Software Standards in Undergraduate Computing Courses

Mike Joy and Michael Luck,
Department of Computer Science,
University of Warwick,
COVENTRY,
CV4 7AL,
UK
email: {M.S.Joy,Mike.Luck}@dcs.warwick.ac.uk

Abstract

High-quality software must be robust, reliable and maintainable. The design and coding of such software is no longer a craft; it is an engineering discipline, and the teaching of computer programming must reflect this. Consequently, students need to be aware of the importance of formulating accurate specifications for programming tasks, and of coding programs which correctly implement such specifications. However, the increase in student numbers experienced in recent years has caused manual methods of assessing students' programming assignments to become inefficient, and consistency more difficult to enforce. In order to support and motivate a rigorous approach in the context of these difficulties, we have developed an innovative on-line assessment system for programming language modules which addresses both concerns. In this paper we describe the package and discuss its importance in the delivery and assessment of undergraduate programming modules.

Keywords

programming assessment; programming skills; automated marking; software standards

1 Introduction

As computer science matures as a discipline, the skills required to program a computer system effectively are becoming better understood. The task of writing a computer program is no longer mysterious; it can be divided into separate processes — the "software design cycle" — underpinned by solid theoretical foundations. However, the so-called "software crisis" (Brooks, 1987, Gibbs, 1994) is upon us; theory and practice are no longer in step. It is rare for large programs to be constructed using the wealth of theory and "good practice" normal in other engineering disciplines. In consequence, much commercial software contains errors, and corporate IT budgets are often overspent in attempting to correct those errors and maintain the software.

First year undergraduate computer scientists, fresh from school or college, and excited by the courses on which they are embarking, are naturally fired by an overpowering enthusiasm. This often manifests itself — and rightly so — in unstructured experimentation with the software to which they have access. All too often, however, this can leave them oblivious to the importance of being able to write good specifications for software, and being able to code those specifications correctly. These are practical skills which are needed to overcome the software crisis, and to enable programmers to write correct programs to specification and to time.

If we are to address the problems that cause the software crisis, we must begin to do so at this point, when students are first developing their programming styles and practices.

In this paper we look at aspects of the design and delivery of introductory programming courses which encourage students to adopt good practice. These are considered in the context of innovations at Warwick which relate to the automated submission and testing of programming assignments. The paper is divided into three main sections. First we look at general issues involved with assessing a student's programming ability, followed by a discussion of factors relating to automating the assessment process. Finally we present an overview of the assessment system adopted at Warwick, which

efficiently and securely allows programming assignments to be managed on-line.

2 Programming Skills

The purpose of a programming course is to teach students to code in a particular language; it is not to introduce them to formal specification tools, such as Z (Wordsworth, 1992) and VDM (Jones, 1990), which are generally regarded as a separate discipline. Nor would such a course examine in depth the choice of algorithm for solving a particular programming problem. There are strong similarities between a solution to a course assignment, and a software solution to a problem which might be encountered in an industrial or commercial context. Both demand a well-specified problem together with an accurate programmed solution. Neither allows much, if any, design choice on the part of the programmer over and above that which is necessary (algorithm, modularisation and data structures) to perform the coding.

A client of a software company will have, as a main consideration, the confidence that the company will write *correct* and *maintainable* software for them. If such a company fails to deliver working software, it will incur heavy penalties; a good computer professional is used to working with well-specified problems, and is judged on their ability to solve those problems. This principle should be applied to student assignments.

2.1 Introductory Programming Modules

It is normal practice for a computer science student to be exposed in their first term or semester to an introductory programming module, so that basic programming skills are available to be built upon later in the course. It is important at that stage to ensure that students have understood correctly, or bad habits will be carried over to subsequent course units.

This module typically takes the form of a unit in which the students are taught to program in a high-level language, such as Pascal, Modula 2 or Lisp. A student who has successfully completed such a module is able to write programs in that language

— and most students do acquire that skill. Furthermore, they will have been assessed on that programming ability, and will have demonstrated to their examiners that they are indeed proficient at programming.

Consider, for a moment, how this process would be carried out. Following lectures, seminars and laboratory sessions, a student is given an assignment. This is the point at which the student is required to exhibit their competence in producing an item of software which is of an acceptable standard. The assignment takes the form of a task for which the student must write a program (and probably also produce supporting documentation). The solution will at a later time be handed in, either as a printout on paper, or as a machine-readable copy of the program. The submitted piece of work must then be assessed, and criteria must be chosen by which it can be measured.

2.2 Specification of Assignments

It is vital that a programming assignment be specified with a high degree of accuracy. If the specification of the task is imprecise, assessing a submitted assignment becomes relatively difficult, as conformance with the specification may not be easy to ascertain. However, with a tight specification, the submitted program can be tested against suitable test data, and for each set of data the output of the program can be compared with the expected output.

These mirror the demands encountered in industry. A software house which has produced a program for a client will typically be required to test the program on sets of data, provided by the client, and which the client expects to be processed. The examiner is the client of the student being assessed, and is entitled to make similar demands on the student.

2.3 Marking Assignments

A major criterion which is used to evaluate the student's standard of programming is simply whether or not the student's program correctly solves the problem. In other words, whether or not it implements the examiner's specification. Other factors will

also be taken into consideration when arriving at a mark or grade for that student's submission, such as *readability*, *modularisation* and *maintainability* of the code. However, it would be difficult to argue that a high grade should be awarded to a non-functioning program, or conversely, that a program which implements the specification should fail to reward the student with a good mark.

We divide the measurement of a program into two distinct parts. One part is concerned with the *correctness* of the program, determined by the specification, while the other part is concerned with the *style* in which it has been written and the *supporting documentation*. This partition serves a useful purpose — correctness can be checked but style and documentation are somewhat subjective. It is, in principle, possible to test the correctness of a program with a high degree of accuracy.

By placing a high priority on students producing working programs which conform precisely to the required specification, they are encouraged to adopt good habits from the start.

3 Managing Programming Assessments

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.

It is not sufficient to require a student to submit programming assignments which fulfil the required specifications. The specifications themselves must be of sufficient accuracy and clarity that students will not accidentally be misled — the quality of the tasks set to the students must be high. Furthermore, measurement of the correctness of students' solutions must itself be both correct and seen to be so. Ideally, both the examiner and the student should be able to test submitted programs against the specifications.

Until recently, it was commonplace for submitted programs to be printed out on paper, and for the printed copies to be read and assessed by an examiner. There are serious problems with this approach.

- There will inevitably be *inaccuracies* in the marking process, as even the most proficient programmer will find it difficult to check the actions of a program.
- The amount of *time* needed to read and fully understand a program is substantial.
- If a student is required to submit test output from a program, that output can easily be *forged*.
- It is only possible to require students 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.
- The *volume of paper* required is high, resulting in time spent physically handling it, together with possible errors when documents are accidentally mis-filed.

Automation of the testing and marking process would, in principle, solve these problems. Partial automation is now not uncommon; students are often asked to provide the examiner with a copy of their program, which the examiner can then run and test. There are problems with this approach.

- Extreme care must be taken to ensure *privacy* of submitted programs. For example, if a networked computer system is used, and a file containing the program is made readable to the examiner, it is undesirable for it to be read by other users of the system.
- There are many ways in which the process can be *frustrated*, such as students failing to make their programs "readable" by the examiner.
- A program could contain a *"Trojan Horse"* which, when run by the examiner, might damage the security or integrity of the system. On stand-alone computers, such as PCs, viruses are an unpleasant fact of life.
- The examiner is still required to spend *time* locating the program, compiling it and running it on suitable data before examining results.

Although much of the testing and marking process has the potential to be automated, this is not as simple as it might at first appear. We can test whether a program meets its specification (at least, in part — the problem is in general formally undecidable). That is, we can run a program against various sets of data, and then check whether its output matches that required by the specification. Furthermore, we can measure the source code using various metrics and arrive at concrete indicators of programming style (modularisation, commenting, consistency of indentation, and so on). The former tests make up a major part of the testing process, as discussed above. The latter is also valuable and serves to refine the final mark arrived at — but in any event, it can be argued that measuring programming *style* is an inexact science (Hung et al., 1993). The relative importance of style against correctness when designing a marking scheme for an assignment is a matter best decided for each individual course – for some programming courses, it may be desirable to emphasise style and readablility more strongly.

The numbers of students following computer programming courses — either within a computing degree or as part of another degree course — are increasing. This is the case in the UK and elsewhere. At the same time, university staff are under considerable pressure to deliver such courses using available resources with maximum efficiency. This is a major incentive to implement an automated testing and submission mechanism.

Having established the desirability of devolving a substantial portion of the marking process to the computer system itself, there are specific issues which such a system must address.

- The system must be *easy to use* — both for the students and for the examiner. For the students, in particular, the complexities of learning new programming skills need not be compounded by unfriendly support software.

- *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).

4 The Warwick Solution: *BOSS*

There have been attempts elsewhere to write software tools to perform this type of function. These we examined, but none addressed *all* the concerns noted above. They included:

- *Ceilidh* — a system developed by Eric Foxley et al. at Nottingham (Benford et al., 1993a, 1993b). Ceilidh contains many more features than our system, which we did not require, including on-line exercises and teaching aids.
- A package called *Submit* developed by Cameron Shelley at Waterloo.
- Other packages have been developed by, for instance, Collier at Northern Arizona University, Kay at UCLA, Isaacson and Scott at the University of Northern Colorado (Isaacson & Scott, 1989) and Reek at RIT (Reek, 1989).

A major problem with most of these initiatives is that of *security*, and we finally took a decision that our own software was required. The other packages are written principally as collections of interrelated UNIX$^{\mathrm{TM}}$ shell scripts, a path we did not wish to follow. Rather, we wrote our software using standard C, conforming to the emerging UNIX standard known as POSIX (recently renamed PASC). Our software therefore maximises portability across different UNIX systems, and minimises being compromised by bugs in UNIX shell interpreters.

Systems such as Ceilidh are packages which provide a full programming environment, handling not only submission and testing of assignments, but also providing

tutorial material and a "user-friendly" interface to the machine. Students are then artificially isolated from the underlying operating system. There are strong arguments in favour of such an approach, which we do not discuss here, and it is superficially very attractive to install and use such a system. However, the more complex a software package becomes, the more complex and time-consuming it becomes to *maintain* and *update*. Furthermore, if that system does not match exactly the requirements of the course on which it is intended to be used, it may not be possible to *customise* its functionality.

We therefore resolved to separate the *tutorial* and *assessment* functions of our system, and for the latter to implement a limited and well-directed package which would satisfy our security requirements and would be easy to maintain in the future.

The package we developed, together with Chris Box, is named *BOSS* ("Bob's Online Submission System"), 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 suitable for *programming* courses only, and not for modules which involve more generalised software design. Introductory programming courses in high-level computer languages are thus the typical target for *BOSS*.

The system very much considers a program to be a "black box" defined only in terms of the relationship between input to the box and output from it. We use the concept of "output" from a program in its broadest sense. A program may write to a terminal (text and/or graphics), create and modify files, and alter other aspects of the state of a computer on which it is run. Similarly, "input" can take a variety of forms. If it is felt desirable to impose constraints on the students' programs (for example, conformance to a specific style), by considering a copy of a student's program as itself an item of output the *BOSS* system can be used to check such requirements. In principle, any aspect of a student's assignments which is *measurable* can be regarded as output

for the purposes of the *BOSS* system.

A benefit of using UNIX is the flexibility offered by the operating system, and it is straightforward to compare output from students' programs against expected output. This holds true even when a program exhibits complex behaviour. Furthermore, the algorithm used to determine whether program output matches expected output can be tailored so that more or less approximate matches can be recognised, and is not restricted to exact comparisons of text fragments.

For each assignment for which *BOSS* is used, the examiner is able to create a number of datasets on which submitted programs will be run. Each such input dataset is linked to an output specification which defines the expected output when a program is run on the input data. The specification may take the form of (for instance) a file, the contents of which must be matched exactly. Alternatively, a range of possible outputs can be allowed by using a "regular expression" as the specification, or a shell script to perform the comparison.

We decided to follow the UNIX philosophy of creating a number of software units, each of which can be run alone, and each performing a very specific task. The individual component programs of *BOSS* are as follows.

4.1 The program `submit`

This program reads a student's program, and stores a copy of it so that the lecturer can at a later date test it and mark it. The copied program is protected so that unauthorised users of the computer system cannot read its contents, thus addressing the *privacy* requirement. It is a "user-friendly" 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, generated by the "snefru" algorithm (the *Xerox Secure Hash Function*, ©Xerox Corporation 1989), 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. An audit file is maintained with copies of all such receipts issued. This addresses concerns on *security*.

A student can also submit extra files (such as might contain documentation) which will also be available to the lecturer to mark. The `submit` command does not perform any further processing on the files.

4.2 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. Each student's programs will be run by a dummy usercode which has minimal system privileges, thus minimising the potential for the student's program to damage the system — we believe it would be very difficult for a "Trojan Horse" to be introduced into our system. Time and space limits are placed on the execution of a program, so as to prevent a looping program from continuing unchecked. The output from the student's program is checked against the expected output for each set of data, and the results stored. If a program fails against a particular dataset, the differences between the actual and expected output are also stored.

Most of the datasets are hidden from the students, thus requiring them to perform their own checks on their programs to ensure the programs run on unexpected data.

4.3 The program `mark`

This utility also can only be run by a course tutor or examiner. 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.

4.4 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. It is common for a user to "tailor" their interface to the UNIX system, and what will work for them may not necessarily work for someone else. 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.

4.5 Experience with *BOSS*

We have run the system so far on 3 courses, two involving Pascal and one which covered UNIX Shell programming, and each attracting roughly 150 students. No student has yet broken the security of the system.

The system can be tailored so that its behaviour can be changed. For example, the default system limits on a student's program during testing can be changed. The number of times a student is allowed to submit a program can be restricted. The availability of the `submit` command can be restricted to a specified list of students.

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.

The *BOSS* system has provided us with a number of benefits, including those following.

- *Large numbers* of students can be handled efficiently by the system.
- *Security* of assignment submission is 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, thus *more efficient use is made of secretarial time*.
- The *time needed to mark* an assignment is reduced considerably.
- The *volume of paperwork* involved is reduced to (almost) zero both for the lecturer and for administrative and secretarial staff.
- The *accuracy* of marking and testing is improved, and consequently the confidence enjoyed by the students in the marking process.
- *Consistency* of marking is improved, especially if more than one person is involved in the marking process.
- There is potential for *further checks* to be built into the system, such as automatic checks for plagiarism.

It follows from the necessity of exact specification for a programming task that there is less scope available to students for novel solutions. This might be considered a criticism of the system. However, the point at which the innovation should occur is in the initial program design.

Another drawback of this approach is that by specifying a program simply in terms of expected output for a given input, the internal structure of a program cannot be checked automatically. An example where this would be important might be a program to perform bubblesort on a list of numbers. The algorithm employed by a student must be checked by the lecturer when marking, by examining the program source code. Whether or not a program specification should include requirements relating to the algorithm to be employed is an issue we do not address here.

4.6 Student Response

We sought the views of our Computer Science students on *BOSS*, by means of a questionnaire. These were generally favourable, and most students considered it an easy system to use. The ability to use `testsubmit` to check the conformance of their programs to the specification was also widely appreciated.

The principal concerns expressed fell into two categories. First of all, the user interface is somewhat raw — and in particular when a program *fails* `testsubmit` the messages delivered are not very lucid. This is a fairly simple task to correct.

The second — and more interesting — criticism is that the output expected was *too precisely specified*. *BOSS* is far too "fussy". All the students who have used the system have been first year undergraduates, many of whom have had considerable programming experience prior to joining our course. Many of them are thus used to programming in an unstructured fashion. We wonder to what extent these concerns are fuelled by a "culture shock", simply not being used to being required to follow precise specifications.

5 Future Developments

As it stands, the systems is functioning well. The generally favourable student response has already been discussed above, 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 a module which will perform automatic checks to indicate possible instances of plagiarism.

The user interface is at present quite rudimentary. In the next few months we intend to produce windowed software for the `mark` program, which will help to speed up marking of assignments even further.

Though these extensions are not yet complete, results so far 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.

6 Conclusion

At Warwick we have developed a system which enables students to submit programming assignments on-line, each of which can then be tested, under secure conditions, against its specification. Our system has been successfully tried on several undergraduate programming modules. This has greatly speeded up the process of marking assignments, and has improved the consistency and accuracy with which it is performed. By requiring specifications to be followed precisely, we encourage an appreciation amongst our students that programming is an exact science. This is a significant factor helping us to produce graduates well-placed to help solve industry's software crisis.

7 References

Benford, S.D., Burke, K.E., Foxley, E., Gutteridge, N.H. & Mohd Zin, A. (1993). Early Experiences of Computer-Aided Assessment and Administration when Teaching Computer Programming. *Association for Learning Technology Journal* **1**, 2, 55–70.

Benford, S.D., Burke, K.E. & Foxley, E. (1993). A System to Teach Programming in a Quality Controlled Environment. *The Software Quality Journal* **2**, 177–197.

Brooks, F.P. (1987). No Silver Bullet: Essence and Accident of Software Engineering. *IEEE Computer* **20**, 4, 10–19.

Gibbs, W.W. (1994). Software's Chronic Crisis. *Scientific American* **271**, 3, 72–81.

Hung, S., Kwok, L. & Chan, R. (1993). Automatic Programming Assessment Metrics. *Computers and Education* **20**, 2, 183–190.

Isaacson, P.C. & Scott, T.A. (1989). Automating the Execution of Student Programs. *ACM SIGCSE Bulletin* **21**, 2, 15–22.

Jones, C.B. (1990). In *Systematic Software Development using VDM*. Prentice–Hall, Englewood Cliffs, NJ.

Reek, K.A. (1989). The TRY System — or — How to Avoid Testing Student Programs. *ACM SIGCSE Bulletin* **21**, 1, 112–116.

Wordsworth, J.B. (1992). In *Software Development with Z.* Addison–Wesley, Cambridge, MA.