Architecture to Support Automatic Grading Processes in Programming Teaching

Caiza J. C.*; Del Alamo J. M.**

**Escuela Politécnica Nacional, Departamento de Telecomunicaciones y Redes de Información, Quito, Ecuador*
e-mail: julio.caiza@epn.edu.ec
*** Universidad Politécnica de Madrid, Departamento de Ingeniería de Sistemas Telemáticos, Madrid, Spain*
e-mail: jmdela@dit.upm.es

Resumen: La calificación automática de tareas de programación es un tema importante dentro del campo de la innovación educativa que se enfoca en mejorar las habilidades de programación de los estudiantes y en optimizar el tiempo que el profesorado dedica a ello. Uno de los principales problemas vigentes está relacionado con la diversidad de criterios para calificar las tareas de programación. El presente trabajo propone e implementa una arquitectura, basada en el concepto de orquestación de servicios, para soportar varios procesos de calificación automática de tareas de programación. Esto es obtenido a través de las características de modularidad, extensibilidad y flexibilidad que la arquitectura provee al proceso de calificación. La arquitectura define como pieza clave un elemento llamado Grading-submodule, el mismo que provee un servicio de evaluación del código fuente considerando un criterio de calificación. La implementación se ha llevado a cabo sobre la herramienta Virtual Programming Lab; y los resultados demuestran la factibilidad de realización, y la utilidad tanto para el profesorado como para los estudiantes.

Palabras clave: evaluación de tareas de programación, proceso de calificación automática, arquitectura.

Abstract: Automatic grading of programming assignments is an important topic in academic research. It aims at improving students' programming skills and optimizing the time of teaching staff. One important gap is related to the diversity of criteria to grade programming assignments. This work proposes and implements an architecture, based on the services orchestration concept, to support many kinds of grading process of programming assignments. It is achieved due architecture's features including modularity, extensibility, and flexibility. The cornerstone of the architecture is a new software component named Grading-submodule, which provides of an evaluation service for the source code considering a grading criterion. The implementation has been done on Virtual Programming Lab. Results show workability, and uselfulness for teaching staff and students.

Keywords: programming assignments assessment, automatic grading process, architecture.

1. INTRODUCCIÓN

The learning path in programming-related courses involves the development of an increasing amount of skills and techniques by students. Correspondingly, lecturers must assess the acquired knowledge and practice, by applying and combining different grading criteria, and provide students with proper and timely feedback that allows them to improve their abilities.

Delays in providing feedback after the submission deadline reduces the impact of these feedback comments drastically, as the student may not be concentrated on the subject any longer, has no means to improve his knowledge and skill on that particular topic and for that submission and, therefore, reduces the engagement of students towards analyzing and applying them. It is therefore of great importance that the assessment procedure be done for each student several times per assignment such that, when the students are fully dedicated to the subject, they spend time assimilating and incorporating the feedback before resubmitting the assignment improving their final grade as well as their comprehension of the different topics. The above assessment procedure is an unmanageable task if dealing with a numerous group with allowed resubmissions per assignment.

Among others, current gaps identified include:

- Supporting many grading processes, which considers many and variable criteria.
- Supporting the fast, and easy development of new assessment tasks.

- Supporting users and feedback interoperability by integrating the evaluation and grading processes with a LMS.

This work aims to propose a services-based architecture to deal with the identified gaps. The service orchestration co has been taken from IT domains and applied into automatic grading processes, to provide it of features as modularity, extensibility, and flexibility. Further, this proposal could help with an important challenge as automatic grading in Massively Open Online Courses.

2. RELATED WORK

Douce *et al*. [3] analyzed the systems for assessing programming assignments up to 2005 and identified three different generations comprising 1) tools for internal use in each university or department where the assessment was only made considering a right or a wrong answer; 2) command-line tools that leverage on operating system commands or shell scripts to assess features beyond functional correctness; and, 3) web-based tools that allow engaging a wider audience. This study highlighted security, flexibility, and interoperability as major issues for future work.

Douce study was updated by Ihantola *et al*. [7] and Romli *et al*. [12], including those tools developed from 2005 to 2010. The authors reported security improvements through the introduction of secure environments (sandboxes) that support the controlled and isolated execution of the code submitted by students. However, flexibility and interoperability remained an issue, since nearly every single tool managed their own users and grades, and defined a limited and closed set of grading metrics and procedures, which let lecturers little freedom to introduce new assessment criteria and schemas in the evaluation processes.

Regarding the grading criteria, several studies [6, 12] have highlighted their enormous diversity. Most authors split the grading criteria into two rough groups, namely static and dynamic. The former focuses on the source code while the latter focuses on testing the runtime behaviour of programs. For example, static assessment may include checking the use of specific structures, proper coding styles such as indentation and variable names, measuring the program complexity, etc. On the other hand, dynamic assessment may include checking the functional correctness, measuring the performance by means of e.g. latency and throughput, etc.

The CourseMarker system [6, 5], previously known as Ceilidh, is probably one of the grading systems that incorporates more different grading criteria. It defines about 120 marking tools that wrap UNIX commands, shell scripts, or c and Java programs, which in turn mark one quality of the student submission. A marking scheme dictates which marking tools must be used to mark a specific assignment, the order in which the tools must be called, and the weight assigned to each mark obtained, so that after calling all of them an overall mark and feedback are provided. CourseMarker supports the development of new marking schemes by means of creating new Java classes. While this provides a greater degree of control over the marking process, it also means that the system must be restarted whenever the marking process changes. In addition, CourseMarker works in standalone mode, providing their own Graphical User Interfaces for all the users involved e.g. lecturers and students. This allows for a greater control over the submission and grading processes, but on the other hand forces the users to learn a new environment focused just on programming assignments.

Lately, the interoperability issue has been further investigated by Queirós and Leal [11]. The authors identified three interoperability facets required for flexible assessment systems, namely easy configuration of new exercises, management of users, and report of assessment results i.e. marks and feedback. They further evaluated 15 programming assignments assessment tools according to these criteria, including the previously mentioned. The conclusions of the survey highlighted the need for interoperability and propose their integration with LMSs, since these systems are ready for production, most universities have them deployed, and already include interoperability features for users, grades and feedback.

Further, LMSs provide lecturers and students with a usable GUI: a complete set of tunable parameters for nearly any kind of assignment management for the former; and, an integrated, overall vision of the learning process including feedback and grades for the latter. Although, they usually lack support for assessing programming assignments of any kind, they allow for the development of new modules that provide more functionality to the basic installation. For example, Virtual Programming Lab [10], JUnit Question Type [8], Online Judge [14], JAssess [13], or EPAILE [1] are some Moodle extensions that allow assessing programming assignments. Unfortunately, they allow just for basic grading criteria such as compilation and functional correctness, and developing a new, customized assessment requires modifying deep parts of the system.

In [2] and [4] a comparison among relevant tools was carried out, one key feature considered was the grading criteria used for the automatic grading. It is shown in Table 1. The conclusion was that every institution and even every teacher has his own criterion to grade an assignment, then the lack of a common model to grade is still an important and persistent problem. Although, some of the reviewed tools offer the possibility of support any grading criterion through the building of plugins. The authors recommend that considering a complete grading process would be better. This grading process would have as features: a high level of configurability and flexibility to support any metric or criterion.

Therefore, we propose an architecture, based on the services orchestration concept, to support many grading processes, based on software units to provide support to any grading criterion; which has been tested in a Moodle extension.

Table 1. Grading criteria applied by automatic assessment tools [2, 4]

Tool's name	Grading criteria
CourseMarker	Typography Correctness Structures use Objects design Objects relations
Marmoset	Dynamic and static analysis
WebCat	Code correctness Completeness Test validity Extensible by plugins
Virtual Programming Lab	Correctness based on test cases Open for new methods
Grading Tool (Magdeburg University)	Compilation Execution Dynamic tests
JavaBrat	Correctness
AutoLEP	Static analysis Dynamic analysis
Petcha	Based on test cases
JAssess	Compilation
RoboLIFT	Unit testing (public and private)
Moodle ext. (Slovak University of Technology)	Compilation Syntactic analysis Functionality by comparison
BOSS	Characters comparison
BOSS2	Dynamic analysis based on Plagiarism and JUnit
SAC	Dynamic analysis
Automata	Rubrics based on regression models
eGrader	Static and dynamic analysis
CAP	Static and dynamic analysis
YAP3 + APAC	Functional testing
IT VBE	Dynamic analysis through white box testing

3. PROPOSED ARCHITECTURE

The proposed architecture is based on the services orchestration concept [9], then some features have been inherited:

- The use of an orchestration engine to control the process, and services' calls, and the provision of a compound service (the automatic grading process itself).

- The context preservation among the different grading components inside the grading process.

- The use of an XML document to model and define the grading process.

- The use of request/response between the orchestrator and the components inside the grading process.

There are two main components, the orchestrator and a new component called Grading-submodule. The former orchestrates the process and calls one by one a set of services provided by the Grading-submodules execution. Every Grading-submodule's call is considered as an independent service. This software component has been designed to provide of modularity, flexibility and extensibility to any programming grading process considering diversity of criteria, and grading metrics.

The architecture will support many kinds of grading processes, which can be seen as grading services, because each of them can be modeled as a set of Grading-submodules. The Grading-submodules can be arranged in any sort, and they can be reused. Fig. 1 shows the proposed architecture in a layer-based approach, where the two top layers are static but the three bottom layers are completely dynamic.

3.1 Grading-submodule

The cornerstone in the architecture is the Grading-submodule. This component allows evaluating code considering one grading criterion, and this last could consider one or more metrics. Then, the Grading-submodule provides of a grading service depending of a given criterion.

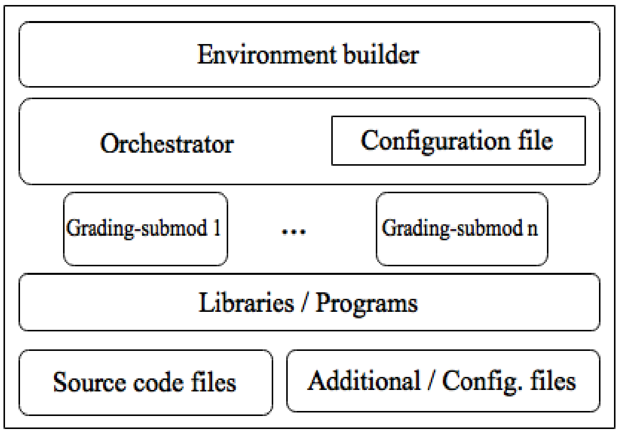


Figure 1. Proposed architecture for grading process.

The goal of the Grading-submodule is performing an action on the code to get values for the considered criterion's metrics, and then helping to establish a grade. Therefore, it is mandatory an associated program to perform that action.

Initially, the program could be written from scratch, but it could be seen as a wrapper too, which could use other already built tools. Depending on the goal a set of parameters may be required, so the Grading-submodule supports the inclusion of those parameters. In Java, tools like JUnit or CheckStyle, could be supported.

The communication from a Grading-submodule to the Orchestrator is through the Configuration file.
The idea in a system, which implements the architecture, is that when a Grading-submodule is built and registered, it could be used/reused in many grading processes.

Finally, it is worth saying that every Grading-submodule has to be well defined; it means it needs to be associated to an only grading criterion. This feature will provide of modularity to a grading process.

3.2 Components

The architecture is expressed in components, and each of them has a role well defined, so they can be improved or set as you need.

Environment builder.- This layer aims to set the environment up to run a grading process. This level could be implemented to move or copy files, to code or decode data, to export environment variables, among other actions. After setting up the environment, it has to call the orchestrator.

An additional advantage of the proposed architecture is that the environment builder can act as an interface to allow integration with other systems that could provide the front-end.

Orchestrator.- This layer aims to control the whole grading process, based on the information provided by the Configuration file.

The Orchestrator has a set of ordered tasks:

- It has to read the Configuration file, and load all the information that it contains.

- It has to call and communicate with every Grading-submodule associated program. The communication from the Orchestrator to the Grading-submodule associated program is not a trivial work. When a system is deployed by the first time, there are not Grading-submodules registered, so the system does not know about the future associated programs to call. The Orchestrator has to use a dynamic way to be able to call a program in execution time. In Java it is done by using Reflection technology.

- It has to process every Grading-submodule results, which were located in the Configuration file during the execution of the associated program, to calculate the final grade and to collect comments.

- It has to send the feedback (grade and comments).

The Orchestrator controls the order inside a grading process.

Submission Configuration File.- This file is aimed to contain two kinds of information. The first one is submission's metadata to be used by the Orchestrator to manage the grading process. The information may include the student's identification, the submission's identification, the grade-base, and the list of Grading-submodules to call. The second kind of information includes all the required parameters to each Grading-submodule associated program, and their values.

This file saves information about the results of each Grading-submodule associated program, and the whole process. All the information has to be ordered and structured, so the configuration file is an XML.

This file is quite important inside the grading process, it is required by the Orchestrator to start the process, and acts as a communication mean between the Grading-submodule associated program and the Orchestrator.

Grading-submodules.- This layer includes a Grading-submodules set. There is not a limit for the number of Grading-submodules registered in a system which implements the architecture and they can be added as the teaching staff needs, it implies **extensibility.**

The number and the arrangement of the Grading-submodules are not limited, so there is **flexibility** inside the grading process. The number of Grading-submodules inside the process, the order and how they are called are defined in the configuration file.

Grading-submodule associated programs are called by the orchestrator inside a grading process.

Libraries and Programs.- This layer includes external programs, libraries, or packages required by any Grading-submodules associated program. This component gives a very important advantage because we can take already built good tools and include them inside our architecture.

The Grading-submodule associated program will call any already built library or program. In this case, the Grading-submodule acts as wrapper and we avoid "reinvent the wheel".

Source Files.- This layer refers to source files written and sent by the students in a submission to accomplish with an assignment.

The students only have to take their source files and send them to the system which implements our architecture.

Additional or Configuration Files.- This component includes files defined by the teaching staff and required by the Grading-submodules associated program or by Libraries and Programs inside the grading process. For instance test cases, rules files, among others.

4. TESTING THE ARCHITECTURE

To validate the proposed architecture, we could have used an existing tool or creating a new tool from scratch. The first choice was selected. Some criteria have been used to compare a set of existing tools:

- License, it will allow accessing the complete code to make changes on this.
- Availability, it is necessary to know if the access to download the code is possible.
- Architecture suitability, it means a tool's architecture which allows testing our architecture.

Additionally, due to the fact that our goal is to use the architecture in our programming classes, we have three more requirements:

- Support for Java and extensibility for other programming languages.
- Communication with Moodle LMS.
- A safe environment to evaluate the code.

VPL (v 1.32) was used as base tool due its next features: the GNU/GPL license, so it is possible to use and modify this regarding the own necessities; the easiness to access the documentation, help and to download the source code; the feature of working as Moodle plugin; its module for plagiarism detection; its security features regarding authentication and working with a safe test environment; its ability to allow defining assessment scripts, it gives the possibility to consider more metrics and criteria to grade; and it can support automatic and semi-automatic processes.
Then, VPL has been adapted to consider the proposed architecture. The new features added to VPL were:

- Management of Grading-submodules.
- Management and configuration of grading process.
- Automatic grading process considering Grading-submodules.

4.1 VPL's Customization

VPL uses two subsystems VPL-Moodle and VPL-Jail (a sandbox environment), each of them is deployed in a different server. The first one is a Moodle-plugin oriented to be the graphical interface for managing the programming lab; and the second is oriented to provide of a sandboxed environment for the grading process.

Each VPL subsystems have been modified, in the VPL-Moodle a grading process management module has been implemented. In the VPL-Jail, the programs that start the process were modified, and the orchestrator and the Grading-submodules were implemented.

4.2 Grading Process Customization (VPL-Jail)

VPL starts the grading process when all the required files are inside the jail. The Jail server, a service running in the VPL-Jail subsystem, executes the evaluation script and an ordered process starts. Fig. 2 shows the whole grading process through the interaction among a set of necessary programs.

The Jail server, the evaluation script and the execution file are part of the VPL architecture. The execution file has been carefully modified to start the new grading process (which support the proposed architecture). The orchestrator calls every Grading-submodule associated program, calculates the final grade, forms the feedback and prints it. Finally, the Jail server collects the feedback; send it as a HTTP response; and delete the sandboxed environment.

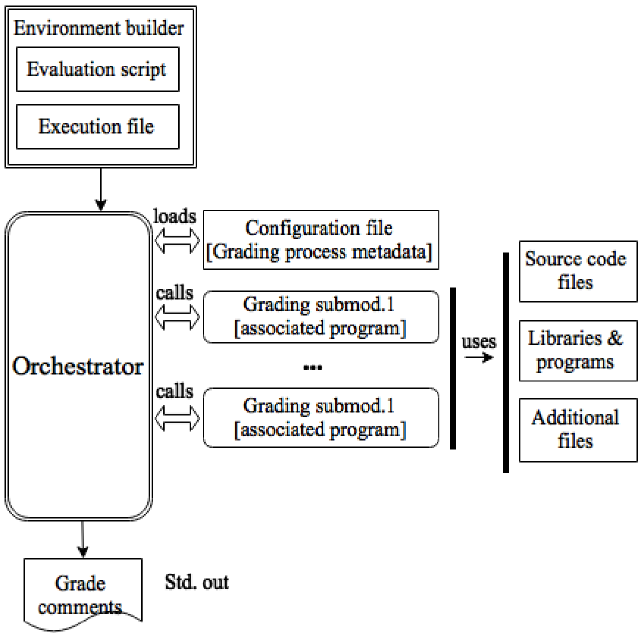


Figure 2. Elements and calls inside the grading process.

Object Oriented Programming has been used to implement all the required programs; so, Fig. 3 shows the class diagram used. Additionally, the implementation has been done using Java language.

The *SubmissionConf* and the *GradingSubmoduleConf* classes have been abstracted from the configuration file. The first one includes information about the whole submission and will be used by the orchestrator to start the grading process. The second one represents information to be used for every Grading-submodule associated program.

The *GradingSubmoduleProgram* class has been abstracted from the Grading-submodule associated program and has been defined as abstract because it acts as 'intermediary' between the orchestrator and any Grading-submodule associated program that the teaching staff or administrator will add.

The *AnyGradingSubmoduleProgram* class is depicted to represent any Grading-submodule associated program that will be considered inside the grading process. The *Orchestrator* class is quite important because control the whole grading process, its operations include:

- Loading the data inside the instance of *SubmissionConf.*

- Orchestrating the process. It refers to iterate the list of *GradingSubmoduleConf* inside the *GradingSubmissionConf* to operate sequentially every Grading-submodule.

- Creating dynamically an instance of *AnyGradingSubmoduleProgram* and run a defined operation on the code. The dynamic creation was possible through the use of *Reflection* technology.

- A final processing to calculate the final grade, to collect the individual comments and to establish a general comment of the whole process.

- Outputting the response. All the processed information is output in a format to be recognized as response feedback by the Jail server.

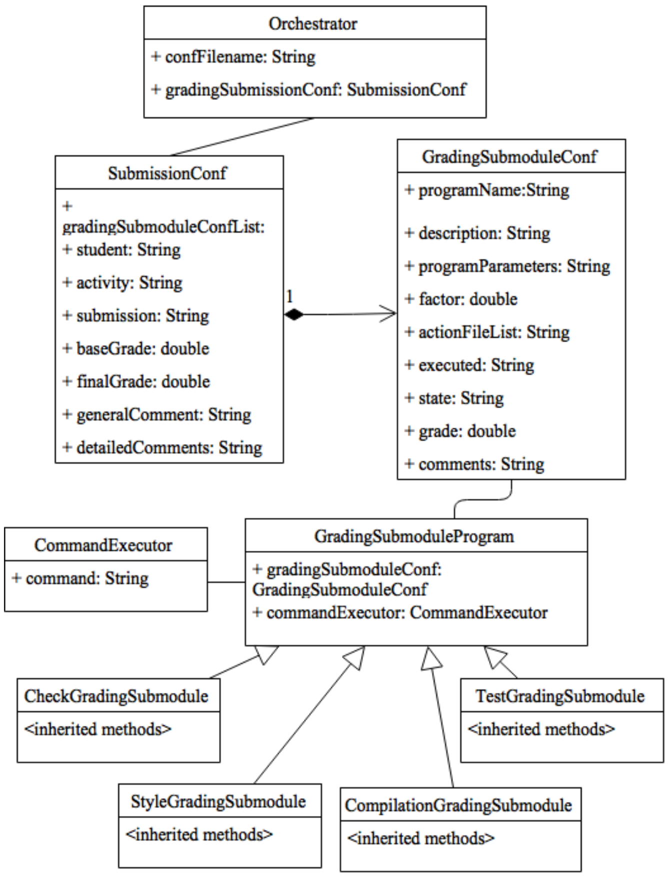


Figure 3. Classes' Diagram

To validate the architecture, the implementation of some classes to test the code has been necessary. These new classes are oriented to check the structure of a set of files (*CheckGradingSubmodule*), to compile a set of source code files (*CompilationGradingSubmodule*), to test a set of source code files against test cases (*TestGradingSubmodule*), and to evaluate the style of a source code file (*StyleGradingSubmodule*).

The sequence diagram shown in Fig. 4 is helpful to understand in a better way the real interactions inside the system. It is useful to highlight the importance of the orchestrator. The *SubmissionConf* and *GradingSubmoduleConf*classes have not been represented because they represent the configuration file (parsing files), a kind of static element. They do not perform any action as well.

The configuration file has a remarkable importance. The orchestrator requires of the submission information, and the Grading-submodule associated program needs the information related to each Grading-submodule. The XML configuration file is shown in Fig. 5. *XML Mapper (JAXB)* was used to parse the configuration file.

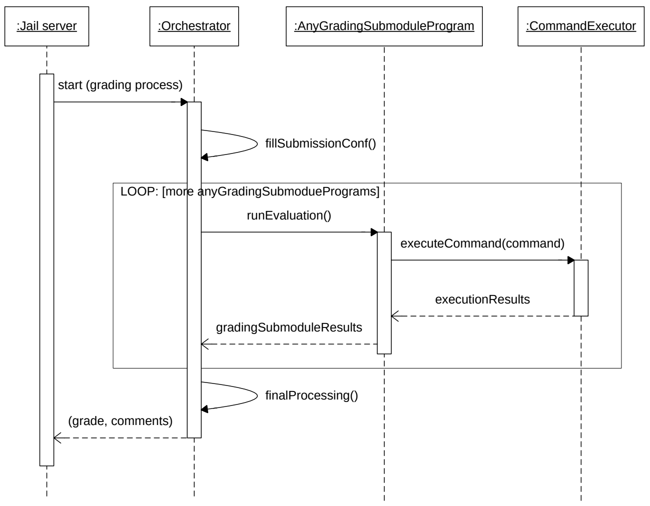


Figure 4. Grading Process Interaction

The information fields related to a submission includes:

- Student, it has information of the student. This information can be the name or an id for instance.
- Activity, it has information to identify the activity. It can be the VPL activity's id.
- Submission, it has the submission number or the submission identification.
- Base grade, it is the base over which the final grade will be calculated.
- Final grade, it is the grade for the submission.
- General comment, it stores a short comment for the submission.
- Detailed comments, it stores the comments of every Grading-submodule.

The information fields about every Grading-submodule include:

- Program name, it contains the full name of the Grading-submodule associated program (including the package). The .class extension is not included.

- Description, it contains a short description for the current submodule. It has to express the main action that the submodule will do.

- Program parameters, it has additional data required by the associated program. It is a string, which includes parameters' values separated by a semicolon and without blank spaces. The parameters can be pathnames, numbers, and so on. If one parameter has many values, commas should separate them.

- Factor, a percentage that represents the submodule weight in the final grade calculation. The addition of this field in all Grading-submodules has to be 100.

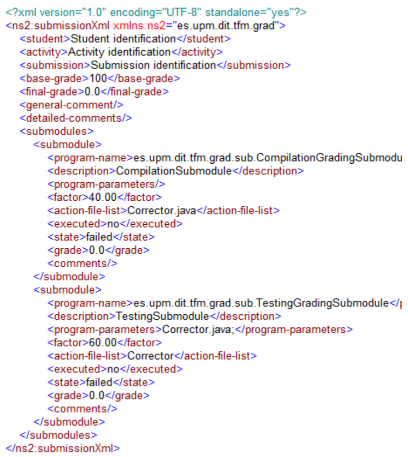
- Action file list, it has a list of filenames over which the main action of the submodule will be executed. The list will be composed of full names (including the package name) or relative names (just the filename) and the file extension depending on every submodule.

- Executed, it shows if the submodule has been executed; independently of success or fail.

- State, it indicates if the submodule execution finished perfectly (*success*), getting a full grade; or if there were some troubles *(failed)* and a partial grade was obtained.

- Grade, it is the grade for the current Grading-submodule. It is a numeric value between 0.00 and 100.00 with 2 decimal places. There always has to be a value in this field since its creation.

- Comments, detailed information about the execution of the Grading-submodule associated program.



```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ns2:submissionXml xmlns:ns2="es.upm.dit.tfm.grad">
   <student>Student identification</student>
   <activity>Activity identification</activity>
   <submission>Submission identification</submission>
   <base-grade>100</base-grade>
   <final-grade>0.0</final-grade>
   <general-comment/>
   <detailed-comments/>
   <submodules>
      <submodule>
         <program-name>es.upm.dit.tfm.grad.sub.CompilationGradingSubmodu
         <description>CompilationSubmodule</description>
         <program-parameters/>
         <factor>40.00</factor>
         <action-file-list>Corrector.java</action-file-list>
         <executed>no</executed>
         <state>failed</state>
         <grade>0.0</grade>
         <comments/>
      </submodule>
      <submodule>
         <program-name>es.upm.dit.tfm.grad.sub.TestingGradingSubmodule</
         <description>TestingSubmodule</description>
         <program-parameters>Corrector.java;</program-parameters>
         <factor>60.00</factor>
         <action-file-list>Corrector</action-file-list>
         <executed>no</executed>
         <state>failed</state>
         <grade>0.0</grade>
         <comments/>
      </submodule>
   </submodules>
</ns2:submissionXml>
```

Figure 5. Structure of the XML Configuration File

4.3 VPL-Moodle Customization

There is a necessity of implementing new features inside the VPL Moodle's plugin, they include: the Grading-submodules Management, the Grading Process Management and a mean to communicate with the VPL-Jail; all of them implemented in a new VPL's module called Grading Process Management Module. The implementation of these features required of some changes in the data infrastructure, the directory system, and the database.

Regarding to directory system, two new directories were created in the VPL data directory, one of them is to store the source code of all the Grading-submodules associated programs, and the other one is to store additional files for every VPL activity that has a grading process associated.

Regarding to database, four new entities were added:

- Grading submodule. It is the representation of the Grading-submodule already defined. This has an important attribute, the *programfilename* which saves the absolute path to the location of the program file associated to the Grading-submodule.

- Grading parameter. Every Grading-submodule associated program could require of parameters to its proper working. This entity will save the definition of each of them.

- Process grading submodule. Every VPL activity will have a set of Grading-submodules to be used inside its grading process. When a Grading-submodule is selected to be part of this grading process, this is converted in Process grading submodule.

- Process grading parameter. This entity saves all the values for parameters required by the Process grading submodule entity.

Besides the data model, the web pages were coded considering the VPL architecture and the Moodle API, so, all of them are very related components.

Finally, when the teaching staff uploads the Grading-submodule associated program, and when the student sends his code, it is necessary a connection between the VPL-Moodle and the VPL-Jail subsystems. The technologies XML-RPC and base64 were used to pass all the necessary data between the subsystems.

5. USING THE ARCHITECTURE

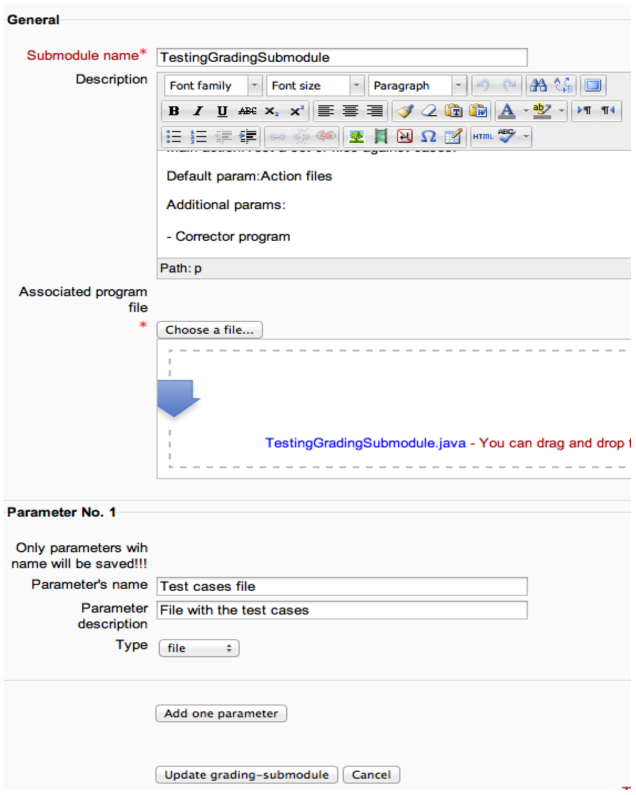
To use the tool, which implements the proposed architecture, it is necessary to create and register the Grading-submodule. To implement a Grading-submodule, it is mandatory to define the associated criterion. Four Grading-submodules were created; *CheckGradingSubmodule,* to check the structure of a set of files; *CompilationGradingSubmodule*, to compile a set of source code files; *TestGradingSubmodule*, to test a set of source code files against test cases; *StyleGradingSubmodule*, and to evaluate the style of a source code file.

Any Grading-submodule the teaching staff needs can be created. The Grading-submodule related program has to inherit from the class GradingSubmoduleProgram and can use some useful methods already implemented. After the creation of the program, it is necessary register the new Grading-submodule in the system. The required data includes the name, description, associated program, and any parameter required. This is shown in Fig. 6.

The registration is done once, and other teaching staff member can reuse it.

After creating a VPL activity (the process can be reviewed in the official page), we can set up and configure the grading process.

The teaching staff can add any Grading-submodule already registered. It is necessary to configure the factor of each of them inside the grade calculation. The user can establish the order of each Grading-submodule inside the grading process, and set every value for parameters required. It can be seen in Fig. 7 and Fig. 8.



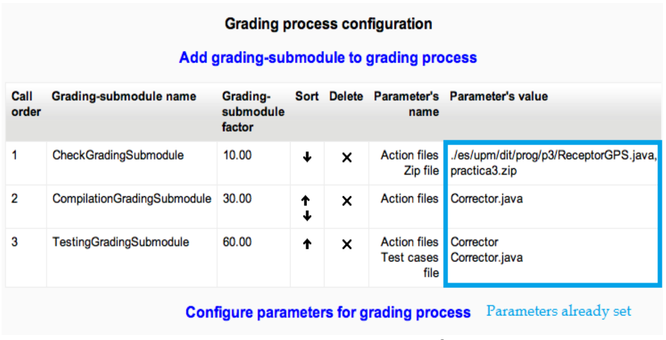
General

Submodule name*	TestingGradingSubmodule
Description	Default param:Action files Additional params: - Corrector program Path: p
Associated program file *	TestingGradingSubmodule.java - You can drag and drop f

Parameter No. 1

Only parameters wih name will be saved!!!	
Parameter's name	Test cases file
Parameter description	File with the test cases
Type	file

Figure 6. Registering Grading-submodules



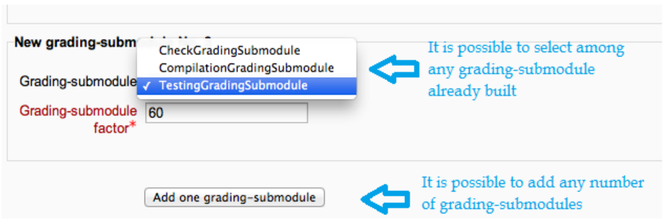
Grading process configuration

Call order	Grading-submodule name	Grading-submodule factor	Sort	Delete	Parameter's name	Parameter's value
1	CheckGradingSubmodule	10.00	↓	×	Action files Zip file	./es/upm/dit/prog/p3/ReceptorGPS.java, practica3.zip
2	CompilationGradingSubmodule	30.00	↑↓	×	Action files	Corrector.java
3	TestingGradingSubmodule	60.00	↑	×	Action files Test cases file	Corrector Corrector.java

Add grading-submodule to grading process
Configure parameters for grading process
Parameters already set

Figure 7. Grading Process Configuration

For the student, the process is very simple; he has to send his code by selecting a file and uploading to the VPL activity. After having done that, he receives the feedback in detail as shown in Fig. 9.

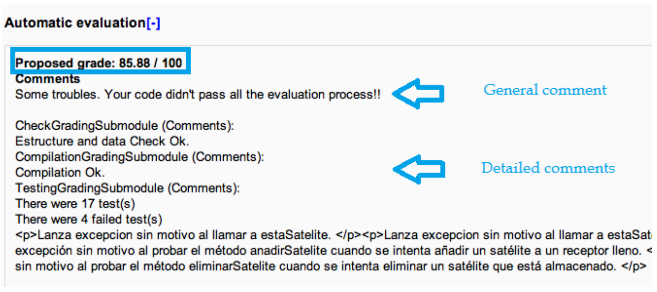


New grading-subm

Grading-submodule	✓ TestingGradingSubmodule
Grading-submodule factor*	60

CheckGradingSubmodule
CompilationGradingSubmodule
TestingGradingSubmodule
It is possible to select among any grading-submodule already built
Add one grading-submodule
It is possible to add any number of grading-submodules

Figure 8. Grading-submodule Addition



Automatic evaluation[-]

Proposed grade: 85.88 / 100
Comments
Some troubles. Your code didn't pass all the evaluation process!!

CheckGradingSubmodule (Comments):
Estructure and data Check Ok.
CompilationGradingSubmodule (Comments):
Compilation Ok.
TestingGradingSubmodule (Comments):
There were 17 test(s)
There were 4 failed test(s)
<p>Lanza excepcion sin motivo al llamar a estaSatelite. </p><p>Lanza excepcion sin motivo al llamar a estaSat
excepción sin motivo al probar el método anadirSatelite cuando se intenta añadir un satélite a un receptor lleno. <
sin motivo al probar el método eliminarSatelite cuando se intenta eliminar un satélite que está almacenado. </p>

Figure 9. Students' Results Interface

4. CONCLUSIONS

The main contribution of this work is **the architecture proposal** based on the services orchestration concept, which defines an orchestrator and Grading-submodules (in any number and any arrangement) providing their services, which could be implemented with any technology. This architecture can be used by already implemented tools or by new ones.

It is worth highlighting that the idea of the Grading-submodule artifact can be used or improved to define new ways of grading or new architectures. In addition, the elements of the proposed architecture are already implemented and can be reused to work in new implementations. It can help to save implementation time in related projects.

The applied technologies shown can be helpful to provide a first sight of them, and to think about them as possible solutions for issues in other projects with similar functionalities. The considerations made in the different stages can be useful for other similar projects or those that follow a similar process as performed in this work.

The present work has validated the proposed architecture. It means that the architecture works as expected but it does not mean that it could not be improved. Some improvements and future work include the next ones.

Measuring and comparing time in the grading processes definition. After the creation and registration of Grading-submodules, the time to define and configure grading processes associated with assignments could be shorter than using other solutions. It could be probed through measuring the time that the configuration of a grading process takes in this solution against the time needed by other solutions' configuration.

Defining a management module for grading processes. The case-studies have shown that sometimes the grading process could be very similar. The grading process (without the parameters' values) could even be the same among different assignments. So if it were possible to define a management of grading processes, it could help to reduce the time of the grading process definition.

Developing a drag-and-drop interface to define a grading process.

Annotating the Grading-submodules. By considering a possible increment in the number of Grading-submodules registered and a way to sort and filter them when defining the grading process, it is possible to create tags to make a classification. These tags could be metrics, criteria, and even the programming language associated.

Improving the deployment of ancillary programs. The current solution supports the use of ancillary programs; these programs have to be placed manually in the *libs* directory. It could be possible to implement a management interface for these programs.

It is possible to think about a solution which can store the data and maintain the jail environment, and provides everything as a service. It means that it could provide a service to access an assignments' repository, a service to copy and to store the data inside that system, a service to start with the grading process, and so on; in this case this solution would be completely independent and could connect any system (just a front-end), which would provide interfaces to connect the solution.

Regarding the Grading-submodule associated program, it acts as a wrapper written in Java that can call another libraries or ancillary programs, which have to be located in the *libs* directory. But it is possible to think about the possibility that the wrapper supports calls to other programs in other hosts through services. The idea appeared because there are already built tools which can provide the evaluation of some metrics as a service. In this case the wrapper could be more powerful.

The XML configuration file could be changed to support more ways to calculate the final grade and additionally to stop the process if some Grading-submodule was not passed. These features could be configurable.

REFERENCES

[1] M. Amelung, K. Krieger and D. Rosner, "E-Assessment as a Service," Learning Technologies, IEEE Transactions on, vol. 4, pp. 162-174, 2011.

[2] J. C. Caiza, J. M. Del Alamo, "Programming assignments automatic grading: review of tools and implementations," in 7th *International Technology, Education and Development Conference (INTED2013),* pp. 5691, 5700.

[3] C. Douce, D. Livingstone and J. Orwell, "Automatic test-based assessment of programming: A review," *Journal on Educational Resources in Computing (JERIC),* vol. 5, pp. 4, 2005.

[4] M. Guerrero, D. S. Guamán and J. C. Caiza. (2015, Feb.). Revisión de Herramientas de Apoyo en el Proceso de Enseñanza-Aprendizaje de Programación. *Revista Politécnica.* [Online]. 35(1), pp. 82-90, 2015. Available: http://www.revistapolitecnica.epn.edu.ec/revista_archivos/revista_volumen_35/TOMO_1.pdf

[5] C. A. Higgins, G. Gray, P. Symeonidis and A. Tsintsifas, "Automated assessment and experiences of teaching programming," *Journal on Educational Resources in Computing (JERIC),* vol. 5, pp. 5, 2005.

[6] C. Higgins, P. Symeonidis and A. Tsintsifas, "Diagram-based CBA using DATsys and CourseMaster," in *Computers in Education, 2002. Proceedings. International Conference on,* 2002, pp. 167-172.

[7] P. Ihantola, T. Ahoniemi, V. Karavirta and O. Seppälä, "Review of recent systems for automatic assessment of programming assignments," in *Proceedings of the 10th Koli Calling International Conference on Computing Education Research,* 2010, pp. 86-93.

[8] *JUnit Question Type*. Available: https://docs.moodle.org/20/en/Junit_question_type.

[9] E. Newcomer .(2005). *Understanding SOA with Web services.* [Online].Available: https://www.safaribooksonline.com/library/view/understanding-soa-with/0321180860/ch01.html

[10] J. C. Rodríguez-del-Pino, E. Rubio-Royo and Z. J. Hernández-Figueroa, "A Virtual Programming Lab for Moodle with automatic assessment and anti-plagiarism features," 2012.

[11] R. A. P. Queirós and J. P. Leal, "PETCHA: A programming exercises teaching assistant," in *Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education,* 2012, pp. 192-197.

[12] R. Romli, S. Sulaiman and K. Z. Zamli, "Automatic programming assessment and test data generation a review on its approaches," in *Information Technology (ITSim), 2010 International Symposium in,* 2010, pp. 1186-1192.

[13] N. Yusof, N. A. M. Zin and N. S. Adnan, "Java Programming Assessment Tool for Assignment Module in Moodle E-learning System," *Procedia-Social and Behavioral Sciences,* vol. 56, pp. 767-773, 2012.

[14] S. Zhigang, S. Xiaohong, Z. Ning and C. Yanyu, "Moodle plugins for highly efficient programmin courses," in *Moodle Research Conference,* 2012, pp. 157-163.