



UNIVERSIDAD AUTÓNOMA DE ZACATECAS

---

## Practica: Arreglos multidimensionales.

---

*Estudiante:*

José Francisco Hurtado Muro

*Profesor:*

Dr. Aldonso Becerra Sánchez

## Tabla de Contenidos

<b>1</b>	<b>Introducción</b>	<b>2</b>
<b>2</b>	<b>Actividades:</b>	<b>3</b>
2.1	Actividad 1: . . . . .	3
2.2	Actividad 2: . . . . .	3
2.3	Actividad inicial: . . . . .	3
2.3.1	TDA . . . . .	5
2.3.2	Extraer color de cada pixel . . . . .	6
2.3.3	Escala de grises . . . . .	9
2.3.4	Modificar Brillo . . . . .	10
2.3.5	Invertir imagen eje X . . . . .	12
2.3.6	Invertir imagen eje Y . . . . .	13
2.3.7	Ejecución de operación transpuesta . . . . .	15
2.3.8	Agregar marco a la Imagen . . . . .	16
2.3.9	Transparencia . . . . .	17
<b>3</b>	<b>Código Agregado - UML</b>	<b>18</b>
<b>4</b>	<b>Pre-evaluación del Alumno</b>	<b>19</b>
<b>5</b>	<b>Conclusión</b>	<b>19</b>
<b>6</b>	<b>Referencias:</b>	<b>19</b>

## 1 Introducción

"La facilidad que los arreglos multidimensionales tienen para permitir guardar datos en forma de columnas/renglones, los hace pertinentes para la resolución de muchos problemas donde se requiere esta situación. El único detalle con esta cuestión radica en que es poco flexible el número de elementos que podemos manipular, ya que se requiere conocer a priori la cantidad de elementos a guardar. Adicionalmente se incrustan los usos de registros para darle una funcionalidad más completa al procedimiento de control"

## 2 Actividades:

### 2.1 Actividad 1:

Lea primero toda la práctica. No inicie a programar sin leer todo cuidadosamente primero. Recuerde que debe generar el reporte en formato IDC.

### 2.2 Actividad 2:

Primero genere la Introducción.

### 2.3 Actividad inicial:

Las propiedades del color pueden ser definidas matemáticamente usando modelos de color, los cuales describen los colores que vemos y con los que trabajamos. Cada modelo de color representa un método diferente de descripción y clasificación del color, pero todos utilizan valores numéricos para representar el espectro visible. Existen muchos modelos, uno de los principales es RGB (canales red, green, blue). Si ha trabajado con un programa de edición de imágenes, quizás les resulte familiar otros modelos como Escala de grises. En RGB, además de los tres colores base se puede tener el canal alfa. En computación gráfica, la composición alfa o canal alfa es la que define la opacidad de un píxel en una imagen. El canal alfa actúa como una máscara de transparencia que permite, de forma virtual, componer (mezclar capas) imágenes o fondos opacos con imágenes con un cierto grado de transparencia.

Para indicar con qué proporción es mezclado cada color, se asigna un valor a cada uno de los colores primarios, de manera que el valor "0" significa que no interviene en la mezcla y, a medida que ese valor aumenta, se entiende que aporta más intensidad a la mezcla. Aunque el intervalo de valores podría ser cualquiera (valores reales entre 0 y 1, valores enteros entre 0 y 37, etc.), es frecuente que cada color primario se codifique con un byte (8 bits). Así, de manera usual, la intensidad de cada una de las componentes se mide según una escala que va del 0 al 255 y cada color es definido por un conjunto de valores escritos entre paréntesis (correspondientes a valores "R", "G" y "B", y en su caso alfa) y separados por comas.

De este modo, el rojo se obtiene con (255, 0, 0), el verde con (0, 255, 0) y el azul con (0, 0, 255), obteniendo, en cada caso un color resultante monocromático. La ausencia de color, es decir el color negro, se obtiene cuando las tres componentes son 0: (0, 0, 0). La combinación de dos colores a su máximo valor de 255 con un tercero con valor 0 da lugar a tres colores intermedios. De esta forma, aparecen los colores amarillo (255, 255, 0), cian (0, 255, 255) y magenta (255, 0, 255). El color blanco se forma con los tres colores primarios a su máximo valor (255, 255, 255). Adicionalmente a estos valores, se puede obtener el canal alfa, como cuarto elemento.

Bajo este supuesto una imagen de 800x600 está compuesta por 800 pixeles de ancho por 600 pixeles de alto. Donde cada pixel es representado por un byte por cada canal, si se tiene 3 canales (RGB) se disponen de 24 bits, si se tienen 4 canales (alfa y RGB) se disponen de 32 bits en la configuración de un pixel. En otras palabras un pixel se puede ver así:

La **profundidad de bits** es determinada por la cantidad de bits utilizados para definir cada píxel. Cuanto mayor sea la profundidad de bits, tanto mayor será la cantidad de tonos (escala de grises o color) que puedan ser representados. Las imágenes digitales se pueden producir en blanco y negro (en forma bitonal), a escala de grises o a color. ¿Cuántos colores puede tener un pixel? De todos es sabido que los archivos y dispositivos digitales almacenan números, concretamente números en sistema binario, es decir ceros y unos, nada más. El color que queremos mostrar en un determinado pixel lo representamos pues con un número. ¿cuántos colores podremos mostrar? Depende de la cantidad de información asociada con cada pixel.



- **1 bit:** 0 1. El ejemplo más sencillo sería este: utilizar 1 bit para la información de color. Un bit puede valer 0 o 1. Por tanto sólo podremos representar 2 tonos. Suele utilizarse 0 para el blanco y 1 para el negro.

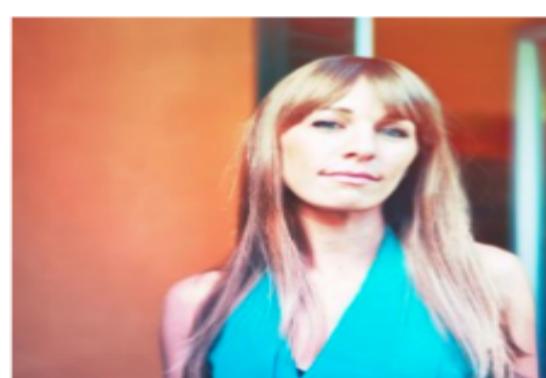


- **2 bits:** 00 10 01 11, con dos bits podemos representar 4 tonos diferentes.

- **3 bits:** 000 001 010 100 110 101 011 111, es decir 8 colores diferentes.

- **4 bits:** 16 colores.

- **8 bits:** 256 colores. Esta es la profundidad de color habitual en las imágenes de "escala de grises": los píxeles pueden tomar uno entre 256 tonos de gris.



- **24 bits:** 16,7 millones de colores. El estándar para imágenes en color es una profundidad de 24 bits (8 bits por canal RGB: 256 tonos de Rojo, Verde y Azul). Esto permite  $256 \times 256 \times 256 = 16.777.216$  colores. Se denominan imágenes en **color verdadero** (true color).

Partiendo de un tipo de dato (en un lenguaje de programación) que permita guardar 32 bits (int, por ejemplo), los primeros 8 bits del lado izquierdo representan al canal alfa (opacidad), los siguientes 8 bits representan el color rojo, los siguientes 8 bits representan el color verde, los siguientes 8 bits (lo más a la derecha) representan el color azul.

<b>Alfa</b>	<b>Red</b>	<b>Green</b>	<b>Blue</b>
8 bits	8 bits	8 bits	8 bits
01010101	00001111	01011100	00001110

Se le pide que realice lo siguiente:

### 2.3.1 TDA

Se proporciona un código de programa (en la parte inferior de este documento) que lee una imagen indicada como ruta. El método `Imagen.getRGB(i, j)` obtiene un número de 32 bits (un pixel) representando los 4 canales mencionados (alfa, r, g, b). Se le pide que guarde cada pixel de la imagen en un TDA, donde las dimensiones requeridas son posiblemente obtenerlas con base en las dimensiones de la imagen (`Imagen.getWidth()` y `Imagen.getHeight()`).

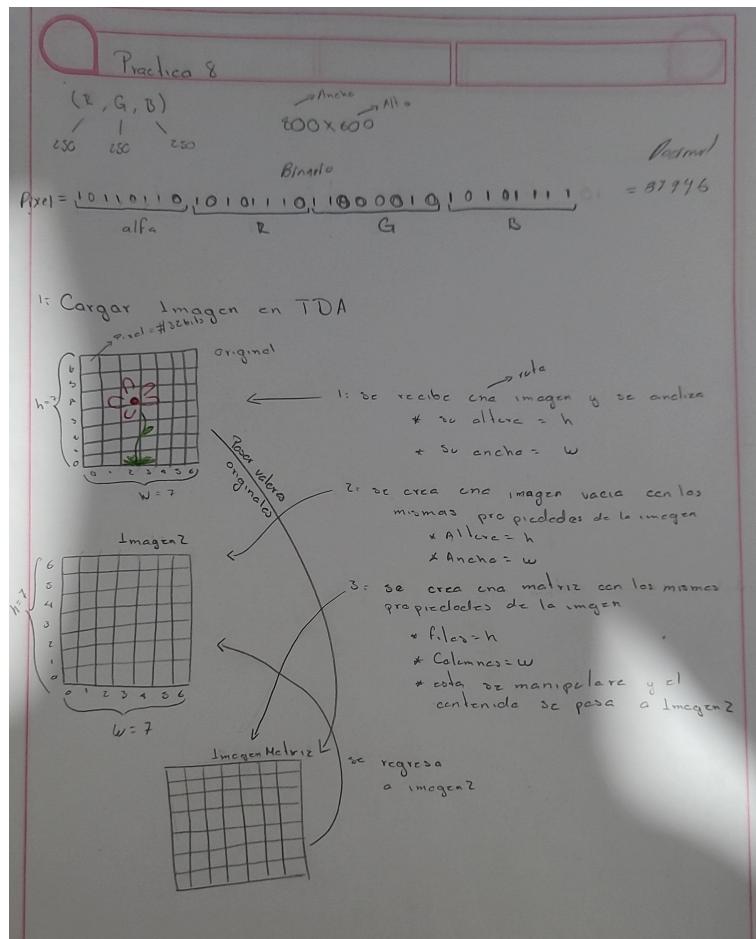
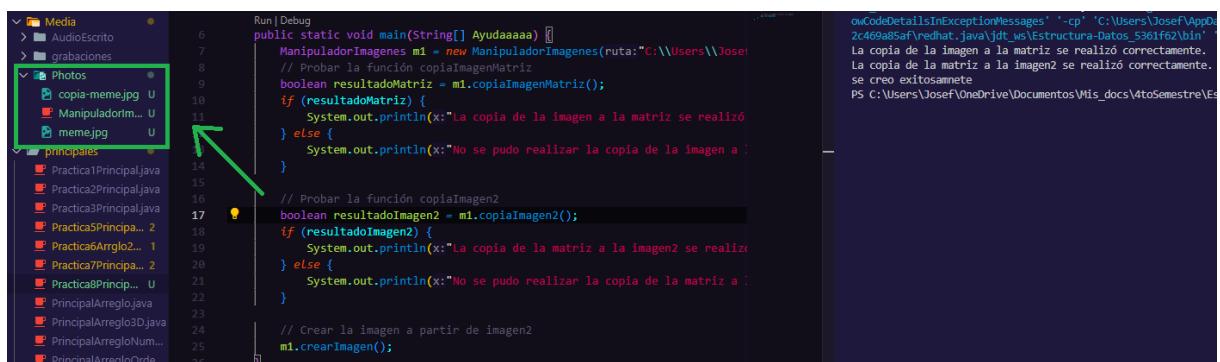


Figure 1: Análisis: TDA

## Explicación

Se planteo que el TDA de manera natural en el constructor, reciba una imagen por medio de una ruta dada, y este la dividirá en pixeles, los cuales habrá dos posiciones de ellos **width = w** que presenta el ancho de la imagen, también esta **height = h** que es lo alto de la imagen, una vez con estos valores dados se crean dos cosas sumamente importantes, **BufferedImage imagen2** que sera nuestra imagen resultado, en esta se pondrá todos los cambios o manipulaciones que hagamos en **Arreglo2DNumerico imagenMatriz** que como ya mencione en esta ultima se harán todas las manipulaciones, ya que este es una copia directa a la imagen original, y se deberia de ver algo como en la figura 2. ademas de esos metodos se tomo en consideración el método auxiliar **crearImagen** que lo que hace es tomar todo de **BufferedImage imagen2** y volver a transformarlo en una imagen, mismo que vemos en la figura 2.



```

    public static void main(String[] Ayudaaaaa) {
        ManipuladorImagenes m1 = new ManipuladorImagenes(ruta:"C:\\Users\\Josef\\AppData\\Local\\Temp\\image.png");
        // Probar la función copiaImagenMatriz
        boolean resultadoMatriz = m1.copiaImagenMatriz();
        if (resultadoMatriz) {
            System.out.println(x:"La copia de la imagen a la matriz se realizó correctamente.");
        } else {
            System.out.println(x:"No se pudo realizar la copia de la imagen a la matriz.");
        }
        // Probar la función copiaImagen2
        boolean resultadoImagen2 = m1.copiaImagen2();
        if (resultadoImagen2) {
            System.out.println(x:"La copia de la matriz a la imagen2 se realizó correctamente.");
        } else {
            System.out.println(x:"No se pudo realizar la copia de la matriz a la imagen2.");
        }
        // Crear la imagen a partir de imagen2
        m1.crearImagen();
    }
}

```

Figure 2: Funcionamiento: TDA

### 2.3.2 Extraer color de cada pixel

Extraiga de cada pixel (número de 32 bits) los correspondientes valores de cada canal en una variable separada para cada uno. Utilice operaciones a nivel de bits, como corrimientos hacia la derecha o izquierda para manipular los bits que se requieran (inclusive se pueden utilizar AND o OR lógicos según sea el caso). Utilice su raciocinio para deducir estos corrimientos con base en los 32 bits que vienen indicados más arriba. Recuerde que un número entero en java es de 32 bits, y que un pixel utiliza los 32 bits para guardar su valor. De esta manera se requiere por ejemplo, para extraer el Green, que sus 8 bits se localicen hasta la derecha de los 32 bits (se recorrieron 8 bits hacia la derecha), y que todos los demás bits sean ceros (los 24 de la izquierda). Así se tiene guardado el color Green en los primeros 8 bits una vez que se ha movido el conjunto de bits. Este valor está guardado en un entero de 32 bits, pero representando solo los bits de ese color (Green, por ejemplo). Este proceso lo tiene que hacer para extraer los demás colores por individual.

Por ejemplo, para obtener en una variable entera el color **blue**, se debería tener algo así (int blue):

blue			
8 bits	8 bits	8 bits	8 bits
00000000	00000000	00000000	00001110

Por ejemplo, para obtener en una variable entera el color **green**, se debería tener algo así (int green):

green			
8 bits	8 bits	8 bits	8 bits
00000000	00000000	00000000	01011100

Por ejemplo, para obtener en una variable entera el color **red**, se debería tener algo así (int red):

red			
8 bits	8 bits	8 bits	8 bits
00000000	00000000	00000000	00001111

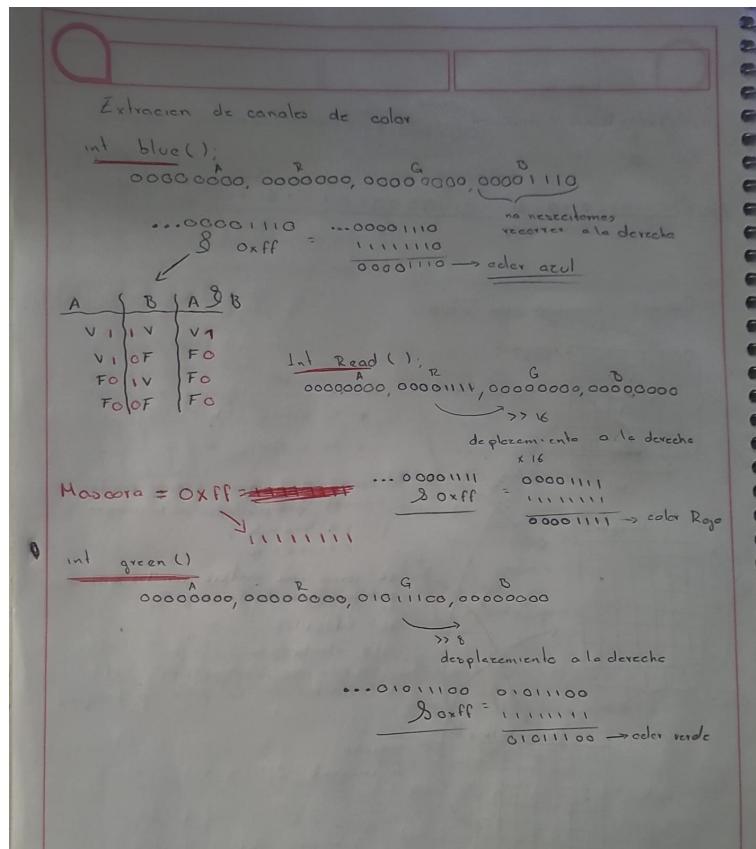
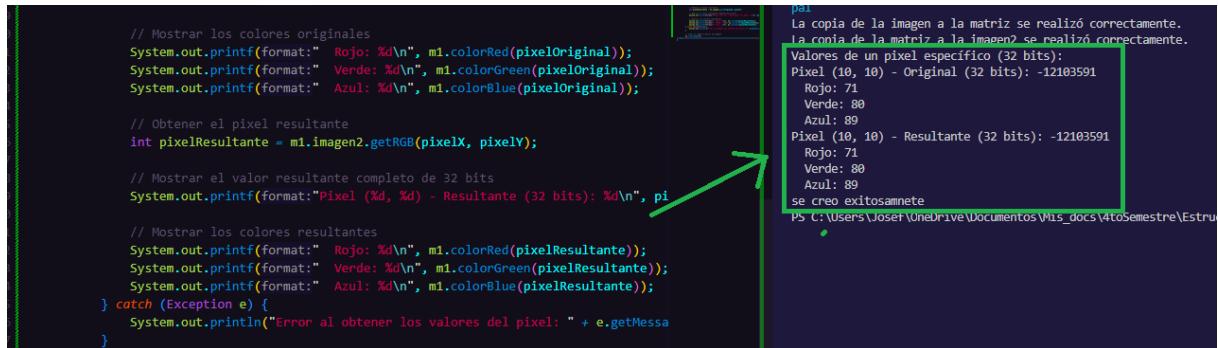


Figure 3: Análisis: Extraer color de cada pixel

### Explicación

En este caso se tomo en cuenta las "**Operaciones a nivel de bits**", donde se destacan 2 operaciones que nos sirvieron para este método, **Desplazamiento a la derecha** y **AND** donde con el desplazamiento a la derecha lo que se consigue es mover los bits hacia la derecha, insertando 0 por la izquierda y según el color que queramos obtener es cuantos espacios tenemos que mover, por ejemplo: rojo »16, verde »8 y azul»0, pero aparte para solo dejar el puro valor que representa a ese color se le aplico lo que se conoce como mascara donde la mascara nos ayuda a ocultar datos que no queremos ver y para este caso usaremos un como mascara **0xff** que visto de manera binaria se vería como **0xff = 1111 1111**, que junto con la operación **AND** mostraría solo aquellos donde se coincide en verdad and verdad o en este caso 1 and 1, esto aplicado en un pixel tendríamos como resultado valores en un rango de 0 a 255 como podemos ver en la figura 4



```

// Mostrar los colores originales
System.out.printf(" Rojo: %d\n", m1.colorRed(pixelOriginal));
System.out.printf(" Verde: %d\n", m1.colorGreen(pixelOriginal));
System.out.printf(" Azul: %d\n", m1.colorBlue(pixelOriginal));

// Obtener el pixel resultante
int pixelResultante = m1.imagen2.getRGB(pixelX, pixelY);

// Mostrar el valor resultante completo de 32 bits
System.out.printf("Pixel (%d, %d) - Resultante (32 bits): %d\n", pi

// Mostrar los colores resultantes
System.out.printf(" Rojo: %d\n", m1.colorRed(pixelResultante));
System.out.printf(" Verde: %d\n", m1.colorGreen(pixelResultante));
System.out.printf(" Azul: %d\n", m1.colorBlue(pixelResultante));
} catch (Exception e) {
    System.out.println("Error al obtener los valores del pixel: " + e.getMessage());
}
    
```

La copia de la imagen a la matriz se realizó correctamente.  
 La copia de la matriz a la imagen se realizó correctamente.  
 Valores de un pixel específico (32 bits):  
 Pixel (10, 10) - Original (32 bits): -12103591  
 Rojo: 71  
 Verde: 88  
 Azul: 89  
 Pixel (10, 10) - Resultante (32 bits): -12103591  
 Rojo: 71  
 Verde: 88  
 Azul: 89  
 se creo exitosamente  
 PS C:\Users\Josef\OneDrive\Documentos\WIS\_docs\4toSemestre\Estru

Figure 4: Funcionamiento: Extraer color de cada pixel

Este método fue posible entender y tener conocimiento para el resto de la práctica gracias al canal de **Jose Colbes** quien tiene un temario completo de operaciones a nivel de bits, donde explica el funcionamiento de estas operaciones, así como ejemplos de su funcionamiento y ejemplos básicos sobre estas operaciones, quien ademas de eso explica cierta teoría y herramientas que se pueden utilizar en estas operaciones, como vemos en la figura 5.



**Operaciones a nivel de bits**

Un ejemplo del operador binario OR es el siguiente:

$x = x | \text{SET\_ON};$

que asigna 1 a los bits en  $x$  que están como 1 en  $\text{SET\_ON}$ .

a	b	a b
0	0	0
0	1	1
1	0	1
1	1	1

El operador binario XOR asigna un 1 en cada posición de bit en donde sus operandos tienen diferentes bits, y cero cuando son los mismos.

10:52

Figure 5: Mención honorifica: <https://www.youtube.com/watch?v=Cdv8oa9Br6Q>

### 2.3.3 Escala de grises

Tome como base una imagen jpg de su elección, y pase la imagen a escala de grises. Este proceso se obtiene promediando los valores individuales de cada canal. El resultado del promedio será el nuevo valor que tendrá cada canal (el mismo valor para cada canal). Con estos valores se debe formar el nuevo valor del pixel de 32 bits (con la posición de bits que le corresponde a cada canal). Este proceso se obtiene nuevamente usando operaciones a nivel de bits. El proceso siguiente es pasar cada pixel a la imagen que se va a grabar con el método `Imagen.setRGB(i, j, valorPixel32bits)`. El resultado de esta imagen ya se puede guardar en un archivo nuevo.

NOTA: la siguiente línea crea una nueva imagen en blanco de tamaño w, h: `BufferedImage`  
`Imagen2 = new BufferedImage(w, h, BufferedImage.TYPE_INT_RGB);`

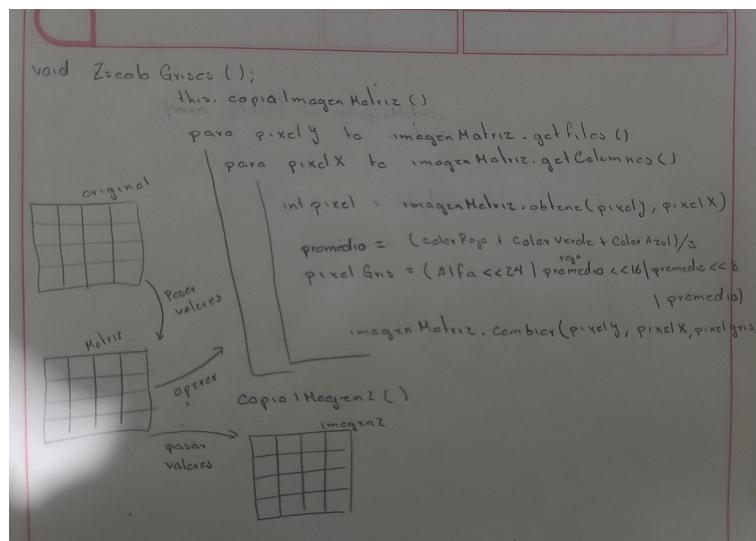


Figure 6: Análisis: Escala de grises

### Explicación

Para este caso lo que se hizo fue pasar los pixeles de la `ImagenOriginal` a una `ImagenMatriz`, la cual a cada pixel se le descompone en sus colores, para después entre los 3 sacarles un valor promedio que este sera el nuevo valor de RGB, alfa se mantiene exactamente igual, y este valor se tiene que reconstruir a 32 bits que en este caso se hizo por medio de **Desplazamiento a la izquierda**, y con el operador **OR** se fue conectando los bits, que esto lo que hace es generar un pixel gris, para al final copiar la `ImagenMatriz` en una `Imagen2` que es el resultado, el resultado podemos verlo en la figura 7.

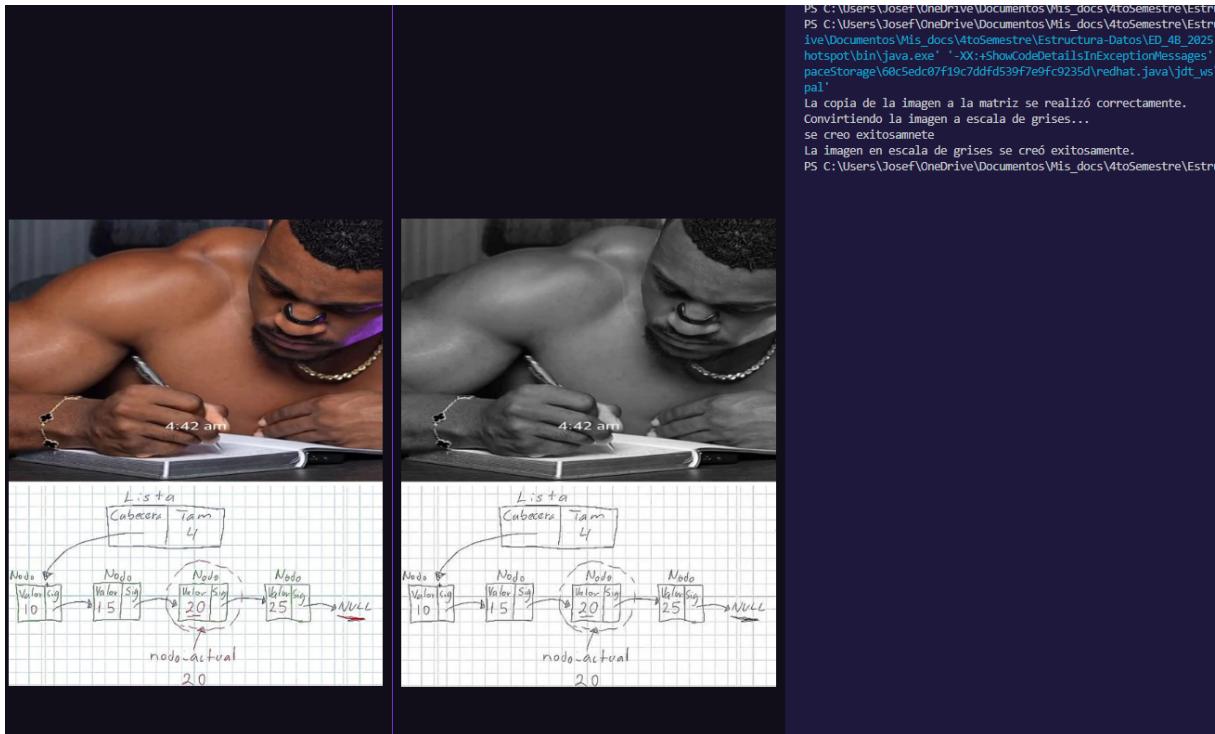


Figure 7: Funcionamiento: Escala de grises

### 2.3.4 Modificar Brillo

Ahora se le pide que modifique el brillo de la imagen (en la imagen de escala de grises). Este proceso se obtiene sumando o restando un valor numérico a cada pixel por igual.



```

void brillo (int brillo)
    para pixely to imagenMatriz.getFilas()
        para pixelx to imagenMatriz.getColumnas()
            pixel = imagenMatriz.obtener(pixely, pixelx)
            int alfa = colorAlfa(pixel)
            int gris = colorBlue(pixel)
            int nuevoGris = gris + brillo
            si (nuevoGris <= 0) {nuevoGris = 0} //que el brillo no sea negativo
            si (nuevoGris > 255) {nuevoGris = 255} //que el brillo no sea mas que 255
            pixel.Gris = (alfa << 24) | nuevoGris << 8 | nuevoGris
            imagenMatriz.combinaly(x, nuevoGris) | nuevo gris)
    capturarImagen()

```

Figure 8: Análisis: Modificar Brillo

### Explicación

Para este caso lo que se hizo fue que a la copia existente que se genero cuando se hizo la escala de grises fuera sacando pixel por pixel, al cual se le saca el alfa y se mantiene guardado, y se obtiene cualquier color del RGB ya que el color gris se obtiene cuando los 3 valores del RGB son el mismo, entonces una vez con este dato, le sumamos un valor dado por el usuario que representa el brillo, si es negativo esta se podrá mas oscuro el pixel, pero si es positivo se ira poniendo mas claro, pero para esto se valida que los valores estén dentro del rango de 0 a 255, donde si es menor que cero pues el valor del gris se queda en en cero y si es mayor que 255 , se queda en 255, una vez validado el nuevo color de gris, se convierte a valor de 32 bits el cual se regresa a la misma posición de la matriz, una vez con los datos con un nuevo gris, se regresa los valores a imagen2 y el resultado es el que podemos ver en la figura 9.

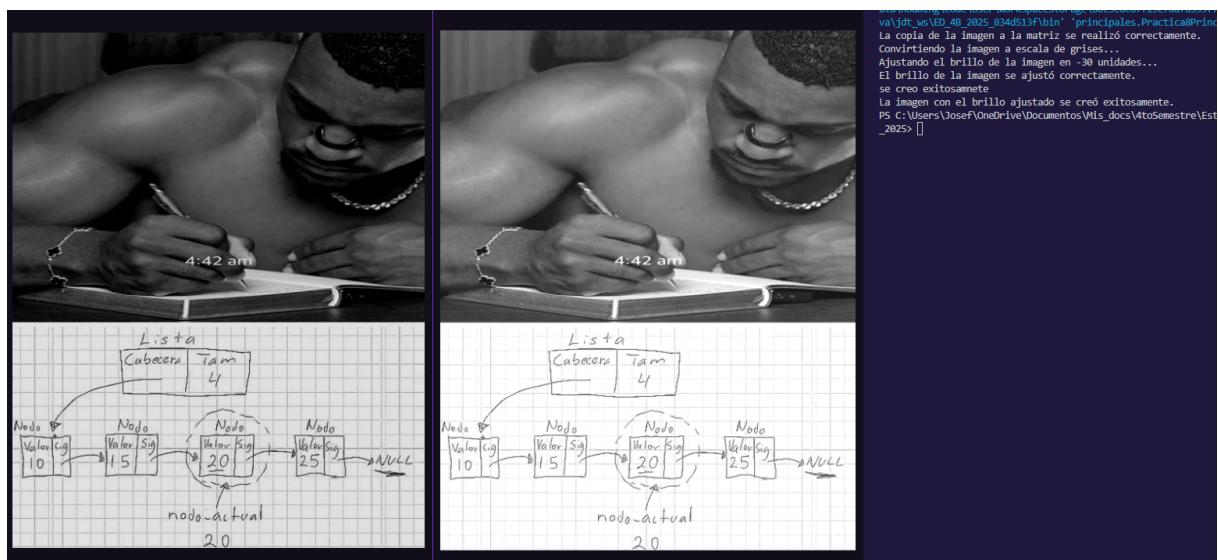


Figure 9: Funcionamiento: Modificar Brillo

### 2.3.5 Invertir imagen eje X

Tome como base la misma imagen y ahora invierta la imagen en posición horizontal.

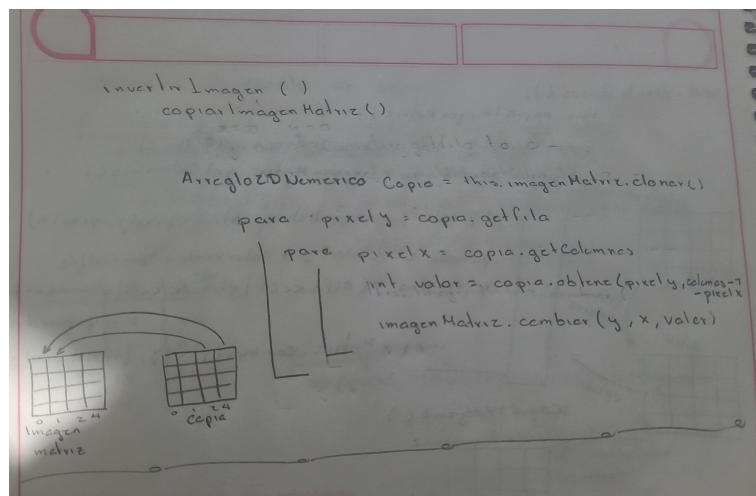
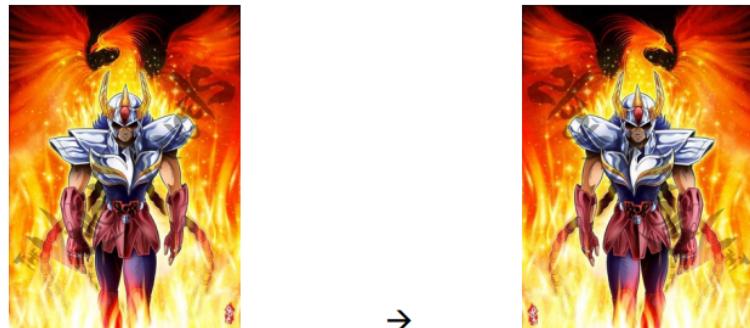


Figure 10: Análisis: Invertir imagen eje X

### Explicación

Aquí lo que se tuvo que hacer es volver a copiar los valores originales de la imagen matriz, después se hizo una copia de la matriz, donde recorremos y vamos sacando los datos desde el final, donde, al número de columnas se le quita uno ya que se inicia en cero, pero además se le resta la posición del eje x, esto para obtener la posición en la matriz original donde se inserta el el valor, y en la figura 11 podemos ver el resultado.

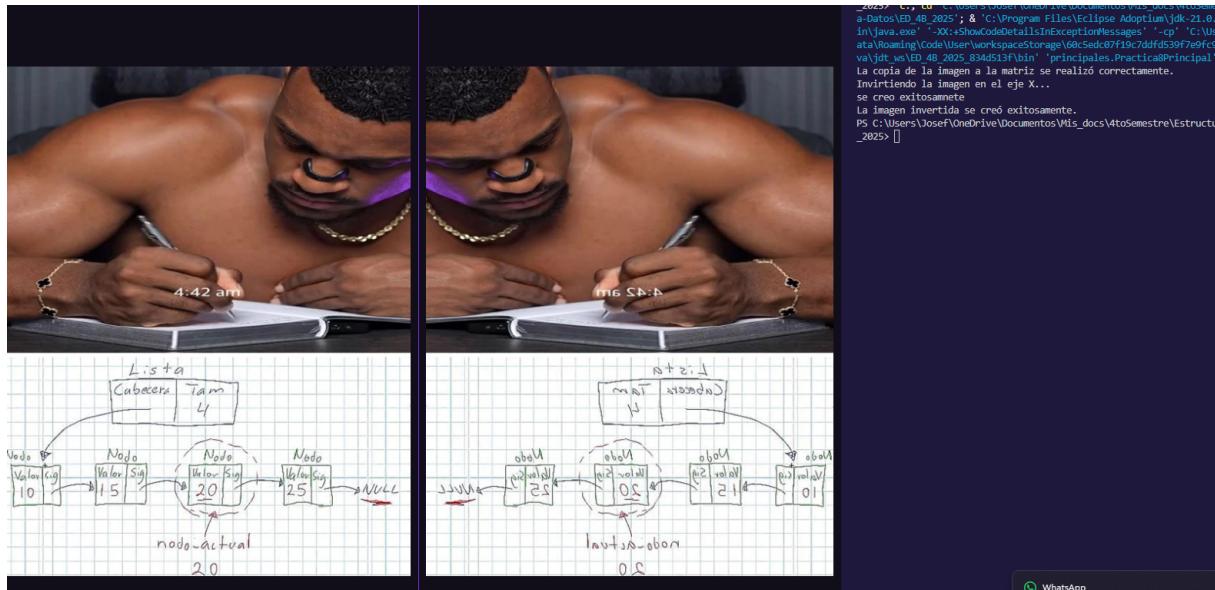


Figure 11: Funcionamiento: Invertir imagen eje X

### 2.3.6 Invertir imagen eje Y

Tome como base la misma imagen y ahora invierta la imagen en posición vertical

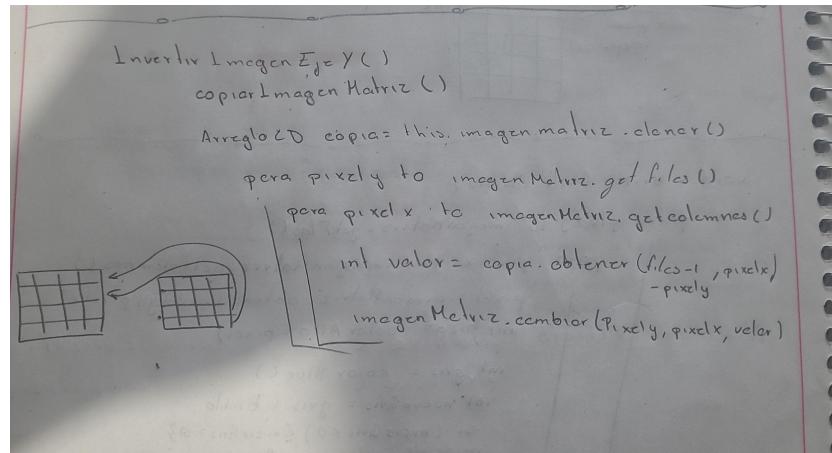


Figure 12: Análisis: Invertir imagen eje Y

### Explicación

Aquí lo que se tuvo que hacer es volver a copiar los valores originales de la imagen matriz, después se hizo una copia de la matriz, donde recorremos y vamos sacando los datos desde el final, donde, al numero de filas se le quita uno ya que se inicia en cero, pero además se le resta la posición del eje y, esto para obtener la posición en la matriz original donde se inserta el el valor, y en la figura 13 podemos ver el resultado.

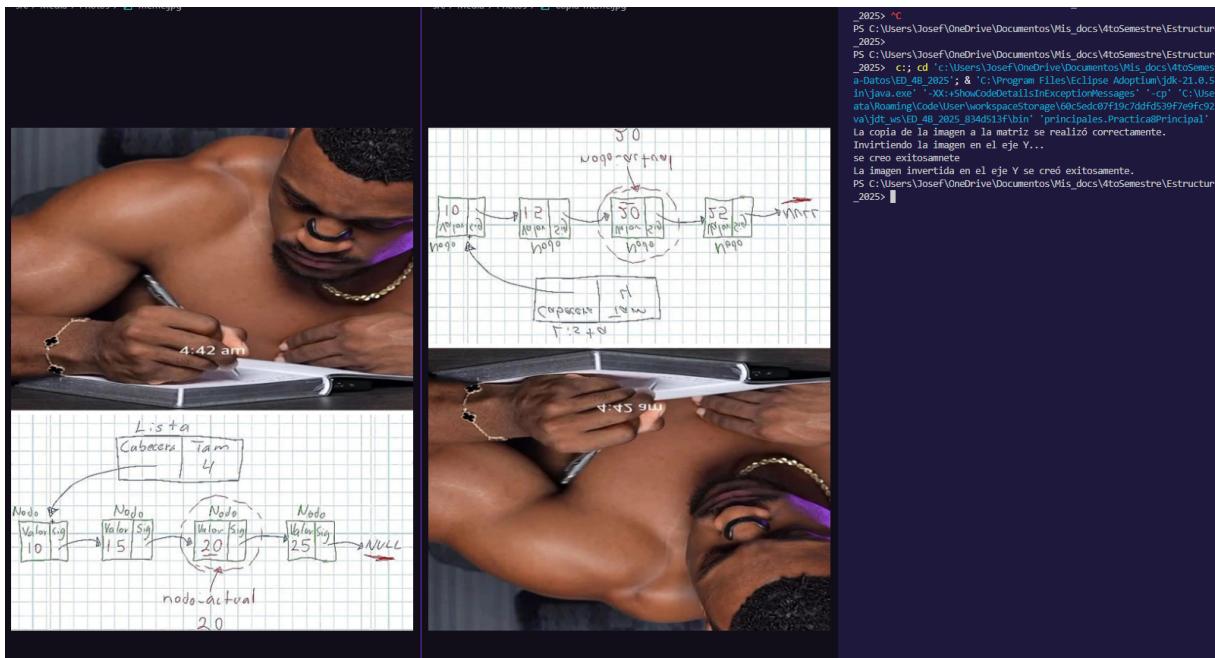


Figure 13: Funcionamiento: Invertir imagen eje Y

### 2.3.7 Ejecución de operación transpuesta

Si utiliza las operaciones que se tienen en el TDA matriz, y ejecuta la operación transpuesta sobre la imagen usada de ejemplo, qué obtendría. Muestre el resultado y ejecute ese proceso

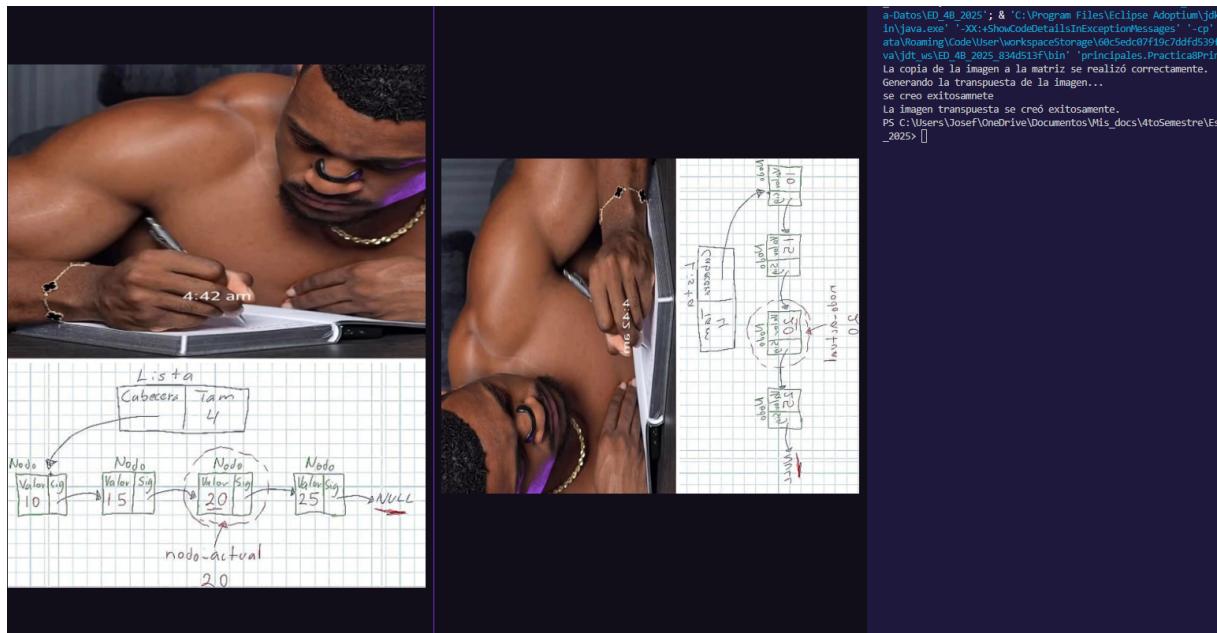


Figure 14: Funcionamiento: Ejecución de operación transpuesta

### Explicación

En esta parte se uso funciones que ya existen el el TDA Arreglo2DNumérico que es transpuesta, que lo que haces es que la matriz la convierte en transpuesta pero para esto se tuvo que hacer dos variaciones de a métodos ya realizados, **boolean copiarAMatriz(BufferedImage image)** que ahora recibe de tipo BufferedImage image doonde se van a guardar los datos de la matriz y también **void crearImagen(BufferedImage image)** que igual recibe un BufferedImage image que es de donde va a sacar los elementos para crear la nueva imagen, como podemos ver en la figura 15.

### 2.3.8 Agregar marco a la Imagen

Se le pide que le haga un marco a la imagen sin robar espacio de la propia imagen (el usuario debe elegir un color y el grosor), por ejemplo:

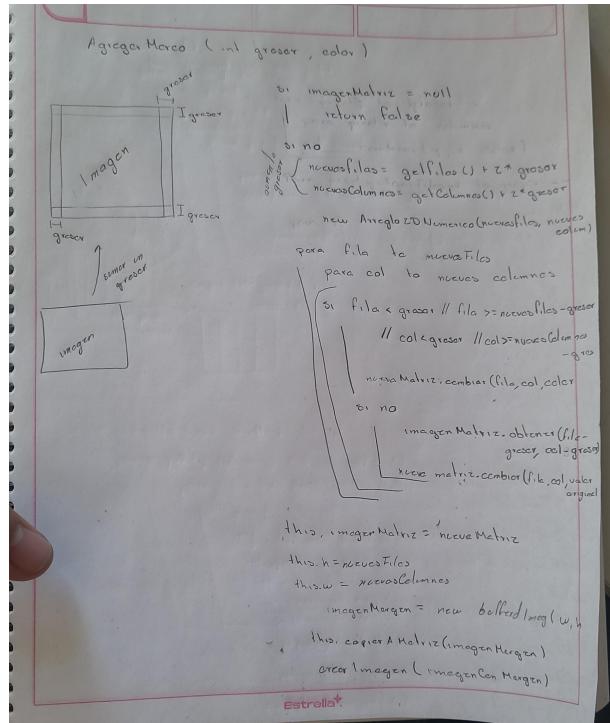


Figure 15: Análisis: Agregar marco a la Imagen

### Explicación

Para este metodo lo que se hizo crear fue crear una matriz nueva con el tamaño original mas el grosor por 2 del grosor dado por el usuario, donde cada se recorre ese rango de agregando el color color, pero si no pertenece al rango del margen, agrega pixeles de la imagen original, una vez que este proceso acabe se le pasara estos valores a una imagen nueva la cual sera creada como podemos ver en la figura 16.

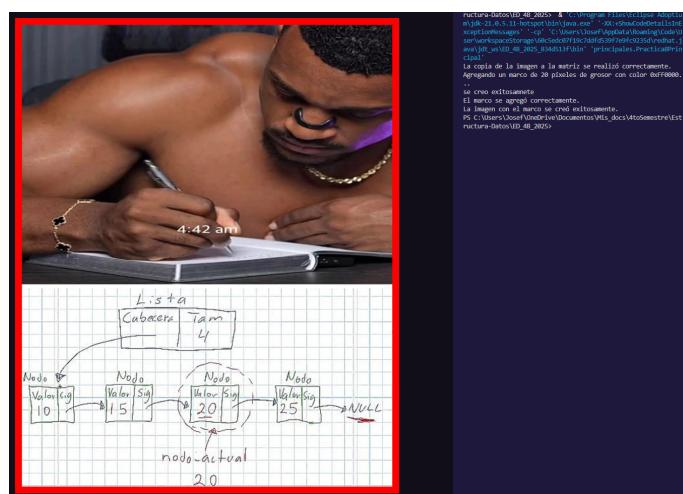


Figure 16: Funcionamiento: Agregar marco a la Imagen

### 2.3.9 Transparencia

Se le pide que invente un nuevo método a partir de los ejemplos de arriba, de tal manera que usted indique qué operación haría y cómo quedaría el resultado. Por obvias razones debe ser un método nuevo que manipule la imagen de alguna manera diferente a las ya mostradas arriba.

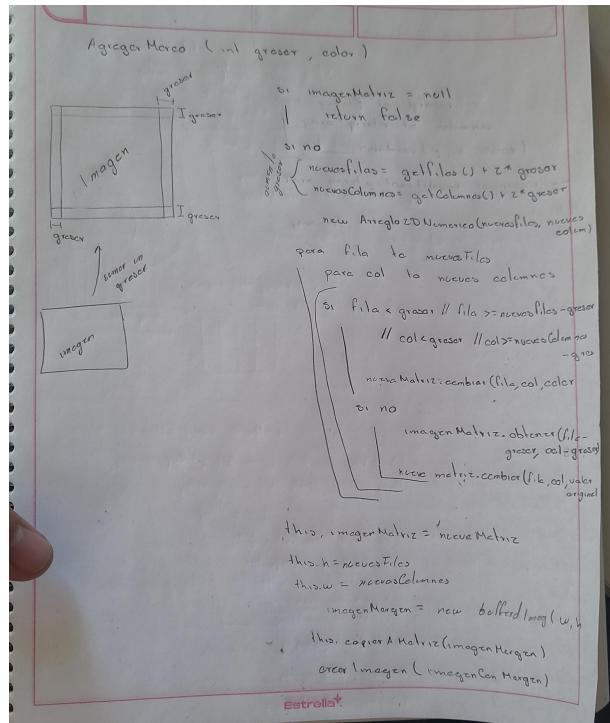


Figure 17: Análisis: Transparencia

### Explicación

Para este método lo que se planteo fue que el alfa cambie, el nivel de transparencia según un valor dado por el usuario, lo cual regresaría una imagen con una opacidad menor o mayor

### 3 Código Agregado - UML

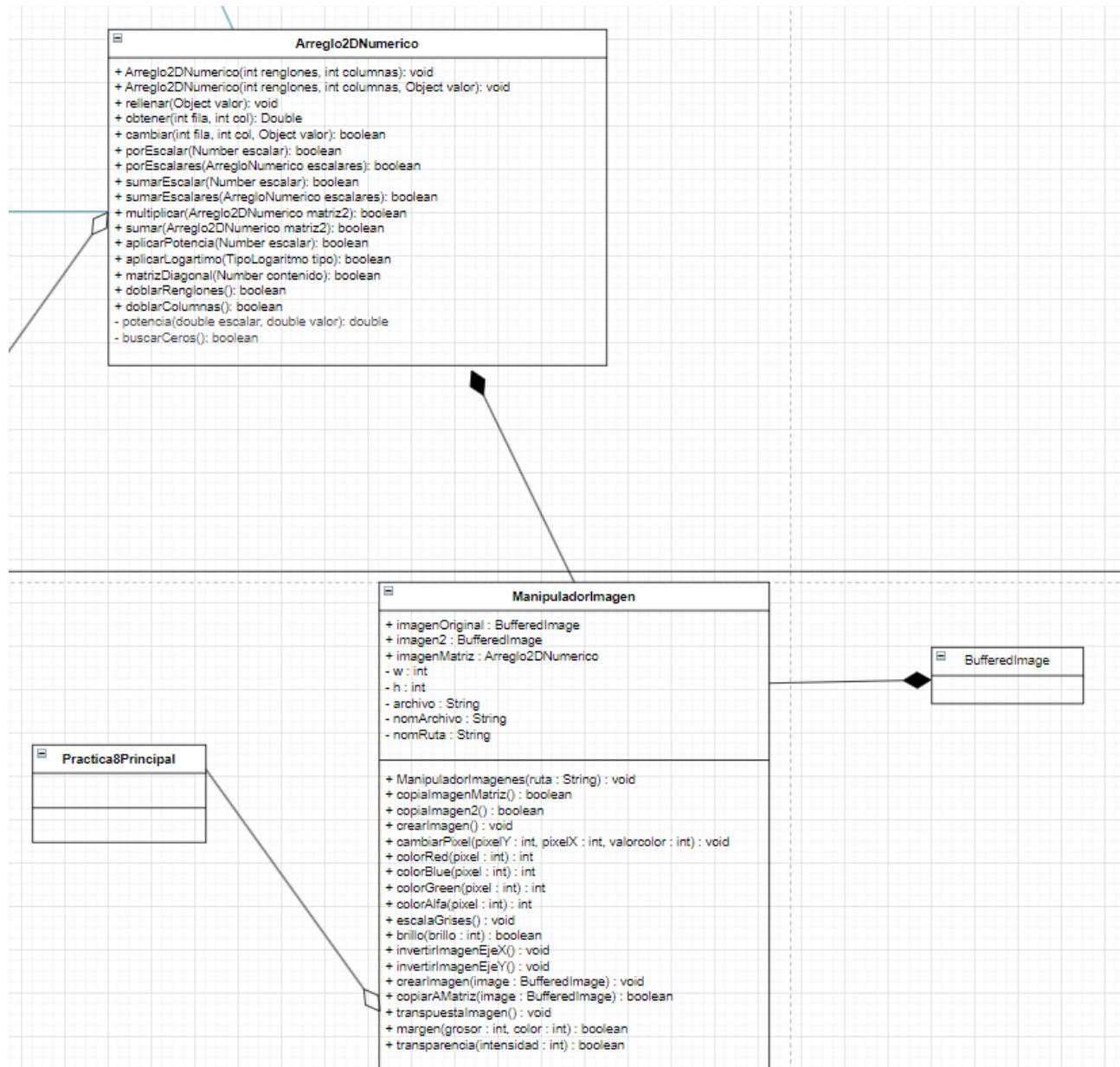


Figure 18: UML

## 4 Pre-evaluación del Alumno

Criterio	Evaluación
Cumple con la funcionalidad solicitada	Sí
Dispone de código auto-documentado	Sí
Dispone de código documentado a nivel de clase y método	Sí
Dispone de indentación correcta	Sí
Cumple la POO	Sí
Dispone de una forma fácil de utilizar el programa para el usuario	Sí
Dispone de un reporte con formato IDC	Sí
La información del reporte está libre de errores de ortografía	Sí
Se entregó en tiempo y forma la práctica	No
Incluye el código agregado en formato UML	Sí
Incluye las capturas de pantalla del programa funcionando	Sí
La práctica está totalmente realizada (especifique el porcentaje completado)	85%

Table 1: Evaluación de la práctica

## 5 Conclusión

se identifico y se vio de manera correcta como usar los conceptos ya tomados, ademas de su relación en procesos dentro de sistemas y cuales son sus datos que los componen

## 6 Referencias:

- Cairo, Osvaldo; Guardati, Silvia. *Estructura de Datos, Tercera Edición.* McGraw-Hill, México, Tercera Edición, 2006.
- Mark Allen Weiss. *Estructura de datos en Java.* Ed. Addison Wesley.
- Joyanes Aguilar, Luis. *Fundamentos de Programación. Algoritmos y Estructuras de Datos.* Tercera Edición, 2003. McGraw-Hill.