IO4 - Rubric for assessing the quality of the students' code

1. Introduction

In the present document we introduce the first version of a rubric which should allow us to assess the degree of quality that students' code has. The following important issues should be taken into consideration before using the rubric:

- **Any item of the rubric is optional**. This means that each lecturer must decide which items apply for her course or even for each assignment. For instance, design items may not make sense for the first assignment of an introductory course, but they may be critical for the last one. **Please use as many items as possible to increase comparability between courses**.

- **This rubric must be used both in the baseline stage and in the validation stage**. The baseline can even be done by using the rubric with submissions from previous semesters (previous to QPED project).

- The rubric is devised so that it requires evaluators making the minimum effort. As a result, **only 10 items must be evaluated**. These items try to encompass most of the aspects related to quality software. These items are based on other rubrics that have been found in the literature.

- Each item is evaluated by using a **4-point scale**. For each feature, we provide some examples of good and bad performance. For each feature, **choose the value that you think that is most appropriate according to your own/course criteria**.

2. Rubric

Information

Institute:

Course ID:

Course name:

Run period:

Assignment:

Which features of the rubric are included?

☐ **TILE**	**Version:**
☐ **PG**	**Version:**
☐ **Feeback Tools**	**Version:**

Feature	Scale and criteria examples			
Modularity ☐	1 ☐	2 ☐	3 ☐	4 ☐
	• The project contains spaghetti code, e.g. it lacks a clear organization. • Most of the classes and functions perform many unrelated tasks and/or their bodies are large. • The degree of coupling is high, e.g. classes are not well encapsulated and interaction between objects does not take place only through public methods (or preferably through an interface).		• Project structure is clear since the code is organized in coherent packages, folders, files, etc. • Most of the classes and functions perform a limited set of tasks and their bodies are limited in length.	
Data types ☐	1 ☐	2 ☐	3 ☐	4 ☐
	• The choice of some data types is wrong, e.g. an integer is used when a boolean is enough. • Complex data structures are used when are not needed, e.g. primitive-data array vs Object-data array.		• Appropriate data type selection for variables and attributes. • Appropriate data type selection for function/method return.	
Readability ☐	1 ☐	2 ☐	3 ☐	4 ☐
	• Formatting is usually missing, poor or it is used wrongly. • The layout of the code is not easy to read. • Some names appear unreadable, meaningless, misleading and/or do not meet naming conventions. • Comments are generally missing or explain obvious issues, such as what the code statement is doing.		• Indentation, line breaks, spacing and brackets fully clarify program structure. • Meaningful identifiers which meet naming conventions are used as variables, functions and class names. • Comments do not explain what the code is doing, instead explain tricky or important decisions. • Comments are present where strictly needed and enhance understanding of the code.	
DRY principle ☐	1 ☐	2 ☐	3 ☐	4 ☐
	• Repeat snippets of code quite often. • Use magic (hard-coding) numbers or string literals.		• Helper functions are used in order to reuse code. • Constants are used and they are kept in a common place.	
Flow ☐	1 ☐	2 ☐	3 ☐	4 ☐
	• There is deep nesting, e.g. many branches, nested loops, etc. • Dead code (unreachable code) is present. • Unnecessary steps were performed.		• Flow is simple so that the most common path through the code is clearly visible. • Traceability: it is easy to verify know which code line corresponds to which program requirement/s.	

API Documentation ☐	1 ☐	2 ☐	3 ☐	4 ☐
	• Information is generally missing, redundant, incomplete or misspelled at the top of the file. • Documentation about the author is missing. • Documentation about the class/module is missing or incomplete. • Documentation about the fields is missing or incomplete. • Documentation about the methods is missing or incomplete.		• At the top of the file, there is a block comment in which the programmer provides author's names. • The summary of the goal of the file and its version. • Documentation about attributes is correct. • Documentation about methods is correct. • Information is generally present and provides a brief description. • It contains pre and post conditions. • The meaning/role of each parameter is clear.	
Output correctness ☐	1 ☐	2 ☐	3 ☐	4 ☐
	• The code does not compile and run cleanly. • The program does not meet some of the specifications.		• Program conforms to the specifications provided by the assignment. • It produces correct results for correct inputs.	
Program Robustness ☐	1 ☐	2 ☐	3 ☐	4 ☐
	• Errors or abnormal conditions are not all handled.		• The program reacts properly to abnormal conditions and erroneous inputs.	
Test Completeness ☐	1 ☐	2 ☐	3 ☐	4 ☐
	• Some specification/requirement is not thoroughly tested, e.g. a test case checks correct input but it does not check the behavior of the program with anomalous/exceptional inputs		• For each specification/requirement that a test suite covers, there are enough tests cases to validate it.	
Traceability (test coverage) ☐	1 ☐	2 ☐	3 ☐	4 ☐
	• It is hard to verify that the tests cover all the program requirements/specification.		• Tests are clear so that it is easy to detect if any requirement is left out of the tests. • It is easy to know what requirements are evaluated by each test case/suite.	

Instructions

- For each feature, tick the checkbox if it is applicable.
- For each feature, circle the level of accomplishment that is most representative of the code that you are evaluated.
- If it is possible, please highlight the examples that are present in the code. You can highlight both correct and incorrect examples.