# Automated Assessment of Programming Assignments

VREDA PIETERSE
University of Pretoria

This is a position paper in which I argue that massive open online programming courses can benefit by the application of automated assessment of programming assignments.

I gathered success factors and identified concerns related to automatic assessment through the analysis of experiences other researchers have reported when designing and using automated assessment of programming assignments and interpret their potential applicability in the context of massive open online courses (MOOCs).

In this paper I explain the design of our own assessment software and discuss our experience of using it in relation to the above-mentioned factors and concerns. My reflection on this experience can inform MOOC designers when having to make decisions regarding the use of automatic assessment of programming assignments.

Categories and Subject Descriptors: K.3.2 [**Computers and Education**]: Computer and information Science Education; K.3.1 [**Computers and Education**]: Computer Use in Education

General Terms: Human Factors

Additional Key Words and Phrases: Assessment software, automatic assessment, MOOC

## 1. INTRODUCTION

A massive open online course (MOOC) is a model for delivering free learning content online without entry prerequisites. Typically, a MOOC is not for credits and has no limit on attendance [Thompson 2011]. Recent trends in tertiary education with a number of elite universities now offering free education through the provision of massive open online courses (MOOCs) are bound to change the model of learning.

MOOCs are as a rule not-for-credit [Daniel 2012], consequently the need for summative assessment may become unnecessary. The need for formative assessment, however, may become more important. Participants in MOOCs learn for the sake of learning. The role of formative assessment in learning is paramount [Earl 2003]. Therefore, continuous formative assessment is needed to enhance learning and to assist students to evaluate their own learning.

When a MOOC is designed to teach a skill like programming, the inclusion of a service for the automatic assessment of programs has the potential to contribute to the quality of such MOOC. Apart from its formative value, it may also provide opportunities to practise programming, which is needed for students to be able develop programming skills. Research by Ericsson et al. [1993] [K. Anders Ericsson and Cokely. 2007] gave rise to numerous findings in support of the notion that expertise is a consequence of years of deliberate practising coupled with coaching, rather than talent, as aptly described by Colvin [2006]:

> *The best people in any field are those who devote the most hours to what the researchers call "deliberate practice." It's activity that's explicitly intended to improve performance, that reaches for objectives just beyond one's level of competence, provides feedback on results and involves high levels of repetition.*

Offering assignments as opportunities to practise is essential. These are more valuable if fast and accurate feedback is provided. In the context of MOOCs this is a real challenge.

Automatic assessment of programming assignments has been applied as long as programming has been taught. Automatic assessment had already been suggested by Hollingsworth [1960] long before personal computers existed. In many programming courses at tertiary institutions the use of automated assessment has been proved useful through the use of systems like QuizPACK [Brusilovsky and Sosnovsky 2005], BOSS [Joy et al. 2005], PASS [Yu et al. 2006], ALOHA [Ahoniemi and Reinikainen 2006], EasyAccept [Sauvé et al. 2006], Peach [Verhoeff 2008], Web-CAT [Edwards and Perez-Quinones 2008], ProgTest [de Souza et al. 2011], Kattis [Enström et al. 2011], Automatic Marker [Suleman 2008], and many others. Most of these, and many more, are discussed in more detail by Ihantola et al. [2010].

Several advantages of automatic assessment in programming courses have been observed. Ala-Mutka [2005] mentions speed, availability, consistency and objectivity of assessment, while Vujošević-Janičić et al. [2012] highlight the advantage of immediate feedback for students, especially for novices who can benefit from early disambiguation of misconceptions.

Advantages that are particularly relevant for MOOCs are its potential to facilitate learning and allowing students to practise and get feedback at any time and anywhere [Malmi et al. 2002], as well as the possibility to give more tasks to the students [Enström et al. 2011]. Chen [2004] asserts that their system allowed students to be held to a higher standard and enabled students to meet this standard. Unfortunately the use of automatic assessment is coupled with its own set of challenges.

I gathered success factors and identified concerns related to automatic assessment from the experiences of the use automated assessment of programming assignments. These are summarised in Sections 2 and 3. Our experience of using our own assessment software is discussed in Section 4. Finally, I reflect on this experience and provide insight that MOOC designers can use when making decisions regarding the use of automatic assessment of programming assignments.

Authors address: vreda.pieterse@up.ac.za.

DOI:http://dx.doi.org/10.1145/1559755.1559763

## 2. SUCCESS FACTORS

This section is a discussion of some factors that are likely to contribute to the successful application of automatic assessment of programming assignments. These were identified through a literature review of reported cases where automated assessment of programming assignments were applied coupled with our own experience of using an automatic assessment tool in a fairly large resident course. These are interpreted in terms of their possible generalisation to their impact when applied in the context of a MOOC.

### 2.1 Quality assignments

The quality of the assignments is equally important for any course and MOOCs are no exception. Feldman and Zelenski [1996] state that first-rate homework assignments are integral to the success of courses. Hundley and Britt [2009] remark that an important part of a successful course is good assignments. When assignments are assessed manually, it is usually possible to compensate for poor assignment design by giving credit for creativity of solutions while assessing. This is, however, not always the case when automatic assessment is applied. According to Ala-Mutka [2005], the use of automatic tools increases the need for careful pedagogical design of the assignment and assessment settings. Thus, the importance of quality assignments is very important for MOOCs.

### 2.2 Clear formulation of tasks

Hollingsworth [1960] already realised that assignments need to be properly formulated in order for automatic assessment to be effective. Douce et al. [2005] remarks that the specification of requirements for an automated assessment always needs to be more precise than for the equivalent manually assessed assignment.

Additional care has to be taken to avoid ambiguities. If the specification is ambiguous it allows for different interpretations; consequently, it is likely that some valid solutions by students may be rejected by the automatic assessment program simply because the assessment instructions may not be configured to recognise it. This observation is a consistent theme mentioned by most authors who have used automatic assessment tools.

Ala-Mutka [2005] cautions that no ambiguities are allowed in the problem specification, especially when considering input/output formats. Formulating the requirements for assignments that are subjected to automatic assessment has to be meticulous, as Cerioli and Cinelli [2008] point out:

> *To make this possible, the project specification must be extremely precise, so that the behaviour of the implementations is totally predictable.*

Clear formulation of tasks is important in all situations, including MOOCs.

### 2.3 Well chosen test data

In most systems applying automated assessment, the functionality of a program is tested by running the program against several test data sets. Ala-Mutka [2005] points out that the coverage of the assessment depends on the test case design. The accuracy, and consequently the formative value of the assessment, is highly dependant on the design of the test cases it uses. Vujošević-Janičić et al. [2012] remark that the grading is directly influenced by the choice of test cases. Montoya-Dato et al. [2009] point out that the set of test cases needs to be “well thought out” to prevent wrong programs from passing test runs.

When wrong programs are falsely identified as correct the students who created the erroneous solutions may remain unaware of their failure, which may have a negative impact on their level of mastery of the material.

The design of test cases based on the problem statement may be as challenging as creating the solution to the problem statement, if not more challenging. It is equally important to carefully select variable values to cover all important paths throughout the program [Vujošević-Janičić et al. 2012].

The test data are the Achilles’ heel of any system that applies automatic assessment of programming assignments. Using a system in a MOOC does not change this.

### 2.4 Good feedback

The importance of the feedback aspect of formative assessment can not be overemphasised. Carless et al. [2011] describe feedback as a key ingredient of the development of quality student learning. Ahoniemi and Reinikainen [2006] remark that novice students require profound and personalised feedback on their programming assignments to support them to improve their weaknesses. Feedback on assignments allows students to revise their submissions [Malmi et al. 2002]. When students know what exactly went wrong with their submissions and where their programs failed, they can use the information to learn from their mistakes.

Providing fully automated feedback may be extremely challenging. Many existing systems can be improved in this regard. The system designed by Ahoniemi and Reinikainen [2006] is able to provide quality feedback, but is not fully automated. It uses semi-automatic phrasing which allows lecturers to provide consistent and objective feedback regarding assignments. The only system that I am aware of that provides fully automated feedback is the system offered by Vujošević-Janičić et al. [2012]. Their system provides support for meaningful and comprehensible feedback to students that students found helpful to correct their errors.

When applying automatic assessment in a MOOC the importance of good feedback increases when compared with its importance in a traditional setting. This is because in a traditional setting there are various other communications between the instructor and the student which can augment the feedback provided by the system, while in a MOOC the additional communication opportunities are diminished.

### 2.5 Unlimited submissions

If automatic assessment is used for formative reasons, it is essential that students be allowed to resubmit improved programs in response to the feedback. Rapid gratification upon the correct improvements reinforces the learning that is intended by the feedback. Suleman [2008] mentions that iterative learning is supported by the possibility of multiple submissions.

Douce et al. [2005] observed that in a traditional setting unlimited submissions may lead to undesirable behaviour. When an unlimited number of submissions are allowed, students are not forced to think about their solutions and they may merely use the automatic assessment tool as a tester program. In a MOOC, where the learning model is different, the presenters of the course need not be concerned if some of the participants revert to such shallow learning. The facility for unlimited submissions should be available for the sake of participants who want to use it appropriately.

### 2.6 Student testing maturity

Edwards [2003] observed that emphasis on test-driven development had a positive effect on the ability of their students to test their programs. Ala-Mutka [2005] deem it necessary that students learn to design test cases and test their programs thoroughly before submitting. Cerioli and Cinelli [2008] advise that the students at least be given part of the test sets to evaluate their projects. They assert that it improves the student's understanding of the specifications.

The application of automatic assessment when coupled with adequate training in software testing provides a learning experience where students learn to program in a more professional manner, favouring robustness and precision over quick-and-dirty solutions [Suleman 2008]. The following opinion of Ben-David Kolikant and Mussai [2008] supports the notion that it is more likely that the students reach higher levels of proficiency in programming when subjected to automatic assessment of their assignments coupled with appropriate training in software verification:

> *Knowing the craft of correctness verification is essential to being a successful programmer.*

When automatic assessment is applied in a MOOC, it is strongly advised that testing techniques relevant to each assignment be emphasised. It will support the students in gaining more value from the automatic assessment and at the same time develop the testing maturity of students in conjunction with their programming skills.

### 2.7 Additional support

There seems to be a general agreement that the need for personal guidance remains [Enström et al. 2011; Ben-David Kolikant and Mussai 2008]. In the traditional learning model the onus remains on the instructors to provide remedial advice and to support the students to interpret the automatic feedback they get from the system. Montoya-Dato et al. [2009] report that questions answered by instructors promote both independent learning and reflective thinking which are deemed important for deep learning.

When entering the MOOC learning model, instructor involvement becomes less pertinent. Most of the MOOC participants remain self-reliant and the vast majority of MOOC participants hardly ever have any direct contact with the instructors. Often a MOOC presentation is a replication of a resident presentation when considering the learning content, but is very different with regard to face-to-face contact. Lectures are provided in the form of recordings. Students are not able to participate in these lectures. Furthermore, in a MOOC there is no face-to-face instructor involvement while students are doing their assignments [Mackness et al. 2010; Vihavainen et al. 2012; Gal-Ezer et al. 2009].

The following options should be considered to compensate for diminished instructor involvement:

—Providing notes on how to use the automatic assessment tool.

—Providing supporting notes on the interpretation of the feedback provided by the automatic assessment tool and how to correct errors that were identified.

—Providing means by which students can ask for support. Ideally this also has to have some automation. A request tracker, for example RT: Request Tracker [Best Practical Solutions INC 2012], can be employed.

—Providing a discussion forum where students can help one another.

—Sharpening the accuracy of the automatic assessment tool as well as the quality of the feedback it can provide.

## 3. ISSUES

This section discusses some issues that need to be taken into account when considering the option to apply automatic assessment of programming assignments. These have been accumulated by combining experience reports found in the literature with our own practical experience. The consequences of applying automatic assessment of programming assignments are interpreted in the context of a MOOC.

### 3.1 Challenging effort

According to Ala-Mutka [2005], the creation of automatically assessable programming assignments requires special attention, since even small mistakes in the marking definitions can cause problems. Chen [2004] reports that it took a substantial amount of thought and care to write good test cases. Douce et al. [2005] parallels the writing of test cases with designing good multiple-choice questions which are also considered to be "notoriously difficult" to write.

MOOC designers should realise that these challenges may outweigh the benefits of being able to assess programs automatically and should be prepared to rise to these challenges when considering the option to use automatic assessment in their MOOCs.

### 3.2 Increased effort

Enström et al. [2011] mention that automated assessment assists in reducing the teacher's workload by removing the tedious work of manually verifying correctness. On the contrary, it has been reported that use of automatic assessment does not reduce the workload of the instructors, but merely moves the required effort from manual assessment to the design and implementation of assessments.

Ala-Mutka [2005] states that the design of test cases for the automatic assessment of programs is more burdensome than designing manual assessments, whereas Cerioli and Cinelli [2008] confess that the most time consuming activity of project evaluation with the support of their automated grader is the definition and of tests. Chen [2004] admits that building a robust feedback system amounts to a heavy workload, especially if the student project instructions allow for some design freedom.

More time is needed to create assignments that will be automatically assessed. As explained in Section 2.2, the assignments themselves need careful and detailed explanations. Furthermore, as discussed in Section 2.3 the test suites to evaluate the solutions are subject to careful design. This is coupled with the increased onus on the instructor to provide more personal attention to students [Enström et al. 2011; Ben-David Kolikant and Mussai 2008; Montoya-Dato et al. 2009]. The instructor who uses automatic assessment as an option has to realise that it may require substantially more effort than would be needed when manual assessment is applied, especially if the number of students is manageable. In the context of a MOOC, the large volumes of participants may warrant the extra effort. If done well, it is likely that it is more reliable than peer evaluation, which is a popular alternative for assessment in many MOOCs.

It is advised that one should capitalise on this effort by reusing assignments and their test sets. Chen [2004] reports that projects from prior semesters were heavily reused. The system by de Souza et al. [2011] provides for an oracle assignments base that can be used instead of defining and implementing new assignments.

## 3.3 Suppressing creativity

Owing to the requirement that assignments that are subjected to automatic assessment need to be extremely precise, the creativity of the students is suppressed. Enström et al. [2011], split tasks into-sub tasks so that the correctness and efficiency of each sub-task could more easily be established. When doing this, the instructors inhibit creativity and at the same time rob the students of the opportunity to practise the skill of stepwise refinement.

CodeAssessor by Vander Zanden et al. [2012] offers a complete program where students have to fill in missing code blocks. It presents an effective method to let students practise the writing of code snippets and test its validity without having to write scaffolding code. It is beneficial, especially for novices, to gain practical experience sooner, but it provides limited opportunities to practise program design.

The consequences of the possible lack of ability to allow for creativity will ultimately depend on the goals of the MOOC. Automatic assessment may be more challenging in a MOOC where software design is an objective of a MOOC, while this lack may not be as problematic in a MOOC that aims to introduce basic programming skills.

## 3.4 Plagiarism

Lukashenko et al. [2007] see plagiarism as a kind of social illness and discuss methods for the *prevention* and for the *treatment* of this illness. Prevention entails education to foster an attitude that condemns plagiarism while treatment entails plagiarism detection aimed at averting its occurrence. These go hand in hand. Wagner [2004] discusses cheating in computer science education and gives sound practical advice in this regard.

When programs are automatically assessed, the inclination among students to copy from one another seems to rise. Suleman [2008] reports that his teaching assistants voiced their concern about increased probability of copying that may occur when the code is not scrutinised by people during manual assessment. Chen [2004] remarks that cheating is always an issue, especially where projects are reused. Ihantola et al. [2010] maintain that students might be more inclined to knowingly submit weak or incorrect solutions that get accepted by a machine than trying to cheat a person. For this reason automatic assessment systems often incorporate plagiarism detection; for example plagiarism detection is an integral part of BOSS [Joy et al. 2005].

Many dedicated systems for program similarity testing are available, of which MOSS [Aiken 1994], JPlag [Prechelt et al. 2002], Plaggie [Ahtiainen et al. 2006], YAP3 [Wise 1996], SID [Chen et al. 2004], SSID [Poon et al. 2012], and PlaGate [Cosma and Joy 2012] seem to be applicable in educational settings.

It remains important to address plagiarism in all academic settings. The new learning model offered by MOOCs, where learning is no longer for credit, may eliminate most of the factors contributing to students wanting to copy someone else's work. As one of our students remarked:

> *Study the code and understand what it does. Then code your own solution; this makes you better at coding and eliminates the need to plagiarise. If you just copy code all the time what do you learn?*

This is an important difference between MOOCs and traditional courses. In the latter, the need for plagiarism detection and action against plagiarism is paramount while in MOOCs it may serve no purpose.

## 3.5 Higher skills expectations

The higher level of understanding of program verification that is required was discussed in Section 2.6. In addition to having to master software testing skill at a higher level, having to use an automatic assessment tool itself may pose additional challenges. Woit and Mason [2003] noticed that their students needed experience to work successfully with their system in an online examination situation.

Even when the use of a program may seem very easy, its simplicity may be deceiving. Chen [2004] reports that submission to his system was as simple as issuing the following command-line instruction:

```
mail < program.c
```

He discovered that the consequence of an honest small mistake can be disastrous; for example, the program is overwritten rather than submitted if this command is issued as:

```
mail > program.c
```

The benefits of the application of automatic assessment should outweigh the additional challenges imposed on the students for having to deal with the system that applies the assessment.

## 3.6 Malicious programs

Hollingsworth [1960] observed as early as 1960 that it is possible for a student to submit a program that may cause deliberate damage. Security is an issue that has to be considered continually. Ala-Mutka [2005] points out that it is not rare that student programs contain bugs that may cause damage or hinder other processes. He emphasises the urgency to provide a secured running environment (sandbox) for running programs written by students without risks to the surrounding environment. The system of Cerioli and Cinelli [2008] verifies the authenticity of the instructor-provided code and executes the student code in a sandbox, with the lowest possible privileges. In our experience, it is far more likely that student programs are erroneous than deliberately malicious. Chen [2004] reports that in five years no real problems with malicious students have occurred.

It is likely that the occurrence of malicious code will remain low in traditional settings. In a MOOC, however, the system is made available to anyone in the world to use for its designed purpose. While using it for its designed purpose, the threat remains equally low. Unfortunately it can not be assumed that all users will use it for its designed purpose. It is likely that some users would attempt to utilise it to do harm or to gain unauthorised access to other applications within an institution's network. It is, therefore, strongly advised that MOOC designers rather be too cautious than running unnecessary high risks when applying automatic assessment of programming assignments.

## 3.7 Limited capability

Unfortunately there is a limit to what can be evaluated in assignments when using an automated process [Douce et al. 2005]. There is far more to a good program than correct functionality. Ideally a program should be evaluated for compliance with functional as well as quality requirements.

Evaluation of a program for compliance with functional requirements is usually done through the application of various forms of testing. Some forms of testing, such as unit testing, are easier to automate than others. As discussed in Section 2.3 and some issues mentioned in Section 2.6, testing a program even using simple input-output testing can be challenging. Testing the functionality of event-driven systems with graphical user interfaces and other kinds of interactive software components is more difficult and consequently more challenging to assess using automated procedures [Douce et al. 2005].

Some of the quality requirements can easily be assessed through static analysis of the source code. Automatic assessment systems like ASSYST [Jackson 2000] and Style++ [Ala-Mutka et al. 2004] use other criteria such as metric scores that correspond to complexity and style when assessing. Other aspects such as conforming to a particular coding standard, code efficiency, and compliance with sound object-oriented design principles are not as easily verifiable [Zimmerman et al. 2011]. Suleman [2008] points out that it would be extremely difficult to automatically check if meaningful variable names were used. Douce et al. [2005] remarks that subtle software quality metrics such as reusability and platform portability may be very difficult to quantify in this context.

Ideally, better results can be achieved through the combination of various techniques and application of a wider range of metrics. Vujošević-Janičić et al. [2012] give examples how certain program deficiencies can be detected by some assessment methods and not others. They argue that automatic assessment should combine functional testing, automated bug finding, as well as similarity measurement of the control flow graph of the student solution with an optimal solution. AutoGradeMe by Zimmerman et al. [2011] employs the same concept by integrating a number of Eclipse plugins to measure an array of aspects of the student submissions and calculate a weighted average to serve as the grade awarded to the program evaluated by the system.

The more comprehensive the assessment the better it would be. Traditional courses as well as MOOCs will benefit equally well by versatile assessments. In MOOCs, however, it may be more important because here there is no option to combine automatic assessment with hand-marking or compensate for weaknesses of the assessment with one-on-one instructor involvement.

## 4. OUR SYSTEM

My colleagues and I applied automatic assessment of assignments of elementary programs written by first-year students on an introductory course on imperative programming using C++. We used an in-house system called Fitchfork for this purpose. This section substantiates our choice to build our own system and describes its current features. This information serves as the backdrop to the experience report which follows in Section 5 and our reflection in Section 5.5.

### 4.1 Rationale to develop our own system

Despite the availability of many other automatic assessment tools that could be used for our purpose, we decided to develop our own for the following reasons:

—Most of the available systems support assessment of only java programs while we had a need to assess C++ programs. The design of our system is platform and language independent. We implemented a module for the assessment of C++ programs compiled on a Linux platform to suit our specific needs.

—In South Africa bandwith is still a cost factor that needs to be taken into account. To cut costs we prefer to use a system that is hosted on our own server and maintained by our own staff.

—We deem the opportunities for real world practical programming experience and research opportunities provided to our post-graduate students beneficial.

Fitchfork is currently not yet ready for public release. It is based on program testing. Like most other automatic assessment systems it is limited to testing the functionality of programs. Currently only input-output behaviour of elementary types is supported. Modules to support platforms other than Linux and programming languages other than C++ is still in development.

### 4.2 Submission

Students submit their assignments using a web interface on our departmental website. The upload slot for an assignment is automatically closed at the specified deadline.

Students are required to upload an archive containing all the files needed to compile and run their code. There is no restriction on the number of files or on the folder structure of the archive. This allows the system to, in theory, be able to assess fairly complex systems. In practice, however, the specification of complex systems might be too difficult to specify crisply enough for fair automatic assessment. In our course the most complex system required less than ten .cpp files, their .h files, a makefile and some data files.

### 4.3 Assessment process

Our system uses a process similar to the system used by Chen [2004] to evaluate the students' programs. The overall strategy is to run a program submitted by a student on a suite of test cases to evaluate the functionality of the program.

Figure 1 is an activity diagram showing the strategy applied by our system for each upload. The following sections discuss the aspects of the process in more detail.

### 4.4 Sandbox

The uploaded archive is extracted in a sandbox where it is compiled and executed. After the archive is extracted and before it is compiled the files in the sandbox may be altered according to specifications provided by the instructor. Table I describes the changes that are currently supported.

Table I. Supported file changes in the sandbox

| Change | Example |
|---|---|
| Delete | Delete specified files. This is typically used in our case to delete files that may have been included by a student to avoid recompilation of portions of their code; for example, we may specify that all executables and all files containing object code have to be deleted. |
| Add | Additional test data or libraries needed for more sophisticated testing can be added. |
| Override | Replace specified files. This can be used to replace some of the files that students have included with other versions of these files; for example, files that were given can be replaced by the original given files to ensure that they were not tampered with. |

### 4.5 Makefile

After the sandbox is set up, the program is compiled using the makefile that should be in the sandbox at this time. The instructor decides which makefile should be used. The options are to use a makefile provided by the instructor or to require that students write their own makefiles.

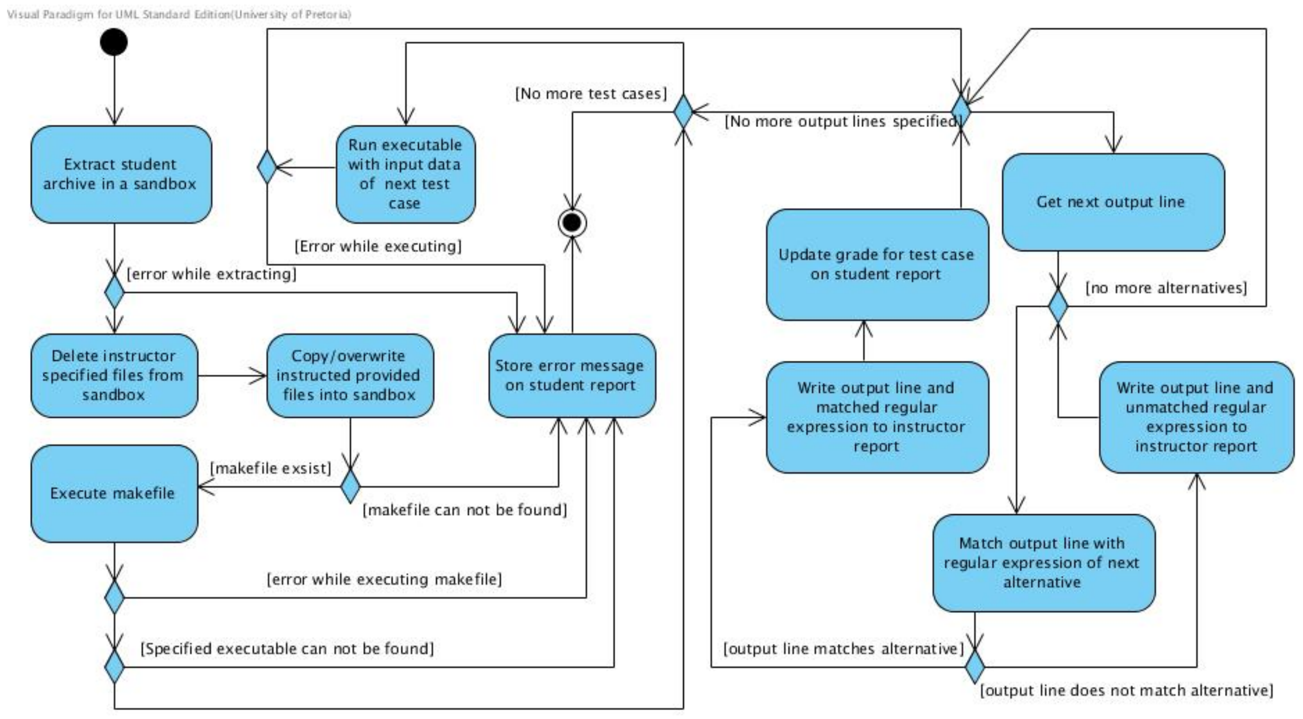
Visual Paradigm for UML Standard Edition(University of Pretoria)
Extract student archive in a sandbox
Run executable with input data of next test case
[No more test cases]
[No more output lines specified]
Get next output line
Update grade for test case on student report
[Error while executing]
[error while extracting]
[no more alternatives]
Delete instructor specified files from sandbox
Copy/overwrite instructed provided files into sandbox
Store error message on student report
Write output line and matched regular expression to instructor report
Write output line and unmatched regular expression to instructor report
Execute makefile
[makefile exsist]
[makefile can not be found]
Match output line with regular expression of next alternative
[error while executing makefile]
[Specified executable can not be found]
[output line matches alternative]
[output line does not match alternative]

Fig. 1. The assessment of a student submission

When a makefile is provided by the instructor, the instructor has the freedom to use the compiler flags of his own choice without the students being aware of his choice. The instructor can ensure that the required compiler flags are used and also that the created executable has the correct name. For it to work, the onus rests on the students to structure their solutions and to name all files needed by this makefile exactly as specified.

If, on the other hand, the students have to write their own makefiles, the students have more freedom in terms of the structure of their solutions and the naming of the files comprising their solutions. This freedom is, however, associated with the risk of having some errors in their makefiles or not complying with the specification of the file name of the executable. This file name is crucial owing to how our system is programmed. The instructor has to accept that the students are also free to use the compiler flags of their choice which may include undesirable ones like `-w` which disables compiler warnings.

## 4.6 Testing

The student's program is tested by executing the created executable in the sandbox with each of a number of specified test cases. The test suite used for the assessment is designed by the instructor. Detail of the test suite is specified in an XML file that is associated with the assignment.

Each test case is specified in terms of test input data and its expected output when given this input. The expected output is specified for each line of output the program should produce. For each line any number of alternatives can be specified. The marks that should be awarded for matching each of the alternatives are also specified using the appropriate XML tags recognised by our system.

When specifying the assessment rules, the instructor can choose whether to use exact matches or matching against regular expressions when comparing the student's output-line with an expected output-line. When using regular expressions, a certain degree of freedom in the output format can be allowed at the risk of inaccurate assessment. If the regular expression is too general, it is possible that wrong output may accidentally be identified as correct.

When performing the evaluation, our system considers the alternatives in the order in which they are specified. If a match is found the remaining alternatives are skipped. If all the alternatives have been considered without any match, no marks are awarded and the evaluation process continues with the next output-line.

The XML code in the listing in Figure 3 is an example of a specification for the second output-line of the assignment specified in Figure 2. It specifies a number of acceptable alternatives for the output-line assuming the input is 'm'. According to this specification four marks are awarded for the exact match with the sample output specified in the assignment. Four marks are awarded if the spacing between the words is different but the rest of the line matches exactly. Three marks are awarded if the correct ASCII is shown allowing for more lenience in terms of the wording in the output line. The next two alternatives each award one mark if the ASCII value is off by one. These alternatives are not as lenient as the previous alternative in terms of the rest of the text on this output-line.

Our system supports assessment of sub-fields within an output-line. To use this feature, the instructor has to specify the delimiter character that has to be used when splitting the output line into fields. Thereafter the specification for the evaluation of the fields is specified in the same manner that the evaluation of lines are specified.

Write a program that asks the user to enter a character and then displays the ASCII value of the character. The following is the output of a typical test run of the required program (User input is shown in bold):

```
Enter a character: A
The ASCII value of A is 65
```

Fig. 2. Example of an assignment

We acknowledge that one needs expertise in regular expressions, knowledge of XML and expertise in software testing to be able to set up the assessment criteria for assignments using our system.

### 4.7 Assessing quality

The only quality assessment currently directly supported by our system is to enforce an efficiency limit. If the execution time of a student program exceeds the specified limit, it times out and does not get assessed. This feature is handy when running programming contests. It is also needed to be able to kill a student program that goes into an endless loop.

It is, however, possible to cash in on feedback that can be provided by the compiler that is used. The GCC compiler [GCC Team 2012] can be directed via the use of some compiler flags to produce warnings on unused variables, implicit type conversions, and language features that are not following the language standards, amongst other things. This detail is currently simply echoed to the student feedback report and not incorporated in the marking scheme. In a future release of our system we plan to incorporate these in the assessment criteria and add the evaluation of other quality measures.

### 4.8 Reporting

While the program is assessed, two reports are generated. These are saved in the database and can be viewed at any time later by using an interface provided on the module web page. The one report is intended for the instructor while the other is the feedback given to the student whose program is assessed.

The report to the student includes all errors and output that may have been generated during the process of preparing the student upload for assessment. These include error messages during the extraction of the archive, error messages generated by the make utility as well as compiler errors and warnings like those mentioned in Section 4.7. It also contains the total mark awarded while assessing the program against the entire test suite, but no intermediate marks.

The report that can be viewed only by the instructor includes a trace of each output line of the student program and each alternative against which it was matched during the assessment process. This information is used to improve the specifications for the assessment for future use.

After the deadline of an assignment these reports are analysed to identify common errors that students have made. The result of this analysis is usually included in general feedback that is given to the class after the marks for the assignment are finalised. This information is also handy when students are individually coached. We prefer not to give the students general access to these reports because it reveals the exact test cases that have been used to evaluate the program. Knowledge about the test cases can result in students programming to adhere to the specific tests instead of programming to solve the general case.

## 5. OUR EXPERIENCE

This section is an experience report of the use of our own automatic assessment system described in Section 4. It supports our arguments that were presented in Sections 2 and 3 and informs our reflection presented in Section 5.5.

### 5.1 Student participation

The assignments that were given to the students each consisted of a number of tasks. The students had to submit their solution to each task separately. A task may be a complete program or a code snippet that has to be inserted in given scaffolding. In some cases consecutive tasks required further development of a partial solution required in the task preceding it.

Figure 4 shows the number of students that were registered for the module at the time each assignment was given, as well as the average number of uploads for the different tasks in each of the assignments. It also shows the average number of uploads per task that could not be assessed owing to one of the reasons mentioned in Table 0**??**, as well as the average number of uploads that was identified to be exact copies of previously submitted work.

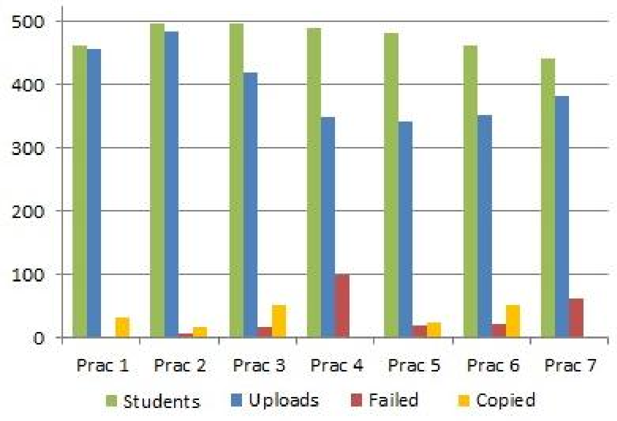

Fig. 4. Uploads, failures and copies

The number of submissions for the first two assignments almost equals the number of registered students at the time, but shows a sharp decline for the next two assignments. As assignments are progressively more challenging, it is expected that the number of students who completed them declined at the later assignments. It is likely that many students did not attempted these assignments. It is also likely that a number of students attempted them but did not upload them because they knew that they did not yet comply with the requirements at the time the assignments were due.

From the fourth assignment onwards the number of uploads seems to have stabilised. The number of uploads shows a slight inclination over these assignments. It is comforting to observe that it seems as if the students who managed to stay involved to the halfway mark (assignment 4), remained involved to the end.

None of the student uploads for the first assignment failed before they were assessed. This can be attributed to the amount of scaffolding that was given for this assignment. The purpose of the assignment was to give a gentle introduction to the system.

```
<line>
  <alt><exact mark='4'>The ASCII value of m is 109</exact></alt>
  <alt><regexp mark='4'>.*The\s+ASCII\s+value\s+of\s+m\s+is\s+109</regexp></alt>
  <alt><regexp mark='3'>.*(ASCII|ascii).*m.*109</regexp></alt>
  <alt><regexp mark='1'>.*The\s+ASCII\s+value\s+of\s+m\s+is\s+110</regexp></alt>
  <alt><regexp mark='1'>.*The\s+ASCII\s+value\s+of\s+m\s+is\s+108</regexp></alt>
</line>
```

Fig. 3. Example of a specification of expected alternatives for one output line

Apart from the large numbers of failed uploads in assignment 4 and assignment 7, the number of failed uploads shows a steady increase during the first three assignments and some flattening towards the end. This can be explained by the increasing difficulty of the assignments and is consistent with the overall upload pattern.

The larger numbers of failures in assignments 4 and 7 can be explained by the fact that these assignments were conducted under conditions where the students were required to complete the assignment within a limited time after the assignment was released. In addition they had to complete the asiignment in the lab during that time. The other assignments could be completed over a longer period during which the students could also work on the assignments at home. The students were also encouraged to upload their solutions for the fourth and seventh assignments even if their solutions were incomplete. These assignments were, in addition to the automated assessment, evaluated manually for compliance with coding standards. It is therefore likely that some students uploaded knowing that their programs do not work, but hoping to get marks for compliance with coding standards anyway.

The nature of the failed uploads is discussed in more detail in Section 5.2 while Section 5.4 describes how we identified and counted the number of copied uploads. This section includes an interpretation of the facts revealed by this analysis in more detail.

## 5.2 Failing before assessment

As can be seen in the diagram in Figure 1, a number of events can cause a student submission to fail even before it is assessed. These are the following:

—The submitted archive cannot be extracted because its format is not supported or it is corrupt.
—The submitted archive does not contain all the required files.
—There is no makefile in the sandbox that was prepared.
—The makefile contains some errors.
—The uploaded code does not compile.
—The program does not produce any output.

These kinds of failures can be avoided by implementing a strategy like the one offered by Zimmerman et al. [2011] at the cost of not supporting exercising program design. Some of these failures can be avoided by providing some of the scaffolding along with clear instructions on how to include the given scaffolding in the submission.

Analysis of the 820 failed uploads that occurred in the course of the semester is shown in Figure 5. It reveals that most of these can be attributed to compiler errors, which would anyway have occurred in an environment where students are not required to provide their own scaffolding. Even if marked manually, the functionality of programs that don't compile cannot be tested. The same applies to programs that compile and can be executed, but do not produce any output.

The missing files in most cases were test data files which were specified in the assignment as required items. They include 42 cases where the missing file is the makefile. These may also include cases where the required file was included but may have been misspelled.

The large number of makefile errors is alarming if taking into account that the systems that the students write in this module mostly consisted of a single source file, which requires a single-rule makefile. These makefile errors include 55 cases where the only error was misspelling the name that was specified for the executable.

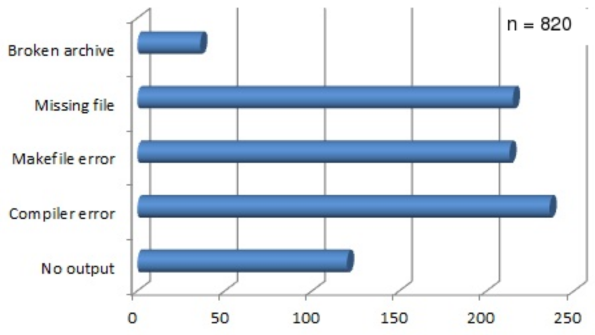

Fig. 5. Analysis of failed submissions

## 5.3 Failing owing to formatting

A serious drawback of automatic assessment mentioned by Vujošević-Janičić et al. [2012] and by Suleman [2008], is that if a student's program does not produce the correct output in the expected format, a system may fail to give an appropriate mark.

Analysis of the 679 uploads that produced output, but were awarded zero marks, is shown in Figure 7. As can be seen, 314 (45%) of these cases were functionally wrong, meaning that the assessment is likely to be fair.

| Expected | Actual |
|---|---|
| `Enter a value: 23`<br>`5 + 23 = 28` | `Enter a value:`<br>`23`<br>`5 + 23 = 28` |

Fig. 6. Misaligned output

The 225 uploads counted as misaligned had output of which the lines were out of sync with the memorandum and consequently were not awarded any marks because the actual output lines did not match the corresponding lines of the memorandum. Figure 6 shows an example of such misalignment.

Another unfortunate consequence of automatic assessment is the difficulty of allowing some freedom in the formatting. A total of 140 uploads were identified where students did not get marks owing to one or more of the following mistakes or similar errors:

—misspelling prescribed text.
—using more or less white space than specified.
—missing or adding some punctuation.

Only tasks that failed were analysed. It is likely that a substantial number of uploads passed but did not receive full marks owing to minor deviations in their formatting. These happened despite the use of regular expressions to specify the expected output. This illustrates the importance of taking extra care when thinking about the different acceptable ways the desired output may be formatted.

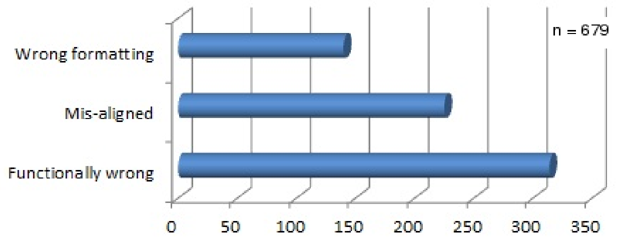

Fig. 7. Analysis of failed submissions

### 5.4 Plagiarism

We observed that some of our students would not hesitate to upload someone else's correct solution when their own seemed to be inadequate. A student who was found guilty of this practice said:

> *My program was exactly the same as his. I compared it and could not see the difference. Yet my program failed and his program got full marks. I think I deserve these marks. Fitchfork is wrong for not giving me my marks.*

It is likely that in this case the culprit had a misalignment or only a non-significant formatting difference which was unfortunately not one of the acceptable alternatives in the memorandum that was used during the assessment.

A script was written to calculate the MD5 hash for each archive that was uploaded. MD5 is believed to produce a unique 28-bit value for any given file. It is commonly used to check data integrity [Wikipedia 2012]. If two archives produce the same MD5 hash, it can be assumed that the content of all the files in those archives is identical in every aspect. When more than one copy of the same MD5 hash value occurred, the number of occurrences was counted and then the number of copies of archives found for each task of each assignment was summed. Figure 4 shows the average number of copies identified over the tasks of each assignment.

The high number of submissions of this kind of copied work is alarming. The absence of copies for assignments 4 and 7 can be explained by the fact that these assignments were conducted in examination conditions where the possibility to copy was eliminated. The higher occurrences of copying coincide with the tasks where the average marks of the students were the lowest. This confirms the observation by Ramzan et al. [2012] that difficulty in the assignment is one of the top reasons for plagiarism.

### 5.5 Reflection

Our experience showed that careless formulation of assessment criteria can result in unfair assessment, which in turn may give rise to increased tendencies among students to cheat. I realise that the quality of the automatic assessment needs to be improved to avoid unfair assessment and reduce cheating.

Out of a total of 11233 uploads, 820 failed before they could be assessed, 365 were awarded zero marks as a consequence of ill-formatting or misalignment. The rest of the uploads i.e 89.45% of all the uploads, were fairly marked according to the memoranda. Out of a total of 11233 uploads, 902 assignments were copied involving 330 students. This means that about 69.3% of the students were involved in the re-uploading of identical archives by other students. This is shown in Figure 8.

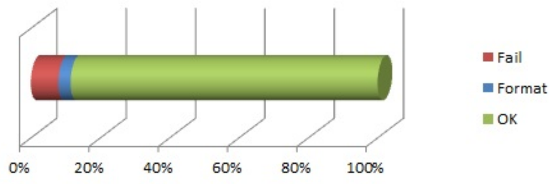

Fig. 8. Percentage of unfair marking owing to automation

The high number of student uploads that failed before they could be assessed motivated us to improve our system to allow instructors to provide more of the scaffolding needed by the automatic assessment system. Further refinement of our assignments and their corresponding memoranda may also contribute to reducing the occurrence of unfair assessment in the next presentation of this module.

As a result of the observation of high occurrence of copied work, we decided that we will submit all the uploads to MOSS [Aiken 1994] after the closing date of each assignment during the next presentation of this module. We have already implemented a script that will automatically submit the uploaded assignments and save the report in a format that we can easily use to follow up on the detected cases and hold the offending students accountable.

A further improvement that needs urgent consideration is refining the feedback reports to the students. It should be changed to show the marks students got for different test cases separately instead of only the total mark for a task as a whole. It can also be extended to support qualitative feedback which may be a textual hint about what is wrong with a specified alternative for which less than full marks was awarded.

We are aware that there is room for improvement in our system regarding assessment of the quality of the student submissions. Following the recommendation of Douce et al. [2005], we will investigate options to extend our system to include functions to assess the quality of the source code uploaded by our students through the use of applicable software metrics. It might be beneficial to automate these aspects that are currently still assessed manually.

## 6. RECOMMENDATION

Ideally, every programming course should aim to provide substantial opportunities for deliberate practise in the from of exercises. These exercises should be designed to reinforce the programming concepts taught in the course. They should also be accompanied by sufficient coaching to be effective.

The practical implications of such an undertaking for a MOOC is not trivial. The goal can only be achieved through the availability of a large number of high quality assignments coupled with coaching in the form of guaranteed rapid and accurate feedback. Automatic assessment of student programs is a promising option with which to address this problem.

Applying automated assessment is in itself not a solution. The quality of the automated assessment is important. Automated assessment can only add value to a MOOC if it is of high quality. Providing high quality automatic assessment can be very challenging and demands increased effort from the instructor.

To ensure the quality of automated assessment I recommend that the system that is used support the following functions:

—A secure environment where student programs can be executed. The system must safeguard against malicious programs. It has to be robust enough to withstand unintended damage caused by faulty student programs. It should also be secure enough to avoid the possibility that the system be utilised to harm the system or to gain unauthorised access anywhere.

—A versatile interface to specify the test cases and their expected solutions to be used when assessing a student submission. The quality of the assessment is extremely dependant on the quality of the test cases used in the system. The system should support the creativity of the instructor in the design of test cases.

—Options to provide scaffolding and support to enable automatic assessment. Ideally, students should not be required to master skills not related to the course they are taking merely to get their assignments assessed.

—An option to allow students to resubmit their work. Ideally, unlimited resubmission should be possible.

—Leniency in the way that student programs are evaluated regarding the format of their program results. Expecting the output of a student program to exactly match a specified output is not feasible. It should be easy to allow variations in formatting of output.

—The option to provide qualitative feedback; for example, if the student's program output is the expected result when a common error was made, the instructor should be able to provide a textual hint associated with the specific output that can be given to the student as to what is wrong with the solution.

—Options the evaluate the quality of student submissions. For example to be able to report on quality metrics such as the complexity of the solution or the degree of its compliance to specific coding standards.

—An option to show students statistical information about the uploads made by them. One can learn a great deal from knowing what common errors have occurred. Students can be motivated by knowing how they perform in relation to their peers.

Functions for plagiarism detection is a deliberate omission from the above list. Plagiarism is a pressing issue when automatic assessment is summative. The new learning model offered by MOOCs, where learning is no longer for credit and assessment is only formative, the urge to copy as well as the need to identify copying is obviated.

Automatic assessment has the potential to provide powerful support to MOOC participants. It should, however, not be seen as sufficient. I recommend that MOOCs offer additional support to their students in the form of comprehensive notes on how to use the automatic assessment tool in their learning. There should also be channels to get additional help, for example the availability of discussion forums where students can help one another.

The success of the use of automated assessment ultimately relies on *how* it is applied. MOOC designers are advised to choose an automatic assessment system that supports the functions they deem essential and to use these functions appropriately.

## REFERENCES

Tuukka Ahoniemi and Tommi Reinikainen. 2006. ALOHA-a grading tool for semi-automatic assessment of mass programming courses. In *Proceedings of the 6th Baltic Sea conference on Computing education research: Koli Calling 2006*. ACM, New York, NY, USA, 139–140.

Aleksi Ahtiainen, Sami Surakka, and Mikko Rahikainen. 2006. Plaggie: GNU-licensed source code plagiarism detection engine for Java exercises. In *Proceedings of the 6th Baltic Sea conference on Computing education research: Koli Calling 2006 (Baltic Sea '06)*. ACM, New York, NY, USA, 141–142. DOI:http://dx.doi.org/10.1145/1315803.1315831

Alex Aiken. 1994. MOSS: A System for Detecting Software Plagiarism. http://theory.stanford.edu/~aiken/moss/. (1994). [Online: accessed 12-January-2013].

Kirsti M Ala-Mutka. 2005. A Survey of Automated Assessment Approaches for Programming Assignments. *Computer Science Education* 15, 2 (2005), 83–102. DOI:http://dx.doi.org/10.1080/08993400500150747

Kirsti M Ala-Mutka, Toni Uimonen, and Hannu-Matti Järvinen. 2004. Supporting students in C++ programming courses with automatic program style assessment. *Journal of Information Technology Education* 3, 1 (2004), 245–262.

Yifat. Ben-David Kolikant and Maurice Mussai. 2008. "So my program doesn't run!" Definition, origins, and practical expressions of students' (mis)conceptions of correctness. *Computer Science Education* 18, 2 (2008), 135–151. DOI:http://dx.doi.org/10.1080/08993400802156400

Best Practical Solutions INC. 2012. RT: Request Tracker. (2012). [Online] accessed 2012-08-28.

Peter Brusilovsky and Sergey Sosnovsky. 2005. Individualized exercises for self-assessment of programming knowledge: An evaluation of QuizPACK. *Journal on Educational Resources in Computing (JERIC)* 5, 3, Article 6 (Sept. 2005), 22 pages. DOI:http://dx.doi.org/10.1145/1163405.1163411

David Carless, Diane Salter, Min Yang, and Joy Lam. 2011. Developing sustainable feedback practices. *Studies in Higher Education* 36, 4 (2011), 395–407. DOI:http://dx.doi.org/10.1080/03075071003642449

Maura Cerioli and Pierpaolo Cinelli. 2008. GRASP: Grading and Rating ASsistant Professor. In *Proceedings of the ACM-IFIP IEEIII 2008 Informatics Education Europe III Conference*. Venice, Italy, 37 – 51.

Peter. M. Chen. 2004. An automated feedback system for computer organization projects. *IEEE Transactions on Education* 47, 2 (may 2004), 232 – 240. DOI:http://dx.doi.org/10.1109/TE.2004.825220

Xin Chen, Brent Francia, Ming Li, Brian Mckinnon, and Amit Seker. 2004. Shared information and program plagiarism detection. *Information Theory, IEEE Transactions on* 50, 7 (july 2004), 1545 – 1551. DOI:http://dx.doi.org/10.1109/TIT.2004.830793

Geoffrey Colvin. 2006. What is takes to be great. http://money.cnn.com/magazines/fortune/fortune_archive/2006/10/30/8391794/index.htm. (October 19 2006).

Georgina Cosma and Mike Joy. 2012. An Approach to Source-Code Plagiarism Detection and Investigation Using Latent Semantic Analysis. *Computers, IEEE Transactions on* 61, 3 (march 2012), 379 –394. DOI:http://dx.doi.org/10.1109/TC.2011.223

John Daniel. 2012. Making sense of MOOCs: Musings in a maze of myth, paradox and possibility. *Journal of Interactive Media in Education* 3, Article 1 (2012), 1 pages.

Draylson Micael de Souza, Jose Carlos Maldonado, and Ellen Francine Barbosa. 2011. ProgTest: An environment for the submission and evaluation of programming assignments based on testing activities. In *Software Engineering Education and Training (CSEE T), 2011 24th IEEE-CS Conference on*. Institute of Electrical and Electronics Engineers, Piscataway, NJ, 1 –10. DOI:http://dx.doi.org/10.1109/CSEET.2011.5876088

Christopher Douce, David Livingstone, and James Orwell. 2005. Automatic test-based assessment of programming: A review. *Journal on Educational Resources in Computing (JERIC)* 5, 3 (2005), 4.

Lorna M Earl. 2003. *Assessment As Learning: Using Classroom Assessment to Maximize Student Learning*. Corwin, Thousand Oaks, CA.

Stephen H. Edwards. 2003. Improving student performance by evaluating how well students test their own programs. *Journal of Educ. Resour. Comput.* 3, 3, Article 1 (Sept. 2003), 24 pages. DOI:http://dx.doi.org/10.1145/1029994.1029995

Stephen H. Edwards and Manuel A. Perez-Quinones. 2008. Web-CAT: automatically grading programming assignments. In *Proceedings of the 13th annual conference on Innovation and technology in computer science education (ITiCSE '08)*. ACM, New York, NY, USA, 328–328. DOI:http://dx.doi.org/10.1145/1384271.1384371

Emma Enström, Gunnar Kreitz, Fredrik Niemela, Pehr Soderman, and Viggo Kann. 2011. Five Years with Kattis – Using an Automated Assessment System in Teaching. In *Frontiers in Education Conference (FIE), 2011*. Institute of Electrical and Electronics Engineers, Piscataway, NJ, T3J–1.

K. Anders Ericsson, Ralf Th. Krampe, and Clemens Tesch-Römer. 1993. The role of deliberate practice in the acquisition of expert performance. *Psychological review* 100, 3 (1993), 363.

Todd J. Feldman and Julie D. Zelenski. 1996. The quest for excellence in designing CS1/CS2 assignments. In *Proceedings of the twenty-seventh SIGCSE technical symposium on Computer science education (SIGCSE '96)*. ACM, New York, NY, USA, 319–323. DOI:http://dx.doi.org/10.1145/236452.236564

Judith Gal-Ezer, Tamar Vilner, and Ela Zur. 2009. The professor on your PC: a virtual CS1 course. *SIGCSE Bullitin* 41, 3 (July 2009), 191–195. DOI:http://dx.doi.org/10.1145/1595496.1562938

GCC Team. 2012. Website of GCC, the GNU Compiler Collection. http://gcc.gnu.org/. (2012). [Online: accessed 12-January-2013].

Jack Hollingsworth. 1960. Automatic graders for programming classes. *Commun. ACM* 3, 10 (1960), 528–529.

Jacqueline Hundley and Winard Britt. 2009. Engaging students in software development course projects. In *The Fifth Richard Tapia Celebration of Diversity in Computing Conference: Intellect, Initiatives, Insight, and Innovations (TAPIA '09)*. ACM, New York, NY, USA, 87–92. DOI:http://dx.doi.org/10.1145/1565799.1565820

Petri Ihantola, Tuukka Ahoniemi, Ville Karavirta, and Otto Seppälä. 2010. Review of recent systems for automatic assessment of programming assignments. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*. ACM, New York, NY, USA, 86–93.

David Jackson. 2000. A semi-automated approach to online assessment. *SIGCSE Bullitin* 32, 3 (July 2000), 164–167.

Mike Joy, Nathan Griffiths, and Russell Boyatt. 2005. The BOSS online submission and assessment system. *ACM Journal on Educational Resources in Computing* 5, 3, Article 2 (Sept. 2005), 28 pages. DOI:http://dx.doi.org/10.1145/1163405.1163407

Michael J. Prietula K. Anders Ericsson and Edward T. Cokely. 2007. The making of an expert. *Harvard business review* 85, 7/8 (2007), 114.

Romans Lukashenko, Vita Graudina, and Janis Grundspenkis. 2007. Computer-based plagiarism detection methods and tools: an overview. In *Proceedings of the 2007 international conference on Computer systems and technologies (CompSysTech '07)*. ACM, New York, NY, USA, Article 40, 6 pages. DOI:http://dx.doi.org/10.1145/1330598.1330642

Jenny Mackness, Sui Fai John Mak, and Roy Williams. 2010. The ideals and reality of participating in a MOOC. In *Proceedings of the 7th International Conference on Networked Learning 2010*. University of Lancaster, Lancaster, 266 – 275.

Lauri Malmi, Ari Korhonen, and Riku Saikkonen. 2002. Experiences in automatic assessment on mass courses and issues for designing virtual courses. *ACM SIGCSE Bulletin* 34, 3 (2002), 55–59.

Francisco J. Montoya-Dato, José Luis Fernández-Alemán, and Ginés García-Mateos. 2009. An experience on ada programming using on-line judging. In *Reliable Software Technologies - Ada-Europe 2009, 14th Ada-Europe International Conference, Brest, France, June 8-12, 2009. Proceedings (Ada-Europe)*. Springer, London, UK, 75 – 89.

Jonathan Y.H. Poon, Kazunari Sugiyama, Yee Fan Tan, and Min-Yen Kan. 2012. Instructor-centric source code plagiarism detection and plagiarism corpus. In *Proceedings of the 17th ACM annual conference on Innovation and technology in computer science education (ITiCSE '12)*. ACM, New York, NY, USA, 122–127. DOI:http://dx.doi.org/10.1145/2325296.2325328

Lutz Prechelt, Guido Malpohl, and Michael Philippsen. 2002. Finding Plagiarisms among a Set of Programs with JPlag. *Journal of Universal Computer Science* 8, 11 (2002), 1016 – 1038.

Muhammad Ramzan, MuhammadAsif Munir, Nadeem Siddique, and Muhammad Asif. 2012. Awareness about plagiarism amongst university students in Pakistan. *Higher Education* 64 (2012), 73 – 84. Issue 1. DOI:http://dx.doi.org/10.1007/s10734-011-9481-4

Jacques Philippe Sauvé, Osório Lopes Abath Neto, and Walfredo Cirne. 2006. EasyAccept: a tool to easily create, run and drive development with automated acceptance tests. In *Proceedings of the 2006 international workshop on Automation of software test (AST '06)*. ACM, New York, NY, USA, 111–117. DOI:http://dx.doi.org/10.1145/1138929.1138951

Hussein Suleman. 2008. Automatic marking with Sakai. In *Proceedings of the 2008 annual research conference of the South African Institute of Computer Scientists and Information Technologists on IT research in developing countries: riding the wave of technology (SAICSIT '08)*. ACM, New York, NY, USA, 229–236. DOI:http://dx.doi.org/10.1145/1456659.1456686

Kelvin Thompson. 2011. 7 Things you should know about MOOCs. http://net.educause.edu/ir/library/pdf/ELI7078.pdf. (November 2011). [Online; accessed 20 May 2013].

Brad Vander Zanden, David Anderson, Curtis Taylor, Will Davis, and Michael W. Berry. 2012. CodeAssessor: An Interactive, Web-Based Tool for Introductory Programming. *The Journal of Computing Sciences in Colleges* 28, 2 (2012), 73 – 80.

Tom Verhoeff. 2008. Programming Task Packages: Peach Exchange Format. *Olympiads in Informatics* 2 (2008), 192–207.

Arto Vihavainen, Matti Luukkainen, and Jaakko Kurhila. 2012. Multi-faceted support for MOOC in programming. In *Proceedings of the 13th annual conference on Information technology education (SIGITE '12)*. ACM, New York, NY, USA, 171–176. DOI:http://dx.doi.org/10.1145/2380552.2380603

Milena Vujošević-Janičić, Mladen Nikolić, Dušan Tošić, and Viktor Kuncak. 2012. Software Verification and Graph Similarity for Automated Evaluation of Students' Assignments. *Information and Software Technology* (2012). DOI:http://dx.doi.org/10.1016/j.infsof.2012.12.005 Online-first.

Neal R. Wagner. 2004. Plagiarism by student programmers. http://www.cs.utsa.edu/~wagner/pubs/plagiarism.html. (2004). [Online: accessed 12-January-2013].

Wikipedia. 2012. MD5 — Wikipedia, The Free Encyclopedia. http://en.wikipedia.org/w/index.php?title=MD5&oldid=530584383. (2012). [Online: accessed 12-January-2013].

Michael J. Wise. 1996. YAP3: improved detection of similarities in computer program and other texts. *SIGCSE Bullitin* 28, 1 (March 1996), 130 – 134. DOI:http://dx.doi.org/10.1145/236462.236525

Denise Woit and David Mason. 2003. Effectiveness of online assessment. In *Proceedings of the 34th SIGCSE technical symposium on Computer science education (SIGCSE '03)*. ACM, New York, NY, USA, 137 – 141. DOI:http://dx.doi.org/10.1145/611892.611952

Yuen Tak. Yu, Chung Keung Poon, and Marian Choy. 2006. Experiences with PASS: Developing and Using a Programming Assignment Assessment System. In *Quality Software, 2006. QSIC 2006. Sixth International Conference on*. Institute of Electrical and Electronics Engineers, Piscataway, NJ, 360–368. DOI:http://dx.doi.org/10.1109/QSIC.2006.28

Daniel M. Zimmerman, Joseph R. Kiniry, and Fintan Fairmichael. 2011. Toward instant gradeification. In *Software Engineering Education and Training (CSEE T), 2011 24th IEEE-CS Conference on*. Institute of Electrical and Electronics Engineers, Piscataway, NJ, 406 – 410. DOI:http://dx.doi.org/10.1109/CSEET.2011.5876114