Mechanised grading: the next step

Christian Queinnec

To cite this version:

Christian Queinnec. Mechanised grading: the next step. 2009. hal-00391368

HAL Id: hal-00391368

https://hal.science/hal-00391368v1

Preprint submitted on 3 Jun 2009

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Mechanised grading: the next step

Christian Queinnec
Université Pierre et Marie Curie
LIP6, 4 place Jussieu, 75252 Paris Cedex 05
France – Email: Christian.Queinnec@upmc.fr

ABSTRACT

Mechanised grading is now mature: lots of experiments proved it. In this paper, we propose elements for the next step: standardisation.

We propose — an architecture for a grading component able to be embedded into various learning environments, — a set of extensible Internet-based protocols to interact with this component, and — a self-contained file format to thoroughly define an exercise in order to ease deployment. The architecture was designed to be scalable, robust and secure.

First experiments with a partial implementation of this platform show the versatility and the neutrality as well of the grading component. It does not impose an IDE, it respects the tenets of the embedding learning environment or course-based system.

1. HISTORY

Around 2000, with the help of some colleagues from UPMC, we set up two experimentations involving mechanised grading. In the first experiment (reported in [9]), we adjoined to the DrScheme programming environment (an Integrated Development Environment (or IDE) devoted to the Scheme programming language [13]) a mechanised grading facility: the student chooses an exercise, reads the stem, types then debugs his program and finally hits the "Check" button to obtain a mark and a page explaining the tests that were performed and led to that mark. This mechanised grader is a stand-alone plug-in to DrScheme; every year since 2001, around 700 students have had used it on their home computer (at every moment) or during lab sessions.

In the second experiment (reported in [19]), we organised general programming examinations or contests where we had several hundreds of undergraduate students to grade in a short time. Centralised mechanised grading was our sole option. Therefore we wrote unit tests to check students' programs considered as black boxes. Soon after-that, we designed a framework and implemented some libraries to ease the production of mechanised graders that is, specialised programs that grade students' files for a given exercise. From 2001, we have been grading several hundred examinations per year.

Around 2005, our various experiences with different programming languages such as Scheme, C, Ada, php, Perl, Java, make or shell convince us that mechanised grading was mature enough for courses that mainly rely on some programming language(s) both for exercises and/or examinations. We proposed to unify our previous experiments into a generalised framework and a multi-language grading architecture to ease the production of graders and to lessen the associated chores. Our goal was to build

1. an autonomous and scalable grading component that may be embedded within various systems,
2. REST-based protocols to interact with this component,
3. a file format for self-contained deployable exercises.

We will first present the advantages of mechanised grading and some of our goals. The second section will present an overview of the framework, nicknamed FW4EX, and the prominent features. In the third section, we will summarise the results of our lattest experiments using the FW4EX platform. Related works is addressed in the fourth section.

2. MECHANISED GRADING

Among the many ways that exist to grade students, quizzes are probably the most usual. Every virtual leaning environment such as Blackboard [1] or Sakai [2], proposes tools to build assessments with questions such as true/false questions or multiple choice questions. Even if more elaborate quizzes exist [3], quiz technology is poor with respect to programming languages.

Writing a program is a complex activity requiring *(i)* to understand a specification, *(ii)* to imagine a solution, *(iii)* to write it down in some programming language and *(iv)* finally to test it to make sure it complies with the specification. Any problem that arises during this life-cycle forces the student to re-iterate parts of this cycle. A way to assess this skill is to let students program with professional tools and to check programmatically whether their programs comply with the specification. Professionally, this is called "unit testing" and popular frameworks such as JUnit [4] were developed for that activity. To use professional tools and to obey professional rules was something we wanted our students to be exposed to.

There are some differences though with usual unit testing.

- Unit testing is designed to give a binary answer: 1 (pass) or 0 (fail). In a grading context, we want to give a more representative and accurate mark for the student's work say a mark between 0 and 20 (as usual in French universities). This may be done by multiplying the tests and summing their partial results.

- Unit testing frameworks often depend on one programming language. However some specifications leave the choice of the programming language to use up to the student. Therefore, a grading framework has to cope with multiple languages.

To design mechanised graders is a hot topic as illustrated by the numerous papers presented at ITiCSE these last years like Quiver [11], RoboProf [10], TorqueMOODa [16], gradem [18], APOGEE [14] or ALOHA [8]. Among well known systems containing graders are CourseMaker [15] and BOSS [17].

Mechanised graders offer multiple advantages:

- consistency: students are graded uniformly (and anonymously) without the biases due to (multiple) human grader(s).

- fairness: if one student finds a problem in the specification or the associated tests then all students benefit from the upgraded grader.

- persistence: once created, assessments are forever available for students who may practise them whenever they want, in the same conditions the assessment was initially offered. We dubbed this "dynamic annals" [19].

Therefore a grading component must be able to grade exercises in various programming languages, various settings: complete programs (stand-alone, client or/and server), program fragments (function, method, class). A grading component must be able to be put to work as part of a learning environment or other systems: it must not dictate whether exercises are summative or formative, it must be as independent as possible from scholar databases.

Of course, it must also be simple to use, robust in face of crashes, secure (respect privacy and anonymity, resistant to pirates and malicious student's works). Finally, our dreamt grading component ought to be scalable in order take benefit of new technologies such as cloud computing (Amazon EC2 for instance) or Internet-based storage (Amazon S3 for instance) if demand for grading should increase.

3. THE FW4EX PLATFORM

The FW4EX framework takes benefit of our previous experiments and tries to address a variety of use cases. In this Section, we present the main lines of the architecture, the protocols and the format of exercises.

Figure 1 shows a simplified sketch of the architecture of the FW4EX platform. This architecture supposes the presence of Internet and heavily relies on REST-based services [12].

From the student's point of view, after choosing (or be assigned) an exercise, his browser (or any FW4EX-compliant client, see Section 3.4) fetches a zipped file (from an exercises server e) containing the stem and all the necessary files or documents required to practise the exercise. When ready,

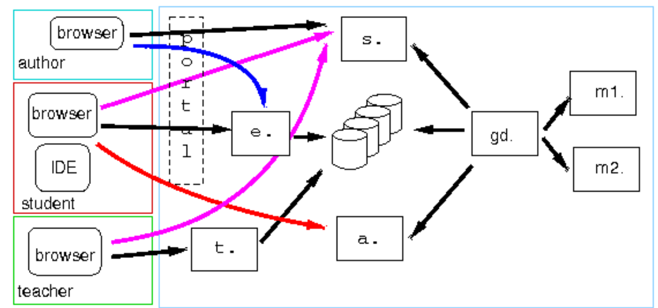

Figure 1: sketch of the FW4EX platform. The arrow tells which machine initiates connections.

the student submits his work (to an acquisition server a) where it will be picked by a grading server gd that will instantiate an appropriate grading slave (m_1 or m_2) to grade the student's files. Once graded, a final report (an XML file) is made available to the student (on some storage server s) as well as its initial submitted work.

Acquisition Servers a are stand-alone servers that must be as robust as possible since they must be always available to accept students' works. To increase their robustness, they do not depend on the presence of the central database. Storage Servers s only serve reports i.e., static files and therefore are robust: students may then get feedback whenever they want. To preserve anonymity, reports may only be fetched via non guessable md5-based urls.

Teachers prescribe sets of exercises to students and may access their resulting reports. A teacher may also collect students' works (jobs) by any convenient mean and submit a batch of jobs for grading. Teachers have (authenticated) access to grades and names through the t server.

The author of an exercise also has access to the anonymous reports produced by his exercise in order to improve it.

3.1 Exercise format

An exercise is a tar gzipped file (not dissimilar to a Java `jar` file) containing an XML descriptor and all the files required to operate the exercise. This central descriptor rules the various aspects of the life-cycle of the exercise, successively: autocheck, publish, download, install, grade. These phases are shortly explicited hereafter in increasing complexity order.

download The descriptor specifies the stem and the files required by the student in order to practise the exercise. The descriptor also describes for each question, the expected work and may contain additional hints such as a template to fill for an answer or the size of the expected work. In the case of an exercise with a single question expecting a single file, an FW4EX-compliant client (see Section 3.4) may use these hints to display an appropriate widget to capture interactively an answer on one or multiple lines.

grade The descriptor defines which scripts (written in whatever scripting language) are used to grade student's files within a grading slave and how these scripts may be combined.

install The descriptor defines how an exercise is installed

on a grading slave (usually a virtual machine (QEMU, VMware)) or on a student's computer. This installation may require to uncompress data files, to compile libraries, etc. The installation on a grading slave is different from the installation on the student's computer since grading scripts are not disclosed to the student nor they are communicated to the student's computer.

autocheck The descriptor defines a number of "pseudo jobs" paired with the expected grade they should get. This allows — to check whether the grading slave contains all the software required to grade a specific exercise, — to ensure non regression when an author evolves an exercise.

Most often it is advised to have at least three pseudo jobs: the null one that contains nothing and expects to obtain a 0/20 grade. The perfect one that should be graded 20/20 and an intermediate one that checks that in-between grades are indeed possible!

publish When installed and autochecked, an exercise is ready to grade real students works. It is then publicly available.

The descriptor also defines some meta-data to characterise the exercise: name, requirements, summary, tags, etc. These are needed by the *e* server to inform students and teachers about the exercises that are ready to be selected.

The XML descriptor is the file that ties all the information required to operate the exercise through all the steps of its life-cycle. Stems, reports and other information coming out of the grading platform are XML documents (JSON and CSV are also possible) therefore, they are skinnable via XSLT style sheets to fit university look and feel. They may also be processed to fill databases or any other tool that needs information about grading.

3.2 Confinement

Our experience shows us that students make mistakes (infinite loops, deadlocks, etc.). Their programs need to be confined in time and cpu consumption. Their production must also be limited: not much than a given number of bytes written to files or on output streams. Though rare, some students' programs are malicious and must be tightly restricted (are they allowed to post mails, to open sockets, to destroy resources to prevent grading other student's files, etc.) For all these reasons, we run students' programs under students' accounts (so they do not harm teachers' files) within virtual machines (QEMU or VMware) and pay much attention to offer the same initial conditions for all jobs.

Our experience also shows us that authors make mistakes. Their programs need to be similarly confined otherwise they may bog down servers, destroy common resources and harm student's experience. To grade an exercise is therefore a complex task where student's errors should appear in student's report, author's errors should appear in author's reports and FW4EX errors should be routed to FW4EX maintainers.

3.3 Grading script

Scripts are regular programs written in whatever language fits the task. They are run within the directory where student's files are deployed, they may read or run author's owned files (stored elsewhere) as well as read or run FW4EX libraries (stored elsewhere).

Grading may be performed by comparison or by verification. The general (suggested) shape of a grading script is the following:

1. **Check expectations** — The framework verifies that the files expected from the student are present, non empty, executable, etc. as specified in the exercise descriptor.
2. **Feedback** — Make student's files appear in the grading report. This is useful for the student since he may check that these are really the files he submitted. It is also useful for the author of the exercise since the report is self-contained.
3. **Loop** — For all test cases:
   (a) **Setup** — Set up the test that is: populate then jump to a directory, uncompress some data files, etc. Depending on the author of the exercise, the test may be explicited or kept secret.
   (b) **RunStudent** — Run the student's programs
   (c) **ShowStudentAnswer** — Show the results of the student's programs. The result may be additionally formated in order to improve its appearance (use multiple columns, plot the result, split lines, etc.)
   (d) **NormalizeStudentAnswer** — Normalise the results, for instance: remove superfluous spaces, apply regular expressions, etc.
   (e) **Gauge** — Gauge the normalised result (see below).
   (f) **EvaluateGain** — Determine the final grade for this test case. This phase may be disseminated through the previous phases but it is cleaner to separate these phases so the ponderation may evolve independently.

When comparing to a known correct solution, the **Gauge** phase is often done as follows:

1. **RunAuthor** – Run the author's programs
2. **ShowAuthorAnswer** — Show the results of the author's program.
3. **NormalizeAuthorAnswer** — Normalise the results as before. These results may be computed ahead of time in the install phase of the exercise.
4. **Compare** — compare the student's and author's results. The comparison may compare integers, exit codes, small strings (with a Levenshtein distance) or files (with the `diff` utility), etc. The comparison may also be approximate using regular expressions.

Verification is specific to the exercise and must check that some property is obtained or not. As a side-remark, the author must pay attention to the fact that in response to a yes/no property, it is too easy for students to come with a program that constantly answer 'yes' and pass half of the tests.

For all these phases, libraries were developed to ease writing these grading scripts.

3.4 FW4EX-compliant clients

We strongly believe that programming should be done with professional tools such as usual programming environment (IDE). Browsers, applets cannot be considered as decent substitutes for real IDE. An FW4EX-compliant client is an IDE (often an IDE plug-in) able to speak to the three types of public servers (e, a and s) according to some REST protocols [12].

These protocols allow a student to authenticate, to obtain stem, to submit work and to, finally, get a report. When submitting a job and following REST principles, the client receives the URL where the report will pop up on some s server.

From an author's point of view, it is similarly possible to submit an exercise, get the corresponding autocheck report leading to the grading reports of the pseudo-jobs. Additional services exist for FW4EX maintainers to manage jobs, exercises, configuration.

Presently we only offer an AJAX FW4EX-compliant client running in any regular browser. It fulfills all of the needs but for the installation of the exercise on the student's computer which cannot be fully automated for security reason. It manages the display to accommodate various types of exercises (see experiments in Section 4): one-liners are captured via a one-line text widget, short scripts are captured via a textarea widget, multiple files are captured via a file upload widget.

Since it mostly handles XML documents, the AJAX FW4EX-compliant client imposes its own XSLT style sheets to display them with the wished look and feel. Additional treatments may also be performed such as using a tree widget to holophrast series of lengthy tests in the grading report. The report is therefore largely independent of the display technology.

The AJAX FW4EX-compliant client may be embedded in a university portal.

We plan to provide two more clients: one for Eclipse (for Java, C, php, etc.) and one for Scite [5]. In these IDE, similarly to what we did for DrScheme [9], a single button or menu will manage authentication, show stem, install accompanying files, pack student's work, submit it and display the associated report possibly, if available, as annotations to source code.

4. EXPERIMENTS WITH FW4EX

We deployed the FW4EX platform in September 2008. Two experiments were conducted in the first semester, a third one will take place at the end of January 2009. The first two experiments will be reconducted for the next semester with different modalities.

4.1 One-liners

For a course on Posix skill (mainly shell and Makefile) we deploy approximately 40 exercises. These exercises are "one liners" since they contain a single question that asks for options (for utilities such as `tr`, `sort`, `sed`, etc.) or for whole commands for various tasks (sifting, sorting, regexping, etc.). Stems are terse (but always give an example of use) in order to let students think about limit cases such as empty streams, empty files, files out of the current directory, files with weird names, etc. The goal of these exercises was to encourage students to read the associated man pages in order to be familiar with the 2 to 4 most useful options with each of these utilities.

These exercises are permanently available for all students, they are not limited in number of submissions. Reports are very detailed, every test is explicited, results are shown and reasons for success or failure are verbalised. Students have to analyse these reports and sometimes discover that some limit cases are indeed possible with respect to the stem. In the actual deployment, grading reports are obtained after 20 seconds. Such a duration compels students to consider that the grader is not a fast debugging tool so debug should be done appropriately on their computer before asking to be graded.

We offer roughly 5 exercises per week for 7 weeks. We start from exercises asking for options and end with exercises asking for small shell scripts (less than 10 lines). On the 120 students enrolled for the course, half of them tried at least one exercise. The platform had been serving 3500 submissions.

The exercises were not prescribed, only voluntary students try them. As already reported in [20], students did not volunteer easily. Apart the initial curiosity of the first week, we noted the usual peak during the week that precede examinations.

Next semester, these one-liners will become mandatory, the resulting marks will be taken into consideration.

4.2 Examinations

In the same course, we use the FW4EX platform to grade the mid-term examination and the two final examinations. The examinations contain three exercises each containing 1 to 6 rather independent questions. Contrarily to the one-liners, examinations are very carefully specified. Stems are precise, they ask for scripts (or Makefile rules) performing various tasks. Examples are always given, deliverables are specified and some hints are given about how grading will be performed.

These examinations last 3 hours. As usual the mid-term examination mainly serves to prepare the students for the final examinations. The student's works are sent to the grader after the examination therefore students do no have any feedback during the examination.

In this setting, we had 120 sets of students' files to grade. The exercise file (containing the examination) was prepared with 6 pseudo jobs. We grade the whole set of students' files 4 times in order to tune the grader: we had to cope with misnamed scripts or files, to fix an encoding problem (UTF8 versus Latin1) and to slightly alter the weight of two questions. Approximately 60 seconds were necessary to grade one student.

The mid-term examination was also made available as a regular (although huge) exercise provided by the platform so students may (re-)try it in a less stressed context. Only 6 students retried it, 1 among them reach a grade greater than 19/20 in 4 attempts, the other topped at 6/20 with 1 to 6 attempts.

Final examinations were similarly proposed and graded. We tried to correlate the success to the final examination with the overall usage of the FW4EX platform (see Figure 2). We only consider students having attempted at least 5 exercises. The best found correlation was between the final mark and the ratio of completed versus attempted exercises.

The examination was held on the computers of the laboratories where no plagiarism was possible. Plagiarism detection will be considered as an option for batch grading.

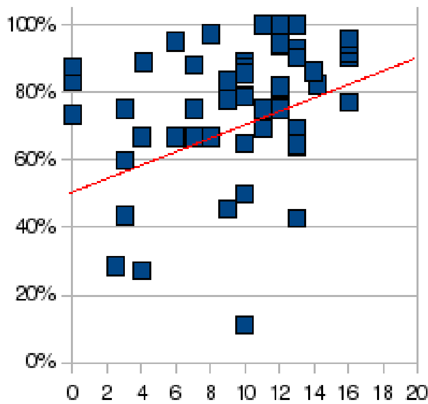

Figure 2: Percentage of completed/attempted exercises with respect to the final mark (between 0 and 20). Only students having attempted at least 5 exercises are considered. The line is the principal axis of the cloud of points.

4.3 Programming contest

The "Journée Francilienne de Programmation" is a programming contest for undergraduate students from various Parisian universities. The contest lasts 5 hours during which 12 teams of voluntary students regularly submit their work (120 uploads) and get their grade (the average grading time was 45 seconds) in order to appreciate where they are with respect to the other teams. Besides individual grading, additional scripts use the reports from FW4EX to deliver bonus points to teams' works (mainly based on the speed of programs).

5. RELATED WORK

While there are many mechanised graders on the market, we only compare FW4EX to CourseMaker [15] and BOSS [17].

The BOSS Online Submission System [6] is a course management tool. It allows students to submit assignments online securely, and contains a selection of tools to allow staff to mark assignments online and to manage their modules efficiently. Plagiarism detection software is included. Interaction with BOSS may use an application or a web front end.

CourseMaker [7] is a web-based, easy-to-use course creation package commercialised by Connect. It is now roughly similar in functionalities to Blackboard or Sakai. Like BOSS, it contains a number of tools – to check typography, syntax, comments, – and to detect plagiarism.

These systems are complete solutions that contain courses documents, exchange information with scholar databases, deploy student clients application or web-based forms to collect work. Contrarily, FW4EX is only an Internet-accessible grader that offers some libraries and runs teachers scripts. FW4EX is helped with a small constellation of specialised servers to take into account scalability.

Strong points of FW4EX are its focus on grading: — confinement to ensure security is tantamount and is not to be improvised — accessibility via Internet easing delegation of grading by learning environments and freeing the platform from any programming language dependency. The exercise self-contained file format allows a very simple deployment. We propose these protocols and format as steps towards standardisation.

6. CONCLUSION

In this paper, we present the FW4EX platform, an attempt to build a grading component that can be embedded into various learning environments. A set of REST-based protocols allows these systems to operate the grader, a "standard" descriptor and a file format specification are proposed to ease the deployment of new exercises. We finally described the different experiments that show the neutrality and versatility of this platform: one-liners, full examinations, programming contests.

For the future, our next work is to develop an authoring IDE (based on Eclipse) to help authors design, test, deploy exercises onto the platform.

More information on the FW4EX project is available on the site `paracamplus.org`.

7. REFERENCES

[1] www.blackboard.com.
[2] www.sakaiproject.org.
[3] www.gradiance.com/idea.html.
[4] www.junit.org.
[5] www.scintilla.org.
[6] http://sourceforge.net/projects/cobalt/.
[7] www.coursemaker.co.uk.
[8] T. Ahoniemi, E. Lahtinen, and T. Reinikainen. Improving pedagogical feedback and objective grading. In *SIGCSE '08: Proceedings of the 39th SIGCSE technical symposium on Computer science education*, pages 72–76, New York, NY, USA, 2008. ACM.
[9] A. Brygoo, T. Durand, P. Manoury, C. Queinnec, and M. Soria. Experiment around a training engine. In *IFIP WCC 2002 – World Computer Congress*, Montréal (Canada), Aug. 2002. IFIP.
[10] C. Daly and J. Waldron. Assessing the assessment of programming ability. In *SIGCSE '04: Proceedings of the 35th SIGCSE technical symposium on Computer science education*, pages 210–213, New York, NY, USA, 2004. ACM.
[11] C. C. Ellsworth, J. James B. Fenwick, and B. L. Kurtz. The quiver system. In *SIGCSE '04: Proceedings of the 35th SIGCSE technical symposium on Computer science education*, pages 205–209, New York, NY, USA, 2004. ACM.
[12] R. T. Fielding and R. N. Taylor. Principled design of the modern web architecture. *ACM Trans. Interet Technol.*, 2(2):115–150, 2002.
[13] R. B. Findler, J. Clements, C. Flanagan, M. Flatt, S. Krishnamurthi, P. Steckler, and M. Felleisen. Drscheme: A programming environment for scheme. *Journal of Functional Programming*, 12:369–388, 2002.
[14] X. Fu, B. Peltsverger, K. Qian, L. Tao, and J. Liu. Apogee: automated project grading and instant feedback system for web based computing. In *SIGCSE '08: Proceedings of the 39th SIGCSE technical symposium on Computer science education*, pages 77–81, New York, NY, USA, 2008. ACM.
[15] C. A. Higgins, G. Gray, P. Symeonidis, and A. Tsintsifas. Automated assessment and experiences of teaching programming. *J. Educ. Resour. Comput.*, 5(3):5, 2005.
[16] C. Hill, B. M. Slator, and L. M. Daniels. The grader in programmingland. In *SIGCSE '05: Proceedings of the 36th SIGCSE technical symposium on Computer science education*, pages 211–215, New York, NY, USA, 2005. ACM.

[17] M. Joy, N. Griffiths, and R. Boyatt. The boss online submission and assessment system. *J. Educ. Resour. Comput.*, 5(3):2, 2005.

[18] R. E. Noonan. The back end of a grading system. In *SIGCSE '06: Proceedings of the 37th SIGCSE technical symposium on Computer science education*, pages 56–60, New York, NY, USA, 2006. ACM.

[19] C. Queinnec and E. Chailloux. Une expérience de notation en masse. In *TICE 2002 – Technologies de l'Information et de la Communication dans les Enseignements d'Ingénieurs et dans l'industrie – Conférences ateliers*, pages 403–404, Lyon (France), Nov. 2002. Institut National des Sciences Appliquées de Lyon. version complète disponible en `http://lip6.fr/Christian.Queinnec/PDF/cfsreport.pdf`.

[20] D. Woit and D. Mason. Effectiveness of online assessment. In *SIGCSE '03: Proceedings of the 34th SIGCSE technical symposium on Computer science education*, pages 137–141, New York, NY, USA, 2003. ACM.