N.J. (1984).