Adrià Soria, Carlo Marcello

PRÁCTICA 4 OOP

1. INTRODUCCIÓN

La práctica 4 de Programación Orientada a Objetos consiste en la creación y desarrollo de varias clases, entre ellas *Matrix*, *Vector*, *AudioBuffer*, *Frame*, *BWFrame* y *ColorFrame*. Cuando se relacionan e implementan de forma eficiente y adecuada, respetando en todo momento los tipos de relaciones que existen entre ellas, podemos obtener un código final muy simplificado debido a las relaciones de herencia que existen en las clases y a la reutilización del código. Partiendo de que nuestras clases principales serán *Vector* y *Matrix*, podremos crear las demás clases para acabar generando un programa que pueda cargar una imágen y que mediante unos métodos podamos cambiarle el brillo de la imágen. Esto lo haremos teniendo en cuenta que una imágen es una matriz de píxeles y que si multiplicamos esta matriz por un vector de su tamaño podemos modificar el aspecto de dicha imagen. Para todas las clases previamente mencionadas deberemos crear e inicializar todos los métodos necesarios para que las clases se conecten entre ellas y tengan una funcionalidad en el programa. Para ello, nos hemos ayudado con el Lenguaje Unificado de Modelado (UML) que creamos en el seminario 4.

Tal y como hemos hecho en las tres últimas prácticas, hemos inicializado todos los atributos y métodos de cada una de las clases de nuestro programa. A continuación se muestra un ejemplo de una de las clases principales del programa:

```java
public class Matrix {

    //atributes

    private Vector[] values;
    private int row;
    private int col;


    //constructor
    public Matrix(int r, int c){
        this.row = r;
        this.col = c;

        this.values = new Vector[r];
        for (int i = 0; i<r; i++){
            this.values[i] = new Vector(c);
        }
    }
```

Aquí podemos observar la clase Matrix con sus tres atributos fila, columna y valores, que en este caso estos valores serán una array de tipo *Vector* debido a la relación de agregación que existe entre las clases *Matrix* y *Vector,* donde tenemos que una matriz está compuesta por varios vectores.
En el constructor podemos ver cómo se inicializan los valores del vector mediante un *for* que recorra todos los valores hasta el valor de filas que contenga la matriz.
Seguidamente también hemos creado los métodos *setters* y *getters* de la matriz que los necesitamos para la implementación de otras clases.

Por otro lado hemos hecho lo mismo con la clase *Vector* donde en este caso los valores son una lista de *double* y solo tiene el otro atributo de dimensión del vector. Para el constructor hemos llamado al método *zero()* que inicializa todos los valores del vector a cero. Aparte también hemos creado los métodos *setters* y *getters* del vector.

```java
public class Vector {
    //Atributes
    private double[] values;
    private int dim;

    //Constructor

    public Vector(int d){
        this.dim = d;
        this.values = new double[this.dim];
        zero();

    }
```

También hemos creado un método para multiplicar una matriz por un vector en la clase *Vector* ya que lo vamos a necesitar en otras clases para hacer este cambio de volumen o cambio de brillo que nos pide el enunciado. Este método lo podremos llamar en otras clases debido a la relación de herencia que tiene la clase *Vector* con las demás clases y haremos la reutilización de código pertinente para simplificar el programa. En este método podemos ver como le entra una matriz y creamos el vector v1. Primero debemos verificar que el vector se pueda multiplicar por la matriz ya que si la dimensión fila del vector no coincide con la dimensión columna de la matriz no se pueden multiplicar entre ellos. Una vez verificado esto debemos recorrer todas las filas y columnas de la matriz y multiplicarlos por el valor del vector que tenemos en esa misma posición. A continuación podemos observar el código que hemos implementado:

```java
public void matrixMultiply(Matrix m){
    Vector v1 = new Vector(m.getRow());
    if(dim == m.getCol()){
        double aux;
        for(int i = 0; i<m.getRow(); i++){
            aux = 0;
            for(int j=0;j<m.getCol();j++){
                aux = aux +m.get(i, j)*this.get(j);
            }
            v1.set(i,aux);
        }
    }else{
        System.out.print(s: "No se puede multiplicar un vector con tamaño diferente a la matriz");
    }

}
```

Seguidamente hemos creado las otras cuatro clases del programa como son *AudioBuffer*, *Frame*, *BWFrame* y *ColorFrame*, que debido a la relación de herencia que tenían estas clases con las dos clases de *Matrix* i *Vector,* hemos tenido que implementar pocos métodos para el funcionamiento de todas las clases.
Por ejemplo podemos ver la clase *Frame*, en la que hemos hecho un *super* de sus atributos para inicializarla en el constructor. También hemos creado un método abstracto que utilizaremos y llamaremos en las clases *BWFrame* y *ColorFrame:*

```java
public abstract class Frame extends Matrix{
    //Constructor

    public Frame(int n, int m){
        super(n, m);
    }

    public abstract void changeBrightness(double delta);
}
```

Seguidamente podemos observar una de las clases que hereda de la clase *Frame* como puede ser *ColorFrame* en la que podemos llamar tanto a los métodos de *Matrix* como a los de *Frame* debido a esta relación de herencia. A continuación vemos uno de los métodos principales del programa como puede ser *ChangeBrightness* en el que llamamos a funciones como *getCol* o *getRow* de la clase *Matrix* y recorremos toda la matriz de píxeles para multiplicarla por un *double* para así obtener una brillantez diferente en la imágen que carguemos en el *main* del programa:

```
public void changeBrightness(double delta){
    for(int i = 0; i< this.getRow(); i++){
        for(int j = 0; j<this.getCol();j++){
            int[] pixel = getRGB(i, j);
            pixel[0] = (int) (pixel[0] * delta);
            pixel[1] = (int) (pixel[0] * delta);
            pixel[2] = (int) (pixel[0] * delta);
            set(i, j, pixel[0],pixel[1],pixel[2]);
        }
    }
}
```

2. DESCRIPCIÓN DE POSIBLES ALTERNATIVAS Y RAZONAMIENTO DE NUESTRA DECISIÓN

A lo largo del desarrollo de la práctica han sucedido algunos casos donde hemos dudado de cómo implementar según que método y el funcionamiento de este, a continuación argumentamos las soluciones empleadas para cada uno de estos casos.

En primer lugar, para la clase *ColorFrame* tenemos un método llamado change brightness que dado un valor tipo *double* este se multiplica por cada uno de los componentes de cada pixel RGB con la finalidad de aumentar o disminuir la exposición de nuestro frame. En el caso que nuestro objetivo sea reducir el brillo, nuestro valor delta se definirá entre cero y uno. Por otro lado, la función encargada de dividir cada píxel de un frame en los tres componentes RGB(Red, Blue y Green) tiene como retorno una lista de tres enteros. Así pues, para evitar la incompatibilidad entre tipos del delta y cada uno de los componentes con los que queremos trabajar, primero efectuamos la multiplicación y acto seguido, convertimos el resultado a un entero para que este pueda ser devuelto esta vez a la función *changeRGB* y nuestro programa no de error.

```
public void changeBrightness(double delta){
    for(int i = 0; i< this.getRow(); i++){
        for(int j = 0; j<this.getCol();j++){
            int[] pixel = getRGB(i, j);
            pixel[0] = (int) (pixel[0] * delta);
            pixel[1] = (int) (pixel[0] * delta);
            pixel[2] = (int) (pixel[0] * delta);
            set(i, j, pixel[0],pixel[1],pixel[2]);
        }
    }
}
```

Otra alternativa al problema sería convertir el valor delta a entero antes de realizar el producto. Consecuentemente, en el caso de que queramos reducir la exposición, tras la conversión el delta decimal pasaría a valer cero, y el producto siempre nos resultaría en cero también. Por lo tanto, esta solución no nos interesa.

3. CONCLUSIÓN

Una vez terminado el programa estamos satisfechos con el resultado obtenido. Esta práctica la hemos encontrado bastante sencilla ya que teníamos que implementar en bastantes casos la relación de herencia que ya habíamos puesto en práctica en otros labs y ya teníamos bastante asumido como crear todas las clases con sus respectivos atributos y métodos.

Pese a haberle dedicado un gran cúmulo de tiempo a la parte opcional de la práctica, tan solo hemos logramos que el programa muestre la imagen de forma gráfica. Al aplicar cambios en el brillo o en la tonalidad del RGB la interfaz gráfica siempre retornaba una imagen completamente blanca. Creemos que puede haber incompatibilidad en la transformación de RGB a valor double ya que este siempre retorna un entero positivo.

En conclusión, llegados a este punto damos la práctica por concluida conscientes de que, durante el desarrollo de ésta, hemos logrado mejorar en el entorno ‘java’ y a su vez familiarizarnos un poco más con el paradigma de programación orientado a objetos el cual hace unas semanas apenas éramos conscientes de su existencia.