# Automatic Test-Based Assessment of Programming: A Review

CHRISTOPHER DOUCE
Feedback Instruments, Crowborough, UK
DAVID LIVINGSTONE AND JAMES ORWELL
Kingston University, Kingston-upon-Thames, UK

Systems that automatically assess student programming assignments have been designed and used for over forty years. Systems that objectively test and mark student programming work were developed simultaneously with programming assessment in the computer science curriculum. This article reviews a number of influential automatic assessment systems, including descriptions of the earliest systems, and presents some of the most recent developments. The final sections explore a number of directions automated assessment systems may take, presenting current developments alongside a number of important emerging e-learning specifications.

Categories and Subject Descriptors: K.3 [**Computing Milieux**]: Computers and Education
General Terms: Design, Documentation, Measurement, Performance
Additional Key Words and Phrases: Education, learning, computer-based training, programming assessment

## 1. INTRODUCTION

Programming problems and assignments are considered essential elements of software engineering and computer science education. Programming assignments can help students become familiar with the attributes of modern programming languages, become acquainted with essential tools, and to understand how the principles of software development and design can be applied.

The *assessment* of these assignments places significant demands on the instructor's time and other resources. The instructor must determine the extent to which the submission has satisfied the requirements of the rubric, and possibly also verify the originality of the submission. This article details a number of attempts to automate the test-based assessment of student programming assignments for both summative and formative purposes. While the authors acknowledge that other automated approaches that may be relevant have been developed, for example peer review [Gehringer 2001] and intelligent training or tutoring systems [Sykes and Franek 2004], they are deemed outside the scope of this article, which is concerned specifically with approaches that automatically assess the success of student attempts to solve programming problems.

The ASAP project was funded by the e-learning tools strand of the UK government's Joint Information Systems Committee.

A literature review serves several purposes. By examining the projects that have been undertaken, it is possible to gain an appreciation of the approaches adopted by others in the past. This is useful from both technical and didactic perspectives. Previous

Authors' addresses: C. Douce, Feedback Instruments, Crowborough, UK; D. Livingstone and J. Orwell, Kingston University, Kingston-upon-Thames, U.K.

projects may be able to inform current development by illuminating the kinds of problems that were encountered and how they were overcome, whether a particular application was successful and whether other system developers had any insights into how contemporary systems should be constructed.

The following sections are broadly categorized according to age. The first section describes some of the first documented attempts at developing automated marking systems. Of course, it is possible that earlier mechanically-oriented testing and evaluation schemes may have been devised. This article only includes systems that use “programming languages,” that is, a form of written symbolic instruction that is executed on a digital system.

The next section presents the second generation of assessment systems. Here we see the emergence of a number of different approaches, primarily through the adoption of automated tools and utilities to simplify the tasks that both instructor and student had to carry out so that the assignment could be assessed automatically. This is an improvement on the earlier generation, where engineers were required to modify their low-level tools like operating systems and programming languages. The need for such drastic measures was removed due to the increasing sophistication of the tool sets and operating systems.

The current (third) generation of assessment systems, adopt web-based technologies to deliver testing tools to both students and tutors, and invariably include assessment maintenance and management utilities. These utilities allow the educator to obtain an overview of how groups of students are progressing, potentially allowing educators to take remedial action or to provide additional support to students who may be encountering difficulties.

In Section 3, there is a description of the pedagogic issues that any designer or user of such a system will encounter. As programming languages and systems change and evolve, assessment demands made on students will change accordingly. This will naturally affect any automated testing framework that may be adopted. The penultimate section explores a number of potential directions that automated programming assessment systems may take, making reference to both existing and emerging assessment initiatives.

Before continuing, a note on a methodology is necessary. The papers described in the following sections were found by a text search of ACM journals and conferences. The ACM Special Interest Group on Computer Science Education contains a particularly large number of relevant papers. For each paper found, references were followed, and where appropriate, authors and project workers were contacted directly to obtain further insight into the design and the operation of their assessment systems.

## 2. GENERATIONS OF ASSESSMENT SYSTEMS

### 2.1 First Generation – Early Assessment Systems

Automated assessment systems seem to have existed for as long as educators have asked students to build their own software. The earliest example of automated testing of programming assignments may be found in Hollingsworth [1960]. Rather than using compilers and text editors, students submitted programs written in assembly language on punched cards. A grader program was run against a student program and two different results were returned, either “wrong answer” or “program complete”. The advantages of this system did not only consist in a good use of tutor resources -- a key advantages was also the efficient use of computing resources, which allowed a greater number of students to learn programming.

As programming systems evolved, so did assessment systems. Forsythe and Wirth, along with Naur, present a grader system that examines programs written in Algol [Naur 1964; Forsythe and Wirth 1965]. The system, for an introductory programming and numerical analysis course, was used intermittently (since 1961) at Stanford University. The system operates by using a "grader" program to test submitted programs. The three functions of the grading programs are to supply test data, keep track of running time, and maintain a "grade book".

Several new ideas were introduced in Hext and Winnings [1969]. To implement their automatic assessment system, modifications to both compilers and operating systems were necessary. Program testing was done by comparing stored test data to data obtained by executing the students' assignments. Following execution of program testing, a report was generated, including detailed test results. Interestingly, the authors say that it may be possible to use the results to check for cheating (based on the size of the program and time spent on execution); later system designers have often addressed the issue of assignment plagiarism.

Hollingsworth made an interesting observation that is just as relevant today as it was when his paper was written: he believed it was possible for a student to submit a program that would caused deliberate damage to the grader software. Security is an issue that must be considered continually.

## 2.2 Second Generation – Tool-Oriented Systems

To develop and use first-generation testing mechanisms successfully demanded a great deal of expertise. A second-generation system, however, can loosely be labeled as "tool-based". First, assessment systems are developed using pre-existing tool sets and utilities supplied with the operating system or programming environment. Second, testing engines and systems are often used and activated in the form of command-line or GUI programming tools.

An example of a second-generation assessment tool can be seen in the work of Isaacson and Scott [1989]. The authors state that assessing programming assignments involves two activities: checking the program to see that it operates correctly and checking the program to see that the programming style has been applied sensibly. Like the earlier assessment systems (and all systems since), the focus of the methodology is the correct functioning of the program submitted.

Isaacson and Scott describe a "script-based" approach to testing. The script takes a set of files in a collection of directories and attempts to compile each file in turn and test each program using a set of predefined test data. The results and compilation attempts are written to a file which a tutor can review. Of course, some students regard the load-testing of the assessment system as a legitimate activity. As a consequence, a number of measures were introduced to ensure that a script runs successfully in the face of deliberate or misunderstood programming.

In the same year a different system was introduced by Reek [1989]. Rather than behaving as a tutor-oriented tool, the TRY system introduced automated testing to the student. TRY allows students to test their programs using a tester program. When the tester program is executed, the student is presented with a set of results and the test attempt is recorded. Like other systems of that period, testing is performed by a simple character-by-character comparison of results generated against expected ones.

Reek makes two particularly interesting points. First, the execution of alien code in a live environment may result in undesirable outcomes, for example, damage to or disclosure of the system or the data held therein. Reek also holds a different view regarding the didactic implications of using an automated testing system. Instead of allowing the student an unlimited number of tries, his system limits students to a set number of submissions per assignment. Thus students are forced to think about the operations of their programs more thoroughly before submitting them for evaluation, rather than relying on the computer to act as a formal functionality testing mechanism.

The Kassandra project [von Matt 1994] represents an interesting development, since it facilitates testing programs written in Matlab and Maple (mathematical languages), as well as in the more traditional Oberon, a successor to Modula-2. The correctness of the submissions is again determined by comparing output data to stored test data, which is created by the tutor. An innovation is introduced by letting assignment submissions to be made to a different process via internet socket technology, allowing a degree of isolation between the operation of the assignment code and the test management system.

The ASSYST system developed by Jackson and Usher [1997] introduces a scheme that analyzes submissions across a number of criteria. ASSYST analyzes whether submissions are correct (again, comparing the operation of a program to a set of pre-defined test data), whether submissions are efficient in their use of CPU time, and whether they have sensible metric scores that correspond to complexity and style. One of this project's greatest contributions is the understanding that an automated assessment system can also become a grading-support system. This system provides mechanisms to handle submissions, create and generate reports, and allow weightings to be assigned to particular aspects of tests.

The BOSS system originated at the University of Warwick in the UK. Its initial specification was similar to that of ASSYST [Joy and Luck 1998; Luck and Joy 1999]: they both ran on the Unix operating system and assessed programs written in C. The first version of BOSS was comprised of a suite of "easy-to-use" command line programs. A student could use one program to perform a test to determine whether his or her assignment was correct and another to submit the program to a secure location which could then be reviewed by a tutor.

Like many computer science departments at around that time, the University of Warwick began to use Java as a teaching vehicle. This necessitated a redesign -- the resulting system is made up of two main elements – an assignment submission and testing program that is a Java GUI application, and a tutor grading and assignment management application, which has since been extended. Like ASSYST, BOSS has become an assessment management system.

One of the most notable developments of the mid-eighties was the Ceilidh system, engendered at Nottingham University [Higgins et al. 2003], where it was used for thirteen years before being superseded by CourseMarker, which is described in the following section.

Ceilidh supports the management and administration of computer-based assessment through the stages of development, execution, and administration. Ceilidh also implements what could be regarded as content management features, such as dividing courses into units and exercises and allowing lecture notes to be disseminated to students. It also supports customized views of content and implements teaching, monitoring, and recording grades.

While the majority of the assessment systems examine software written in C and Java, there are some that assess code written in more exotic languages. Scheme-Robo, described by Saikkonen et al. [2001], automatically assesses programs written in Scheme, a functional language. One of the differences between this project and the others is that the testing service is via an e-mail interface, originally developed for the *algorithm* assessment system, TRAKLA.

These second-generation systems have continued to evolve and develop. The Scheme-Robo project has been supplemented by a graphical user interface and an algorithm-animation component [Korhonen and Malmi 2000]. Some second-generation systems have evolved into third-generation web-oriented ones, which is the focus of the next section.

## 2.3 Third Generation – Web-Oriented Systems

While the second-generation tools can often be characterized by command-line interfaces and the manual operation of scripts, third-generation assessment systems make use of developments in web technology and adopt increasingly sophisticated testing approaches.

CourseMarker, developed at Nottingham University, builds on Ceilidh: it supports four types of users: students, tutors, teachers, and developers/ administrators and contains a number of content management components. One of the most interesting developments is the introduction of a "diagram" assessment system and its support for an impressive variety of languages including Prolog, SQL, and FORTRAN. Assignment assessment is done by an automated mechanism that analyzes the program across a number of criteria, with the emphasis on exploring program design. The design criteria are program format (typographic, lexical structure, and presence of particular features), program operation in response to test data (dynamic operation), program complexity and execution efficiency (time spent on execution). The marking actions are defined in a mark-action file that is constructed by the tutor.

While the original Ceilidh system gave a simple indication of the number of marks for a submission, CourseMarker provides the student with richer feedback. As well as presenting the student with a percentage of an optional alphabetic scale, a 'feedback tree' is given, allowing a student to interactively identify where marks were lost. There is also a system that suggests further reading.

CourseMarker provides a wealth of course, student, and assessment administration facilities. Using administration tools the tutor can add and delete users, edit course documents, create and install new courses, and assign permissions to users. The tutor is also given wizards that allow exercises to be constructed, and for the creation of new exercises from existing ones. There is also provision for reporting, including information on cohort statistics and access to data logged as a part of the assessment submission process. Further developments include the implementation of multiple-choice questions and the conversion of all existing Ceilidh projects to the CourseMarker form.

The BOSS system has also continued to develop. As well as providing a set of graphical user interfaces to students and tutors, the latest development also includes a web server component. This allows the tutor to review submissions using a traditional web-browser. BOSS system also introduces the notion of plagiarism detection.

BOSS is currently used in two different forms: an open-source version that can be installed onto a system utilizing the Java platform, and an implementation that is specific to Warwick University. The latter department-specific version is integrated with the university information services.

Daly and his colleagues have developed a Java-oriented assessment system called RoboProf, deployed in an honors degree program at Dublin City University [Daly 1999; Daly and Waldron 2004]. The system presents programming problems within a web-browser and the student is asked to type a program into a text box. When complete, the assignments are submitted, compiled, and results returned. If the program is valid, the student is invited to move onto the next programming activity. The system is comprised of 39 problems, divided among the areas of variables, control flow, arrays, and strings.

RoboProf has a number of particularly attractive features: The notion of levels gives the student a view of progress; the automatic programming assessment element is integrated with multiple-choice assessment, which complements individual programming tasks. The multiple-choice questions are generated randomly, thereby reducing a student's potential to copy the results submitted by others. It is interesting to note that Daly has also contributed to developments in plagiarism detection [Daly and Horgan 2005].

Jackson argues that assessment of programming assignments should include a human component as well as an automatic one [Jackson 2000]. In this work, Jackson presents a series of tests using predefined test data, the results are given to the tutor, who in turn examines the software for appropriate documentation and evidence of good design in the program structure.

The Automated System for the Assessment of Programming (ASAP), from Kingston University, is another development that has much in common with earlier ones. ASAP focuses primarily on the Java teaching language, which is used at Kingston University. Testing is carried out using a combination of IO stream analysis and by testing individual member functions.

ASAP adopted an implementation strategy that is somewhat different from others. The initial intention was to develop a submission system through a university-wide virtual learning environment. The submission system would then be tied to a proprietary "grade book" mechanism that allows tutors to view the students' submissions. Hence, a marking service was constructed that could be accessed by other software components through a web-service, and this approach informed other, subsequent, design decisions.

As in other projects, plagiarism was considered a potential classroom issue. The ASAP project worked with a partner at another university to implement a program interface to an internationally recognised Java plagiarism-detection engine, called JPLAG. Additional components such as randomized fixed-response questions [Daly 1999] were also made available through a separate web service interface.

## 3. A PEDAGOGIC REVIEW

Arguably the most critical component of programming assessment is the nature of the task set before the student. The parameters and issues affecting this component are examined in this section.

### 3.1 The Test Interface

The means by which any test can access and evaluate a student submission is, naturally, an important consideration. From the earliest records of automated assessment, a common methodology has been to test the functionality of a student submission via its *software interface* [Hollingsworth 1960; Naur 1964]. This makes the academic, summative assessment methodology equivalent to the industrial test methodology: if all test cases result in successful program behavior, then the student has passed successfully. This approach imparts a practical, objective quality to the assessment, and provides the student with an insight into the primary means by which industrial software is verified. In one sense it is a supremely empiricist approach to software evaluation, in that it is only the standard interfaces of the executable code that are used in the evaluation, as opposed to any perceived value to the lines of source code from which that interface was derived.

Similarly, systems can be designed to assess functional completeness via a standard input/output interface, using file I/O or pipes from other programs as the required sources and sinks; the Ceilidh system (described in Section 2.2) uses this interface. One advantage is that a given test can be used to assess programs written in any suitable language, providing versatility and opportunities to investigate differences between languages. But a difficulty with this approach is the implementation of I/O handling that is robust to irrelevant differences between the expected and submitted programs.

However, there are software quality metrics besides functional completeness that can be used in assessing programming assignments. Many languages use indentation to aid the readability; this can be checked automatically, as can the frequency and location of comments. Ceilidh uses execution time; BOSS uses completeness of source code formatting and indentation; and Jackson supplements the automatic tests with a manual inspection of the submission. The Scheme-Robo system (described in Section 2.2) evaluates the internal representations that are used in the submitted program. Subtle software quality metrics such as reusability and platform portability will be more difficult to quantify in this context; however, there may be specific problems that necessitate that mode of development.

### 3.2 Grading Submissions

The technology to deliver the assessment is neutral with respect to the policy for awarding the subsequent grades. Marks may be awarded for the various stages of the submitted solution, e.g., compiling, constructing, and testing each required function (or method) in turn. Although the traditional pass threshold is 40%, there is a sense in which incomplete, malfunctioning solutions could be regarded as unequivocally undesirable (both in terms of software engineering and for student development). A grading system could be defined wherein low marks are offered for complete solutions to relatively easy questions, and more marks awarded for complete solutions to progressively harder problems. No marks are offered for incomplete solutions. The intended effect of this grading system is to increase the importance (in the eyes of the students) of achieving a completely working solution, which they may otherwise regard as unimportant, since they can pass with 40% without ever having achieved one. A complete solution is regarded as an important step in building the confidence of student programmers, even if some initially complete only the simplest of tasks.

### 3.3 Graphics and Graphical User Interfaces

Programs with graphics and graphical user interfaces are perennially attractive to students, since they provide impressive-looking results. But they present a problem for the automatic testing system, in that graphical and interactive software components are more difficult to test for two important reasons. First, the test harness needs to be more sophisticated in order to examine the source on which the graphics have been painted, or to furnish the software submitted with user events (such as pointing and clicking) and to then check subsequent behavior. (Such systems have been developed successfully [English 2002].) Second, the specification of requirements for an automated assessment always needs to be more precise than for the equivalent *manually* marked assessment. The requirements need to be precise to allow the assessment system to use reasonably simple rules to distinguish between inaccurate solutions and correct solutions programmed to a slightly different specification. In graphics problems in particular, this presents potential difficulties. For example, consider the task of drawing the British (Union Jack) flag on an applet window. To allow automatic marking to be reasonably straightforward, the instructions would have to be specified to such an extent that the nature of the assessment would be changed as a consequence.

### 3.4 Test Conditions

A further parameter of assessment is the material available to the student during the test, which determines whether it is a test of memory, adaptability, or initiative. A distinction is commonly drawn between formative assessments (designed to improve student knowledge and skill) and summative assessments (designed to make a judgment about student achievement). Broadly, there are three types of material: documentation about language and libraries; example source code, possibly written beforehand by the student; and instructional material such as slide and textbook content. By controlling the availability of these materials, the instructor can adapt the assessment to the specific pedagogic requirements.

A further, related, condition is the extent to which students are required to adapt and extend existing source code in order to satisfy assessment requirements. The alternative is to specify a question in which students must write source code from scratch. These two conditions represent complementary skills in software engineering: again, the instructor can tune the rubric to achieve particular learning outcomes.

### 3.5 Complex Questions

One of the perceived shortcomings of the methodology for automatic assessment is that its inflexibility prevents assessment of more complex questions. Automatic assessment is inflexible because it specifies in advance the interface through which the student is assessed. Questions requiring the student to design his or her own interface cannot be tested in this way, since a design cannot be easily made available to the assessment system beforehand. However, by combining multiple classes, possibly in code that is provided to the students beforehand, test interfaces can be specified; but this will still require a considerable amount of work.

## 4. EVALUATING AUTOMATIC PROGRAMMING ASSESSMENT

To measure the success of any engineering endeavor, evaluation with respect to the stated objectives is essential. For an automated assessment system, the stated objectives could be whether the system does what it is supposed to do, whether it is liked by its users, and whether it helps students become more proficient at programming.

First, it should be noted that the automation of assessment is somewhat independent of the issue of where the students are located during the assignment: in the lab, or at a distance [Korhonen et al. 2002]. Deploying an automated assessment in the classroom allows an instructor to troubleshoot problems that students may have with the system and to understand their difficulties by observing their actions.

Descriptions of assessment systems usually contain a small evaluative section. In an earlier review of this area, Leal and Moreira [1998] carried out an evaluation of the students' perceptions of automatic grading. Students were reported to have no fundamental objections to the application of an automatic grading system; a similar result was found by von Matt [1994]. There was disagreement as to whether a human or machine grader is preferable. Students stated that support materials such as user instructions or online help should be provided wherever possible and there were some statements on the inability to take programming ambiguity into account; a comment echoed by Oliver [1998].

Woit and Mason [2003] and English [2002] take a different evaluative approach. Instead of merely identifying the students' likes and dislikes, they attempt to evaluate whether assessment systems can help students become more proficient. Woit and Mason performed a five-year study using a combination of teaching methods. Best performance was observed when students were presented with weekly online assessments which were selected from a set of voluntary exercises. In accordance with this research, we can conclude that online assessment, when delivered as an integral part of a course, can generate an improvement in student performance, since it may reduce the stress inherent in high-stakes testing and increase motivation. (In a related questionnaire described by Higgins et al. [2003], students appear to prefer this approach.)

In their discussion of their SQL tutor system, Mitrovic et al. [2000] consider further criteria for evaluating their system: the strength of the student's opinion on what to work on next and the propensity to abandon an assignment before completion. Mitrovic et al.'s automated tutor constructed a model of the student's current state of learning and the effects of allowing students to have access to the model were evaluated.

Automatic assessment systems have a number of problems in common. The first, and perhaps the most significant, is the issue of requirements. For automatic assessment systems to be successful, all assignments given to students must be carefully specified. There is a parallel in other assessment domains. Good multiple-choice questions are considered notoriously difficult to write. Within programming, interpretation is key to success, and instructions must guide interpretation precisely. As discussed in Section 3.1, some educators may take the view that an assessment system should examine the style of a program or take the use of comments into account.

Some of the automatic testing systems explored in this article have an interesting and significant flaw. A program can be submitted that may be correct in its operation yet be pathological in its construction. In this case, only by using several different criteria, can assignment quality be comprehensively assessed automatically.

## 5. DEVELOPMENT DIRECTIONS

When thinking about how automated assessment systems may change, we must consider what may be able to be assessed as software technology evolves. The automatic assessment domain will always be limited to problems that can be assigned a clear marking scheme. The assignments administered to students will reflect innovations emerging from academia and industry and the skills that employers find useful.

Two recent innovations in the literature are worth mentioning. Almost all the programming assignments that students construct are command-line based, providing two advantages. First, the assignments are of a relatively small size, and subsequently rather simple. Second, via input and output redirection, a technique found in second-generation tools, the command-line programs can be tested and easily evaluated.

Command-line applications, although undoubtedly useful, can be somewhat far from a new student's experience. Students embarking on a computer science or software engineering degree will be familiar with graphical user interfaces. Even though a focus on command-line applications may be entirely appropriate for the development of programming skills, it may be seen as uninspiring, given the average student's experience with a contemporary operating system. This raises the question of whether GUI applications, complete with windows, icons, and buttons, could be assessed automatically.

The assessment of Java GUI programs has been explored by English [2004], who proposes developing a framework from which students can create graphical components that can be executed and viewed. The JEWL system is in fact a GUI tool kit, with which the GUI can be replaced by a test harness that can then interpret instructions that the program under test executes. Interactions are carried out by a simple message-loop.

The second innovation corresponds to the relationship between automatic assessment in the teaching domain and in industry, where it is an increasingly established software development practice. Indeed, software development approaches like extreme programming (or XP) advocate test-first strategies [Beck 2003], that is, function tests are developed before or at the same time as the code that performs the required function. Developing tests at such an early stage of software development ensures that problems with the design and operation of a software system are discovered as early as possible in the development cycle. Unit tests are particularly useful in preventing regression problems. The later versions of the BOSS system use the JUnit based testing mechanism, which also has widespread industrial exposure.

Meta-testing, i.e., assessment of the test set that accompanies a given software solution is a further extension of the test domain. Here the design and implementation of the test set is assessed as to whether it is a valid and complete accompaniment to the solution. Edwards [2003a; 2003b] presents a system that automatically tests the *student's* tests with a number of freely available tools. This approach teaches the importance of tests and allows a student to appreciate what constitutes good and bad tests and introduces the concept of code coverage.

The changes in how testing is done are also evident. One of the currently fashionable topics in software development and engineering is that of a service- oriented architecture (or SOA) [Wilson et al. 2004]. Rather than using a traditional programming interface (API), requests can be sent to a machine that implements a web service. This process will take a request, perform a set of actions, and return a response. Some third-generation testing systems are already moving in this direction, due to adoption of earlier but related technologies such as Java remote method invocation (RMI) that separates client applications from server operations.

Management of end-user assessment and submission review components can begin to make use of a service-oriented architecture by requesting results from a distant web service. Once obtained, the results can be rendered using any number of approaches, either through a separate application or through a number of different website portal technologies.

A key challenge, addressed below, is how to expand the accessibility of automatic assessment systems so that they can be used in arbitrary institutional environments. This requires the definition of suitable interoperability standards. Automating testing and assessment of programming assignments is a perennial problem that has been the subject of a significant number of research projects. Currently there is no consensus regarding standardization although there appears to be a large amount of commonality between various projects both in what they do and the technologies they apply. The lack of standardization may be partly due to different educational practices at different institutions and different views about what elements of a system are considered essential.

It is likely that standardization of assessment systems will become an important element in future developments. The ASAP system adopted interoperability standards that were being explored by projects funded by the UK Joint Information Systems Committee. The call for interoperability is also made by the various e-learning communities that are implementing and working with enterprise-level virtual learning environments (VLEs) or managed learning environments (MLEs), which allow students to access both course materials and a myriad of other services such as discussion forums and time tables. It is in the interest of universities to enable different VLE systems to work together in order to prevent the emergence of an inflexible VLE monoculture.

The *learning design* specification is also of interest. The first managed learning environments used a concept known as a *learning object*. This is a set of resources which a student can use to gain instruction on a particular subject. A learning object can include web pages, other documents, multimedia, and automated question items and tests. However, even though the learning object is a useful concept for educational technology, it was difficult to share patterns of object *use* among educators. Learning design allows patterns of resource use to be described, including when and how instructional content is delivered, tests are administered, and remedial instruction in the form of additional material or reading lists is provided.

## 6. SUMMARY

This article has reviewed a number of projects that automatically assess student programming assignments using a test-based approach. Three broad generations of test-based assessment systems have been identified: The first-generation systems represent the initial attempts to automate testing, and are considered real innovations. Their usability, however, is limited to their particular computing laboratories. The second-generation systems are characterized by their use of command-line-based tools, sometimes in association with locally built and maintained GUI interfaces. The third, and current, generation systems make use of contemporary web-based technologies, and sometimes provide additional support for educators in the form of assessment management and reporting facilities. The third generation of assessment systems is beginning to offer services outside the boundaries of a particular laboratory, providing tools directly to student desktops via web browsers.

The automated assessment systems discussed in this article can also be used in a number of ways in conjunction with other nontest-based approaches such as human-based marking, peer review, and intelligent tutoring systems. They can be used by tutors to encourage students to write software by offering regular and frequent feedback. Alternatively, these systems can be a part of a central assessment regime within an institution. Different educators and institutions can apply these systems in different ways.

Some evaluation papers cite a number of advantages that adopting these systems will provide. For example, assessing programming assignments is a difficult and time-consuming task, and an educator's time may be more effectively spent giving guidance to students and explaining concepts that they find difficult to grasp. Another advantage is that human assessors are fallible, whereas machine assessors can give entirely objective responses (provided they are adequately programmed).

There are, of course, disadvantages. The main one is the restrictions that apply to what can be assessed automatically, that is, only clearly defined questions with completely specified interface for the overall solution. Moreover, unless explicitly specified as a necessary component to the solution, an assessment engine cannot award additional marks for creative design or innovative solutions.

There will always be a need for human assessors. Rather than replacing the tutor, these systems can provide support. They can also support students, allowing them to gain more confidence in their work. The systems help tutors and educators by allowing them to identify potential student misconceptions or problems more easily, and thus facilitate programming and software engineering education more effectively.

## ACKNOWLEDGMENTS

Thanks are extended to Steve Grindle and Justin Cobb for all the hard work they have carried out during the period of the ASAP project. The authors would also like to thank the reviewers for their expert and substantive suggestions for how the paper could be improved.

## REFERENCES

BECK, K. 2003. *Test Driven Development: By Example*. Addison-Wesley, Boston, MA.

DALY, C. AND HORGAN, J. 2005. Patterns of plagiarism. In *Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education.*

DALY, C. AND WALDRON, J. 2004. Assessing the assessment of programming ability. In *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education.* 210-213.

DALY, C. 1999 RoboProf and an introductory computer programming course. In *Proceedings of the 4th Annual SIGCSE/SIGCUE ITiCSE Conference on Innovation and Technology in Computer Science Education*. 155-158.

EDWARDS, S. H. 2003a Using test-driven development in the classroom: providing students with automatic, concrete feedback on performance. In *Proceedings of the International Conference on Education and Information Systems: Technologies and Applications.* 421-426.

EDWARDS, S. H. 2003b. Teaching software testing: Automatic grading meets test-first coding. In *Proceedings of the OOPSLA'03 Conference.* Poster presentation. 318-319.

ENGLISH, J. 2004. Automated Assessment of GUI Programs using JEWL. *Proceedings of the 9th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education.* 131-141.

ENGLISH, J. 2002. Experience with a computer-assisted formal programming examination. *ACM SIGCSE Bull. 34*, 3: *Proceedings of the 7th Annual Conference on Innovation and Technology in Computer Science Education.* 51-54.

ENGLISH, J. AND SIVITER, P. 2000. Experience with an automatically assessed course. In *Proceedings of the 5th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*. 168-171.

FORSYTHE, G. E. AND WIRTH, N. 1965. Automatic grading programs. *Commun. ACM* 8, 5, 275-529.

GEHRINGER, E. F. 2001. Electronic peer review and peer grading in computer-science courses. In *Proceedings of the 32nd SIGCSE Technical Symposium on Computer Science Education*. 139-143.

HEXT, J. B. AND WININGS, J. W. 1969. An automatic grading scheme for simple programming exercises. *Commun. ACM 12*, 5, 272-275.

HIGGINS, C., HEGAZY, T., SYMEONIDIS, P., AND TSINTSIFAS, A. 2003. The CourseMaster CBA system: Improvements over Ceilidh*. J. Edu. Inf.Technol. 8*, 3, 287-304.

HOLLINGSWORTH, J. 1960. Automatic graders for programming classes. *Commun. ACM 3*, 10, 528-529.

HUNG, S.-L., KWOK, L. F., AND CHAN, R. 1993. Automatic programming assessment. *Comput. Edu. 20*, 2, 183-190.

ISAACSON, P. C. AND SCOTT, T. A. 1989. Automating the execution of student programs. *SIGCSE Bull. 21*, 2, 15-22.

JACKSON, D. AND USHER, M. 1997. Grading student programing using ASSYST. In *Technical Symposium on Computer Science Education, Proceedings of the 28th SIGCSE* (San Jose, CA), 335-339.

JACKSON, D. 2000 A semi-automated approach to on-line assessment. In *Proceedings of the 5th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, 164-167.

JOY, M. AND LUCK, M. 1998. Effective electronic marking for on-line assessment. In *Proceedings of the 6th Annual Conference on the Teaching of Computing* (Dublin City University, Ireland). 134-138.

KORHONEN, A. AND MALMI, L. 2000. Algorithm simulation with automatic assessment. In *Proceedings of the 5th Annual SIGCSE/SIGCUE ItiCSE Conference on Innovation and Technology in Computer Science Education*. 160-163.

KORHONEN, A., MALMI, L., MYLLYSELK , P., AND SCHEININ, P. 2002. Does it make a difference if students exercise on the web or in the classroom? In *Proceedings of the 7th Annual SIGCSE/SIGCUE Conference on Innovation and Technology in Computer Science Education, ITiCSE02* (Aarhus, Denmark), ACM Press, New York, 121-124.

LEAL, J. P. AND MOREIRA, N. 1998. Automatic grading of programming exercises. Tech. Rep. DCC-98-4, Dep. di Ciência de Computadores, Universidade do Porto, Portugal.

LUCK, M. AND JOY, M.S. 1999. A secure on-line submission system. *Softw. Pract. Exper.* 29,8, 721-740.

MITROVIC, A., MARTIN, B., AND MAYO, M. 2000. Using evaluation to shape ITS design: Results and experiences with SQL-tutor. In *User Modeling and User- Adapted Interaction 12*. 243-279.

NAUR, P. 1964. Automatic grading of students' ALGOL programming. *BIT* 4, 177-188.

OLIVER, R. G. 1998. Experience of assessing programming assignments by computer. In *Computer Based Assessment Volume 2: Case Studies in Science and Computing.* D. Charman and A. Elmes, eds., University of Plymouth. 45-49.

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

SAIKKONEN, R., MALMI, L., AND KORHONEN, A. 2001. Fully automatic assessment of programming exercises. In *Proceedings of the ITiCSE 2001Conference,* ACM Press, New York, 133-136.

SMYTHE, C., ET AL. 2005. IMS question and test interoperability, Ver. 2.0, IMS Global Learning Consortium.

SYKES, E. R. AND FRANEK, F. 2004. A prototype for an intelligent tutoring system for students learning to program in Java. *Int. J. Comput.Appl.1*, 35-44.

THOBURN, G. AND ROW, G. 1996. PASS - An automated program assessment system. In *Proceedings of the 4th Annual Conference on the Teaching of Computing* (Centre for Teaching Computing, Dublin City University).R. Rory O'Connor and S. Alexander, eds.

VON MATT, U. 1994. Kassandra: The automatic grading system. Tech. Rep.UMIACS-TR-94-59, Institute for Advanced Computer Studies, Dept. of Computer Science, University of Maryland.

WILSON, S., BLINCO, K., AND REHAK, D. 2004. Service-oriented frameworks: Modelling the infrastructure for the next generation of e-learning systems. JISC-CETIS. Available from JISC.

WOIT, D. AND MASON, D. 2003. Effectiveness of on-line assessment. In *Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education*. 137-141.

Received March 2005; accepted May 2005