Documentation of Lab 4

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?

Once the classes are fully designed and their roles clearly defined, we can proceed to implement a program that draws Logo instructions on the screen. To achieve this, we will use the Graphics and JPanel libraries, which handle creating a graphical window where the turtle's movements and drawings will be displayed.

Moreover, we will define some subclasses from TurtleInstruction class to override the method apply. They will dictate the turtle's actions based on the given instruction. Each instruction will trigger a corresponding behaviour, such as moving forward, rotating, or activating the pen for drawing. In addition, we will need to define a class Segment to draw a simple line using the previously mentioned libraries.

Next, the Logo class will also be upgraded to manage the graphical components. It will initialise the screen with a background colour and dimensions, maintain a list of all drawn segments, and handle the final rendering of the drawing. Furthermore, the turtle's position and orientation will be visualised on the canvas, providing real-time feedback on its movements.

By implementing these upgrades, we will be allowed to simulate and visualise Logo programs efficiently.

To start with, we made the **TURTLEINSTRUCTION** class abstract in order to override a method in the subclasses:

- + abstract apply(parameter: int): to be possible to override it.

Given that, we have designed three subclasses inheritance from TurtleInstruction superclass named **PenInstruction**, **FwdInstruction** and **RotInstruction**. These new classes have been defined equally, the difference: the apply method.

- + constructor(word: String, minRange: int, maxRange: int, turtle: Turtle): it only calls the super constructor.
- + apply(parameter: int): depending on the class and the given parameter, it changes the state of the pen, it moves forward or it rotates. It calls turtle's methods.

Moreover, the **SEGMENT** has been designed to represent a straight line segment on the screen:

The class is structured with the following attributes:

- - startX: the x-coordinate of the starting point of the segment.
- - startY: the y-coordinate of the starting point of the segment.
- - endX: the x-coordinate of the ending point of the segment.
- - endY: the y-coordinate of the ending point of the segment.

To create an instance of the class, the constructor is defined as follows:

- + Segment(x1: int, y1: int, x2: int, y2: int): it initialises the segment with the specified start and end coordinates. The parameters x1 and y1 correspond to the starting point, while x2 and y2 represent the ending point of the segment.
- + paint(graphic: Graphics): it draws the segment on the screen using the Graphics library. It takes a Graphics object as an argument and calls its drawLine method to render a line from (startX, startY) to (endX, endY).

The modification only consists of implement a new method:

- + addFunction(name: String, program: Program): it creates a new instance of a Function with the parameters given. Then, add it to the dictionary. Its key is the name itself.

Then, the **LOGO** class has been modified to include new functionalities that enhance its ability to draw elements on the screen. This class is responsible for controlling the turtle's behaviour and drawing its movements.

The new attribute is:

- - segments: this is an ArrayList that stores all the line segments created by the turtle as it moves. Each segment represents a straight line.

Obviously, we have redesigned its constructor:

- + Logo(): it initialises the segments attribute as an empty list. Additionally, it sets the background dimensions and the colour using the setPreferredSize and setBackground methods from. Lastly, it puts “PEN”, “FWD”, “ROT” instructions in the dictionary as instances of subclasses of TurtleInstruction.

The new methods designed are:

- + addSegment(startX: int, startY: int, endX: int, endY: int): it creates a new Segment object using the provided starting and ending coordinates and adds it to the segments list.
- + paint(graphic: Graphics): this method overrides the paint method from JPanel. First, it calls the parent method. It then iterates through the segments list and uses the paint method of each Segment object to draw them on the screen. Finally, it paints the current position and orientation of the turtle on the screen.

To conclude, the **TURTLE** class has been modified to paint the final state of the turtle, moreover we need to redesign the moveForward method to add a segment related to the distance moved.

This class needs a reference to a Logo object in order to implement new methods:

- - logo: reference to a Logo object.

The new or modified methods have been designed as follows:

- + moveForward(distance: int): this modified method calculates the new position of the turtle based on its current angle and the specified distance. If the pen is activated, it creates a new line segment from the current position to the calculated position and adds it to the Logo object using its addSegment method. The turtle's position is then updated to the new coordinates.
- + paint(graphic: Graphics): it visually represents the turtle's current position and direction on the screen as a triangle. The triangle's tip and corners are calculated based on the turtle's current coordinates and angle. Three Segment objects are created to form the triangle and are then painted using their respective paint methods. This provides a clear graphical representation of the turtle's orientation and position.

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.

In the creation of this lab, various alternatives were considered in order to implement the graphical representation of the Logo programs. Below is an overview of such options, along with the reasoning behind the final choices that were made.

- **Handling Turtle Instructions:**
    - Instead of the hierarchy of subclasses for the TurtleInstruction class, there could be one class that has a type field that makes differences between the instruction types. For example: "PEN", "FWD", "ROT". That way, classes would be fewer, but the apply method would then become more complex because it would have to use if-else or switch-case statements based on the specific type of instruction.
    - Chosen Solution: The subclass-based approach was chosen in order to make use of a fundamental OOP concept: polymorphism. An abstract apply method defined in the TurtleInstruction class, and overridden in subclasses, results in

each instruction type encapsulating its behavior, hence giving rise to more maintainable and extensible code.

- **Representing Line Segments:**
    - Instead of using a separate Segment class to represent the line segments, its coordinates of the line could have been directly stored as primitives inside the Logo class or managed by another collection.
    - Chosen Solution: The Segment class was created to adhere to the Single Responsibility Principle of OOP. Each segment now encapsulates its data and behavior, like the ability to draw itself using the Graphics object; this reduces the complexity within the Logo class.

- **Turtle's Reference to Logo Administration:**
    - To manage the interaction between a turtle and the drawing canvas, alternatives provided were to directly pass a Graphics object into the Turtle class or use static methods that store segments globally.
    - Solution Adopted: The Turtle class was provided with a reference to the Logo instance. This maintains encapsulation and ensures that the turtle will interact with the drawing canvas in a controlled object-oriented manner.

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.

Actually, all went pretty much as it should, and we implemented satisfactorily all the requirements in the lab description. Tests were able to reveal that programs executed as expected with the hierarchy of TurtleInstruction and the Logo class managed drawing segments and the graphical presentation of the turtle very well. Listeners were responding accordingly, too, and so was the output with regards to drawing.

However, several problems appeared during the implementation process:

- **Coordinate Transformations not well understood:**
    - The translation of coordinates of the turtle into the coordinate system of the Graphical component was a big pain. This was particularly true in order to ensure that the turtle started in the middle of the screen and pointed upwards. Debugging that helped deepen the understanding of Java's Graphics library.

- **Turtle Rotation Implementation:**
    - Calculating the turtle representation at various angles of a triangle was confusing initially because it dealt with trigonometric functions and adjustments for the y-axis inversion of the graphical component. This required additional research and testing.

- **Object References and Design Decisions:**
    - Having the Logo instance's reference assigned directly to the Turtle class introduced a tight coupling, but within the scope of this lab, it was acceptable, since it kept the code simpler and maintained consistency in segments.