MOOC and Mechanized Grading

Christian Queinnec
Sorbonne Universités, UPMC Univ Paris 06, UMR 7606, LIP6, F-75005 Paris, France

Keywords: MOOC, Mechanized Grading.

Abstract: As many others, we too are developping a Massive Online Open Course or MOOC. This MOOC will teach recursive programming to beginners and will heavily use an already existing infrastructure for mechanical grading (Queinnec, 2010). This position paper discusses how these two components are combined in order to increase students' involvement.

1 INTRODUCTION

Developping a MOOC is now a common activity in the Academia and so are we doing. This paper is a position paper that presents the main characteristics of our future MOOC: it makes an heavy use of an infrastructure to mechanically grade students' programs. How we intend to combine our MOOC with that infrastructure and how we want to create incentives for the students in order to increase their involvement is addressed in this position paper.

Programming exercises are proposed in between the videos of the course. These exercises ask for real programs from students. These programs are then mechanically graded and grading reports are sent back to the students. Reading grading reports allows students to evolve their programs but additional incentives may better help students. We propose to combine three means which should help students to progress by themselves.

- Students may program in pair: video-chatting around a common, shared program, that both students can edit, recreates conviviality. The underlying question is how to provide a good partner for a student who looks for one ?
- We also propose to students to peep slightly better programs from other students' and learn from them. The underlying question is how to select the most adequate programs to be peeped ?
- Finally, we ask student to recommend (or not) the peeped programs that were helpful, this ranking should help answering the previous item.

Section 2 presents the main lines of our MOOC and Section 3 presents the grading infrastructure while its new features appear in Section 4. Proposed incentives are described in Section 5 and Section 7 concludes this position paper.

2 PREPARATION OF A MOOC

We are currently developping a MOOC on recursive programming. The e-learning part is based on a course created in 2000 (Brygoo et al., 2002) and since then delivered every year at UPMC to hundreds of young scientific students as an introduction to Computer Science. The course material was, from 2000 to 2003, provided as a physical CDrom then, from 2004 to 2006, as a CDrom image (300MB) downloadable from UPMC web servers. An innovative characteristics of these CDroms was that they contained a programming environment (based on DrScheme (Felleisen et al., 1998)) with a local mechanized grader, see (Brygoo et al., 2002) for details. The students could then read the course documents, write programs, run them and be graded without requiring an Internet connection.

This new MOOC[1], adequately named "*Programmation récursive*", is an endeavour to extend this course to a broader French-speaking audience, to experiment with the aspects related to social networks and, finally, to collect and study students' answers to the proposed exercises in order to build appropriate error taxonomy and thus better future editions of the MOOC.

Of course, the context has dramatically evolved from 2000 to now. The mechanical grader of 2000

[1]http://programmation-recursive-1.appspot.com

Queinnec C..
MOOC and Mechanized Grading.
DOI: 10.5220/0004942102410245
In *Proceedings of the 6th International Conference on Computer Supported Education* (CSEDU-2014), pages 241-245
ISBN: 978-989-758-021-5
Copyright © 2014 SCITEPRESS (Science and Technology Publications, Lda.)

which was grading Scheme programs (Scheme is the programming language used by that course) has now evolved into a multi-language grading infrastructure running in the cloud (Queinnec, 2010). The MOOC uses CourseBuilder (Google, 2013), is hosted on Google App Engine for elasticity, uses YouTube for video streaming and a Google group for forum.

The programming environment, named MrScheme is now provided as a Scheme interpreter, written in Javascript by Frédéric Peschanski and his colleagues (Peschanki, 2013). This programming environment runs locally in students' browsers. With help of MrScheme, students must test their programs (an habit we enforce) before requesting their program to be graded thus saving servers' computing power. We require students to write programs satisfying a given specification but also to write their own tests for their own programs. The machinery don't accept to grade programs that fail their own tests.

3 GRADING INFRASTRUCTURE

The grading infrastructure is named FW4EX and described in (Queinnec, 2010). This is a cloud-based infrastructure controlled by REST protocols. Teachers uploads exercises that are tar-gzipped sets of files containing scripts to grade students' programs. Students submit their programs and receive a grading report in return. The infrastructure also offers additional services such as the whole history of their submissions and their associated grading reports.

The infrastructure was carefully defined to scale up, see Figure 1. Students submit (1) to some acquisition servers which act as queueing servers that are regularly polled (2) by some marking drivers. As their name implies it, marking drivers grade student's submission and send (3) the resulting grading reports to long term storage servers (Amazon S3 for instance). Finally storage servers are polled (4) by the waiting students. Student's browsers choose the acquisition server which in return tells where to fetch the resulting future grading report.

Marking drivers are isolated, they cannot be queried. Moreover they run inner virtual machines to run the grading scripts. This allows to confine (potentially malicious) students' and teachers' programs in memory, time, cpu, access to Internet, etc.

Marking drivers also record (5), when connected, the details of the performed grading into a centralized database. Acquisition and storage servers don't perform heavy computations but there are more than one in order to offer some redundancy to ensure continuity of service. The number of marking drivers is handled elastically that is proportionally to the number of submissions to grade. To grade a submission more than once is not a problem: this is the price to pay to absorb grading peaks.

In 2011, we added a new service, see (Queinnec, 2011), that tries to rank students according to their skills. To submit a program is considered as a move in a game that this student plays against all other students who try the same exercise. If a student gets a higher mark in fewer attempts then the student wins over all other students who got a lower mark or a similar mark but in more attempts. Using a ranking algorithm inspired from Glicko (Glickman, 1995) or TrueSkill (Graepel et al., 2007), it is then possible to rank students on a scale bounded by two virtual students:

- the best student succeeds every exercise with the right answer at first attempt
- and the worst student fails every exercise with one more attempt than the worst real observed student.

This approach can only rank students having tried a sufficient number of exercises in order to appreciate their skill.

4 GRADING PROCESS

When a student submits a program, this program contains functions and their associated tests, let's call them f_s and t_s. For instance, the next snippet shows a student's submission for an exercise asking for the perimeter of a rectangle:

```
(define (perimeter height width)
  (* 2 (+ height width)) )

(check perimeter
   (perimeter 1 1) => 4
   (perimeter 1 3) => 8 )
```

This snippet contains the definition of the `perimeter` function followed by a `check` clause (an extension we made to the Scheme language) checking `perimeter` on two different inputs. The `check` clause implements unit testing. With other languages, Java for instance, we use the JUnit framework instead, (Beck and Gamma, 2012).

The author of the exercise (a teacher) has also written a similar program that is, a function f_t and some tests t_t. The grading infrastructure uses an instrumented Scheme interpreter and executes the following steps:

1. $t_s(f_s)$ should be correct. Student's program that fail student's own tests are rejected. This ensures

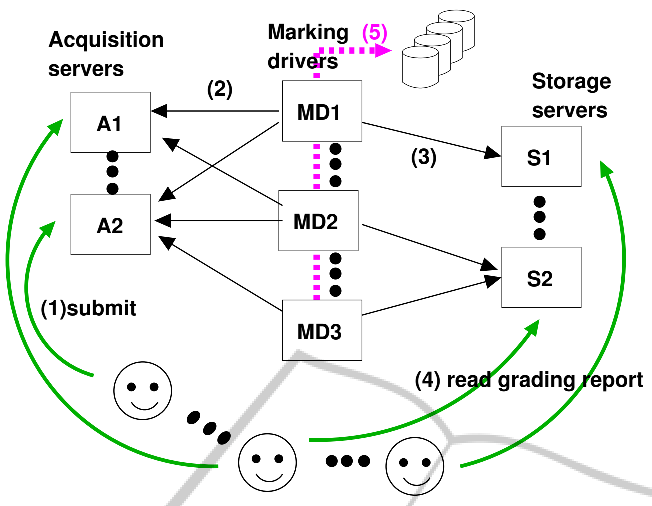

Figure 1: Architecture of FW4EX infrastructure.

that students don't forget to check their own code. We also check that the student does not obviously cheat i.e., t_s really calls function f_s.

2. $t_s(f_t)$ should be correct that is, student's tests should be related to the problem solved by f_t. The number of student's successful tests and the number of time the teacher's function has been called are used to provide a partial mark.
3. $t_t(f_s)$ should be correct that is, student's function should pass teacher's tests. The number of successfully passed tests also provides a partial mark.
4. The Scheme interpreter was instrumented in order to compute coverage profiles. Comparing the coverage of the student's tests with respect to teacher's test provides the last component of the final mark. Student's tests should at least execute all the code parts of their own code that are executed by teacher's tests. This again provides a partial mark.
5. All these partial marks are weighted and combined to form the final mark.

While steps 1, 2 and 3 were already present in the old CDroms, step 4 is new and measures the completeness of student's tests with respect to teacher's tests (which might be not perfect!).

Most exercises are graded in less than 15 seconds. The grading report returned to the student verbalizes what was submitted, which tests had been done and which kind of results were obtained with student's code compared to teacher's code. These reports are often lengthy but reading them carefully to understand the discrepancies develop students' debugging skill.

5 INCENTIVES

Equiped with such a grading machinery, we must offer incentives to the students so they may progress by themselves. Some incentives are currently under development and will be tested when the MOOC starts in 2014. The rest of the Section describes these incentives and the scientific challenges behind them which are not completely solved today.

5.1 Pair Programming

The first incentive is to provide an infrastructure for pair of students to work conjunctly on an exercise. This is pair programming as advocated by eXtreme Programming (Beck, 2000). A pairing server will be provided from which students will get a peer, the server will then provide a shared (Google) doc or equivalent and will let the two students work together and submit together. The server will choose a peer with roughly the same skill as determined by the ranking algorithm explained in Section 3. Of course, this might only work if a sufficient number of students in need of a peer are simultaneously present therefore, for every week of the MOOC, we intend to define peering periods. This feature will also require a widget in the shared doc to submit the joint work and see the resulting grading report. Accompanying this shared document with a Google hangout allowing to share voice and/or video will probably be attractive.

The quality of the peering process depends on the accuracy of the skill ranking. Conversely, to be able to appreciate students' skills allow the pairing server to also provide opponents with roughly the same skill:

the goal is then to propose, at the same time, the same exercise to the opponents and to compare their results in mark and time to reach that mark.

5.2 Epsilon-better Peeping

The second incentive is to propose to students having obtained a mark *m* and eager to progress, to peep two other students' submissions with slightly better marks i.e., $m+\varepsilon$. This is what we called "Epsilon-better peeping". This will favour reading other's code, another important skill worth stressing since beginners often think that they code for computers and not for humans! The slightly better mark may have been obtained by a better definition of the function or by better tests. Both allow to improve students' work. Reading these others' submission carefully and determine why they are better may be eye-opening.

To prevent students to just copy-paste better solutions, we will limit the number of times students may peep at other's submissions.

For students who stick to very low marks, we will probably have to set ε to a bigger value. If a huge number of students attend this MOOC, we may try various settings for this parameter to help students climb the first step.

5.3 Recommendation

After being served other's submissions, students will have to tell whether one of these other's submission was useful or not. This is a kind of recommendation system (or crowd ranking) from which the best helping submissions should emerge. However, differently from recommandation systems where a huge number of persons recommand a few items (movies for instance) here, we have a few students producing a huge number of submissions. Therefore to select the most appropriate submissions is a real challenge.

What we envision is to ask the teaching assistants to write a set of programs with increasing marks and, for the first edition of the MOOC, to favour these programs. This also solves the bootstrap problem since there must be other's submissions in order to implement this incentive.

Accumulating students' submissions should allow to elaborate a taxonomy of programs and errors. This taxonomy will help improving grading reports. Reports may include hints triggered by the kind of recognized error. The recommandation system that selects the best helping submissions, may also use that taxonomy. But this taxonomy will only be taken into account for the next edition of the MOOC.

6 RELATED WORK

Mechanical grading has been used for many years in many different contexts and programming languages. However generic architectures such as (Striewe et al., 2009), that support multiple languages, that are scalable and robust are not so common. Our infrastructure is one of them.

The estimation of students' skill is also a well studied domain (Heiner et al., 2004). Many works exist that try to characterize the student's model that is, its shape and its parameters (Jonsson et al., 2005) (Cen et al., 2006). They often start from an analysis relating exercises and the involved primitive skills then, they observe students' progress (mining the logs) in order to determine the parameters that best fit the model mainly with the "expectation maximization" technique (Ferguson, 2005).

The previous studies use far more information than us since they mine the logs of an intelligent tutor system where are recorded which exercise is delivered, how long the student read the stem, what help he requires, etc. By contrast, our grading infrastructure only gives us access to marks. Our set of proposed exercises is not (yet) related to the involved skills nor the set of skills is clearly stated. Therefore we are currently more interested to provide incentives to work in pairs with an attractive but rigorous feedback.

While recommendation systems are legion, to recommend the slightly better programs that helped to progress may be an interesting idea. We will see if our MOOC stands up to its promises.

7 FINAL REMARKS

In this paper, we present some ideas that are currently under development for a MOOC teaching recursive programming for beginners. This MOOC will start in March 2014 hence results are not yet known.

However and as far as we know, the conjunction of a grading machinery, a skill ranking algorithm and a recommendation system for help seems to be innovative and worth studying.

REFERENCES

Beck, K. (2000). eXtreme Programming. http://en.wikipedia.org/wiki/Extreme_programming.

Beck, K. and Gamma, E. (2012). The JUnit framework, v4.11. http://junit.org/.

Brygoo, A., Durand, T., Manoury, P., Queinnec, C., and Soria, M. (2002). Experiment around a training engine. In *IFIP WCC 2002 – World Computer Congress*, Montréal (Canada). IFIP.

Cen, H., Koedinger, K., and Junker, B. (2006). Learning factors analysis - a general method for cognitive model evaluation and improvement. In *Paper presented at the 8th International Conference on Intelligent Tutoring Systems*, pages 164–175.

Felleisen, M., Findler, R., Flatt, M., and Krishnamurthi, S. (1998). The DrScheme Project: An Overview. *SIGPLAN Notices*, 33(6):17–23.

Ferguson, K. (2005). Improving intelligent tutoring systems: Using expectation maximization to learn student skill levels.

Glickman, M. (1995). The Glicko system. Technical report, Boston University. http://glicko.net/glicko.doc/glicko.html.

Google (2013). CourseBuilder. https://code.google.com/p/course-builder/.

Graepel, T., Herbrich, R., and Minka, T. (2007). TrueSkill^{TM}: A bayesian skill rating system. Technical report, Microsoft. http://research microsoft.com/en-us/projects/trueskill/.

Heiner, C., Beck, J., and Mostow, J. (2004). Lessons on using its data to answer educational research questions. In *Workshop Proceedings of ITS-2004*, pages 1–9.

Jonsson, A., Johns, J., Mehranian, H., Arroyo, I., Woolf, B., Barto, A., Fisher, D., and Mahadevan, S. (2005). Evaluating the feasibility of learning student models from data. In *Proceedings of the Workshop on Educational Data Mining at AAAI-2005*, pages 1–6. MIT/AAAI Press.

Peschanki, F. (2013). MrScheme. http://github.com/fredokun/mrscheme.

Queinnec, C. (2010). An infrastructure for mechanised grading. In *CSEDU 2010 – Proceedings of the second International Conference on Computer Supported Education*, volume 2, pages 37–45, Valencia, Spain.

Queinnec, C. (2011). Ranking students with help of mechanized grading. See http://hal.archives-ouvertes.fr/hal-00671884/.

Striewe, M., Balz, M., and Goedicke, M. (2009). In Cordeiro, J. A. M., Shishkov, B., Verbraeck, A., and Helfert, M., editors, *CSEDU (2)*, pages 54–61. INSTICC Press.