Documentation of Lab 2

1. An introduction where the problem is described. For example, what should the program do? Which classes do you have to define? Which methods do you have to implement for these classes?

The program should create a Program by adding a list of statements to it. After that, the Validator class checks each statement to ensure it is valid or not based on the Logo's instruction dictionary. Once validated, the turtle executes the Program, interpreting each statement to perform the action desired. This piece of the project consists of 4 extra classes added to the 2 ones of the previous lab.

To start with, the **STATEMENT** class has been designed as following:

Its attributes are named as:

- - word: name of the Statement.
- - parameter: value of the Statement, for the case of “END” Statement, there is no parameter.

In order to initialize the class, we call its constructor Statement:

- + Statement(initword: String, initParameter: int): initialize the Statement with the given name and parameter value. Moreover, it saves the name in upper case letters if the name has been given with exactly three letters.
- + Statement(initword: String): the constructor used to initialize an “END” Statement.

Moreover, we have implemented getters and a toString method:

- + getWord(): it returns the name.
- + getParameter(): it returns the parameter value.
- + toString(): it gives the Statement in the form of a string.

In addition, the **PROGRAM** class has been designed as following:

A class whose objects are made of a single attribute:

- - sequence: list of Statement objects to keep track of what actions the turtle will perform.

In order to initialize the class, we call its constructor Program:

- + Program(): when a Program object is created, it is initialized with an empty Statement list.

At the same time, we have implemented several methods to work with this list:

- + addStatement(newStatement: Statement, int initParameter: int): given a certain Statement, we add it to the attribute list. Additionally, we have differentiated between “END” Statement and not “END” Statement. Therefore, if the Statement does not contain a parameter, it will be an argument of the following method.
- + addStatement(newStatement: Statement): to add “END”s statements.
- + sizeProgram(): it returns how many lines (statements) the Program object has.
- + getStatement(line: int): given an integer, it returns the whole Statement

Moreover, we have implemented the **LOGO** class as it follows:

Its attributes are instances of objects of previous labs and seminars:

- - turtle: a Turtle object
- - instructions: a dictionary of valid instructions that the turtle can perform. We have created the dictionary as a HashMap.

Then, we have its constructor:

- + Logo(): the constructor creates a new Turtle and an empty HashMap. Then, it introduces into the dictionary the following instructions with their valid range: “PEN”, “FWD”, “ROT”, “REP”, “END”.

Its unique methods are getters which return instructions:

- + getInstruction(nameInstruction: String): given a name, it searches into the dictionary with the given key and returns the Instruction..
- + getInstructions(): it returns the whole dictionary of instructions.

Lastly, the **VALIDATOR** class has been designed as following:

A Validator itself is an instance of a Logo object.

- - logo: the only attribute is a Logo object.
- + Validator(var1: Logo): it puts the Logo argument object to the attribute “logo”.

The goal of this class is validate both Statement and Program objects:

- + errorCode(var1: Statement): given a Statement, on the one hand, the method checks if its word is a valid instruction by looking at the dictionary, if not it returns 1. On the other hand,

it checks if the parameter of the Statement is in its valid range, in case it is not, it returns 2. Otherwise, it returns 0, which means that the given object is valid and can be used.

- + printError(var1: Statement): it prints the type of error or the validation of a Statement depending on the number returned by the errorCode method.
- + errorCode(var1: Program): given a Program, it iterates over its statements. If at least one Statement is not valid, it should return -1. Moreover, the Statement's loops have to be balanced with their respective "END" statements. If the Program is unbalanced, it returns 1. Otherwise, the whole Program is legal, therefore it returns 0.
- + printError(var1: Program): it prints the type of error or the validation of a Program depending on the number returned by the errorCode method.

2. A description of possible alternative solutions that were discussed, and a description of the chosen solution and the reason for choosing this solution rather than others. It is also a good idea to mention the related theoretical concepts of object-oriented programming that were applied as part of the solution.

We have followed almost all the references from the given Lab2 diagram, although we have added some getters mentioned below.

On the one hand, in order to check whether the instruction word is valid or not, we have used the condition !logo.getInstructions().containsKey(initStatement.getWord()). What we are doing here is to get the instructions array and check whether the instruction word is defined or not in it. In order to do that, we have added the getter public HashMap<String, Instruction> getInstructions() that returns the instructions hashmap.

On the other hand, in order to check whether the program is valid or not, we have used a for loop that iterates through the whole arraylist of the Program class statements, and, later on, checking whether each statement is valid or not. In order to do the for loop, we have added the getter public ArrayList<Statement> getBody() that returns the statements arraylist.

3. A conclusion that describes how well the solution worked in practice, i.e. did the tests show that the classes were correctly implemented? You can also mention any difficulties during the implementation as well as any doubts you might have had.

In order to check how well our solution works and show that the classes are also well implemented, we have initialized a Program and a Logo object that will serve us to check several methods.

To start with, when trying to get a specific Instruction in the program, for example we tried to access the instruction placed at 0, it returned the name of the Instruction “FWD” and its parameter value. It works since we have added that Instruction to the first one. So, when we acceded to the 0th Instruction in the array list, it should give that returned one. Additionally, we asked the program to return its size in terms of statements. Since we added two of them, the implementation works.

Moreover, for the Logo class code validation, it correctly found the instruction "PEN" and showed its range. However, when asked for similar words like "pen" and "pens," it also returned corrected results. Firstly, “pen” is a valid name for an Instruction given that the argument of the getInstruction method is turned to upper case. Therefore, it is the same argument as “PEN”. Then, the method also showed name and range. Secondly, “pens” is not a correct Instruction, only the “PEN” name is valid.

Furthermore, the Validator efficiently checks the program’s validity and provides clear feedback without any issues.

Overall, the classes interact well with the tests, confirming that the solution is well-implemented and reliable in practice.