MOOC and mechanized grading

Christian Queinnec

MOOC and mechanized grading

Christian Queinnec
UPMC LIP6
Christian.Queinnec@lip6.fr

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 for beginnners and makes an heavy use of an already existing infrastructure for mechanical grading (Queinnec, 2010). This paper presents some ideas on how to combine these two components along with some (untested) incentives for students in order to lessen teachers' time and increase students' involvement.

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 MOOC: it makes an heavy use of an infrastructure for the mechanized grading of students' program. 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 to lessen teachers'time is addressed in this paper. Section 1 presents the main lines of our MOOC and Section 2 presents the grading infrastructure while its new features appear in Section 3. Proposed incentives are described in Section 4 and Section 5 will conclude this position paper.

1 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 still delivered today at UPMC to hundreds of young scientific students as an introductory course on 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 related social aspects 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 which was grading Scheme programs (Scheme is the programming language used by that course) is now a multi-language grading infrastructure running in the cloud (Queinnec, 2010). The MOOC is hosted on Google App Engine for elasticity. makes use of YouTube for video streaming and a Google group for forum.

The programming environment, named MrScheme is now provided by a Scheme interpreter, written in Javascript by Frédéric Peschanski and his colleagues (Peschanki, 2013), and thus able to run locally in students' browsers. With help of MrScheme, students can 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. We don't grade programs that fail their own tests.

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

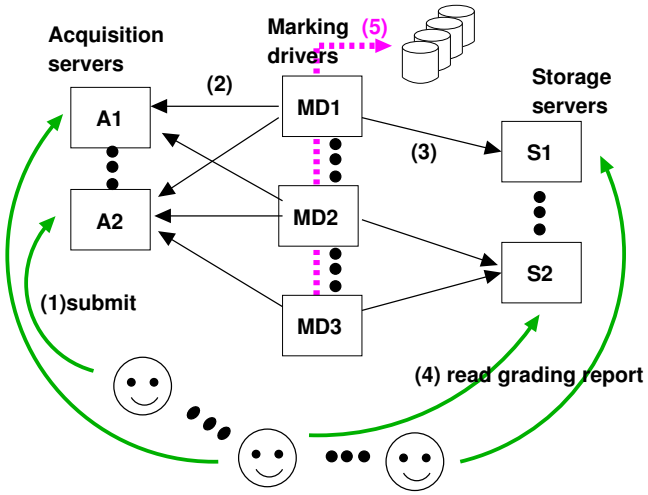

Figure 1: Architecture of FW4EX infrastructure

2 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 to some acquisition servers which act as queueing servers that are regularly polled by some marking drivers. As their name implies it, marking drivers grade student's submission and send the resulting grading reports to long term storage servers (Amazon S3 for instance). Storage servers are polled by the waiting students. Student's browsers choose the acquisition server which in return tells where to fetch the produced grading report.

Marking drivers also record into a centralized database, when connected, the details of the grading performed.

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 the 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 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.

3 GRADING

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 language, Java for instance, we use the JUnit framework instead, (Beck and Gamma, 2012).

The author of the exercise has also written a similar program that is, a function f_t and some tests t_t. The grading infrastructure makes use of an instrumented Scheme interpreter and execute the following steps:

1. $t_s(f_s)$ should be correct. Student's program that fail student's own tests are rejected. This ensures that students don't forget to check their own code. We also check that the student does not cheat i.e., t_s really calls function f_s.
2. $t_s(f_t)$ should be correct that is, student's test should be related to the problem. The number of student's 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 provide a partial mark.
4. The Scheme interpreter was instrumented in order to be able to compute coverage profiles. Comparing the coverage of the student's tests with respect to teacher's test provide the last component of the final mark. Student's tests should at least execute all the code parts of their own code that are used by teacher's tests. This again provide 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 test with respect to teacher's tests (which might be not perfect!).

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.

4 INCENTIVES

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

4.1 Pair programming

The first incentive is to provide an infrastructure for two students to work conjunctly on an exercise. This is pair programming as in the eXtreme Programming trend (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 2. 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 shared Google doc to submit the joint work and see the resulting grading report. Coupling writing this shared document with a Google hangout allowing to share voice and/or video will probably be attractive.

4.2 Epsilon-better peeping

The second incentive is to propose to students having obtained a mark m and eager to progress, two other students' submissions with slightly better marks $m+\varepsilon$. This will favour reading other's code, another important skill worth stressing since beginners often think that they code for computers! The slightly better mark may have been obtained by a better definition of the function or by better tests. Reading them carefully to 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 we provide other's submissions.

For students who stick to very low marks, we will probably have to set the 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.

4.3 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.

Accumulating students' submissions should allow to elaborate a taxonomy of programs and errors. This taxonomy will help improving grading reports: they may then include hints triggered by the kind of recognized error. The recommandation system to select best helping submission may also use that taxonomy.

But this taxonomy will only be taken into account for the next edition of the MOOC.

5 CONCLUSIONS

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 February 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.

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.

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

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

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/.

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. unpublished.