An Integrated Approach to Courseware

Nathan Griffiths
Department of Computer Science
University of Warwick
Coventry, CV4 7AL, UK
`nathan@dcs.warwick.ac.uk`

Mike Joy
Department of Computer Science
University of Warwick
Coventry, CV4 7AL, UK
`M.S.Joy@warwick.ac.uk`

Abstract

Software engineering is becoming increasingly important as an engineering discipline, and its teaching in universities and other higher education institutions should be of high quality. In this paper we describe a tool (BOSS — the Boss Online Submission System) which aids the education of software engineers. BOSS allows students to submit programming assignments online, and to run black-box tests on their programs prior to submission. Instructors can use BOSS to assist in marking such assignments by allowing submitted programs to be tested against multiple data sets. We describe how BOSS helps in the teaching of specific conceptual aspects of software engineering, and how it addresses some of the practical issues involved in teaching large student numbers in a pedagogically neutral manner.

Introduction

The practice of engineering requires a high degree of precision with respect to specifying problems and to designing and instantiating solutions. Effective engineering education should include a similar level of exactitude in both teaching and assessment. The students' learning experience should prepare them for the commercial context in which they are likely to find themselves upon graduation. In this paper we describe a software system that addresses some of the issues related to achieving such a learning experience in the context of teaching a Computer Science MEng programme, in particular with respect to teaching software and information systems engineering.

In common with other engineering disciplines, software engineering requires a precise and exact specification of a problem, and the design and implementation of a solution. Moreover, software engineering in a commercial setting typically involves a team of engineers working towards a common goal, rather than a single developer working alone, so requiring a precise task decomposition. The education process should reflect these needs by requiring students to develop exact software solutions to problems along with the ability to function as part of a group, and both skills should be assessed. We have developed a software system, BOSS (the Boss Online Submission System), to support such assessment across the range of software engineering tasks, from small programs through to large scale group projects. In BOSS an instructor can give precise specifications of tasks and students are able to test and submit their solutions to these tasks. Students' work is automatically assessed using a combination of black-box test harnesses and software metrics to measure the correctness and accuracy of the submission, along with the general quality of the program code. The results of these automatic tests are then available to the instructor during the assessment process.

From an educational perspective, BOSS allows us to teach and assess students in a more realistic setting, and encourages the ethos of precision from the early stages of the course. For example, students may be asked to construct a software module according to a precise description with a view to incorporating the module as part of a larger system. Using BOSS we can assess such tasks by using unit testing to simulate the inclusion of the module in a larger system. Reliability and consistency in marking are increased, and the staff overhead required is reduced. Our solution is scalable, and is regularly used with class sizes in excess of three hundred students.

In this paper we describe the BOSS system, considering both the pedagogical issues and practical constraints. We discuss our experiences with the system, which has now been used for several years, identifying the strengths and weaknesses of utilising an automated system in general, and the BOSS system in particular.

Software Development as an Engineering Activity

Over the last twenty years software engineering has evolved from a relatively obscure idea, practiced by a relatively small number of enthusiasts, to an accepted engineering discipline[13]. Furthermore, as the importance and pervasiveness of computer software increases, the application of good software engineering becomes ever more important. Today's world is reliant on software for almost all aspects of everyday life, from science and business to entertainment — each of these areas requiring the same fundamental software engineering good practice. As educators, it is our role to ensure that our students, who are tomorrow's software engineers, have the skills necessary to operate effectively in such a world.

Software engineering is an active research field in its own right, and there is an abundance of work on the processes involved, from both academic and industrial perspectives. There are many alternative models and methodologies for software engineering, ranging from the formal[11,12], through the business oriented[3,8], to client focused approaches[1,10]. Each model or methodology has advantages and disadvantages, and is applicable in a certain set of circumstances. However, these alternatives all share certain key practical stages: requirements analysis and problem definition, specification, design, implementation, testing, and integration. The extent to which a given stage is present in a particular model or methodology varies, but with only a few exceptions each of the common software engineering approaches contains these stages. From an educational perspective, it is important that we not only teach students details of specific approaches, but also about these stages. We must ensure that students learn about, and have practical experience in, these key stages that apply to any specific model or methodology. Our aim is to teach students to be software engineers rather than to simply have knowledge of some specific model or methodology at a surface level. In other words, we would like our students to develop the intellectual tools to quickly adapt to new situations and models for software engineering, rather than being constrained by academic knowledge of a small number of specific approaches.

In a commercial setting, software engineering typically involves a team of engineers working towards a common goal, rather than a single developer working alone. It is important that students' learning of software engineering equips them to practice as professionals in such an environment. In addition to the social skills needed to work as part of a development team, there are a number of practical skills that are required. Specifically, students need the ability to:

1. effectively decompose problems into subproblems or tasks,
2. be precise in the specification of tasks,
3. be precise in the implementation of tasks, and
4. integrate components from disparate sources into a coherent whole.

As educators, we aim to give students these fundamental practical skills. In doing this we aim to encourage learning at the deeper levels of Bloom's taxonomy[2], engendering understanding and synthesis, rather than surface learning requiring only knowledge recall. A key part of our approach to encouraging this kind of learning is to give students the opportunity to learn through experiential learning[4]. Experiential learning involves more than just doing — it involves analysing what happens and reflecting upon experiences to extract meaning from what took place. Thus, it is not sufficient for students to simply undertake a software engineering task, but rather they must reason about what they are doing and its implications. For students to learn in this manner they must be given an appropriate framework giving suitable guidance and feedback to encourage reflection.

Teaching Software Engineering

Practical computing courses that involve significant amounts of programming continue to attract increasingly large numbers of students. The use of automated submission and testing of student programs is helpful to the instructor by reducing the time spent on administrative tasks, and increasing the accuracy and efficiency of the marking and feedback processes.

The problems of programming a solution to a course assignment, and those of developing software to address a task in an industrial or commercial context, share many qualities. In both cases, the problem must be well-specified so that the task is clear, and the solution must be constrained to provide an appropriate means of achieving it. The design choices faced by the programmer, apart from those critical to implementation itself (including the algorithm, modularisation and the data structures), are minimal.

For these educational reasons, a programming assignment should be carefully and accurately specified. Without such a precise specification of the task, the assessment of programs can become significantly more difficult, as conformance to the specification may not be easy to judge. A tight specification allows the submitted program to be tested against suitable test data, so that the output of the program can be compared with the expected output for each set of data. The requirement that solutions must conform to specification thus serves the dual purpose of enforcing software engineering good practice and enabling a measurable assessment of the program.

Automated "black box" testing is an important software engineering technique. Allowing students to view — and to use — tests which may be applied during the marking process to software they have written, reinforces the engineering character of the software development process.

Any such automated submission and testing package should be user-friendly in that both students submitting assignments and staff involved in marking them should find it easy to use. In keeping with good practice for educational software, it should not, however, constrain students to work on their assignments in fixed ways instead of allowing them the flexibility of choice in development tools. Likewise, it should not constrain instructors to use particular teaching methods or assessment strategies. Feedback must also be provided to students, giving an indication of the performance of assessed programs and offering material for them to reflect upon the software engineering process.

The Local Context

At a practical level there are many constraints governing the manner in which software engineering is taught, depending on the local context. The authors' context is that software engineering is just one of many components in the Computer Science curriculum. However, software engineering issues pervade many other aspects of the curriculum, since students are required to design or implement software as part of other aspects of their course. Our approach to teaching software engineering is to integrate it within this wider curriculum. Students learn about software engineering in each of their four years of study. However, there is only a single module formally devoted to software engineering alone, the remainder of the teaching being embedded within the context of other course components.

In year one students take a number of programming modules, covering a variety of programming paradigms in order to establish a common base on which to subsequently build. The programming tasks are relatively small, and are undertaken individually. In addition to the explicit aims of teaching students particular programming paradigms and languages, we begin to teach them about software engineering. The tasks that students are set are tightly specified and the programs they produce are assessed, amongst other criteria, according to how precisely they meet the specification. In year two students undertake a formal software engineering module, which has an associated practical task, based on a relatively broad specification of a piece of software. Students are divided in groups of five or six and are required to undertake the main stages of software engineering: requirements analysis and problem definition, specification, design, implementation, testing, and integration. This gives students direct experience of the complete software engineering process, and allows them to put into practice the theory they have been taught, and to utilise their experiences from the previous year (especially regarding the importance of precision). During year three students undertake a large individual project, and in the final year students undertake a large group project, and again they should practice good software engineering. A key difference in the assessment of these projects compared to the tasks from years one and two is that students are also assessed on their ability to reflect on the software engineering process. This is important since, not only does it given them chance to consider the more esoteric aspects of software engineering and teamwork, but it also encourages the reflection that is important to experiential learning[4].

Testing

Rigorous testing is fundamental to good software engineering, and all approaches to software engineering encompass some methodology for testing. One of the most common, and widely applicable, approaches to testing is unit testing where each component of a system is extensively tested to ensure that it functions correctly as an individual unit. Once unit testing has been undertaken and unit functionality verified, components can be integrated to form a complete software system. Integration and validation testing ensures that the integration process has been successful. The importance of testing, and the details of unit testing in particular, are taught to students throughout the four years of their course. The tightly specified programming tasks that we use in year one ensure that students understand how important it is that the "units" they produce comply with the specifications. Students are largely responsible for undertaking their own test procedures, to ensure that their software meets the specification. However, to aid the learning experience we provide a mechanism for students to perform certain "black-box" tests on their software automatically. These automatic tests simply represent a subset of the tests that students should themselves be undertaking as part of their testing regime. Moreover, the tests reinforce the importance of precision since a student's software is incorporated into a wider software system over which they have no control.

Programming Style

Good software engineering is about more than expert programming and strict adherence to a development methodology; there are more esoteric aspects related to program and code structure, documentation, and layout. These aspects can be loosely categorised under the banner of "program style". In a commercial setting, where over time several engineers are likely to be involved in a single project, the maintainability and extensibility of the resultant system is in part dependent on program style. Like good writing, good style is partially a subjective judgement. However, there are certain criteria that are universally accepted, and assessment of the extent to which a program meets these criteria can be aided by an automated tool. These criteria form a set of metrics which can be used to characterise the style of a given program. Metrics, such as the ratio of code to comments, complexity of program structure, and the depth of inheritance of object-oriented programs, can aid the assessment of the quality of a program.

Plagiarism

Plagiarism is a perennial problem in teaching across all disciplines. As student numbers rise, the difficulties in detecting plagiarism also increases. Plagiarism detection is especially difficult for a human to detect in a software engineering environment, where a large number of students are required to produce programs adhering to a tightly defined specification. By definition, if the students are all successful then their submissions will produce exactly the same output for a given input. Thus, the only way to detect plagiarism is to consider the internal structure of the programs themselves. Manually attempting to detect plagiarism across large numbers of submissions is an intractable problem, but detection of plagiarism is a necessary activity in order to ensure the validity of students' submissions.

Scalability

Many of our modules are taken by over three hundred students, and checking each submission accurately for its correctness and precision is a large task. In an educational environment, an instructor should be responsible for actually assessing and giving feedback to a student. However, any automation that can assist in this task is highly beneficial. The allocation of marks and feedback remains role of the instructor, but automation greatly increases accuracy when assessing large numbers of students and significantly speeds up the process.

The BOSS System

The BOSS system for automatic submission of assignments was created in an effort to address the problems described above. The system comprises a collection of programs, each of which performs a different task contributing to the overarching goal of effectively managing the process of submitting programming assignments on-line. Although BOSS was designed specifically for courses with large numbers of students, assessed by means of programming exercises, it can be used on other courses as simply a collection tool. It can also be applied to courses where the submitted work does not consist of computer programs, but is in a format suitable for automatic processing.

The individual component programs of BOSS are designed to be used by two kinds of individual. First, students are able to submit programs and gain feedback. Second, instructors and any course tutors involved in assisting the instructor must be able to gain access to the submitted programs in order to test and mark the student submissions.

The structure of the system reflects the conceptual division of the software into three core modules, each well-defined, which can be treated as largely independent components. These are represented as the ovals at the top of Figure 1, and address submission of assignments, their testing and their marking. These components offer the following functionality.

Students may submit programs on-line by means of a user-friendly program that conducts a dialogue with the student to ensure that the correct submission is made. The submission is stored and simple checks are carried out (to ensure the correct programming language is used and to verify the student's identity, for example), so that the instructor can subsequently test and mark it. A receipt is emailed to the student confirming the submission and including signatures of the files submitted for verification purposes.

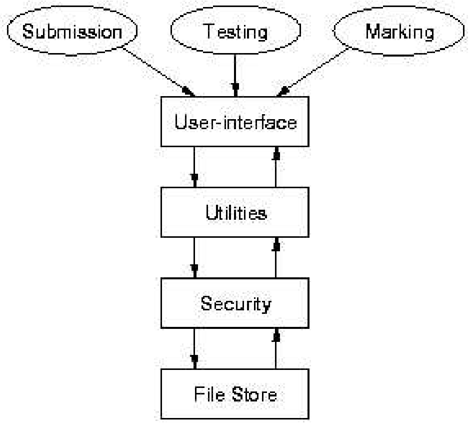

Figure 1. System overview

All submissions for a specified item of coursework can be run against a number of sets of data. The output from the students' programs are compared with the expected output for each set of data, either as text, or using JUnit unit tests. 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 minimise the potential for a program to damage the system.

Submissions, the results of the testing process, and the results of calculating a selected set of metrics on the submissions, can be inspected on-line by authorised staff. Anonymity is preserved by storing data by University ID number (or other code used for the purpose).

Students can test their programs themselves by running them against one or more data sets on which the programs 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. This ensures that the program will work as the student expects when being tested and marked. In addition, it provides students with confidence that their submitted work does pass some minimal requirement.

Final marks are stored in a SQL database and correlated with information from the University database (names and courses versus ID numbers and course registration, for example) to produce final marksheets for examination secretaries.

The BOSS system provides a plagiarism detection tool[6], bringing to the instructor's attention submissions that share many similarities. The criteria for similarity is fully configurable so to be applicable to different tasks. Indeed, it is not restricted to programming tasks, but can also be used for essay based assessments.

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 instructor 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 instructor'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 its significance in the assessment process.

The History of BOSS

When BOSS was originally introduced[9], it was coded in C, and ran with a command-line interface on a single UNIX machine. Since then, it has been re-coded as a Java client-server package, using RMI technology[5] for remote access. Both client and server are platform-independent, and the interface is a GUI with a standard Java look and feel. Data is now stored in an SQL database into which student registration data from the institution's Student Record System may be included via an intermediate database, called Coresoft[7].

It has been deployed on several courses, including those covering Pascal, UNIX Shell, C++ , Standard ML and Java, each course attracting up to three hundred and fifty students. As it stands, the system is functioning well. There has been a generally favourable student response, and this has improved as the culture of automatic submission has become established within the Department. In addition, instructors 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 have also introduced the system into a second-year course which covers the practical application of software tools. Though this normally requires a slightly more involved testing regime, the BOSS toolkit provides a very adequate and appropriate means of automatic submission and testing. More importantly, perhaps, students have had virtually no difficulty in using the system. This seems to imply that the newly established culture has taken root, and that our initial efforts at integrating the system into the fabric of the degree courses are paying off.

In 2002 the software was made open source, and can be downloaded as a single package[14,15].

The Future for BOSS

We believe that BOSS successfully addresses the problems of online submission and assessment in our local context, and there is no intention to expand the functionality of BOSS significantly beyond that which it currently supports. Our methodology of building outwards from a core functionality reflects our concern for ensuring that the tools we use on our courses do not cause problems in the learning process. Future developments will simply keep the system in step with technological developments, both in hardware and software. For example, during Summer 2003 we intend to deploy an enhanced user interface (applying current best practice in HCI design), a web-based interface, and improvements to the installation harness.

Future work will also include the provision of additional measuring harnesses. BOSS currently includes harnesses for testing, software metrics, and plagiarism detection. We intend to continue to refine these existing harnesses, and to consider the inclusion of further harnesses to aid the assessment process. For example, we are considering mechanisms for automating the testing and assessment of programs which have a graphical user interface.

Experiences with using BOSS

The BOSS system has provided us with a number of benefits without compromising the general approach taken of maximising exposure to standard tools and utilities. Large numbers of students have been 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 can be reduced to (almost) zero both for the instructor and for administrative and secretarial staff.

More importantly, perhaps, 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.

We sought feedback from students by means of questionnaires which required students to comment on their experiences of using the system, and also questionnaires which required numerical responses for questions relating to system use. These were generally favourable, and most students considered it an easy system to use. The ability to use the utility to test programs in advance of submission to check the conformance of their programs to the specification was also widely appreciated.

The principal concerns expressed fell into two categories. The first of these covered minor criticisms about the user-interface and the specific messages that the system provides to students when a program fails the test utility. Many of these criticisms have since been addressed in the latest version of the BOSS system and, as noted above, we are continuing development so that the user-interface is improved still further.

The second — and perhaps more interesting — category of criticism was that the output expected was too precisely specified. BOSS is far too “fussy”. This criticism relates to the format of the output specified such as in the precise layout of tabular output, and also to some students’ desire to design their own user-interfaces such as by establishing interactive prompting for input. This is an important point, for it seemed to reflect the preference of first year undergraduates who had had considerable programming experience prior to joining our course. Many of them were used to programming in an unstructured fashion and were unused to being required to follow precise specifications.

Issues Encountered

Where students are asked to produce a complete program, rather than a software unit, the current BOSS system requires a textual interface to be specified. This is adequate for most of the functionality that we wish to assess, and it certainly emphasises the need for precision of software. However, in some situations a textual interface is not ideal, and we would prefer to specify a graphical interface for a programming task. Although BOSS can be used for the submission of graphical programs, and can be used to assess software metrics and detect plagiarism, it cannot currently be used for testing such submissions. As noted above this is likely to be an important area of future extension.

In common with textual interfaces for programs, a high degree of precision is required when defining unit tests for software component tasks. The task descriptions given to students must include a specification of the inputs to the module, and the expected outputs. However, in many programming paradigms there is much scope for ambiguity, since there may be several alternative data types that can be used for a given concept. These data types are typically incompatible with each other, and have subtly different features. Therefore, when setting software component tasks it is important that the instructor be precise about the exact interface that is expected, down to a detailed definition of the data types and structures used.

At an administration level BOSS relies on a database of students, modules, and staff. In related work we have developed an intermediate, institution independent, database (Coresoft) in which this data is stored[7]. We currently have a mechanism for an automated daily update of this data from the University's central Student Record System. However, due to internal procedures, and the propriety nature of the Student Record System, this is not a smooth procedure. BOSS and Coresoft are both open source, and so are available for other institutions to use locally. The use of this intermediate database ensures that BOSS does not rely on a particular Student Record System.

The propensity of students to submit work close to the deadline causes certain scalability issues. In particular, the load on the system tends to be fairly low until a few days before a deadline, at which point it rises and continues to rise until the deadline passes. This means that although the number of student submissions through the system during any one academic year may be relatively low, these submissions tend to be bunched together resulting in prolonged periods of low load followed by short spikes of very high load. This has implications both for the reliability of the software and the hardware. At a hardware level the system must have the physical resources (i.e. network and disk bandwidth, memory and processor capacity) to cope with several hundred submissions within a short space of time. At the software level this is compounded by the fact that often several students are submitting simultaneously, and so there are issues relating to concurrent database and file access. BOSS is in general a stable system, and the model is certainly scalable, however our experience is that hardware constraints can cause limited problems when a very large number of students attempt to use the system simultaneously.

Automatic tests can be made available to students in addition to being run by the instructor during the marking process. Originally, we would make just one test available to students in order that they could ensure that their programs would work as expected in the test harness — this is particularly important where the programming language is heavily dependent on the environment (UNIX shell, for example). Recently, multiple tests have been made available to students, partly in response to student feedback that a single test is insufficient.

Whilst this is pedagogically beneficial, since we expect that students will test their programs themselves and use the BOSS tests as a final "sanity check", the pattern of use does not match our expectations. Many students do not write their own tests and instead incorporate the BOSS test harness into their software development cycle. This results in poor testing by those students, and a restricted learning experience (since they have not fully thought through the testing phase of the software lifecycle). Furthermore, an unexpectedly high load is occasionally placed on the BOSS software, and the machine on which it is running.

Evaluation

As described above, the most common student complaint about BOSS is about its “fussy” nature. For example, the output from a program task may be specified to be “hello world”, but a student may produce a program that outputs “Hello world” (which has different capitalisation). Other examples of such differences are whitespace, and differences in output due to operating system features (e.g. line feeds are different under Linux/UNIX and Windows). This raises the question of whether a submission meets the specified criteria. From an engineering perspective, since their submission clearly differs from the criteria, we can confidently say that the specification is not met. The justification for this is to imagine the student’s software forming part of a larger system that requires a particular interface. From a semantic perspective, however, it is clear that the submission is not significantly different from the specification, although the detail is different. We must, therefore, make a pedagogically informed decision about how to handle such situations. Since we are concerned with teaching software engineering we tend towards the view that such a submission does not meet the specification. However, give our pedagogical concerns it is important that this is correctly perceived by the students, in order for them to reflect upon the implications. BOSS gives an instructor the opportunity to give students suitable feedback, explaining the importance of such seemingly subtle differences. However, an instructor can also choose to ignore such differences if they deem it appropriate.

Prior to university, students tend to be conditioned to specifications being relatively loose — if they are creative and add more functionality are usually rewarded. At the university level, however, we start by teaching them the importance and value of precision. Thus, if they are asked to develop a program with certain functionality, it should have exactly that functionality, no more and no less. An appreciation of the importance of such precision is fundamental to their professional development as software engineers. In the later years we combine these two views — precision is required and is fundamental, but students are able to negotiate a suitable specification if they feel that certain functionality is valuable.

The BOSS system is now relatively mature, and broadly has all the required functionality. However, student expectations and the fast moving nature of the field give rise to something of a moving target. In particular, student expectations of a suitable user interface to the system changes over time. BOSS originally had a simple textual interface, but has developed to included a graphical front end. As computer ownership amongst students has risen, along with the proportion of students having home broadband Internet connections, the desire for a web based interface has risen, and we are currently incorporating such an interface. It is likely that as future technologies emerge, similar changes will be required. We must also ensure that BOSS supports suitable programming paradigms and languages. The system currently supports a range of languages, however as new ones emerge, we will need to adapt to them.

Conclusions

We have described BOSS, a software tool which allows students to submit programming assignments and other coursework online, and allows automatic tests to be run on submitted programs. The software also includes plagiarism detection modules, and a database which is structured to facilitate the upload of student registration data from institution student record systems. We have used BOSS over a number of years on our Computer Science degree courses, and it has benefited our courses by speeding up and increasing the accuracy of the assessment process.

The automatic testing process pervades the students' learning experiences, and reinforces the engineering character of the software development process. BOSS is a pedagogically neutral learning support tool which does not dictate the teaching approach, leaving the instructor free to adopt any teaching methodology, whilst retaining the precision and exactitude needed when viewing software engineering as an engineering discipline.

Bibliography

1. K. Beck, *Extreme programming explained: Embrace Change*. Addison Wesley, 1999.
2. B. S. Bloom and D. R. Krathwohl, *Taxonomy of Educational Objectives: The Classification of Educational Goals, by a committee of college and university examiners. Handbook I: Cognitive Domain*. Longman, 1956.
3. G. Booch, *Object-oriented analysis and design: with applications.* Benjamin Cummings, 1994.
4. G. Gibbs, *Learning by Doing*. FEU Longmans, 1988.
5. W. Grosso, *Java RMI*. O'Reilly, 2001.
6. M. S. Joy and M. Luck, *Plagiarism in Programming Assignments*. IEEE Transactions on Education 42(2), pp. 129-133, 1999.
7. M. S. Joy, N. Griffiths, M. Stott, J. Harley, C. Wattebot and D. Holt, *Coresoft: A Framework for Student Data*. Proceedings of the 3rd Annual Conference of the LTSN Centre for Information and Computer Sciences, LTSN Centre for Information and Computer Sciences, pp. 31-36, 2002.
8. J. Kerr and R. Hunter. *Inside RAD*. McGraw-Hill, 1994.
9. M. Luck and M. Joy, *Automatic submission in an evolutionary approach to computer science teaching*. Computers and Education 25(3):105-111, 1995.
10. R. Martin, *Agile Software Development*. Prentice Hall, 2003.
11. H. D. Mills, M. Dyer and R. C. Linger, *Cleanroom software engineering*. IEEE Software 4(5) 19-25, 1987.
12. B. Potter, J. Sinclair and D. Till, *Introduction to Formal Specification and Z.* Prentice Hall, 1996.
13. R. S. Pressman, *Software Engineering: A Practitioner's Approach*. McGraw-Hill, New York, 2001
14. http://www.dcs.warwick.ac.uk/boss/ (The BOSS home page)
15. http://sourceforge.net/projects/cobalt/ (The BOSS SourceForge.net project from which the package can be downloaded)

Biographical Information

Dr NATHAN GRIFFITHS is a lecturer in Computer Science at the University of Warwick, UK.

Dr MIKE JOY is a senior lecturer in Computer Science at the University of Warwick, UK.