OOP – Laboratory 4 Report

GitHub Repo: LucaFranceschi01/UPF-POO22-G101-07

Introduction:

In this Lab we were asked to implement the needed classes to create and modify vectors and matrices. This has some interesting applications such as creating Frames (matrices that can represent images which can be black and white or colored) or creating Audio Buffers (vectors that can represent audio files). We have also implemented some methods to modify the values of each Vector/Matrix in a very precise way, such as changing the brightness of an image or change its RGB composition or changing the volume in an Audio Buffer.

We'll start by explaining the more meaningful aspects of our project.

Vector class:

The Vector class has two attributes: a list composed of real numbers and the dimension of that vector (the length of the list). There are some parts of that class that need to be clarified:

Constructor method: The constructor method of this class takes as a parameter the dimension of that vector and initializes the values attributes to a zero list.

Method *multiply*: Multiplies all the elements of the vector by a scalar that is passed as a parameter.

Method *sumElements*: Returns a real value that is the sum of all the elements of the vector.

Method *vectorSum*: Adds two vectors element by element and returns the Vector result.

Method *multiplyElements*: Very similar to the vectorSum method but multiplies instead of adding.

Method *matrixMultiply*: Changes the vector to the result of multiplying it by a matrix that fits (the operation cannot always be performed).

Method print: prints a text string to the terminal followed by the vector in a legible way.

Matrix class:

The Matrix class represents a matrix that is made of a collection of vectors. Then, the attributes of this class are an Array of vectors, and the row and column dimensions. We also apply some methods to modify the values of it in different ways. There are some methods that need to be clarified:

Constructor method: Creates a matrix of some desired size of rows and columns by creating an ArrayList of zero vectors (using the Vector constructor, which initializes all values to 0).

Method *multiply*: Multiplies all the elements by a scalar and returns the result, not modifying the value of the current matrix. Basically, it iterates over all the vectors and applies the method *Vector.multiply* to each one of them.

Method *create3DRotationZ*: Creates a 3x3 matrix that has the property of, when multiplied by a vector of dimension 3, rotates it α degrees (in radians, which is passed as a parameter).

Method *print*: prints a string followed by the matrix in a legible way.

Frame class:

The Frame class is an abstract class that extends the Matrix class, creating some abstract methods that will be implemented in some subclasses (such as *changeBrightness* and *getImageFromFrame*) and implementing some others that will simplify the project later. This is helpful for us because we can have the same API with different implementations for different types of Frame (the implementation for color frames is not the same as the one for black and white frame).

ColorFrame class:

The ColorFrame class extends the Frame class and implements the abstract methods changeBrightness and getImageFromFrame. There are some aspects of the code that may need to be clarified:

Constructor methods: We have overloaded the constructor method to be able to create a zero ColorFrame or a ColorFrame from a BufferedImage. That second constructor is a little bit more complex, since it needs to take the value of each pixel of the image and set it to the matrix.

Method *ChangeBrightness*: since a change of brightness is simulated changing the values of red, green and blue components by the same amount, we call the method changeRGB with the same parameter delta.

Method *getImageFromFrame*: from the ColorFrame, generates and returns a BufferedImage with the values of the matrix that defines it. We need to take every value of the matrix and set the pixel of the BufferedImage accordingly. This method helps us to test the classes and methods later in the testing.

Method *changeRGB*: in this method we iterate over all the matrix changing each value of the matrix by its corresponding value. For that, we need to convert the value to its corresponding RGB components and modify them. Then, we need to convert them back into a double value and store it in its position. Of course, we need to take into consideration that each one of the components range between 0 and 255, then we need to check if those values exceed the range in each step.

Method *toBWFrame*: returns a BWFrame from a ColorFrame. What we do is iterate all over the matrix taking the value of each element. Then we take the RGB components of that value, convert it to a grayscale value (for that, we use a method called: Colorimetric perceptual luminance-preserving conversion to grayscale). Finally, we assign that value to the corresponding element of the new BWFrame matrix.

Methods *valToRGB* and *RGBToVal*: this method was given in the problem statement. Converts a double value to the corresponding red, green and blue values by logic and boolean operations.

BWFrame class:

In a similar way to the one for ColorFrame, this class represents an image, but in this case the values correspond to the grayscale, instead of RGB. Due to this, the implementation of some methods vary, even though both classes that extend the Frame class. Below are some more explanations of the code:

Method changeBrightness: similar to the ColorFrame.changeBrightness method, modifies the value of the grayscale value to simulate an increase or decrease in brightness.

Method getImageFromFrame: similar to the *ColorFrame.getImageFromFrame*, creates and returns a BufferedImage, now in grayscale, with the same values of the Frame.

AudioBuffer class:

This class is a specification of the Vector class and represents an audio file through a vector.

TestGUI class:

This class implements a graphical user interface to test the classes and methods in ColorFrame and BWFrame. The most important aspect of this class is the constructor and the *actionPerformed* method.

Constructor method: in this method we create the frames from an actual image, a kitten, that we first have to extract from a file, and then create the frames and create the GUI (frame, labels, buttons).

Method *actionPerformed*: we provided an implementation for the interface ActionListener that we had to override. Basically, for each there is an action assigned to it, for instance that can be to call a changeBrightness method.

TestInheritance class:

This class contains the main method, which tests some of the methods for vectors and matrices and creates a TestGUI instance.

Alternative Solutions:

Our final solution decided to implement the changeBrightness and the changeRGB methods by doing the max of 255(max value for frames, corresponds to black) and the min of 0(min value for frames, corresponds to white) and the current value of the pixel plus a proportion of the max value(255) that was specified by delta. Thus, our solution was demanding a delta from the user as a form of value between 0 and 1, that we used to apply that percentage of 255 to each pixel. In the case of changeRGB, we allowed the user to change the percentage of one R, G, or B value at the time, by applying the method explained above to the specified colors with their corresponding deltas. We are aware that another possible solution could have been just applying the direct multiplication of values or the summation of these values, but we decided to apply this way of increasing and decreasing the brightness of the frame because we think is the most flexible out of the 3 options and allows us to test better the behavior of the image when we change its values. Below we will explain the most significant theory background concepts used to optimize the program and make it easier for the user to understand. The concepts we applied were mainly:

Abstraction and encapsulation. By applying abstraction and encapsulation, we are trying to focus only on the essential and hide the irrelevant things for the general public in order to reduce the complexity of the code for someone that is trying to understand it from the outside. In this lab, we applied this abstraction and encapsulation in some methods such as the *print* methods for Matrix and Vector, as well as for the constructor of the *TestGUI*, that had many operations inside it, but the user was not able to access them and could only interact with the display created by us.

Overriding. Basically, redefining an existing method in another class. In this lab, we had to override some methods for the *BWFrame* and the *ColorFrame* classes, as they had inherited some abstract methods from *Frame* that needed to be overridden. We also did override in the *TestGUI* class, as we had to use a method from the interface *ActionListener*.

Inheritance. Inheritance refers basically to the relation between two classes, in which one of them is the superclass (more general object) and the other one the subclass (more specific object). In this type

of relation, the subclass inherits all the attributes and methods of the superclass and can use them as their own. In this lab, we had the case of inheritance of the *Frame* class, where *Frame* was the superclass and *BWFrame*¸ *ColorFrame* were the subclasses of *Frame.* As said above, one of the advantages was that these subclasses could use the *protected* methods and attributes of the superclass *Frame.*

Abstract classes and methods. Abstract classes are classes that contain one or more abstract methods. Abstract methods are methods that are declared but not implemented in the class, so mainly they are intended to be overridden by the subclasses. So, by logic, abstract classes are meant to have subclasses, as they need their abstract methods to be overridden by other classes for them to work. In this lab, we used abstract classes and abstract methods mainly in the *Frame* class, which is defined as abstract because it contains at least one abstract method, and the abstract methods of the class are: *changeBrightness* and *getImageFromFrame*, which are both overridden in all subclasses.

Interfaces. An interface is something like a purely abstract class, as it only contains abstract methods, and it cannot contain any non-abstract attributes or methods. However, it can be implemented in some classes or not, it is not necessary for its existence to be implemented. If we implement an interface on a class, we need to override all the abstract method that this class contains. In this lab, we use interfaces for the creation of out *TestGUI,* as we needed the interface *ActionListener* to be able to perform actions when clicking a specific button.

Delegation. Delegation refers to calling other methods inside a method to do part of the job, is delegating the responsibility of doing a task to another object. In this lab, we used delegation several times, for example in the method *multiply* of Matrix class we use the *setVector* method to do a part of the task. Most of the methods inside Matrix delegate tasks to other methods of Matrix and Vector.

Use of keywords. In Java there are some special words called *keywords* that fulfill some specific needs and are very useful to do some tasks. In this lab we used for example the words: *super* (used to access the superclass for either the constructor, the attributes of the methods of it), *extends* (that is used for inheritance at the declaration of a class to denote that is a superclass of another class), *this* (that is used to refer to the current object in a method or constructor), *implements* (that is used to express that a given class is implementing the methods of a specific interface), abstract (that denotes that a method or a class is abstract), etc.

Conclusions:

Our solution ended up working well, as we were able to successfully create the two types of frames correctly, with their scales applied properly, and with all their methods working as they were supposed to. We were also able to create a GUI for the representation of the frames, that we extracted from a file, then passed it to Frame and then included it in the display of the GUI. We were also successful in the creation of buttons that could increase and decrease the brightness of both frames, alongside with some specific buttons for the ColorFrame one, that allowed us to increase or decrease a specific value for the R, G, or B values of the Frame. In relation with the first part of the lab, we were also successful in the creation of the Rotation Matrix for some angle. We didn't have any major doubts during this lab, although we found ourselves having to really explore the options out there to manage the scale of numbers for the types of frames and the ways of increasing the brightness or RGB values. However, the most challenging part of this lab for us was the creation of an interactive GUI from scratch.