The BOSS Online Submission and Assessment System

MIKE JOY, NATHAN GRIFFITHS, and RUSSELL BOYATT
University of Warwick

Computer programming lends itself to automated assessment. With appropriate software tools, program correctness can be measured, along with an indication of quality according to a set of metrics. Furthermore, the regularity of program code allows plagiarism detection to be an integral part of the tools that support assessment. In this paper, we describe a submission and assessment system, called BOSS, that supports coursework assessment through collecting submissions, performing automatic tests for correctness and quality, checking for plagiarism, and providing an interface for marking and delivering feedback. We describe how automated assessment is incorporated into BOSS such that it supports, rather than constrains, assessment. The pedagogic and administrative issues that are affected by the assessment process are also discussed.

Categories and Subject Descriptors: K.3.1 [**Computers and Education**]: Computer Uses in Education—*Computer Managed Instruction*

General Terms: Algorithms, Languages

Additional Key Words and Phrases: Online submission, programming languages, automated assessment

1. INTRODUCTION

The number of students enrolling in degree courses in UK universities has increased substantially over the past few years, leading to large class sizes and increased student–staff ratios. A specific problem arising from this concerns the substantial resources required to manage the assessment of practical exercises, so that students receive accurate and timely feedback, which will benefit their progress.

Automation of the assessment process is a potential solution, facilitated by the ubiquity of Internet access and the relative affordability of computing equipment. There has been a rapid expansion of tools, both commercial (such as WebCT [WebCT 2004] and Questionmark Perception [Questionmark 2004]) and within the academic community (for example, the CASTLE Toolkit at the University of Leicester [Leicester University 2004] and TRIADS at Derby

Authors' address: Department of Computer Science, University of Warwick, Coventry, CV4 7AL, United Kingdom; email: {M.S.Joy,N.E.Griffiths,R.Boyatt}@warwick.ac.uk.

University [CIAD 2004]). Most such products include assignment submission and automated assessment as part of the software functionality and deal effectively with assessments, which can be formulated as simple questions (such as "multiple-choice" or "text entry"). This is appropriate for "shallow" or "surface" learning, where knowledge and comprehension are being tested, at the lower levels of Bloom's Taxonomy [Bloom and Krathwohl 1956].

Computer programming is a *creative* skill, requiring "deep" learning, and one which the student must practice in order to master. Existing generic tools do not address such skills, and although there is substantial literature defining best practice for the use of such tools [Bull and McKenna 2001], it has been argued that simple questions *cannot* be used to measure deep learning [Entwistle 2001].

Computer programs are, in principle, ideal subjects for automated assessment. Not only can the *correctness* of a program be measured, but also its *quality*, by the application of *metrics*. Furthermore, due to the regularity of program code, techniques for *plagiarism detection* can be easily incorporated into the automated process. A system is needed to support academics in the assessment of student submissions through collecting submissions, performing automatic tests on them, checking for plagiarism, and providing an interface for marking and delivering feedback.

The "BOSS" Online Submission System has been developed over a number of years, as a tool to facilitate the online submission and subsequent processing of programming assignments [Luck and Joy 1999; BOSS 2004]. In this paper, we discuss the recent development of BOSS as a Web-based tool, which supports the whole of the assessment process, based on the application of engineering principles to the automation of the process, together with a pedagogical foundation, which does not constrain the teacher to present or deliver their assessment material in any given style. We cover new support for software metrics and for unit testing, which allows for a rigorous software engineering-based approach, together with the incorporation of novel technologies for supporting the complex administrative components of the process.

2. AUTOMATIC ASSESSMENT

There is no single correct approach to the problem of assessing programming assignments. Different practitioners may adopt different strategies, depending on the specific aims and objectives of the course they are teaching and on their own style and preferences.

BOSS is a tool for the assessment of programming assignments, which supports a variety of assessment styles and strategies and provides maximum support to both teachers and students. Within this framework, the teacher has access to automatic tools to assist in the assessment process, which can be used as much (or as little) as the teacher deems appropriate.

2.1 The Assessment Process

The process of marking a programming assignment includes three principle components, as illustrated in Figure 1.

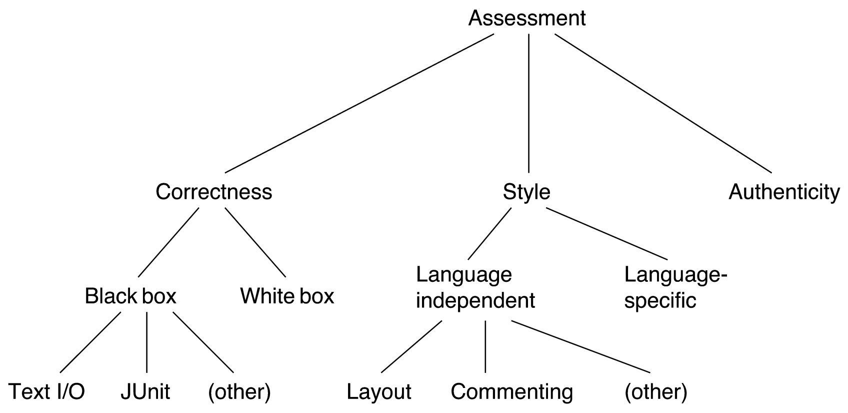


Fig. 1. The assessment process.

The first component, *correctness*, relates to the extent to which a program's functionality matches that of its specification. The second, which we refer to as *style*, describes those attributes of a program that a student's submission is expected to display, but which are unlikely to be explicit in the program specification, and allow for a limited amount of interpretation. The final component, *authenticity*, covers administrative tasks including verification of the student's identity and checks for plagiarism.

These components are conceptual rather than definitive. There are categories for marking, which can be included in the program specification, or can be regarded as stylistic. For example, the performance characteristics of a program may be formally specified and can thus be checked for correctness, but may alternatively be considered as optional (but desirable) program attributes.

2.2 Support Assessment, Not Constrain

Our original motivation was for a tool, which would support computer programming modules, but it was also clear that an effective tool would—with minor adjustments—also be supportive of other types of material. The package, therefore, separates the purely administrative operations (the online submission functionality and data management) from the programming-specific features (automatic testing and marking of students' programs).

2.3 Pedagogic Foundation

There appears to be no agreed set of criteria, which measure a "good" program, and different academics and software practitioners stress different aspects of the process. However, we can identify the following as criteria which are commonly applied:

- *Comments in code.* Best practice dictates that programs should be well documented within the code itself, in order to ensure that the program is understandable and maintainable by third parties (and, indeed, the programmer themselves in the case of large projects).

- *Code style*. Best practice also dictates that programs should have a clear layout, with meaningful choices of identifiers and method names. Again, students' programs should have a code style appropriate to the assessment task. In general, programs should be clear to read, although programs written for specific applications, such as those designed to take up minimal disk space and have minimal memory overhead, may trade clarity for efficiency.
- *Correctness of code*. Programs should be correct, both through adhering to the syntax and semantics of the language used, and by having the required functionality.
- *Code structure*. There are typically many alternative ways of writing a program for a particular task. Students should make use of appropriate language constructs in their code and should structure their code into appropriate modules or classes.
- *Code testing*. Programs should be rigorously and methodically tested using a suitable testing regime.
- *Use of external libraries*. Many programming languages have external libraries of functions for achieving particular tasks. Students should, *where permitted*, make effective use of these libraries.
- *Documentation*. Programs should have supporting design, user, and system documentation.
- *Choice and efficiency of algorithm*. There are typically many alternative methods to program a solution to a particular task. Programs should use appropriate and efficient algorithms.
- *Efficiency of code*. The implementation of the chosen algorithm should be efficient and appropriate language constructs should be used.

These are broadly specified attributes and not necessarily straightforward to define. Not all are applicable to all programs and stylistic criteria, in particular, are subject to individual preference and interpretation. It is, therefore, important to incorporate tools to support criteria, which *can* be automatically measured (such as test harnesses to evaluate program correctness), leaving others in the hands of the teacher, and providing tools to aid the academic judgement where possible.

2.4 Automatic Testing

Code correctness is a marking criterion which is often perceived by students as of primary importance ("Does my program work?"), in contrast to the arguably more subtle requirements for stylistic or algorithmic detail. It is also fundamental to the software engineering process—an incorrect program is seldom useful.

Correctness may be defined by specifying the expected output of a program (or part of a program) for a given input. Precisely *how* the input and output should be described is dependent on the type of program and may take the form of text or data files, or of data values within the program itself. BOSS evaluates correctness by the application of *automatic tests* and two paradigms are currently employed, although the software is structured to allow the incorporation of further testing paradigms in the future.

The first paradigm defines input and output as data files and a test is constructed by specifying the content of the expected output file (or files) for given input data files. A script (such as a UNIX shell script) may be incorporated to postprocess the output generated after a test. Although this is a simple "black-box" technique, which is common for test harnesses and is used in other systems such as CourseMarker [Higgins et al. 2003]. It has the advantage that (almost) any automatic test can be specified in this manner. Furthermore, if the data files are assumed to be text, then each test can be described clearly and concisely and, hence, made accessible to students.

The second approach (which only applies when Java is the language used) uses JUnit tests [Lane 2004]. In this case, input and output are specified as Java objects and a test is constructed by specifying a method to be run, taking the input object as argument and returning the expected output object. This paradigm has the advantage of compatibility with development environments, which support JUnit testing and consistency with both classical and agile development methodologies.

Since the automatic tests will run code written by students, there is a danger that a student's program may (accidentally or otherwise) perform an unsafe operation, potentially damaging the system on which the test is run. The *test harness* used by BOSS protects the system against unsafe or malicious code, using a variety of security techniques. BOSS delivers these paradigms of testing in two modes. The first mode is available to students at submission time to enable them to gain immediate feedback (and allow them to resubmit in the light of this feedback, if they wish). The second mode is postsubmission and allows the course manager to run a batch job of tests after the final submission deadline. This second mode is typically linked to marking categories and creates the starting point for the marking process. The availability of automatic tests both to students and securely to staff, allows their use either as a formative resource, or for summative evaluation purposes, or as a combination of both.

2.5 Automatic Measurement

Assessing a program requires an evaluation of some, or all, of the attributes described earlier in this section. Many of these are subjective and cannot easily be assessed automatically. For example, how well a student has structured and commented their software, or whether they have used appropriate language constructs is better assessed manually by a human marker. We can, however, perform a limited automated assessment of a program to aid the marker in this process. BOSS provides a set of simple program metrics, such as number of comments, percentage of methods declared abstract, etc., which can support the marker in assessing the subjective attributes of a program. The software is extensible and inclusion of other metrics is straightforward.

2.6 Submission and Authentication

A primary administrative function of BOSS is the online collection and storage of work submitted by students. This part of the process requires security features, including the following.

- The identity of a student using the software is verified.
- Integrity of files submitted by a student are assured.
- Transmission of data between student and the system, and the data stored on the system, is protected from unauthorised access.
- Appropriate audit trails are in place so that all parts of the process can be checked.

Source code plagiarism has become a serious problem. Assessed assignments in larger modules may consist of hundreds of submissions, which makes manual comparison, for evidence of plagiarism, of all possible assignment combinations impracticable. BOSS incorporates novel plagiarism detection software [Joy and Luck 1999; White and Joy 2005], which compares submissions automatically to seek for evidence of plagiarism and, if evidence is found, to present the offending submissions to the teacher for manual comparison.

2.7 Administrative Support

An important, but essential, part of the process is its incorporation into the wider institutional processes. Tasks such as the provision of accurate lists of students are central to the smooth operation of software such as BOSS. This has been facilitated by the incorporation of the new "Coresoft" database schema [Joy et al. 2002], which provides an abstract view of the software, and which allows the import of data from, and export to, other databases and data repositories, with minimal software development.

The online marking process is simplified as much as possible, minimizing the number of key strokes required. Dialogs contain all the information required by the teacher, containing the results of any automatic tests applied, access to the original source code, and (where appropriate) the metrics applied to the submissions.

These features allow the teacher to concentrate on the assessment without the necessity of significant time spent on peripheral administration.

Marks are assigned to a student's work during the marking and moderation processes. However, it is desirable for students to receive an explanatory result rather than just a number. Therefore, we allow markers to attach written feedback to each submission. After the marking and moderation procedures have been completed, the module manager may publish the results of the assignment. Students' results are dispatched to them through email, consisting of their final moderated mark and any comments that have been attached by the markers. Since there is a delay between the students submitting their work and the final marks being published, it is important to verify that each student receives appropriate feedback. By choosing to return marks and feedback using email, there is a high degree of confidence that each student will receive and read their feedback and it is for this reason that email, rather than a web solution, is adopted.

2.8 Platform Independence

The authors' department, in common with many Computer Science departments, has adopted widespread use of Linux and UNIX operating systems, whereas the University provides Windows-based solutions to students. It was, therefore, a necessity that BOSS be platform-independent. Java was chosen to form the basis of a complete rewrite of the system, not only because it would run on all major operating systems, but because its object-oriented paradigm, together with a wide selection of supported APIs, were seen to be supportive of rapid and accurate coding.

Two possible solutions to platform independence were considered. The first would involve Java clients and servers (so that BOSS would become an application which would run on student/staff computers) and the second a web-based solution accessible through web browsers. Since there are compelling arguments in favor of each solution, both have been implemented and are currently supported.

3. USER VIEW

Two strategic decisions were taken relating to the architecture of BOSS. First, there should be an overall model for the structure of an assignment, which we refer to as the *component model*, which is designed to support arbitrarily complex rubrics that might accompany an assessed piece of work. Second, the users of the system (students, module managers, administrators, markers, and moderators) should have clearly defined roles.

The software uses a client–server architecture with separate clients for students and for authorized staff (for security reasons). Each client is provided both as a secure web client and as a stand-alone application, so maximizing the flexibility of the system in terms of a user's working environment. There are, consequently, two distinct *views* of the software, according to whether the user is a student or a member of staff.

3.1 Component Model

Assessments take a wide variety of forms, including single tasks (such as essays) or potentially complex entities (such as examinations). It is not uncommon to encounter a rubric such as, *"Attempt question 1, question 2, any three questions from section B, and one question from section C."*

The data model used by BOSS, i.e., the *component model*, is intended to support arbitrarily complex assessment structures. The model is simple, straightforward to store in a relational database, and able to cope with any rubric.

A complex assignment (in terms of choices and paths through the tasks to be completed) may be desirable as a component of an adaptive and individualized learning and teaching strategy. Our purpose in introducing the component model is to free the teacher from restrictions on the structure of an assessment, allowing a complex assessment model to be deployed.

The *component model* is a description of the structure of an assessed piece of work. It is intended to cover all possible assignments given to students, including continuously assessed work, examinations, and tests. The component model is based on the following four fundamental notions.

- A *problem* is a single task (such as a multiple-choice question, or an essay) which is not divisible into subproblems and has a *maximum mark* as an attribute.
- A *multicomponent* is a triple (C, A_C, M_C), where C is a nonempty set $\{c_1, c_2, \ldots, c_n\}$ of components, A_C is an integer in the range $0, \ldots, |C|$, and M_C is an integer in the range $0, \ldots, 100$. A_C represents the number of components a student is expected to attempt. M_C is the maximum mark for the whole multicomponent. If $A_C = 0$, then a student is expected to attempt all subcomponents. The maximum marks for the subcomponents are used to determine the relative weightings of those components.
- A *component* is either a problem or a multicomponent.
- An *assessment* is a multicomponent.

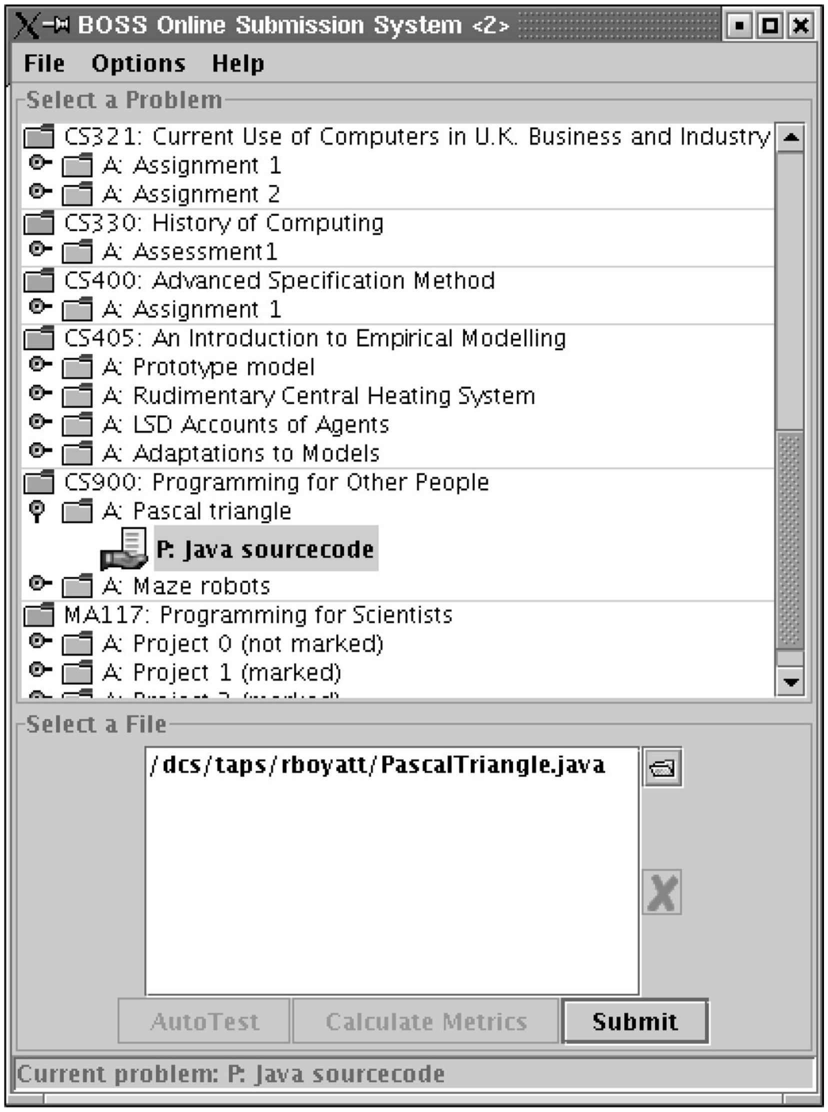
The component model is perhaps best demonstrated with an example. Suppose that an assessment has the rubric, *"Attempt question 1, question 2, any three questions from the five in section B, and one question from the three in section C. Question 1 is worth 20 marks, question 2 is worth 30 marks, section B is worth 30 marks, and section C is worth 20."* Using the obvious shorthand, the assessment decomposes as shown in the following tabulation:

Component	Type	Attributes
A	multicomponent	($\{Q_1,Q_2,S_B,$SC$\}$,0,100)
Q_1	problem	20
Q_2	problem	30
S_B	multicomponent	($\{B_1,B_2,B_3,B_4,B_5\}$,3,30)
S_C	multicomponent	($\{C_1,C_2,C_3\}$,1,20)
B_1	problem	10
B_2	problem	10
B_3	problem	10
B_4	problem	10
B_5	problem	10
C_1	problem	20
C_2	problem	20
C_3	problem	20

The intention is that, where the sum of the "maximum marks" of the components within a multicomponent is different to the maximum mark of the multicomponent itself, appropriate scaling will take place.

The model will also handle unusual cases. For example, if the rubric for an exam (with maximum mark 50) is *"Attempt any two out of the following three questions"*, and the three questions have been coded with different maximum marks (say 20, 30, and 40, respectively), then the components would be as follows:

Component	Type	Attributes
Exam	multicomponent	($\{Q_1,Q_2,Q_3\}$,2,50)
Q_1	problem	20
Q_2	problem	30
Q_3	problem	40



CS321: Current Use of Computers in U.K. Business and Industry
A: Assignment 1
A: Assignment 2
CS330: History of Computing
A: Assessment1
CS400: Advanced Specification Method
A: Assignment 1
CS405: An Introduction to Empirical Modelling
A: Prototype model
A: Rudimentary Central Heating System
A: LSD Accounts of Agents
A: Adaptations to Models
CS900: Programming for Other People
A: Pascal triangle
P: Java sourcecode
A: Maze robots
MA117: Programming for Scientists
A: Project 0 (not marked)
A: Project 1 (marked)

/dcs/taps/rboyatt/PascalTriangle.java

Current problem: P: Java sourcecode

Fig. 2. Student dialogue screenshot.

If a student obtains marks m_1, m_3 from problems Q_1 and Q_3, respectively, then their total mark is then calculated as: $([m_1 + m_2]/[20 + 40])$*50

3.2 Student View

The BOSS software permits students to perform two principle tasks:

(1) Students submit their (programming) assignments online (a typical dialog is illustrated in Figure 2;
(2) students are able to run automatic tests on their programs prior to submission (and afterward if they wish to resubmit within the prescribed deadline).

3.3 Staff View

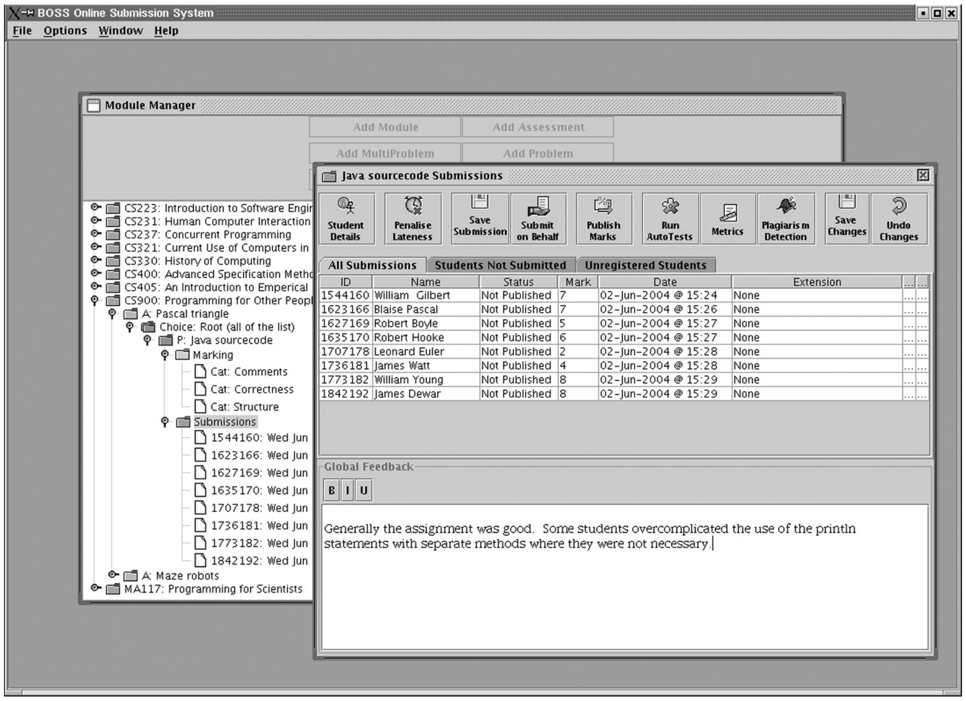
The BOSS software permits staff to perform five principle tasks.

1. Automatic tests can be run on the set of student submissions and as part of the marking process. These tests may be a superset of those that a student can run, or they may be separate. For example, students may be given a (small) set of automatic tests to run on their programs prior to submission, for the purposes of ensuring that they have met minimum requirements for the assignments. Further tests available to staff alone might then be used to assess how well each student has met the full set of requirements.
2. Plagiarism detection software assists in the identification of potential intra-corpal source-code plagiarism.
3. Submissions can be marked online by viewing the results of the automatic tests, running the submitted program, and viewing the submitted source code.
4. Staff authorized by the module organizer can moderate the marks given to students' work by other markers.
5. Feedback can be given on each submission and BOSS collates the feedback from the set of markers of a given submission and provides a mechanism for communicating this back to the student.

In order to deliver these tasks in a manner which ensures data privacy (staff can only perform appropriate tasks) and allows for multiple marking of an item of work to be performed "blind," there are four staff roles, as follows.

- *Administrator*. The administrator may create modules and allocate managers to individual modules. This role is not a "super user" one and the administrator's view of the data is strictly limited.
- *Manager*. Each module is allocated one (or more) managers, who can edit all properties of that module, including assignment details, marking criteria, and allocation of markers and moderators. An example of the manager's view of the system is illustrated in Figure 3.
- *Marker*. Each problem contained within an assignment is allocated one or more markers by the module manager. Each marker is allocated submissions to mark, and will mark online according to the marking criteria authored by the manager. An example of a marker's view of BOSS is illustrated in Figure 4. Weightings of individual marking categories, and the identity of the student, are hidden from the marker in order to ensure fairness and transparency of the marking process.

  The markers have the opportunity to write feedback on the work marked and it is expected that the manager will issue the markers with guidance as to what type of feedback is appropriate for that particular problem.
- *Moderator*. Once a problem has been marked by all markers allocated to it, a moderator is required to review the marks awarded and the feedback given. If multiple markers have been allocated to each student, the moderator's view will contain all the marks awarded, and a "suggested" moderated mark for each marking category, which the moderator is free to alter. The weightings



ID	Name	Status	Mark	Date	Extension	...	...
1544160	William Gilbert	Not Published	7	02-Jun-2004 @ 15:24	None	...	...
1623166	Blaise Pascal	Not Published	7	02-Jun-2004 @ 15:26	None	...	...
1627169	Robert Boyle	Not Published	5	02-Jun-2004 @ 15:27	None	...	...
1635170	Robert Hooke	Not Published	6	02-Jun-2004 @ 15:27	None	...	...
1707178	Leonard Euler	Not Published	2	02-Jun-2004 @ 15:28	None	...	...
1736181	James Watt	Not Published	4	02-Jun-2004 @ 15:28	None	...	...
1773182	William Young	Not Published	8	02-Jun-2004 @ 15:29	None	...	...
1842192	James Dewar	Not Published	8	02-Jun-2004 @ 15:29	None	...	...

Generally the assignment was good. Some students overcomplicated the use of the println statements with separate methods where they were not necessary.

Fig. 3. Manager dialogue screenshot.

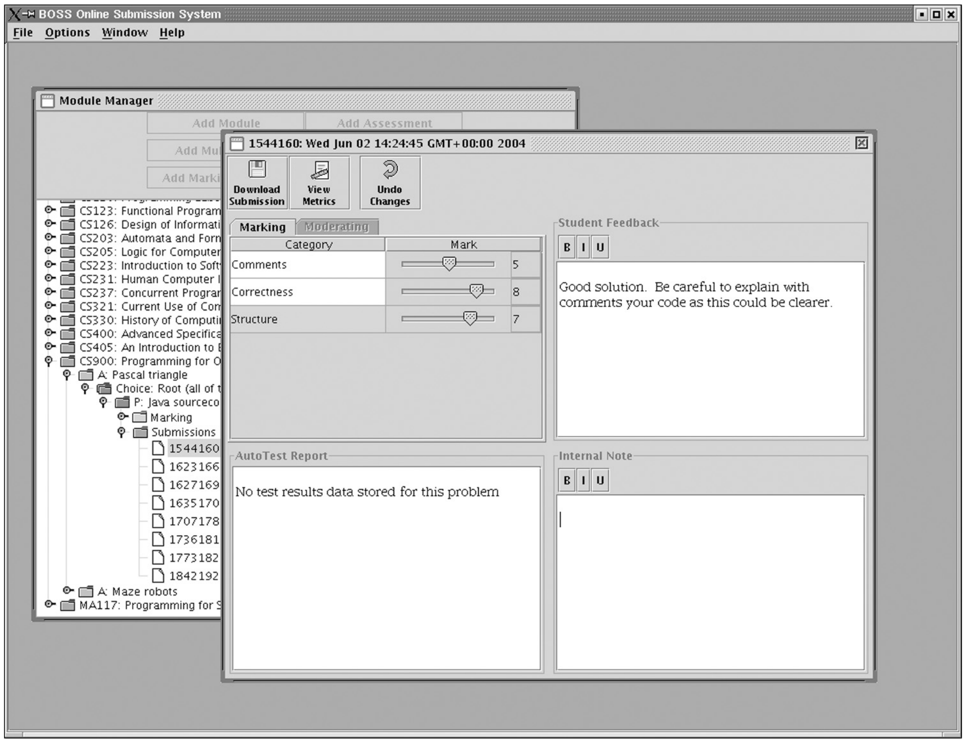
for the individual marking criteria are available to the moderator, but the student's identity is not. Only when a student's work has been moderated are the final results available to the manager.

The ideal model, if resources are available, is for each piece of work to be double marked, moderated by a third person, who may or may not be the module manager. However single marking is permitted by BOSS, in which case the role of moderator becomes one of checking the consistency and accuracy of the marker.

4. ARCHITECTURE

An overview of the system architecture can be seen in Figure 5, showing its primary components. There are four data repositories (represented by grey rounded boxes), which store marks, information about students and their submissions, results of automated tests, the results of plagiarism detection, and other necessary data.

Staff and student servers provide appropriate functions and data access to staff and students, respectively. Both the staff and student servers have a web-based interface and a standalone Java application interface. The web interfaces communicate with the other system components via a secure web-server using SSL. The staff interface also provides access to the plagiarism detection software (called Sherlock) which analyses the stored submissions and stores various metrics for assessment by teaching staff [Joy and Luck 1999; White and Joy 2005]. An "automatic test server" is responsible for performing tests



1544160: Wed Jun 02 14:24:45 GMT+00:00 2004

CS123: Functional Program
CS126: Design of Informati
CS203: Automata and Forn
CS205: Logic for Computer
CS223: Introduction to Soft
CS231: Human Computer I
CS237: Concurrent Prograr
CS321: Current Use of Con
CS330: History of Computi
CS400: Advanced Specifica
CS405: An Introduction to E
CS900: Programming for O
A: Pascal triangle
Choice: Root (all of t
P: Java sourceco
Marking
Submissions
1544160
1623166
1627169
1635170
1707178
1736181
1773182
1842192
A: Maze robots
MA117: Programming for S

Category	Mark
Comments	5
Correctness	8
Structure	7

Good solution. Be careful to explain with comments your code as this could be clearer.

No test results data stored for this problem

Fig. 4. Marker dialogue screenshot.

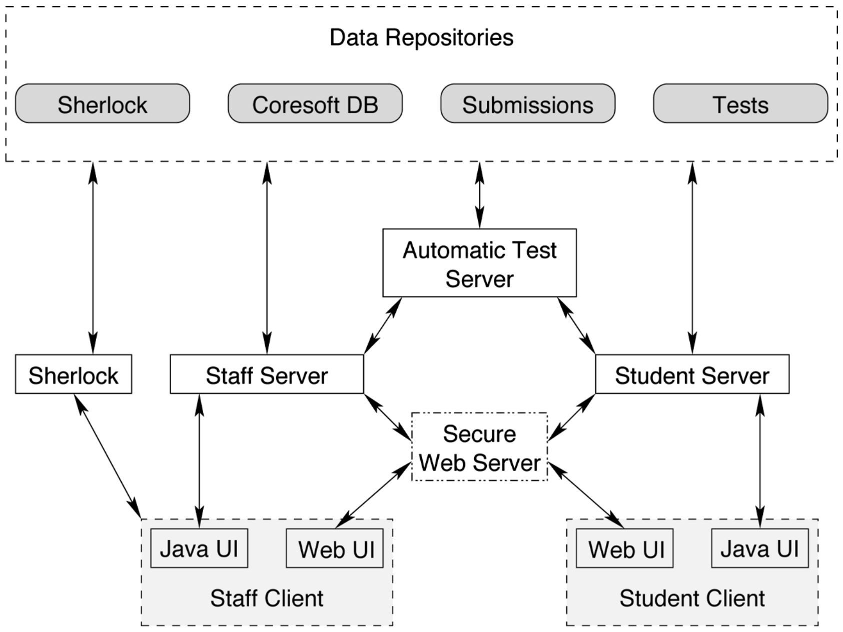


Fig. 5. The BOSS architecture.

on students' submissions and storing the results (or passing feedback to the student if the test is being run prior to submission).

A client–server architecture using RMI (Remote Method Invocation) for data transport forms the basis of the current system. At an early stage in the development of the Java code, we decided that any maintainable and robust solution required a modular approach. Both CORBA and RMI were considered, the latter chosen because of its Java base and consequent ease of coding. The use of Applets was ruled out, since correct functioning of Applet-based clients is dependent on the browser support for Java and the power of the client machine. Not only do some proprietary browsers not support Java fully (and this has been the subject of litigation both in the US and the EU), but at the time of the decision students' personal machines were unable to run complex Applets acceptably fast.

4.1 Client–Server

The BOSS system consists of three servers: student server, staff server, and the testing (or slave) server. These are actually three separate Java processes that are usually run on the same machine but can, if so desired, execute on three physically separate machines. The primary function of the student server is to authenticate students and receive their coursework submission for appropriate storage and logging. In addition, the student server is capable of communication with the testing server if the automatic code tests are available to the student before they make their final submission. The staff server, to which access is only permitted to fully authenticated members of staff, provides testing, marking and moderation functionality to the client software. The testing server is not directly accessible—the staff and student servers communicate with it to request the automatic testing of student submissions.

Each server executes as a process without administrative (superuser) privileges to prevent the compromise of the entire machine should one server be maliciously attacked and compromised, an event which has not yet happened in the lifetime of the project. Filesystem and database privileges are carefully allocated for each server.

The BOSS system currently offers two clients, the web-based and Java application clients, of which further details can found in Section 4.5. The development of two separate interfaces has been possible because of careful design of the client–server interface. Where possible, we have placed functionality in the servers to allow clients to be kept "lightweight," and also to prevent large amounts of sensitive student data from being transmitted data over the network unnecessarily.

4.2 Automatic Test Server

In addition to the student and staff servers, the third part of the BOSS system, namely the test server, is used to run submissions through a series of fully automatic tests. The testing system is functionally separate from the core BOSS system allowing some flexibility in the deployment of a testing system which may, depending on the scale of automatic testing required by the institution, involve separate computing hardware. Transfer of submissions from the student and staff servers to the testing server is encrypted to prevent malicious modification or theft of a submission.

BOSS offers automatic testing functionality in two modes: submission-based and batch-based. The first is available to students at submission time. The course manager can make available tests that give immediate feedback to students. These tests can be used as a simple check of the submission and can help prevent erroneous submissions. Furthermore, based on the feedback given to them, students can revise their submissions if they discover that they have not met the requirements (assuming that the final deadline has not passed). The majority of the automatic testing is performed in the second mode and can not be seen or executed by a student. These postsubmission tests are typically executed by the course manager as a batch job after the final submission deadline. We can construct more elaborate and thorough tests at this stage as there is no requirement for immediate feedback to a student. The results of these postsubmission tests can be linked to marking categories, which assess the correctness of a submission, freeing the marker to spend a greater amount of effort assessing the style of the submission.

Both of the testing paradigms introduced in Section 2 are available in both the submission-based and batch-based testing modes. The first paradigm defines input and output as data files and checks a student's submission against the expected output. Although this is a simple mechanism, it is also very powerful and allows a course manager to model the strict input and output requirements that are often present in real-world software engineering tasks. JUnit tests form the basis of the second BOSS testing paradigm (available only for Java submissions). This form of test consists of testing for the equality of two method calls given a specified input. More specifically, suppose that a Java programming assignment can be specified by: the input C_I, the output C_O, and a method `m` with signature C_O `m(`C_I`)`. Then a test can be defined by: an object O_I, an instance of class C_I, and an object O_O, an instance of class C_O, such that `m(`O_I`)` returns O_O. A student's implementation of method `m` can then be tested by checking `m(`O_I`).equals(`O_O`)`. Of course, we may need to override `equals()` to achieve the desired result. Unlike the first paradigm, this does not allow for the I/O presentation to be specified and testing may require a combination of both techniques.

In both modes, to prevent overloading the testing server close to a submission deadline, we limit the number of concurrent tests. Consequently some students at the submission stage may have to wait several moments for a test result. We see this as an acceptable compromise to prevent overstretching available testing resources. All tests are executed inside a sandbox able to limit the amount of available CPU time and memory and with strictly controlled set of permissions to prevent malicious submissions compromising the host system. There have been no known compromises of the testing system.

4.3 Databases

Central to a data-bound application, such as BOSS is the storage and management of the data. In addition to storage of submitted assignments, as archives on secure backed-up file systems, an SQL database is used for other data, such as times of submissions, basic student identity information, and marks awarded. The initial deployment of a proprietary database was found to be unsuccessful (because of the repeated requirement of systems staff to manage the database) and open-source databases such as MySQL, MSQL and PostgreSQL have since been used. Differences between the dialects of SQL used are a continual source of frustration, although the latest versions of MySQL and PostgreSQL allow interchangeability with minimal intervention, assisted by the use of JDBC to connect with the Java servers.

In order to facilitate the import of data from external sources (such as the University's Student Record System), an "institution-independent" database schema, called CoreSoft, was developed [Joy et al. 2002]. The aim of CoreSoft was to present the minimum data required for BOSS (and other related applications requiring similar data) in a format that would be compact, and use appropriately normalized tables with easy to remember names. The translation of data from external databases to the CoreSoft schema (and vice versa) is—at least in principle—a straightforward task.

This incorporation of Coresoft into a large software package such as BOSS demonstrates that it is an effective tool for providing an interface between the external academic data requirements of the package and the data supply capabilities of the host institution.

4.4 Plagiarism Detection

The department's plagiarism detection software, known as "Sherlock" [Joy and Luck 1999], has been developed in parallel with BOSS and, until 2002, was a separate tool. Sherlock reports on a collection of documents and reports instances of pairs (or larger clusters) of documents that contain similarities. Initially written for use with Pascal (and now Java) programs, Sherlock has been extended to operate on free-text data files. Both its source code and free-text facilities compare well, both in terms of accuracy and of ease of use, with other plagiarism detection tools such as CopyCatch [CFL Software Development 2004].

4.5 HCI

The development of both web-based and application clients is motivated by two main factors. First, students demand a simple to use product to submit their work, both from the campus and when working at home, suggesting a web client as being appropriate. Figure 6 shows screenshot of a dialog from the web-based client for students.

Second, staff who are marking assignments for large classes desire an interface, which is quick to use and minimizes the number of key strokes. This type of interface is simpler to code as an application and, when used on a machine directly connected to the campus network, avoids the delays inherent in the web-based solution. Screenshots of the application interface are presented in Figures 3 and 4 above.

Both interfaces have been coded to take account of appropriate "good practice" [Shneiderman 1998]. For example, the web interfaces are structured as

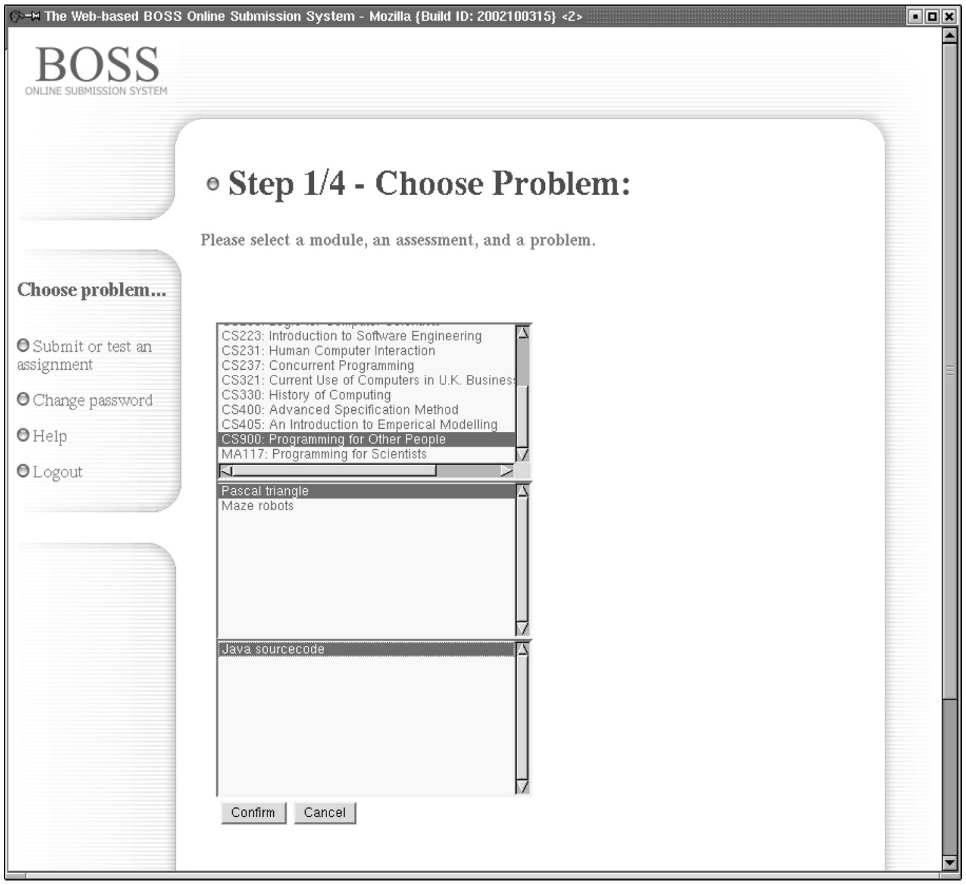

The Web-based BOSS Online Submission System - Mozilla {Build ID: 2002100315} <2>
BOSS
ONLINE SUBMISSION SYSTEM
Step 1/4 - Choose Problem:
Please select a module, an assessment, and a problem.
Choose problem...
Submit or test an assignment
Change password
Help
Logout
CS223: Introduction to Software Engineering
CS231: Human Computer Interaction
CS237: Concurrent Programming
CS321: Current Use of Computers in U.K. Busines
CS330: History of Computing
CS400: Advanced Specification Method
CS405: An Introduction to Emperical Modelling
CS900: Programming for Other People
MA117: Programming for Scientists
Pascal triangle
Maze robots
Java sourcecode
Confirm
Cancel

Fig. 6. Web client screenshot.

collections of relatively small individual pages with many navigation aids and shortcuts, and are appropriate for remote access to the server where the connection may be slow or unreliable. The application interfaces maximize the amount of relevant information available on each screen, to enable the user to navigate through the dialogs and complete their task, and is appropriate for the local high-speed connections normally available to staff. Both student and staff clients have been coded with both types of interface and evaluation of the usage patterns is ongoing.

5. EVALUATION AND DISCUSSION

The development of BOSS is ongoing, driven by the evolving demands of academics and of the technologies that can be applied. In the academic year 2004–05, an opportunity was taken to evaluate BOSS [Heng et al. 2005], and we present a "snapshot" of its current use, and discuss issues which will affect the directions in which the software will change.

Our evaluation is restricted to its use at the University of Warwick, since—BOSS being an open source product—we do not have accurate information as to which other institutions have deployed BOSS, and what local changes they have made to the software.

5.1 Methodology

We employed a combination of techniques in order to gain information to assist us. These included:

- an analysis of end of module questionnaires;
- interviews with staff who have been involved in the BOSS marking or management process;
- interviews with a representative group of students; and
- an analysis of the contents of the database used by BOSS.

We summarize our findings in three main parts. First, we present a technical evaluation of the software and provide evidence which indicates that the software is now stable, and that remaining issues relate principally to the lesser-used dialogs within the staff clients. Second, we discuss the usability of the software. Finally, we consider the pedagogy of BOSS and argue that its original objective of being a "pedagogically neutral" tool has been achieved.

5.2 Technical Evaluation

As evidence of the stable nature of BOSS, we describe how the major design decisions that have influenced its development have led to improved reliability and security. We describe how BOSS was used in the academic year 2004–05, outline the outstanding issues, and explain why they do not pose significant obstacles to general use.

Dividing functionality between separate servers has provided technical and organizational benefits. It has helped ensure that problems with the marking or testing systems do not directly affect the submission of coursework.The student server and client have also, restricted access and limited functionality to help reduce the risk of a security breach.

The large user base has allowed us to quickly identify and resolve any problems with the software. In 2004–05, the performance of BOSS can be summarized:

- over 5500 coursework submissions received electronically;
- no major outages, or downtimes impacting submission deadlines; and,
- no (known) security breaches.

BOSS allows students to resubmit their work multiple times. The administrative burden of doing this with paper submissions would be substantial, but with BOSS this is easy. For example, in one of the first year programming modules, 42% of students resubmitted their work, yet it required no further effort from the course manager. The number of resubmissions per student is presented in the graph in Figure 7.

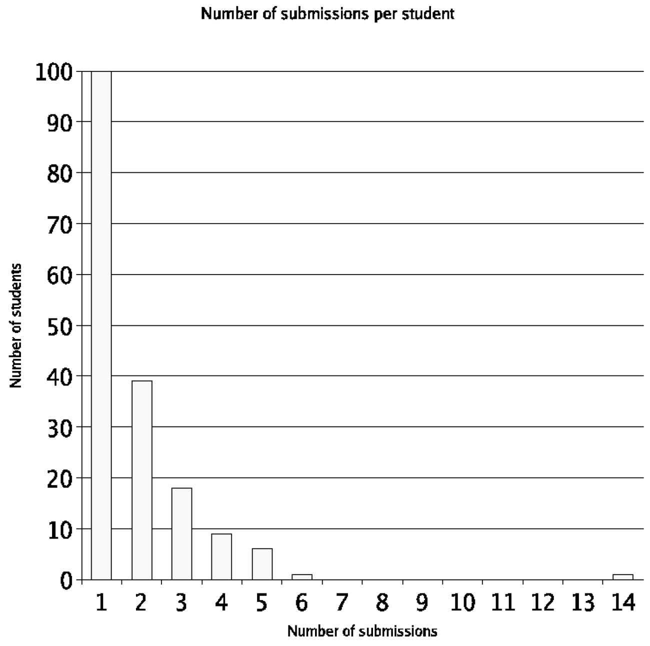


Fig. 7. Number of resubmissions for the “Programming Laboratory” module.

There are some outstanding issues with the BOSS system, specifically with the staff client, which relate to the functionality of the software—staff have differing opinions as to what extensions (if any) should be made to the client. Although these issues are important, they do not impinge upon upon the ability to accept student submissions and store them accurately and securely, nor do they affect the accuracy and stability of the test server.

5.3 Usability Evaluation

Ease of use is important for a tool such as BOSS, which receives heavy usage, and for which mistakes during the processing of data can have very serious consequences.

We report on our evaluation of the usability of the system, from the perspectives of its different users, as evidenced from a series of semistructured interviews, from which all the quotes in this section are taken. The interface is also heuristically evaluated against a list of guidelines and design principles [Nielsen 2005; MIT Usability Group 2005].

This work has been used to design a new web interface for BOSS, which addresses the issues the evaluation has highlighted [Heng et al. 2005]. This has been implemented for the student web client and will be implemented for the staff web client when resources permit.

5.3.1 *Student Client.* All interviewed student users indicated that they had no problems learning to use the student client. Some students explained that a reason for this is their prior experience using other web applications.

Students commented that "it looks nice," and, compared to the previous interface (2002), that the new interface "looks nicer and more friendly." One commented that "I prefer clean design, the less graphics the better," and noted that the web interface is "light-weight and quick."

Minor user interface issues were identified. For example, pages of the client are often long and one of the markers noted that "it looks OK on a large screen, not on a laptop." Users with a screen resolution lower than 1280×1024 need to scroll when using some of the dialogs, with the consequent increase in excise.

A specific issue raised was the lack of immediate feedback when students submit an assignment. Instead of generating a confirmation page *on the browser* indicating the receipt of a student's assignment immediately after a submission is made, an email is sent to the student's registered email address. The receipt of emails by the students is sometimes delayed and some students mentioned that they usually make two or more attempts to submit the same assignment "just in case" the BOSS server did not receive the first one.

5.3.2 *Staff Client.* The result of heuristic evaluation relates the information presentation of the staff client interface to the web site structure. The existing client uses a depth-emphasizing site structure rather than a balanced combination of depth and breadth-emphasizing. The advantage of using a depth-emphasizing site structure is that it expands the size of the site indefinitely by storing information in many levels of the site. Information is revealed gradually and other task buttons can be found in different levels when users click through the pages. However, since a depth-emphasizing site structure lacks linear navigation, many facilities are located in different levels of the site hierarchy, without immediate access for the users. Therefore, this approach may not be ideal when adopted by a software package with many features and functionalities targeted at different group of users.

The staff client received more focused feedback from the staff users. Almost all staff users demonstrated their need to look at the information presented on the interface before deciding on the sequence of actions that need to be performed by the system. They considered that the interface is more geared toward executing actions rather than informing them about the state of the modules and the submissions they are managing. Consequently, some of the staff users considered that BOSS is complicated to use.

Heuristic evaluation and interviews also suggested that although BOSS Online has achieved an overall consistency in look and feel by the use of stylesheets, some details have been neglected in the design. For example, the naming and the position of the buttons, which have the same meaning in the interface, are not always consistent.

Information gathered from the developers and the system designers show that both the student and the staff interfaces have been implemented pragmatically. BOSS was designed-based in order to satisfy the need of the Department of Computer Science to facilitate the submission and the marking process of student assignments. Development over the years has been incremental and new features have been added to BOSS based on users' requirements while retaining the existing functionalities. The views of these two classes of user support a claim that the software effectively fulfills its purpose.

5.4 Pedagogic Evaluation

BOSS is intended to be “pedagogically neutral.” We provide an environment in which a secure online submission platform can be used with existing tools, or alternatively the extra assessment utilities in BOSS can be used in conjunction with them to offer increased functionality [Luck and Joy 1999].

In order to evaluate BOSS against this objective, we identify two sources of evidence. First of all, we examine its patterns of use to test the hypothesis that the tool is indeed used with a *variety* of existing tools. Second, we consider the comments made by interviewed staff, who have used the system which relate to its educational value.

5.4.1 *Patterns of Use.* The Department of Computer Science does not prescribe how an individual academic should manage his/her own modules and each academic is free to use BOSS as much or as little as they desire. An analysis of the patterns of use over the fifteen modules, which used BOSS, highlights the different individual approaches taken to the adoption of the software.

Seven of the modules used the system as a collection mechanism only, allowing the assessment process to be supported by other tools, such as spreadsheets. The reasons for not using the assessment features include:

- assignments are essay-based (two modules);
- assignments relate to formal methods and do not require coding (one module);
- assignments take the form of group projects, which contain added administrative complexity not appropriate for BOSS (one module);
- assignments require students to code, but the nature of the programming paradigm (for example, declarative) requires tests to be performed on students’ code, which is not easily modeled as “expected output for given input” (three modules).

One further (essay-based) module used BOSS for collection and the plagiarism detection software only.

Of the remaining seven modules, which did use the majority of the software features, all involved students programming using procedural or object-oriented languages, as illustrated in Table I, which identifies

- whether the automatic tests were used,
- whether the plagiarism detection software was used,
- whether the marking process was conducted *online* within BOSS,
- the type of code which the module was delivering.

5.4.2 *Plagiarism Detection.* The assessment process involves ensuring that students are marked on their own work and prevention and detection of plagiarism is, therefore, an important part of the process. All staff who used the plagiarism detection software commented that it was effective and, in each

Table I. Programming Modules Using BOSS

Year of study	Automatic tests?	Plagiarism detection?	Online marking?	Code type
1	no	yes	no	simple C++
1	no	yes	no	simple Java
1	yes	no	yes	simple Java
1	yes	yes	yes	simple UNIX Shell and Perl
1	yes	yes	no	intermediate Java
2	no	yes	no	Prolog
2	yes	yes	yes	advanced Java

module, disciplinary action was taken on a number of students as a result. Use of the plagiarism detection software as a regular part of the assessment process, and its associated visibility to students, has resulted in effective deterrence, and although plagiarism has not been eliminated, instances of it have been reduced on large programming modules to typically less than 5% [Joy and Luck 1999; White and Joy 2005].

The one module, which did not use the plagiarism detection component, offered introductory Java programming for science students (principally mathematicians and physicists). The teaching style adopted involved the students being presented with "templates," which shielded them from the complexities of the object-oriented paradigm and enabled them to concentrate on writing and editing relatively small amounts of procedural code appropriate to their disciplines. The use of automatic plagiarism detection software is not effective on data where the amount of individually contributed code is small.

5.4.3 *Automatic Testing.* For four of the seven module assessments, it was appropriate to use the automatic test harness together with the on-line marking dialog (these were first- and second-year modules using Java or UNIX Shell code). The final three modules were supported by alternative assessment and management regimes. The staff involved in each of these modules were skilled in alternative software products and wished to use facilities which would be inappropriate to include in BOSS. For example, one academic remarked:

> A spreadsheet is very flexible, you can sort it in many ways, do lots of things on it, colour it, and so on. You can highlight it, give private comments, comments for yourself, . . . do layout, create graphs . . . .

The pedagogic effectiveness of BOSS is that of the educational paradigm it is used to support. The simplest nontrivial use of BOSS—and one of the original motivations for its creation—is the incorporation of black-box automatic tests into the assessment process for programming modules. Conversations with module leaders suggest that the time necessary to devise and deploy a set of automatic tests is typically 1 or 2 hours and that the time taken to mark a single student's submission may be as low as a couple of minutes; this is strong evidence that the approach is administratively effective. All the staff who used the software for its automatic tests and the online marking agreed that the black-box paradigm worked successfully, although setting up those tests was regarded as complex. This is, in part, a usability issue, but is also a comment on the inherent difficulty of writing a correct test harness together with a clear and comprehensible specification that will enable students to understand what they are required to code.

Incorporation of program metrics into the software is recent and has not yet been fully incorporated into any academic's marking scheme. Similarly, although JUnit unit tests have been included in the system functionality, the technology has not yet been taken up in any module at Warwick.

Automatic testing paradigms are not magic bullets. JUnit tests require technical skill to code and are language-specific. Input versus expected output tests, when interpreted as text files, invites difficulties due to misspellings, control characters and whitespace, which can confuse superficially simple comparisons.

The student view has been generally very positive and comments at the end of module questionnaires indicate that it is regarded as an efficient and convenient system to use. A specific comment often made by students is that the automatic testing is "too fussy." In an environment where we encourage students to take a formal engineering-based approach to software development, it seems appropriate that a tool such as BOSS is precise and this student view might be interpreted positively. However, there are drawbacks to such an engineering approach and the following students' views about the use of automatic tests are not unusual:

> Automatic tests are unfair against those who have tried and only just failed to reach the required outcome.

> Didn't like the auto tests—too picky with spaces, etc.

The number of tests, and their place in the marking scheme, is another focus of discussion. It is neither feasible nor desirable to provide a complete testing suite for use by students while they are learning the basics of programming, since the number of tests would be prohibitive, and communicating the results to students correspondingly problematic. However, it may be desirable to allow students access to some automatic tests to assist them in their program development. It may also be desirable to reserve some tests for the marking process in order to encourage students to think through the specification thoroughly. This is a strategy which is difficult to justify to students:

> I don't understand why there are more tests on the assignment. Can you give us all tests when you give us assignments. Then we can know what you want us to do.

More detailed staff feedback (via interviews) suggests that the functionality of such a system supports colleagues' requirements, but identify the client dialogs as being over complex.

Our claim of "pedagogic neutrality" is further supported by the use of BOSS as a vehicle for innovative pedagogic approaches. For example, the software has successfully been extended to provide a package to support peer assessment [Sitthiworachart and Joy 2004].

5.5 Future Directions

Any initiative that is dependent on technology is also at risk from changes in technology and it would be unwise to speculate what those changes will be. However, the paradigm used by BOSS appears to support a variety of modules successfully, and significant changes are not currently envisaged. The underlying technology will be upgraded as and when suitable new technologies and standards are in place.

The software has been made available as open source under the GNU General Public License [Free Software Foundation 2004]. There are three primary reasons for taking the open-source route.

- The development of BOSS is not commercially viable, given the level of commitment that would be needed to support, maintain, and continue development of the software.
- Making the software open source encourages take-up by other institutions and the subsequent community support and development that naturally follow.
- Placing the source code of the system in the public domain enables other institutions not only to use the system, but to customize and extend it without any license restrictions (and hopefully feed back their extensions to the user community).

5.6 Other Issues

In order for a system, such as BOSS, to be used effectively, it must interact with institution processes efficiently and accurately. Several issues have arisen during the deployment of BOSS that may well apply to many other institutions.

The quality of data received from the SRS (Student Record System) is sometimes poor. For example, delays in updating student data centrally often preclude the automatic generation of accurate lists of students registered for a given module. The schema used by the SRS is required for generation of accurate statistical data for government agencies, in addition to the more general central management functions. The type of statistics required affect the table structure of the database (for example, separate module codes are used for students repeating a module, which is often required if they do not pass the module at first attempt) and cause the import of data into BOSS, through the generic CoreSoft database schema, to be more difficult than expected.

The separation of the BOSS marking process into the three phases of mark, moderate, and finalize is not the paradigm of choice for all academics, some of whom interleave the processes. Flexibility in the mode of use is an adjustment to the functionality of BOSS that is currently being incorporated.

BOSS has been conceived as a tool targeted at a single task, namely, the management of online programming assignments. It is not intended to provide a suite of learning material, and contains no functionality to support students' learning other than that which directly arises from the activity of assessment. The support for learning provided by BOSS is encapsulated by the process of a student getting feedback from automatic tests prior to submission, followed by feedback from markers after submission. Thus, the learning benefits to students of using BOSS are similar to other assessment methods and are primarily dependent on the academic design of the assessment (or preferably a sequence of both formative and summative assessments) and the quality of feedback given by markers.

It is interesting to compare the BOSS approach with that of CourseMarker [Higgins et al. 2003], formerly named Ceilidh, a tool developed at the University of Nottingham, which allows both formative and summative assessment. The formative approach, which can be taken in CourseMarker, is to allow students to present solutions to programming problems multiple times. Each solution is then marked against a "template" and against a variety of metrics, allowing the student to improve their solution prior to final submission by assimilating the frequent feedback presented by CourseMarker. This was an approach that we chose not to follow, since we wished BOSS to focus on the process of online submission and measuring the correctness of students' code, rather than become a tool with a broader support for formative assessment. The CourseMarker approach prescribes a style of programming that, it might be argued, is not always appropriate, and we decided that the formative functionality would be inappropriate for BOSS. Our emphasis is on providing a tool to assist teaching staff and encourage best practice in teaching programming rather than to provide an online learning environment.

It should be noted that, although our primary aim is to support the teaching of programming, BOSS is also useful as a submission and marking tool for other types of assessment, such as essays. For example, an automatic "essay style checker" could be seamlessly plugged into BOSS to support the assessment of an essay-based module. BOSS provides an effective means for the collection of submissions, since students can submit using computers across the campus or from home via the web interface. The Sherlock plagiarism tool allows teaching staff to detect intracorpal plagiarism in the essays submitted by students [White and Joy 2005]. BOSS can also be used as a repository for a marker (or group of markers) to store feedback on each submission. At the end of the marking process, this feedback can be collated and moderated for each submission and then returned to the student.

5.7 Historical Notes

The initial software package was developed in the mid-1990s, when many terminals were still text-only, students would normally interact with the University computer systems from on-campus, and remote communication with central UNIX servers was necessarily text-based. The technology initially deployed was an application with a text interface, which ran on a central UNIX server. Coding was in ANSI C, and designed in as reusable and modular a fashion as the language would easily allow. Security was achieved by means of standard UNIX file permissions and judicious use of the "setUID" mechanism. The Snefru [Merkle 1990] hash algorithm was used to sign each submission and ensure the integrity of submitted data.

This solution was successful, but the rapid advent of higher-quality terminals with graphic capability suggested that an improved user interface was desirable. Not only would staff productivity increase with a "point and click" interface, but student perception of the software would improve, since it would appear more "up to date." The immediate solution was to add a front-end coded in Tcl/Tk, which was relatively easy to implement because of the modular structure of the underlying code [Luck and Joy 1999]. While this solution was effective, it exposed a fundamental weakness in the choice of technology, namely, that the scalability was poor. For example, the modular constructs of Tcl/Tk are few and primitive, and the Tcl/Tk scripting language is weakly typed. It was felt that the software was not amenable to significant development in its current state and a permanent solution was sought. A detailed description of the system at that time has been reported elsewhere [Luck and Joy 1999].

The use of simple UNIX utilities, such as `diff`, have been used to perform the comparison between actual and expected program output since the first version of the software. However, despite the apparent ease of writing tests, care must be taken to ensure that the tests and the specification (as presented to the students) are consistent. The incorporation of JUnit tests is a recent addition to the software and relies on the use of Java as the language being taught. Furthermore, the approach taken must be "object-oriented" and it should be noted that many teachers choose to adopt a procedural approach to the introductory teaching of Java [Yau and Joy 2004; Burton and Bruhn 2003].

6. RELATED WORK

A variety of other tools have been created, which address some of the issues motivating BOSS. We briefly review these in this section.

Other methods used to collect students' work online include email [Dawson-Howe 1995] and requiring students to store their work in a specified folder [Isaacson and Scott 1989; Ghosh et al. 2002].

Tools and techniques to assist in the automatic testing of programs include the use of shell scripts [Isaacson and Scott 1989], copying submitted files to the teacher's filespace [Reek 1989], and transferring ownership of submitted files to the teacher [MacPherson 1997]. Since students' programs may exploit loopholes in the system (either accidentally or deliberately), automatic testing is often performed in a restricted environment designed to minimize the possibility of damage to the system [Cheang et al. 2003; Hurst 1996].

The Online Assessment System (OAS) [Bancroft et al. 2003] supports online assignment submission together with a web-based interface for online marking. OAS interfaces with an institution student database (FITSIS), and uses a custom application F^2M^2 to facilitate the online marking process.

The Online Judge [Cheang et al. 2003] provides a straightforward testing harness where tests are defined in terms of whether (string) output from students' programs matches expected output. Plagiarism detection software is included, but there is no additional functionality.

CourseMarker [Higgins et al. 2003] provides a web-based client–server architecture which supports online submission, automated assessment, and a rich marking interface. A fundamental difference between CourseMarker and BOSS is the paradigm for interacting with a student. BOSS is conceived as a *summative* assessment tool and, although it supports feedback for students, its primary function is to assist in the process of accurate assessment. CourseMarker also supports the *formative* aspects of assessment, allowing students to have their program graded at frequent intervals *prior to submission*. In order for this to be feasible, the profile of the program is constrained by measuring its attributes and its functionality in order to arrive at a grading. It can be argued that this can constrain the student by penalizing an unusual (but correct) solution.

Automatic assessment tools for specific languages have been developed, such as Scheme [Saikkonen et al. 2003] and the use of Knuth's Literate Programming paradigm [Knuth 1984] has also been used to allow automatic annotation of students' programs [Hurst 1996].

Canup and Shackleford [Canup and Shackleford 1998] have developed software for automatic submission, but which does not support automatic testing of programs. Blumenstein et al.[2004] have been developing a generic assessment and marking engine, which can be configured for multiple languages.

7. CONCLUSION

The use of BOSS over a period of years has demonstrated the effectiveness of a focused tool, which addresses the requirements of assessing students' programming skills. The inclusion of a generic database schema and plagiarism detection software, together with a platform-independent client-server architecture, provide a foundation adaptable to changes both in technologies and in pedagogic requirements.

The BOSS system is a modular and extensible tool that has significantly aided student assessment in the authors' Computer Science department. Academics have the flexibility to use BOSS simply as a collection mechanism or as a complete automated assessment suite. BOSS is focused on supporting the *process* of assessment; it contains no functionality to support student learning other than that which directly arises from the activity of assessment. This has been demonstrated, in practice, by its successful use in a range of modules within the authors' department. Academics are free to use their own judgment about how the system can be best used to support students' learning in the context of a particular module. BOSS has proved itself to be successful tool, that supports the assessment process, but does not artificially constrain it.

REFERENCES

BANCROFT, P., HYND, J., SANTO, F. D., AND REYE, J. 2003. Web-based assignment submission and electronic marking. In *HERDSA 2003*. IEEE. Available: http://surveys.canterbury.ac.nz/herdsa03/pdfsref/Y1007.pdf (accessed: 30 May, 2004).

BLOOM, B. S. AND KRATHWOHL, D. R. 1956. *Taxonomy of Educational Objectives: The Classification of Educational Goals. Handbook I: Cognitive Domain*. Longman, London.

BLUMENSTEIN, M., GREEN, S., NGUYEN, A., AND MUTHUKKUMARASAMY, V. 2004. Game: A generic automated marking environment for programming assessment. In *ITTC 2004*. IEEE, 212–216.

BOSS. 2004. The BOSS online submission system. online. Available: http://boss.org.uk/ (accessed 19 December, 2004).

BULL, J. AND MCKENNA, C. 2001. *Blueprint for Computer-Assisted Assessment*. CAA Centre, University of Loughborough.

BURTON, P. AND BRUHN, R. 2003. Teaching programming in the OOP era. *ACM SIGCSE Bulletin 35*, 111–115.

CANUP, M. AND SHACKLEFORD, R. 1998. Using software to solve problems in large computing courses. *ACM SIGCSE Bulletin 30*, 1, 135–139.

CFL SOFTWARE DEVELOPMENT. 2004. Copycatch gold. online. Available: http://www.copycatchgold.com/ (accessed: 30 March, 2004).

CHEANG, B., KURNIA, A., LIM, A., AND OON, W.-C. 2003. On automated grading of programming assignments in an academic institution. *Computers and Education 41*, 121–131.

CIAD. 2004. TRIADS. online. Available: http://www.derby.ac.uk/assess/ (accessed: 25 April, 2004)

DAWSON-HOWE, K. 1995. Automatic submission and administration of programming assignments. *ACM SIGCSE Bulletin 27*, 4, 51–53.

ENTWISTLE, N. 2001. *Promoting Deep Learning through Assessment and Teaching*. AAHE, Washington, DC.

FREE SOFTWARE FOUNDATION. 2004. GNU general public license. online. Available: http://www.gnu.org/copyleft/gpl.html/ (accessed: 25 April, 2004).

GHOSH, M., VERMA, B., AND NGUYEN, A. 2002. An automatic assessment marking and plagiarism detection. In *ICITA 2002*. IEEE.

HENG, P., JOY, M., BOYATT, R., AND GRIFFITHS, N. 2005. Evaluation of the BOSS online submission and assessment system. Tech. Rep. CS-RR-415, Department of Computer Science, University of Warwick Coventry, UK.

HIGGINS, C., HEGAZY, T., SYMEONIDIS, P., AND TSINTSIFAS, A. 2003. The CourseMarker CBA system: Improvements over Ceilidh. *Education and Information Technologies 8*, 3, 287–304. Available: http://www.cs.nott.ac.uk/CourseMarker/ (accessed: 30 March, 2004).

HURST, A. 1996. Literate programming as an aid to marking student assignments. In *Proceedings of the First Australasian Conference on Computer Science Education*. ACM, New York. 280–286. Available: http://www.literateprogramming.com/lpin-assess.pdf (accessed: 30 March, 2004).

ISAACSON, P. AND SCOTT, T. 1989. Automating the execution of student programs. *ACM SIGCSE Bulletin 21*, 2, 15–22.

JOY, M. AND LUCK, M. 1999. Plagiarism in programming assignments. *IEEE Transactions on Education 42*, 2, 129–133.

JOY, M., GRIFFITHS, N., STOTT, M., HARLEY, J., WATTEBOT, C., AND HOLT, D. 2002. Coresoft: a framework for student data. In *Proceedings of the 3rd Annual Conference of the LTSN Centre for Information and Computer Sciences*. LTSN Centre for Information and Computer Sciences. 31–36.

KNUTH, D. 1984. Literate programming. *The Computer Journal 27*, 2, 97–111.

LANE, D. 2004. *JUnit: The Definitive Guide*. O'Reilly, Sebastopol, CA.

LEICESTER UNIVERSITY. 2004. The CASTLE toolkit. online. Available: http://www.le.ac.uk/castle/ (accessed: 25 April, 2004).

LUCK, M. AND JOY, M. 1999. A secure on-line submission system. *Software—Practice and Experience 29*, 8, 721–740.

MACPHERSON, P. 1997. A technique for student program submission in UNIX systems. *ACM SIGCSE Bulletin 29*, 4, 54–56.

MERKLE, R. 1990. A fast software one way hash function. *Journal of Cryptology 3*, 1, 43–58.

MIT USABILITY GROUP. 2005. Usability guidelines. Online. Available: http://www.mit.edu/ist/usability/usability-guidelines.html (accessed: 30 September, 2004).

NIELSEN, J. 2005. useit.com. Online. Available: http://www.useit.com/ (accessed: 30 September, 2004).

QUESTIONMARK. 2004. Questionmark Perception. online. Available: http://perception. questionmark.com/ (accessed: 30 March, 2004).

REEK, K. 1989. The TRY system - or - how to avoid testing student programs. *ACM SIGCSE Bulletin 21*, 1, 112–116.

SAIKKONEN, R., MALMI, L., AND KORHONEN, A. 2003. Fully automatic assessment of programming exercises. In *ITiCSE 2001*. ACM, 133–136.

Shneiderman, B. 1998. *Designing the User Interface, (3rd ed.)*. Addison-Wesley, Reading, MA.

Sitthiworachart, J. and Joy, M. 2004. Effective peer assessment for learning computer programming. In *Proceedings of the 9th Annual Conference on the Innovation and Technology in Computer Science Education (ITiCSE 2004)*. 122–126.

WebCT. 2004. WebCT. online. Available: http://www.webct.com/ (accessed: 30 March, 2004).

White, D. and Joy, M. 2005. Sentence-based natural language plagiarism detection. *ACM Journal of Educational Resources in Computing 4*, 4, 1–20.

Yau, J. and Joy, M. 2004. Introducing Java: A case for fundamentals-first. In *EISTA 2004*. 1861–1865.

Received June 2004; revised December 2004 and April 2005; accepted April 2005