

# Aplicación gráfica para la compresión de información multimedia

Proyecto Final de Carrera

[Ingeniería técnica de sistemas]

**Autor:** Manuel A. Velázquez Sisternas

**Director:** José Vicente Benlloch Dualde

[26 de septiembre de 2014]



# Resumen

A través de la presente memoria se va a realizar un recorrido de cómo se ha venido desarrollando el proceso de trabajo del proyecto “**Aplicación gráfica para la compresión de información multimedia**”. Durante el desarrollo de los puntos que comprende la misma, se detallará los procedimientos empleados, los métodos realizados y los resultados obtenidos. Todo el proyecto queda amparado bajo la idea de lograr una aplicación eficaz cuya interfaz amigable facilite el aprendizaje al alumno que la utilice.

**Palabras clave:** Java, K-means, WindowsBuilder, Huffman, Bitmap, Compresión, LZW, RLE.



# Tabla de contenidos

## 1 Introducción

### 1.1 Consideraciones sobre la compresión de datos

#### 1.1.1 Técnicas de compresión de datos

### 1.2 Definición de la propuesta de trabajo

## 2 Desarrollo del proyecto

### 2.1 Punto de partida

### 2.2 Problemas iniciales identificados

### 2.4 Soluciones desarrolladas. Explicación del algoritmo

#### 2.4.1 El algoritmo RLE (Run Length Encoding)

#### 2.4.2 El algoritmo de Huffman

#### 2.4.3 El algoritmo LZW (Lempel-Ziv-Welch)

#### 2.4.4 K-means

## 3 Funcionamiento de la aplicación

## 4 Conclusiones

## 5 Fuentes consultadas



# 1 Introducción

## 1.1 Consideraciones sobre la compresión de datos

La capacidad de los actuales ordenadores de proyectar imágenes, sonidos y texto de forma simultánea, así como la capacidad de realizar paralelamente el control de todo tipo de dispositivos periféricos (unidades DVD, Blu-ray, webcam, memorias USB ...) resulta casi imprescindible en cualquier sistema de información.

Este tipo de información ocupa gran cantidad de espacio en nuestros dispositivos. Por ejemplo, un vídeo de 640x480 píxeles, con 24 bits/píxel y 30 fps con una duración de un minuto ocuparía algo más de 1'5GB.

Tal cantidad de información resulta imposible de tratar sin la codificación de dichos archivos. Para ello resulta fundamental la reducción del volumen de datos tratables. Así, al proceso de representar una determinada información empleando una menor cantidad de espacio, se le denomina "compresión de datos".

La compresión de datos puede buscar repeticiones en series de datos, para después almacenar sólo el dato junto al número de veces que se repite. Así, por ejemplo, si en un fichero aparece una secuencia como "AAAAAA", ocupando 6 bytes se podría almacenar simplemente "6A" que ocupa solo 2 bytes, con el algoritmo Run Length Encoding (RLE).

El proceso puede resultar sencillo inicialmente, aunque realmente se trata de algo mucho más complejo puesto que es inusual encontrar patrones de repetición tan exactos: Por un lado, existen algoritmos que buscan series largas que posteriormente codifican en formas más breves; por otro lado, algunos algoritmos, como el de Huffman, examinan los caracteres más repetidos para luego codificar, siempre de manera más corta, los que más se repitan. Por último, otros como el Lempel-Ziv-Welch (LZW) construyen un diccionario con los patrones encontrados, a los cuales se hace referencia posteriormente.

### 1.1.1 Técnicas de compresión de datos

El objetivo de la compresión es siempre reducir el tamaño de la información, intentado que esta reducción de tamaño no afecte al contenido. No obstante, la reducción de datos puede o no afectar a la calidad de la información:

Las técnicas de **codificación sin pérdidas** (o *lossless*) son procedimientos de codificación que tienen como objetivo representar cierta información, ocupando un menor espacio y siendo posible una reconstrucción exacta de los datos originales. Es decir, los datos antes y después de ser comprimidos son exactamente iguales. En el caso de la compresión sin pérdidas, una mayor compresión suele implicar más tiempo de proceso.



Las técnicas de **codificación con pérdidas** (o *lossy*) tienen como objetivo representar determinada información utilizando una menor cantidad de información, siendo imposible la reconstrucción exacta de la información original. En la codificación con pérdidas se eliminan datos para disminuir aún más el tamaño, con lo que la calidad puede verse afectada. La idea central de esta aproximación es eliminar la información que se considere irrelevante, basada en las características de la percepción humana.

La codificación de ficheros es un tema crucial tratado en la asignatura **Fundamentos de Sistemas Multimedia.**

## 1.2 Definición de la propuesta de trabajo

En ocasiones, resulta complejo para los alumnos poder comprender la efectividad de dichos algoritmos. Por ello, sería de ayuda para el alumno y posterior apoyo para la asignatura, el **desarrollo de una aplicación** mediante la cual se puedan realizar compresiones y comparar los resultados, siempre a partir del archivo original.

Dicha aplicación nos obliga a tomar una serie de decisiones en pos de su usabilidad:

**Independencia de plataforma**, lo que viene a significar que el programa escrito pueda ejecutarse igualmente en cualquier tipo de sistema operativo (SO). Esta cualidad se hace imprescindible en una comunidad como la nuestra, donde existen usuarios de un amplio abanico de SO.

**Interfaz amigable**, que permita comunicarse con el programa sin que éste suponga ningún tipo de dificultad ni resistencia al usuario que interactúe con él. Así, se hace imprescindible una interfaz de ventana.

**Adecuación a la asignatura**, es decir, los algoritmos implementados han de ser las principales codificaciones estudiadas en Fundamentos de Sistemas Multimedia.

## 2 Desarrollo del proyecto

### 2.1 Punto de partida

En la aplicación se implementarán los algoritmos principales explicados en **Fundamentos de Sistemas Multimedia.** Por un lado tendremos los siguientes algoritmos sin pérdida:

- RLE: (*Run Length Encoding*) dado un flujo de datos, lo que permite este algoritmo es codificar las repeticiones de datos consecutivos en el número que se representan y el número de veces que se repite dicho símbolo. Si la información que vamos a codificar presenta una gran cantidad de datos repetidos de manera consecutiva, el algoritmo dará buenos resultados.
- Codificación de Huffman: Se toma un alfabeto de  $n$  símbolos, con las frecuencias de aparición asociadas a cada uno de ellos, y produce un código de



Huffman para ese alfabeto de frecuencias dadas.

El algoritmo consiste en la creación de un árbol binario que tiene cada uno de los símbolos en las hojas y se construye de manera que, a partir de la raíz y siguiendo hacia cada hoja, se obtienen los códigos de Huffman.

- LZW:(Lempel-Ziv-Welch), algoritmo de compresión de datos basado en diccionarios. Este diccionario se crea y, paralelamente, se le van agregando pares de caracteres consecutivos legibles, aun cuando todavía no sea posible prever si ese código se reutilizará más adelante o no. Es en este detalle donde se encuentra la brillantez del método: Al armar el diccionario paralelamente, se evita hacer dos pasadas sobre el texto -una analizando y la otra codificando- y dado que la regla de armado del diccionario es tan simple, el descompresor puede reconstruirlo a partir del texto comprimido mientras éste es leído.

Y por otro lado, optaremos por K-means como ejemplo de codificación con pérdidas, para crear una paleta de colores óptima:

- **K –means**, es un algoritmo de partición. Básicamente , este algoritmo busca formar *clusters* (grupos) los cuales serán representado por *K* objetos. Cada uno de estos *K* objetos es el valor medio de los objetos que pertenecen a dicho grupo.

## 2.2 Problemas iniciales identificados

El software multiplataforma puede dividirse en dos tipos: Uno requiere una **compilación individual** para cada plataforma que le da soporte, y el otro se puede ejecutar directamente en cualquier plataforma, sin necesidad de preparación especial. En este caso, nos resulta mucho más útil desarrollar una aplicación que sea capaz de correr directamente en cualquier plataforma sin una preparación especial, ya que este hecho facilitará su uso al alumno que desee utilizarlo. He llegado a la conclusión que debo escribir el software en un lenguaje interpretado pre-compilado portable, para el cual un intérprete en tiempo de ejecución sea un componente común o estándar en todas las plataformas. El lenguaje finalmente elegido ha sido Java, ya que tal y como reza su axioma "write once, run anywhere", puede ejecutarse en cualquier SO y su uso está ampliamente extendido.

Java permite la creación de interfaz gráfica mediante las librerías Swing o SWT. El gran problema que se encuentra al programar con estas librerías es que se percibe la ausencia de alguna interfaz que nos facilite el trabajo de creación de interfaces gráficas en Java, algo parecido al IDE de Microsoft Visual Studio.

Las imágenes digitales consisten en un conjunto de valores numéricos ordenados en forma de matrices, donde cada uno de estos valores representa el color de un pixel. Cada formato (jpg, gif, bmp, png ...) realiza internamente sus propias compresiones, por lo tanto, frecuentemente el tamaño del archivo no representa el peso real de su matriz de colores. En Java, para manejar y manipular imágenes, usamos la clase `BufferedImage`. `BufferedImage` describe una imagen con un buffer de acceso de datos de la imagen, y nos permite leerla pixel a pixel. Así, se logra extraer el array de colores



de una imagen, con el inconveniente de que éste pesará más que la propia imagen ya codificada por su formato.

## 2.4 Soluciones desarrolladas. Explicación del algoritmo

Para el desarrollo de la aplicación he decidido usar Eclipse como entorno de desarrollo integrado. Eclipse está compuesto por un conjunto de herramientas de programación de código abierto multiplataforma, adecuada para desarrollar aplicaciones.

**Eclipse** dispone de un Editor de texto con resaltado de sintaxis. La compilación es en tiempo real. Tiene pruebas unitarias con JUnit, control de versiones con CVS, integración con Ant, asistentes (*wizards*) para creación de proyectos, clases, tests y refactorización.

El entorno de desarrollo integrado (IDE) de Eclipse emplea módulos (en inglés *plug-in*) para proporcionar toda su funcionalidad al frente de la plataforma de cliente enriquecido, a diferencia de otros entornos monolíticos donde las funcionalidades están todas incluidas, las necesite el usuario o no. Este mecanismo de módulos es una plataforma ligera para componentes de software. Adicionalmente, permite a Eclipse extenderse usando otros lenguajes de programación como son C/C++ y Python, pudiendo así trabajar con lenguajes para procesado de texto como LaTeX, aplicaciones en red como Telnet y Sistema de gestión de base de datos. Esta característica la he explotado ampliamente con la instalación de WindowsBuilder, *plug-in* para el desarrollo de interfaces visuales que permite arrastrar componentes.

WindowsBuilder es una herramienta que Google compró y liberó en 2011, que permite arrastrar componentes visuales sobre un formulario que genera de forma automática el código Java. Esto permite ser más productivo en el desarrollo de la interfaz de la aplicación y ayuda a focalizar la atención en la lógica de nuestro problema

Para instalar el *plug-in* se deben seguir los siguientes pasos:

Una vez instalado, pasamos a exponer el ejemplo de cómo creamos una ventana con WindowsBuilder:

1.- Abrir Eclipse e ir a la pestaña de Help --> Install New Software

2.- En la nueva ventana, seleccionar la opción de browse. Se deben proporcionar los siguientes datos:

Name: WindowBuilder

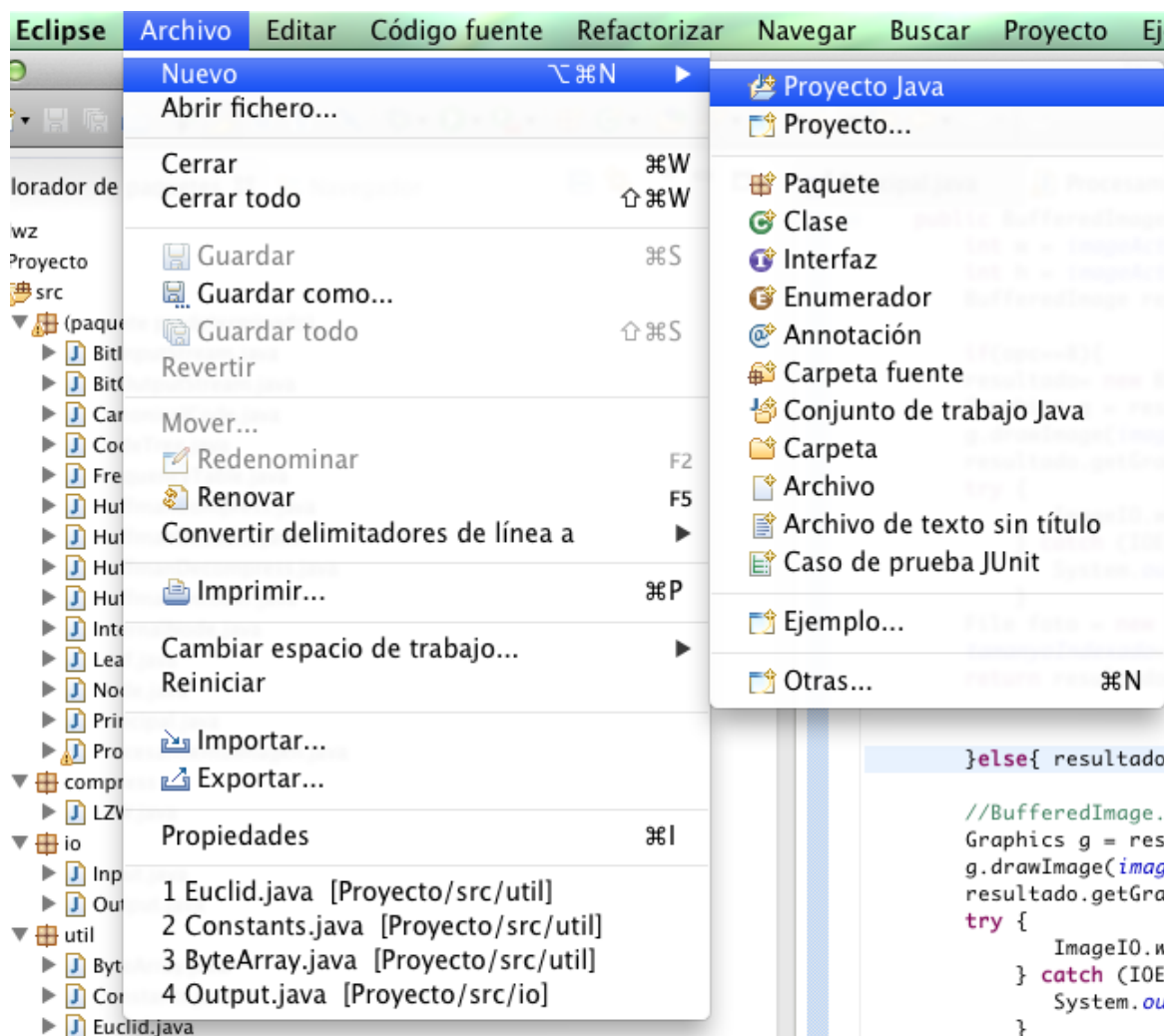
Location: [archive.eclipse.org/windowbuilder/WB/release/R201109201200/3.7/](http://archive.eclipse.org/windowbuilder/WB/release/R201109201200/3.7/)

Dar click en aceptar.



- 3.- Esperar hasta que se muestren los plugins a instalar de Swing Designer, SWT Designer y WindowBuilder Engine (required). Seleccionar las tres opciones.
- 4.- Dar click en aceptar y aceptar los términos para poder realizar la instalación de la herramienta.
- 5.- Al finalizar, reiniciar Eclipse e ir a la pestaña de Help/ About Eclipse / Installation Details y si se encuentran los tres plugins, se habrá instalado de manera exitosa la herramienta WindowBuilder

Crearemos un Proyecto de Java normal (Java Project) normal (**File/New/Java/Project...**).

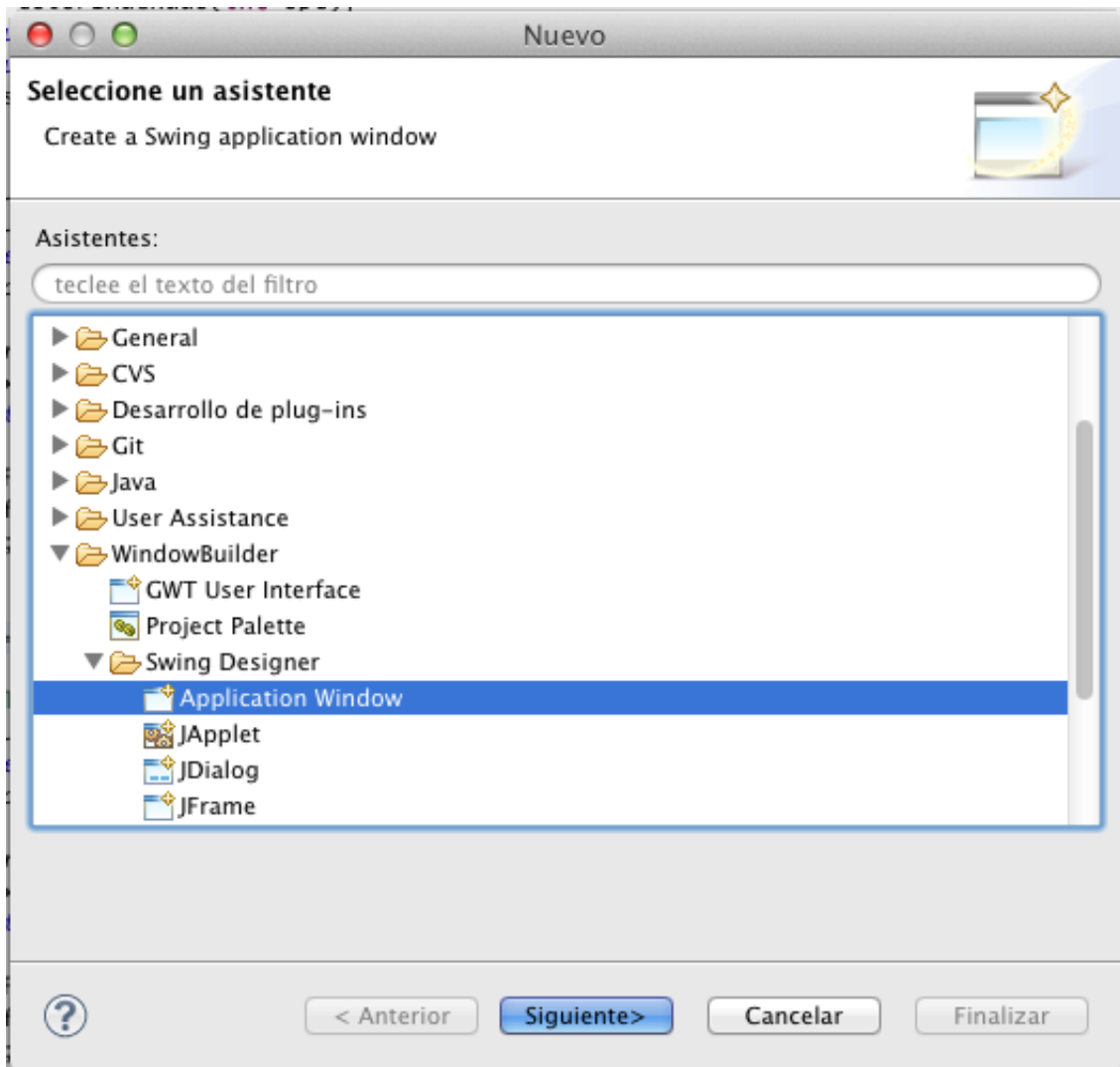




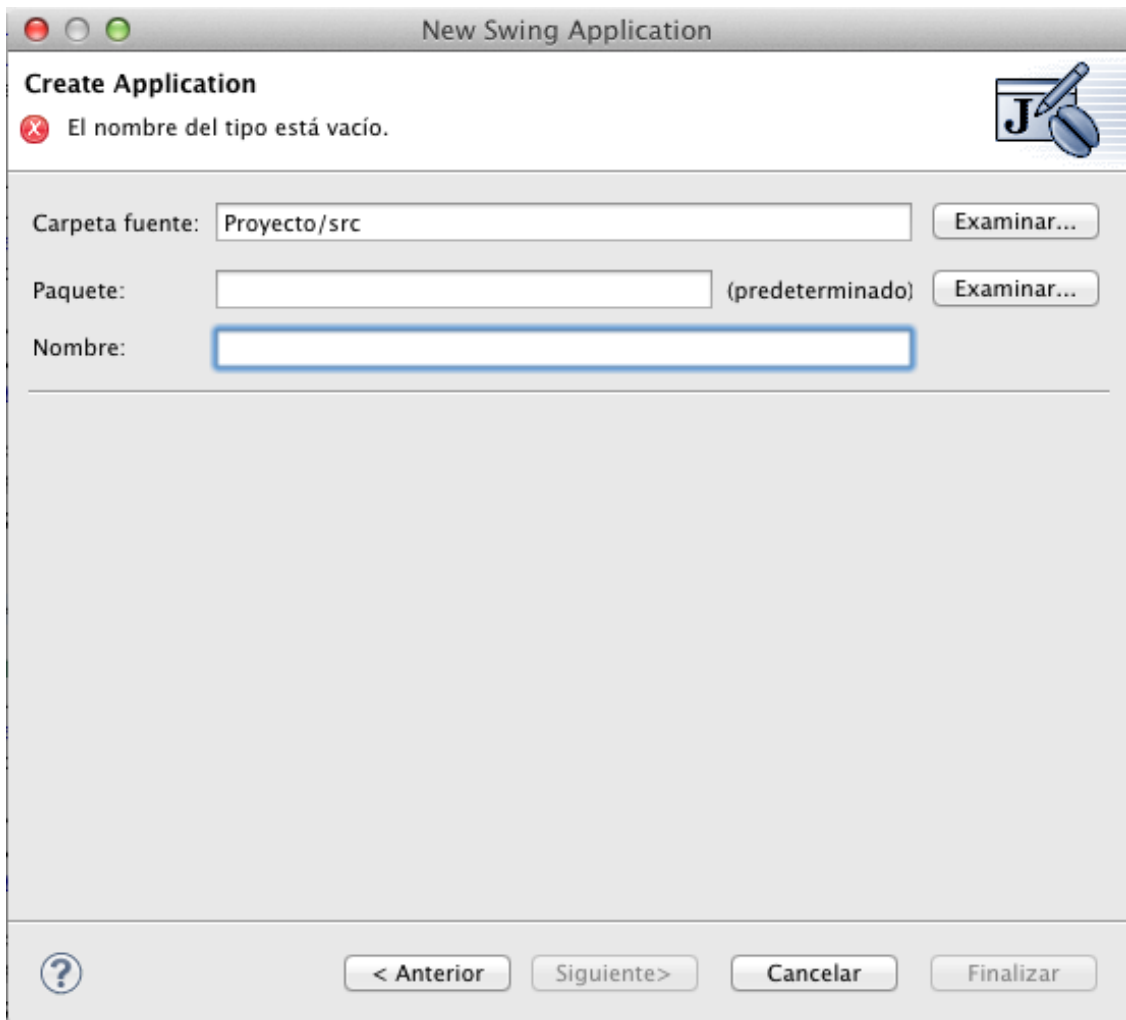
Le damos nombre y pulsamos finalizar.



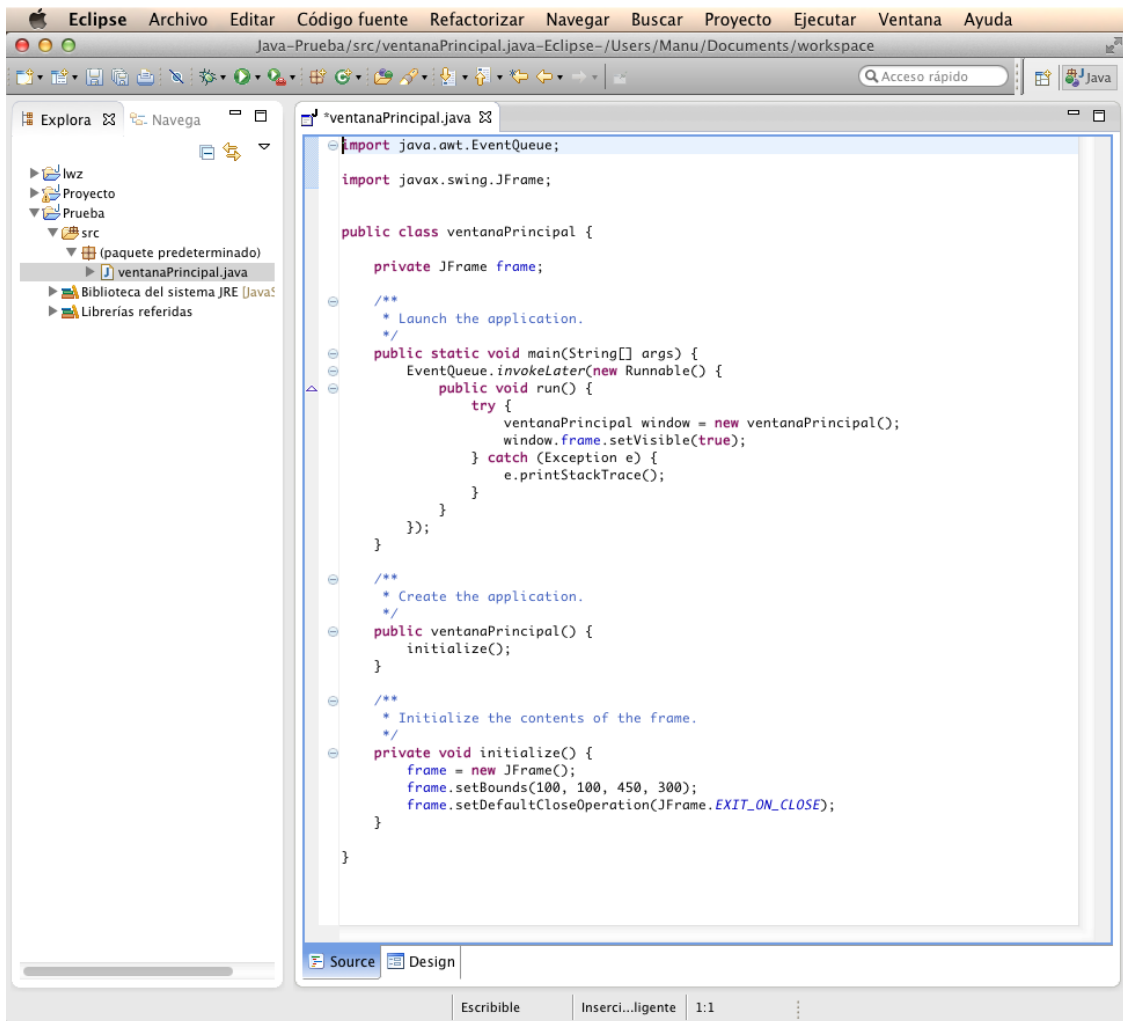
Ahora crearemos nuestra ventana, para eso damos clic derecho en el proyecto o vamos a **File/new/Other...** y buscamos **WindowBuilder**, posteriormente ingresamos a **Swing Designer** y seleccionamos **Application Window** y damos siguiente.



Al hacer esto, se carga otra ventana donde está definido el proyecto. Ingresamos el nombre de nuestra ventana y le damos **finalizar**.



Automáticamente, se crea una clase con el nombre que definimos y el código necesario para nuestra ventana.



The screenshot shows the Eclipse IDE interface. The main editor window displays the source code for the file `ventanaPrincipal.java`. The code is as follows:

```
import java.awt.EventQueue;

import javax.swing.JFrame;

public class ventanaPrincipal {

    private JFrame frame;

    /**
     * Launch the application.
     */
    public static void main(String[] args) {
        EventQueue.invokeLater(new Runnable() {
            public void run() {
                try {
                    ventanaPrincipal window = new ventanaPrincipal();
                    window.frame.setVisible(true);
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        });
    }

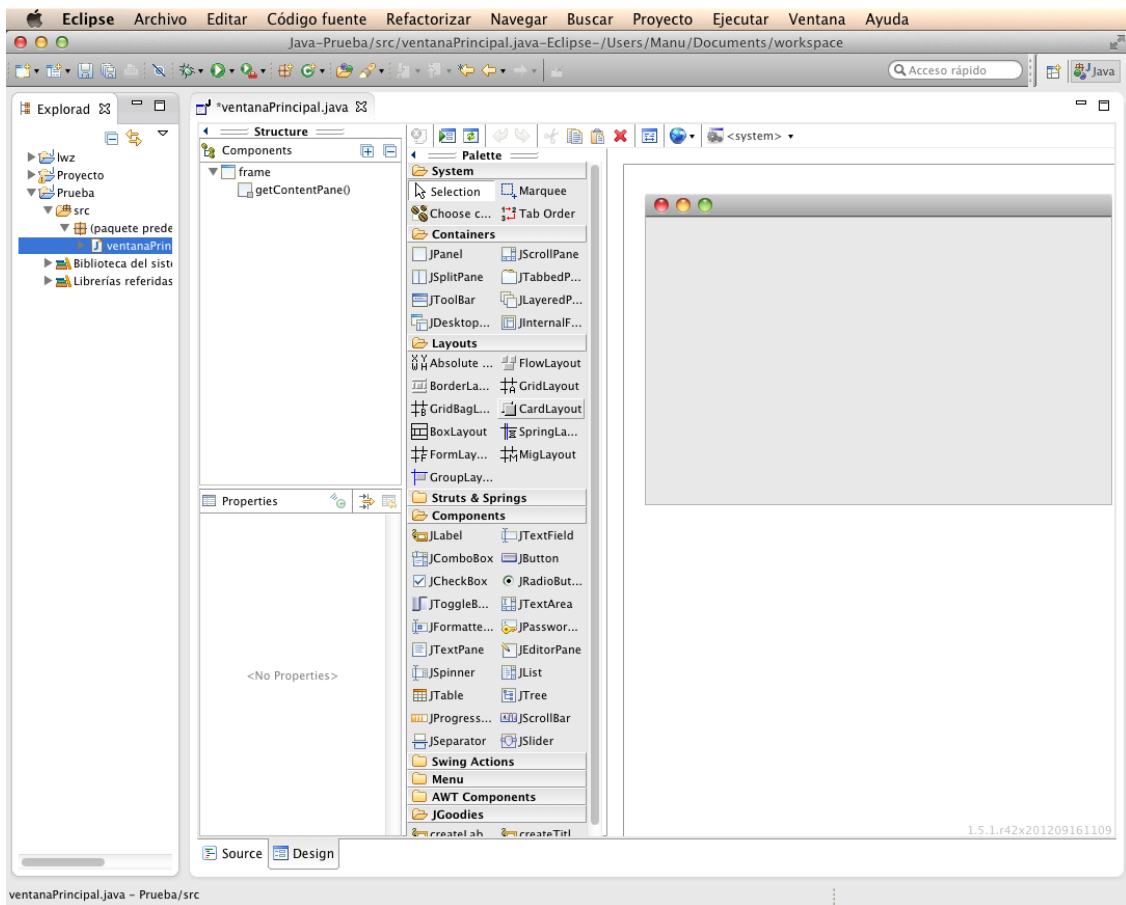
    /**
     * Create the application.
     */
    public ventanaPrincipal() {
        initialize();
    }

    /**
     * Initialize the contents of the frame.
     */
    private void initialize() {
        frame = new JFrame();
        frame.setBounds(100, 100, 450, 300);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
}
```

The IDE interface includes a menu bar (Archivo, Editar, Código fuente, Refactorizar, Navegar, Buscar, Proyecto, Ejecutar, Ventana, Ayuda), a toolbar, and a Project Explorer on the left showing the project structure. The bottom status bar indicates 'Escribible', 'Inserci...ligente', and '1:1'.



También podemos ver 2 pestañas: **Source** donde nos encontramos y vemos el código y la otra **Design**, que nos permitirá ver la paleta de componentes para crear nuestras aplicaciones de forma gráfica.



Encontramos la paleta de componentes, el árbol de componentes usados, las propiedades y la parte gráfica donde vemos la ventana que estamos diseñando. Estos paneles pueden ser cambiados de posición como cualquier panel de los usados en **Eclipse**. Además, podemos modificar el código y nuestra parte gráfica se actualiza según la modificación.

Dentro de la codificación multimedia sin pérdidas vamos a estudiar el algoritmo RLE, la codificación mediante códigos Huffman y el algoritmo LZW.

#### 2.4.1 El algoritmo RLE (Run Length Encoding)

El algoritmo RLE (Run Length Encoding) es un algoritmo de compresión de datos muy simple.

El algoritmo funciona de la siguiente manera: Dado un stream (flujo) de datos, lo que permite este algoritmo es codificar las repeticiones de datos consecutivos en el símbolo que representan y el número de veces que se repite dicho símbolo.



Por ejemplo, el stream de datos:

```
00000000000000001111111111111000000111111111110000000000000000000001
```

Podría codificarse así:

```
1 0:15  
2 1:13  
3 0:6  
4 1:11  
5 0:19  
6 1:1
```

Sin embargo, aunque estemos ante un algoritmo muy simple, puede dar muy buenos resultados en determinadas situaciones. Si la información que vamos a codificar presenta una gran cantidad de datos repetidos de manera consecutiva, el algoritmo dará buenos resultados.

En mi implementación, guardamos los resultados de la codificación en un fichero txt. Los 2 primeros datos almacenados corresponden al ancho y alto de la imagen. Después recorremos el array de pixeles de la imagen seleccionada guardada en un objeto `BufferedImage`, guardando la codificación en el fichero de texto.

```
public static void RLE(){  
  
    //compresion RLE  
  
    int width = imageActual.getWidth(); //ancho  
    int height = imageActual.getHeight();//alto  
  
    int aux1 = 0;  
    int aux2;  
    int cont =1;  
  
    FileWriter fichero = null;  
    PrintWriter pw = null;  
    try  
    {  
        fichero = new FileWriter("imagen.txt");  
        pw = new PrintWriter(fichero);  
        pw.println(width);  
        pw.println(height);  
  
        for (int i = 0; i < height; i++){  
            for(int j=0; j< width; j++){
```



```

        aux2=aux1;
        aux1=imageActual.getRGB(j,i );
        if(aux2==aux1){ //contamos cuantas veces se
repite el color
            cont++;

        }else{
            //imprimimos en archivo el par
repeticiones-color.
            pw.print(cont);
            pw.println(Integer.toString(aux2));
            cont=1;}

        }

    }

} catch (Exception e) {
    e.printStackTrace();
} finally {
    try {
        // Nuevamente aprovechamos el finally para
        // asegurarnos que se cierra el fichero.
        if (null != fichero)
            fichero.close();
    } catch (Exception e2) {
        e2.printStackTrace();
    }
}

File mi_fichero = new File ( "imagen.txt");
tamanoLRE = mi_fichero.length( );

}

```



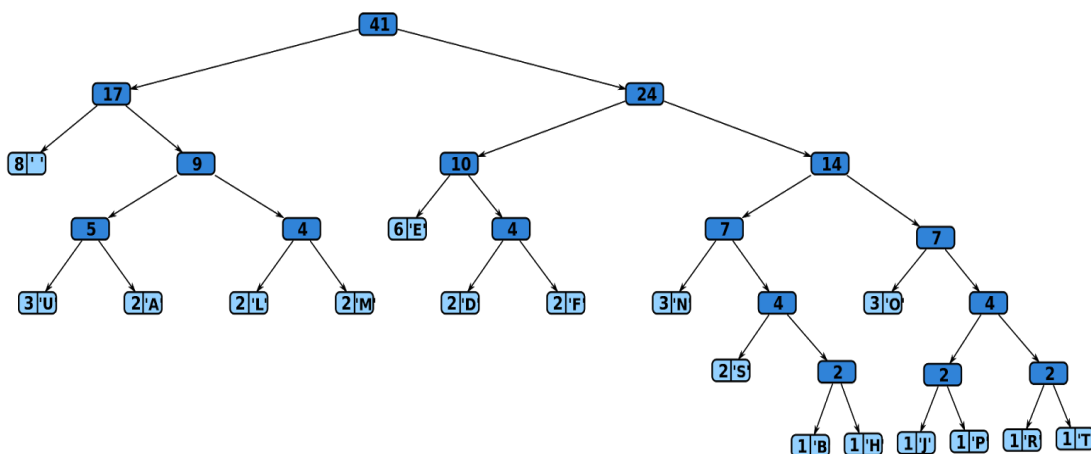
### 2.4.2 El algoritmo de Huffman

El algoritmo de Huffman toma un alfabeto de  $n$  símbolos, con las frecuencias de aparición asociadas a cada uno de ellos y produce un código de Huffman para ese alfabeto y frecuencias dadas.

El algoritmo consiste en la creación de un árbol binario que tiene cada uno de los símbolos en las hojas y se construye de manera que, a partir de la raíz y siguiendo hacia cada hoja, se obtienen los códigos Huffman.

1. Se crean varios sub-árboles, sin hijos, uno con cada símbolo y con su frecuencia de aparición.
2. Se escogen los de menor frecuencia, y se unen, formando un nodo. El nuevo nodo será la raíz del árbol que unía a los anteriores, con la etiqueta de la raíz igual a la suma de las frecuencias de los otros nodos. Estos árboles, a su vez, serán hijos de nuevos árboles. Sus hijos obtienen las etiquetas 0 y 1 al de la izquierda y de la derecha respectivamente.
3. Se repite el paso 2 hasta que quede un solo árbol.

El peso de las aristas de la Figura es, 0 para las de la izquierda, y 1 para las de la derecha. Se trata del árbol correspondiente al texto "ESTO ES UN EJEMPLO DE UN ÁRBOL DE HUFFMAN"



Si queremos codificar un símbolo, debemos seguir los siguientes pasos:

1. Se empieza con un código vacío.
2. Se inicia el recorrido desde la hoja del símbolo a codificar. Se recorre el árbol hacia arriba.
3. Cada vez que se suba un nodo, añadir al código un "1" o un "0".
4. Invertir el código al llegar a la raíz. Éste es nuestro código Huffman asociado al símbolo.



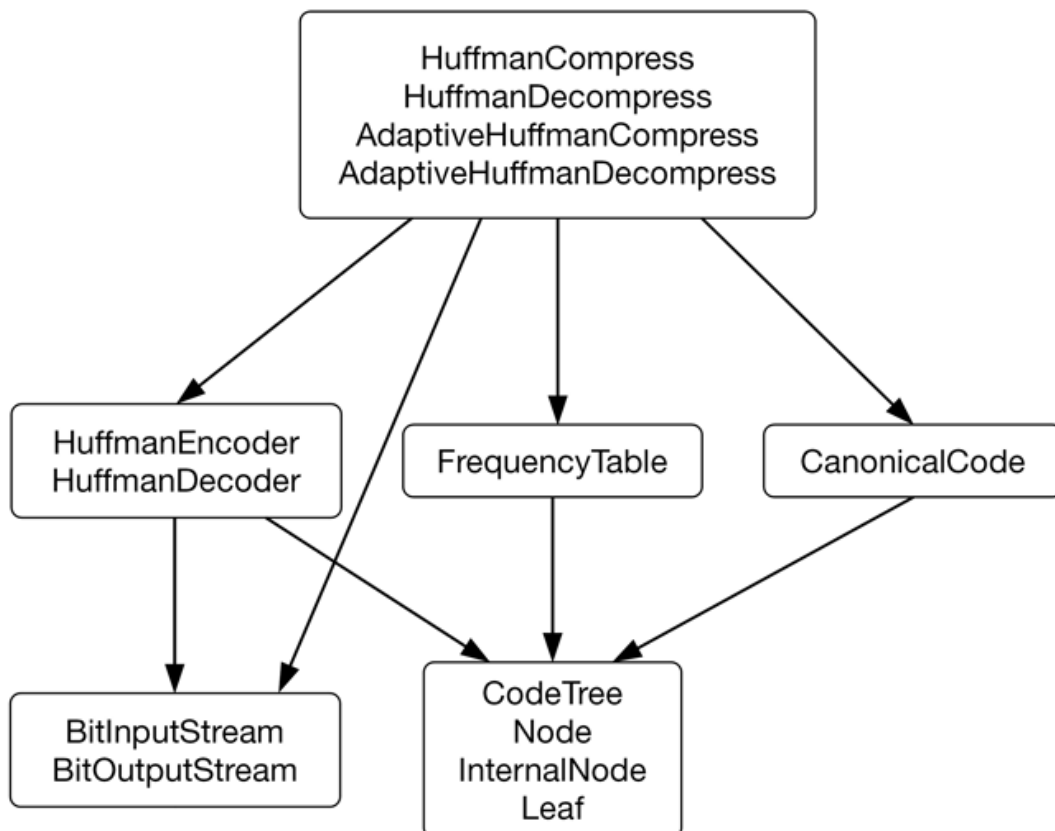


Para decodificar un código Huffman en su correspondiente símbolo debemos seguir estos pasos:

1. Se empieza a partir de la raíz del árbol.
2. Se extrae el primer elemento del código Huffman.
3. Descender por la rama etiquetada con ese elemento ("0" o "1").
4. Volver al paso 2 hasta que se llegue a la hoja del árbol, etiquetada con el símbolo buscado.

Para la implementación del código del algoritmo de Huffman, he tomado el desarrollo elaborado por el proyecto Nayuki. Este proyecto es una implementación de código abierto de la codificación Huffman en Java. El código está destinado a ser utilizado por estudiantes, y está concebido como una base sólida para la modificación y ampliación. Como tal, está optimizado para ser claro y resultar de baja complejidad, dejando a un lado la velocidad / memoria / rendimiento.

Seguidamente, me dispongo a detallar el funcionamiento de cada módulo.

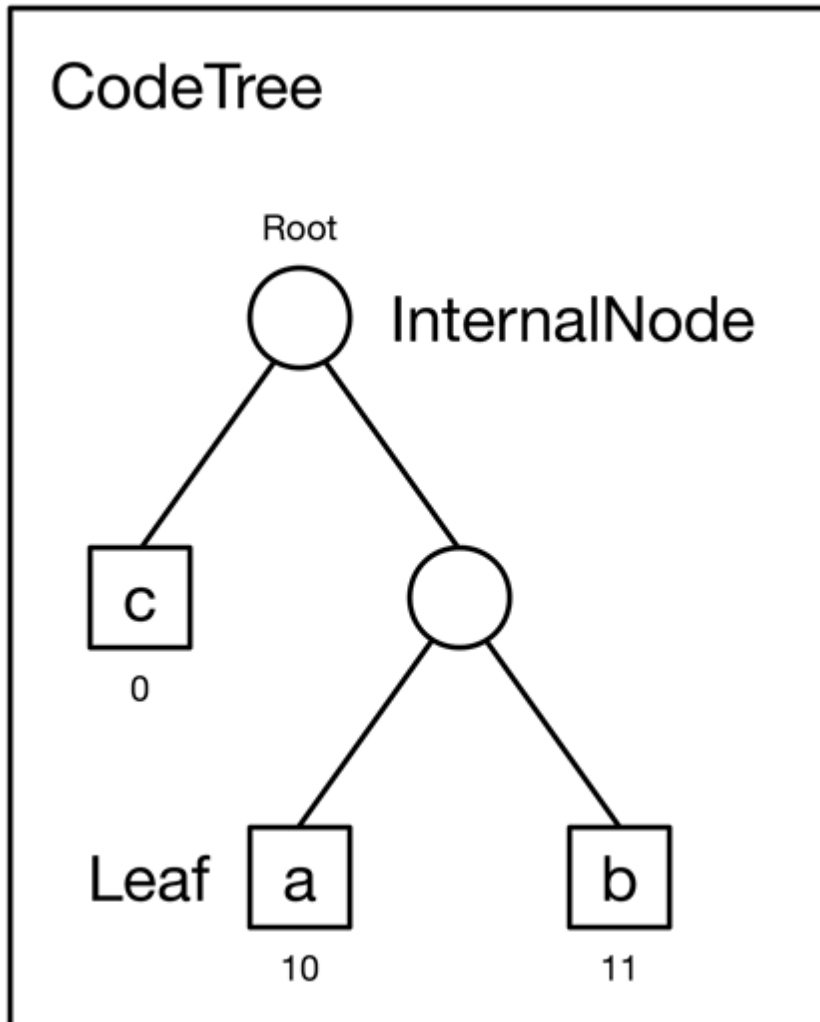


**HuffmanCompress**, **HuffmanDecompress**, **AdaptiveHuffmanCompress** y **AdaptiveHuffmanDecompress** son clases de muestra que implementan de forma estática y dinámica la codificación de Huffman.



Las clases **HuffmanEncoder** y **HuffmanDecoder** implementan los algoritmos básicos para la codificación y decodificación de un flujo de código-Huffman. El árbol de Huffman debe ser establecido antes de la codificación o decodificación. El árbol de Huffman puede ser modificado después de la codificación o decodificación de cada símbolo, siempre y cuando el codificador y el decodificador tengan el mismo árbol de Huffman en igual posición en el flujo de símbolos.

La clase **CodeTree**, junto con **Node**, **InternalNode**, y **Leaf**, representan un árbol de Huffman.



La clase **FrequencyTable** es una matriz de enteros que cuenta frecuencias de símbolos. También es responsable de generar un árbol de Huffman óptimo para su actual gama de frecuencias.

Las clases **BitInputStream** y **BitOutputStream** están orientadas a controlar los flujos de entrada y salida de bits, la longitud total de cada flujo de bits es siempre un múltiplo de 8.

El código implementado por el proyecto Nayuki tiene una serie de limitaciones:

Todo el código relacionado con Huffman sólo funciona con alfabetos de hasta `Integer.MAX_VALUE - 1` (i.e.  $2^{31} - 2$ ) símbolos.

**FrequencyTable** sólo puede realizar un seguimiento de la frecuencia de cada símbolo hasta `Integer.MAX_VALUE`. Si se trata de incrementar la frecuencia de un símbolo más allá de este límite se lanza una excepción. El uso de frecuencias mayores a 231 en la práctica es poco habitual.

El código está optimizado para la comprensión, no para el rendimiento. Una consecuencia es que **CodeTree** utiliza memoria de manera ineficaz para almacenar la cadena de bits de código para cada símbolo. Utiliza `ArrayList <Integer>` a un costo de al menos 4 bytes por bit representado.

Un par de métodos usan recursividad para recorrer el árbol de Huffman entero. Cuando se utiliza un método de este tipo en un árbol muy grande, puede producirse un desbordamiento de la pila de llamadas a métodos entre clases debido a un número excesivo de llamadas, lanzando un error de tipo `StackOverflowError`.

Usando las clases anteriormente explicadas he implementado el siguiente método.

```
public void Huffman()throws IOException{

    File outputFile = new File("COMPRESIONHUFFMAN");
    InputStream in = new BufferedInputStream(new
FileInputStream(imagenSeleccionada));
    BitOutputStream out = new BitOutputStream(new
BufferedOutputStream(new FileOutputStream(outputFile)));
    int[] initFreqs = new int[257];
    Arrays.fill(initFreqs, 1);

    FrequencyTable freqTable = new FrequencyTable(initFreqs);
    HuffmanEncoder enc = new HuffmanEncoder(out);
    enc.codeTree = freqTable.buildCodeTree();
    int count = 0;
    while (true) {
        int b = in.read();
        if (b == -1)
            break;
        enc.write(b);

        freqTable.increment(b);
        count++;
        if (count < 262144 && isPowerOf2(count) || count %
262144 == 0) // Actualiza code tree
```



```

        enc.codeTree = freqTable.buildCodeTree();
        if (count % 262144 == 0) // Reset frequency table
            freqTable = new FrequencyTable(initFreqs);
    }
    enc.write(256);
}

```

### 2.4.3 El algoritmo LZW (Lempel-Ziv-Welch)

El algoritmo LZW (Lempel-Ziv-Welch), es un algoritmo de compresión de datos basado en diccionarios. Este diccionario se crea tanto en la codificación como en la decodificación, siendo en este último caso inferido a partir de la codificación efectuada. El diccionario contiene códigos obtenidos de sucesivas inserciones de subcadenas que no se contienen inicialmente en el alfabeto de la codificación.

Por ejemplo, teniendo el alfabeto inglés:

SIMBOLO	BINARIO	DECIMAL
#	00000	0
A	00001	1
B	00010	2

.  
.  
.

SIMBOLO	BINARIO	DECIMAL
X	11000	24
Y	11001	25
Z	11010	26

Procedemos a codificar la cadena: ESTOYESTUDIANDO.



<b>Secuencia actual</b>	<b>Siguiente caracter</b>	<b>Codigo</b>	<b>Bit</b>	<b>Se inserta</b>
E	S	5	00101	27: ES
S	T	19	10011	28: ST
T	O	20	10100	29: TO
O	Y	15	01111	30: OY
Y	E	25	11001	31: YE
ES	T	27	11011	32: EST
T	U	20	010100	33: TU
U	D	21	010101	34: UD
D	I	4	000100	35: DI
I	A	9	001001	36: IA
A	N	1	000001	37: AN
N	D	14	001110	38: ND
O	#	15	001111	39: O#

Podemos observar que a partir de insertar 32 combinaciones de símbolos, se necesitan 6 bits para codificar. Respecto a este ejemplo, necesitamos 75 bits inicialmente, esto es,  $5 \text{ bits/- símbolo} * 15 \text{ símbolos}$ , que se ven reducidos a 72 ( $(5 \text{ bits/código} * 6 \text{ códigos}) + (6 \text{ bits/código} * 7 \text{ códigos})$ ).



A simple vista la compresión no parece notable, pero eso es debido a que hay pocas repeticiones de combinaciones de símbolos.

Para la aplicación he optado por modificar el código desarrollado por el usuario Madhead de code Google, para integrarlo dentro del mío. Para ello se han integrado dentro de mi código como tres paquetes a parte. Y se ha implementado el siguiente código que hace uso de los mismos.

```
public static void lzwCompression(){
    Scanner cons=new Scanner(System.in);
    try {

        FileInputStream in = new FileInputStream(nomArchivo);
        FileOutputStream out = new
FileOutputStream("compresionLWZ");
        LZW lzw = new LZW();
        lzw.compress(in, out);

    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

#### 2.4.4 K-means

Dentro de la codificación multimedia con pérdidas, vamos a analizar el algoritmo K-means para la creación de una paleta de colores óptima para una imagen.

Una imagen está compuesta por un número de píxeles (ancho por alto píxeles). Cada uno de estos píxeles contiene una información de color, por ejemplo RGB, esta información de color puede tener más o menos bits por cada una de sus componentes teniendo una gama de colores mayor/menor, pero ocupando también un mayor/menor espacio en disco.

Si reducimos los bits de color dedicados a cada píxel tendremos una imagen que ocupe menos espacio, aunque, por supuesto habremos reducido la calidad de la misma.

Para la implementación de K-means, se ha usado las propiedades de la clase bufferedImage.

Al construir un objeto de clase bufferedImage, se ha elegido el tipo de imagen que la define, si elegimos una representación con una paleta de color de 8 bits y copiamos la imagen original dentro de este objeto, Java automáticamente reduce los bits de color dedicados a cada píxel, creando la paleta de color adecuada utilizando k-means.



El código es el siguiente.

```
public BufferedImage colorIndexado(int opc){
    int w = imageActual.getWidth();
    int h = imageActual.getHeight();
    BufferedImage resultado;

    // if(opc==8){
        resultado= new BufferedImage(w, h,
BufferedImage.TYPE_BYTE_INDEXED );//8 bits
        Graphics2D g = (Graphics2D)resultado.getGraphics();
        g.setComposite(AlphaComposite.Src);
        g.drawImage(imageActual, 0, 0, null );
        resultado.getGraphics().drawImage(imageActual, 0, 0, null );
        try {
            ImageIO.write(resultado, "jpg", new
File("foto8bit.jpg"));
        } catch (IOException e) {
            System.out.println("Error de escritura");
        }
        File foto = new File("foto8bit.jpg");
        tamañoIndexado=foto.length();

        return resultado;
    }
}
```

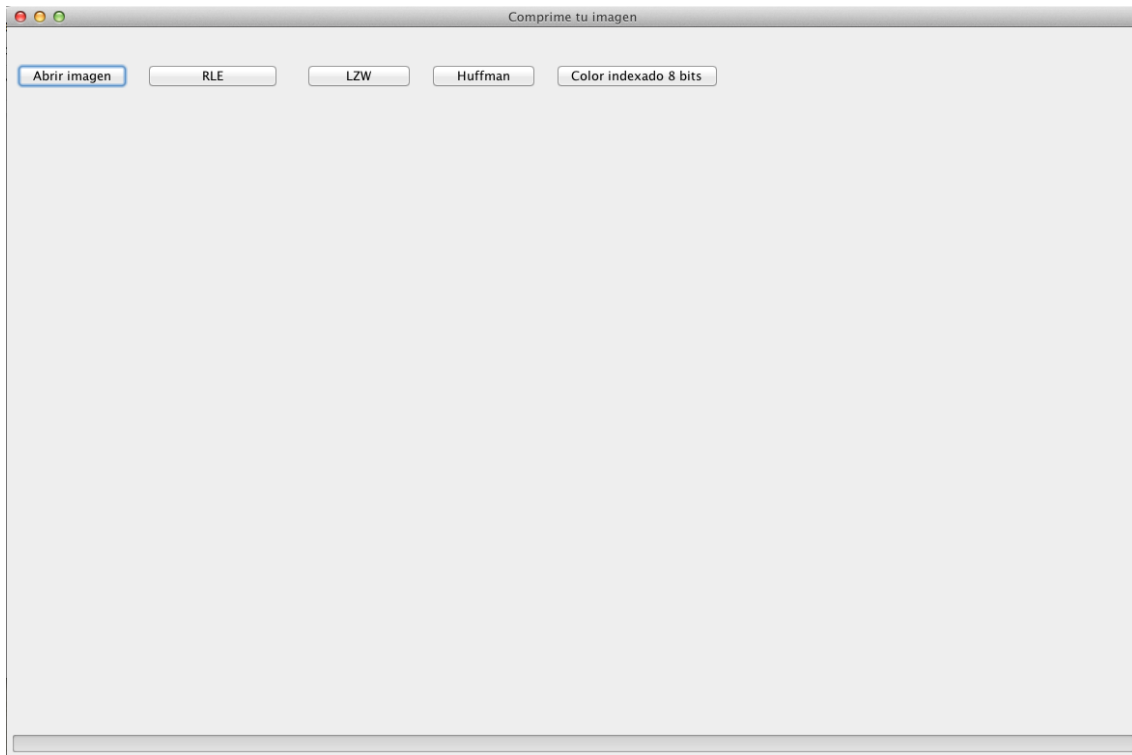
### 3 Funcionamiento de la aplicación

*\*La siguiente explicación gráfica se complementa con un screencast.*

En el siguiente, paso vamos a explicar el funcionamiento de la aplicación.

El primer lugar debemos hacer doble click sobre “Pfc.jar” y se nos abrirá la siguiente ventana.

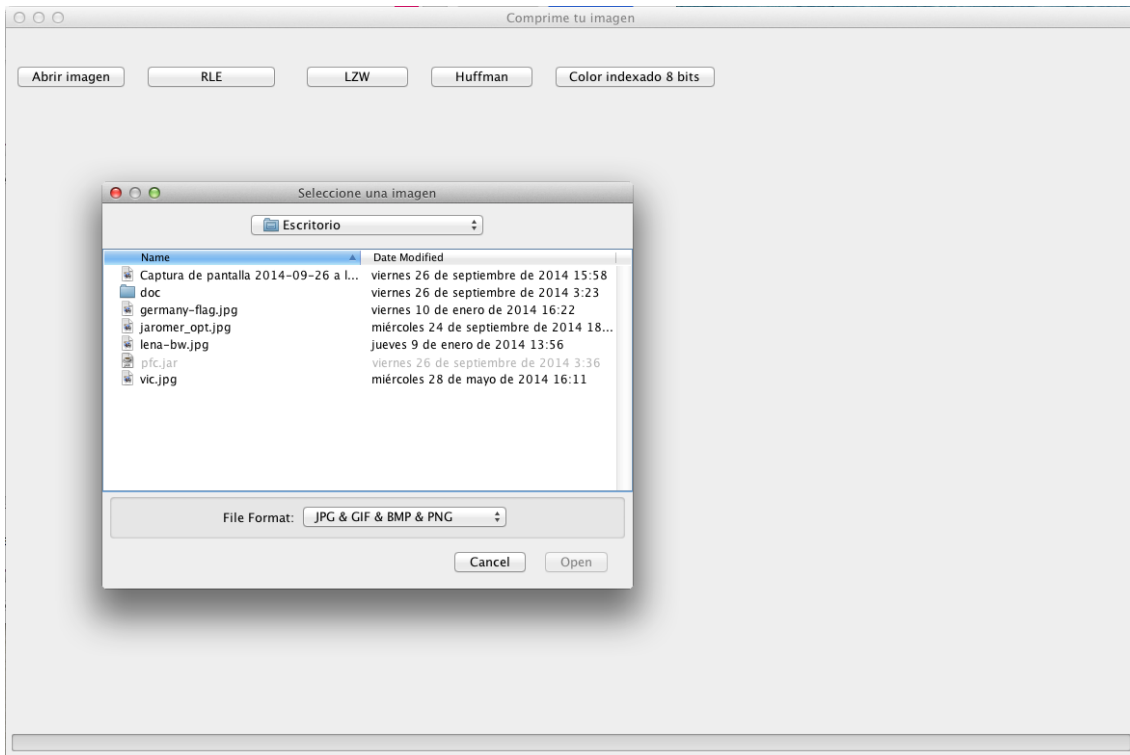




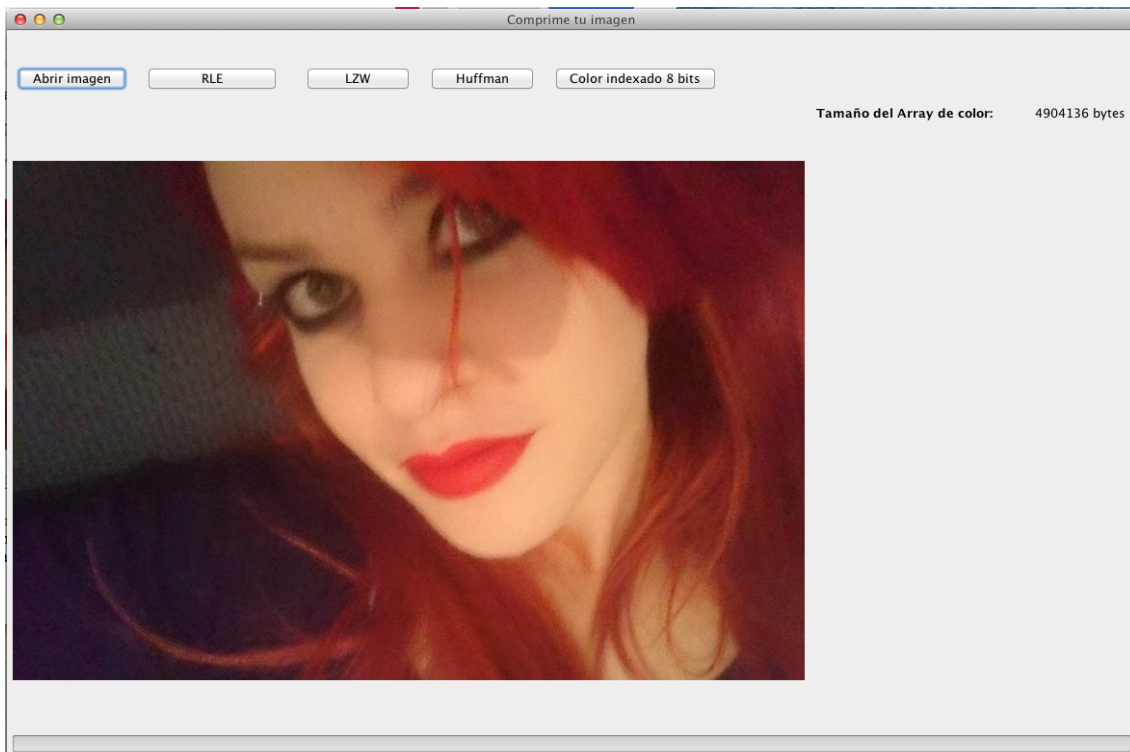
En ella podemos observar 5 botones en la parte superior. En la inferior, nos aparece una barra de progreso vacía.

Lo primero que debemos hacer es elegir una imagen sobre la que queremos aplicar las distintas codificaciones- Para ello, debemos pulsar sobre “Abrir imagen”, se nos abrirá un navegador donde elegir el archivo deseado (dentro de los formatos jpg, gif, bmp y png) una vez seleccionado debemos pulsar open.





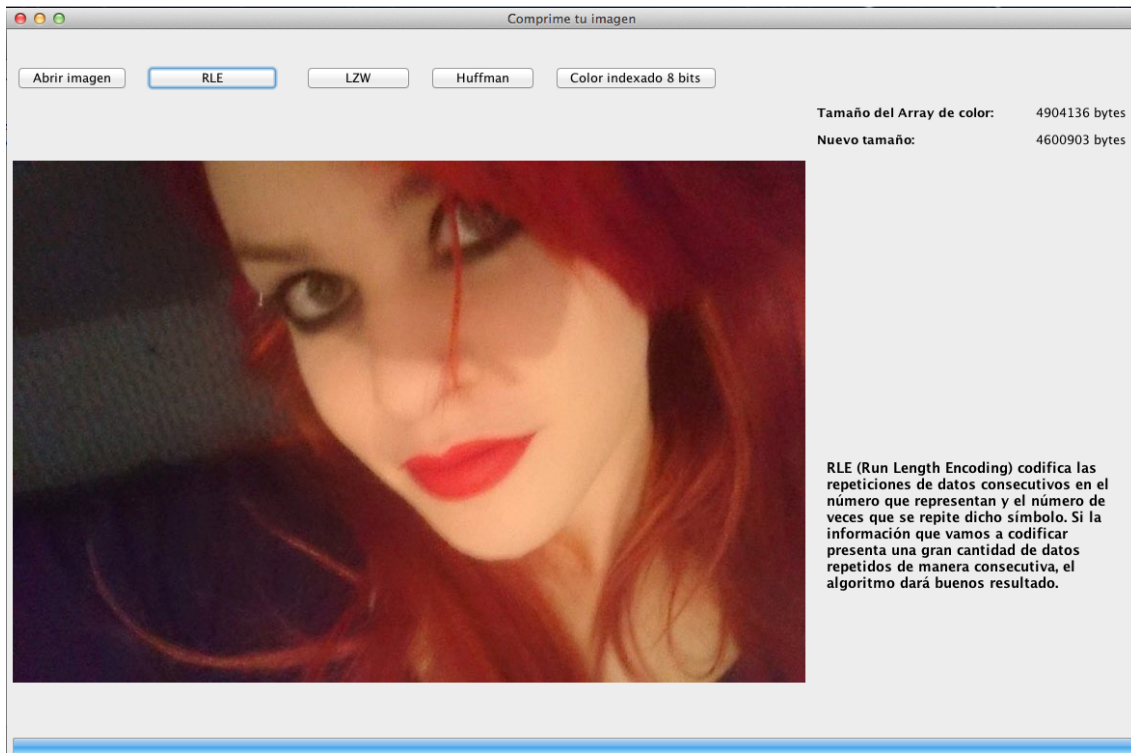
Tras pulsar open, la ventana quedará de la siguiente manera.



Se observa que la imagen se ha abierto y en la parte superior derecha aparece el tamaño del array de color de dicha imagen.

Ahora es el momento de empezar a aplicar codificaciones. Comenzaremos con RLE (Run Length Encoding).

Se pulsa el botón "RLE".



Se observa que la barra de progreso se ha completado y se ha procesado la imagen. Ahora, en la parte superior derecha aparece el tamaño de la compresión y en la parte inferior se ha impreso una breve definición de la codificación.

Observando los datos, la compresión de la imagen ha sido mínima. Esto es debido a que en la imagen varía constantemente el color de los pixels.

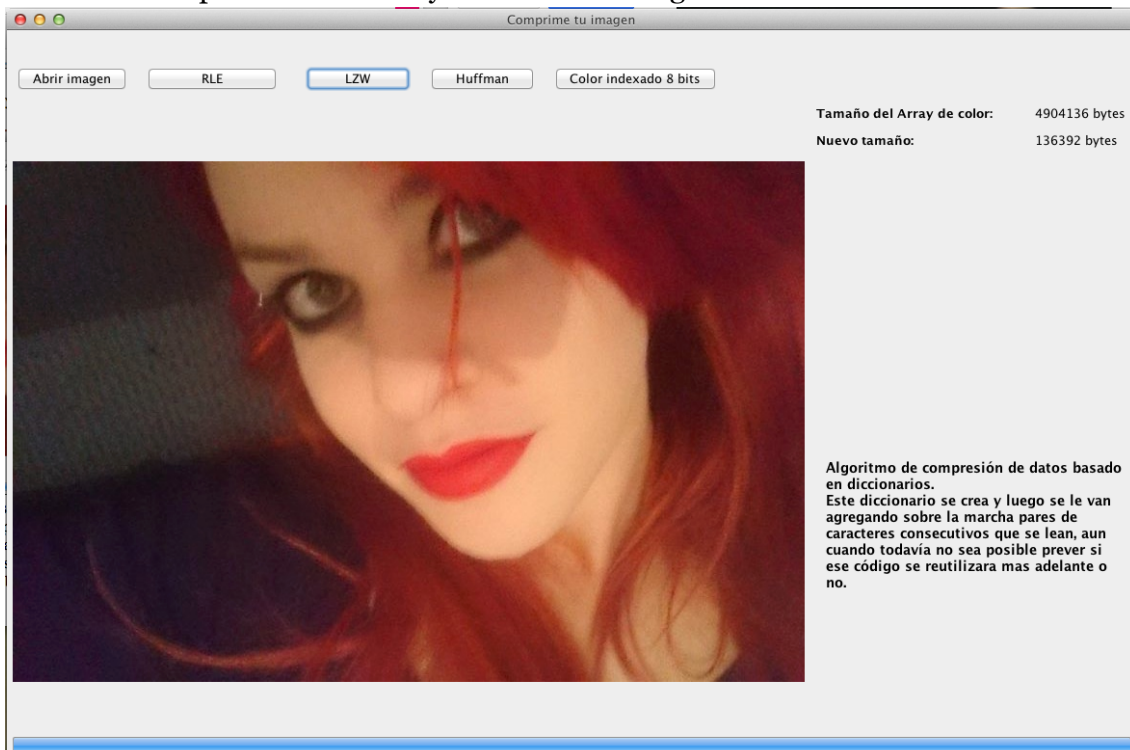
A continuación, se expone otro caso en el que este algoritmo obtiene unos excelentes resultados en la codificación de imágenes. Si utilizamos como imagen la bandera de Alemania, observamos una altísima tasa de compresión debido a la repetición de colores.



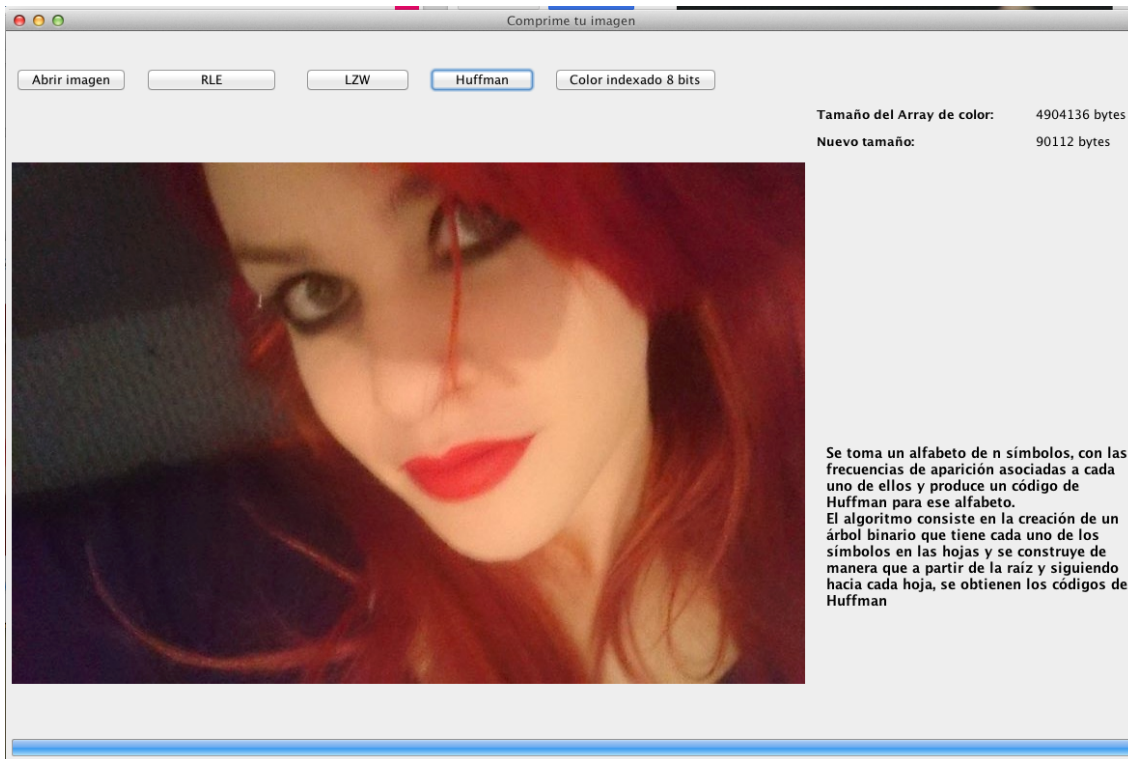
Se abre de nuevo la imagen anterior y pulsamos sobre el botón “LZW” y la aplicación codifica.

En este caso, el archivo que se le pasa como parámetro, pesa 4904136 bytes. Después de la compresión, el archivo creado pesa 136392 bytes, lo que es una reducción espacial muy importante. En la misma carpeta donde tenemos alojada la aplicación, se nos habrá creado un archivo llamado “Compresión LZW” que incluye la foto comprimida.

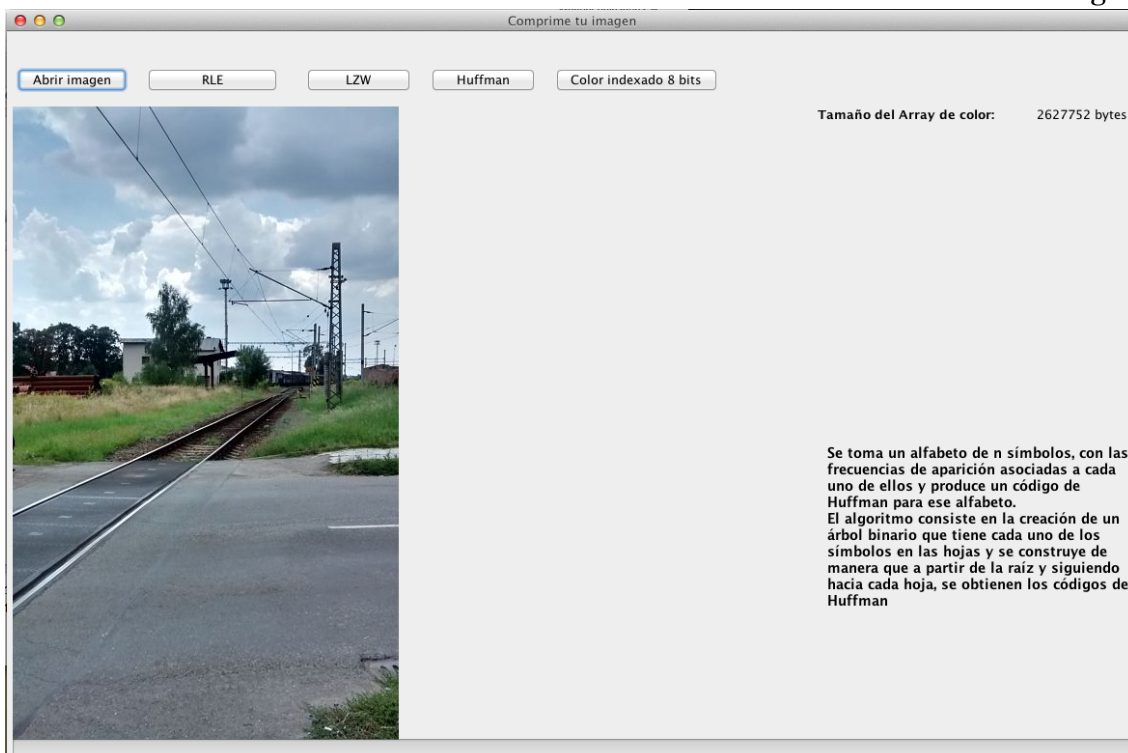
Tras “LZW” se pulsa “Huffman” y obtenemos los siguientes resultados.



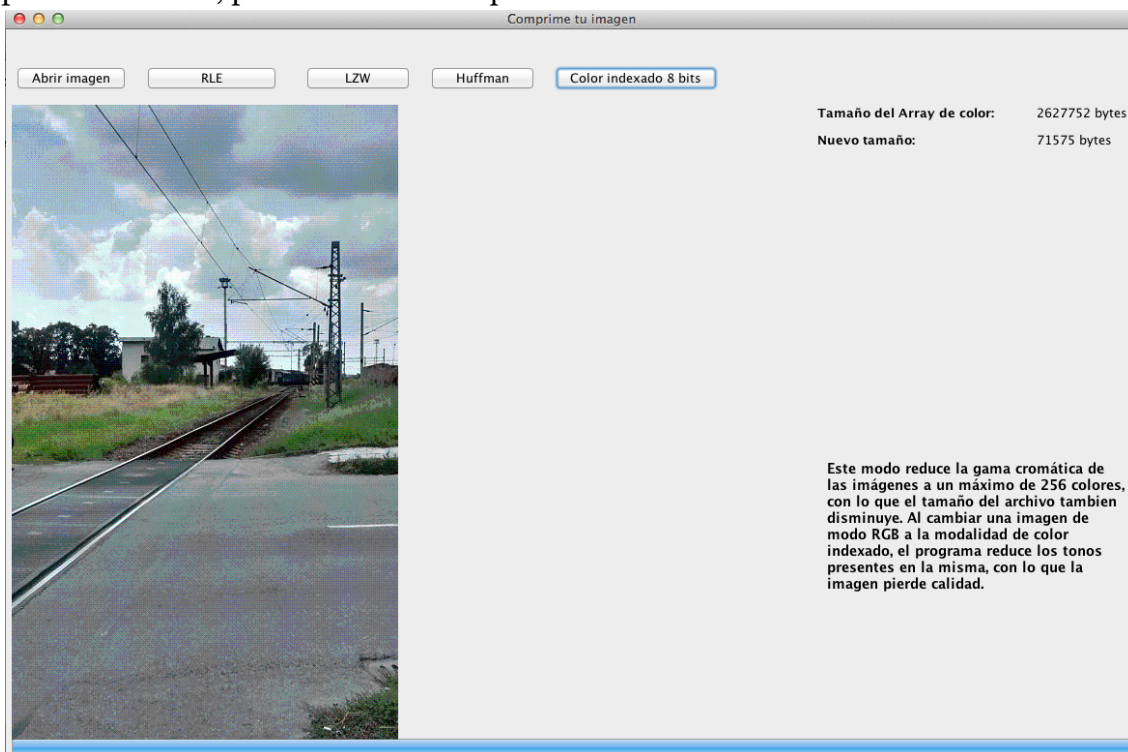
Por una parte, en la carpeta donde está alojada la aplicación se habrá creado un archivo “Compresión Huffman” que alberga la compresión de la imagen. Los resultados obtenidos de la compresión se pueden observar que también son bastante buenos.



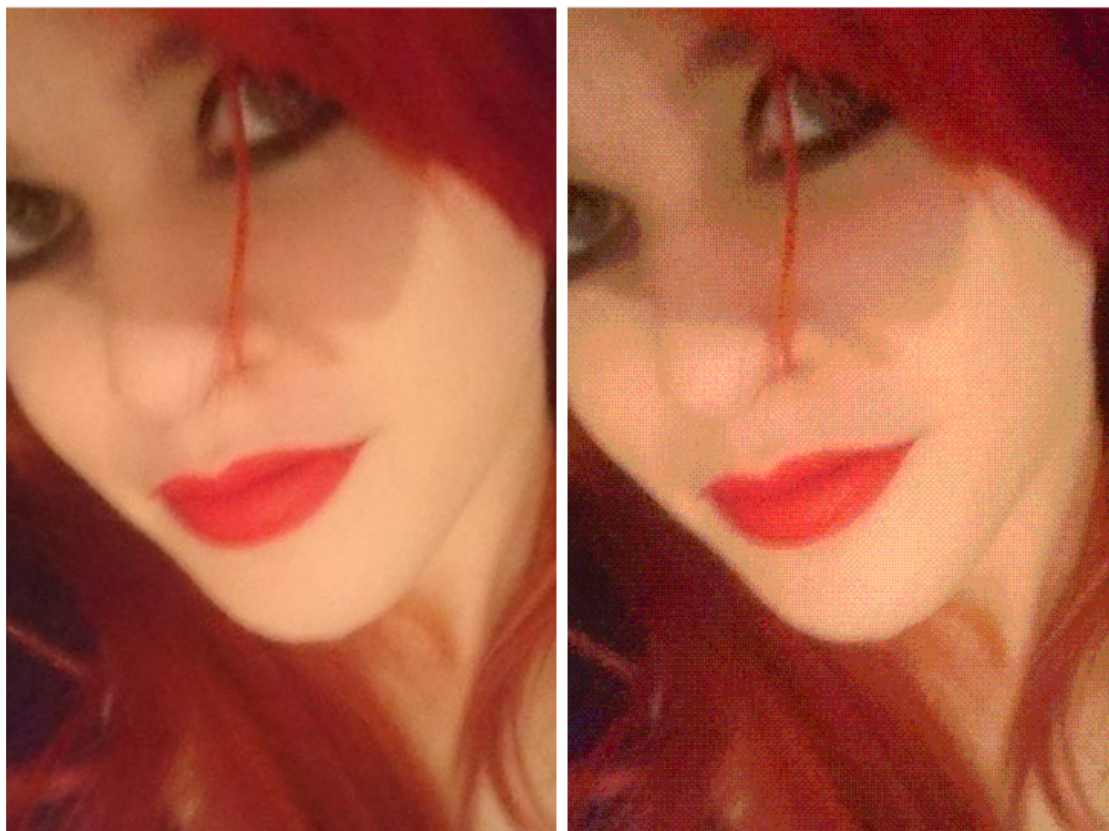
Por último, tenemos “Color indexado”, el único algoritmo con pérdidas implementado en este caso. Se puede observar que hay un botón que corresponde a una paleta de color de 8 bits. Para esta ocasión se abre una nueva imagen



Pulsamos sobre el botón “Color indexado 8 bits” y nos aparecerá la imagen con pérdida de color, podemos observar que el tamaño se ha reducido considerablemente.



Por el lado contrario, se observa una reducción en la calidad de la imagen. Seguidamente, se adjunta dos imágenes, una antes y otra después de aplicarse la compresión. En ellas se pueda observar la pérdida de datos.



## 4 Conclusiones

Tras el desarrollo de la aplicación, podemos afirmar que la compresión de la información multimedia es imprescindible, ya que aunque las capacidades de los discos duros aumentan continuamente, con ellos aumenta también la calidad de la información multimedia a la que accedemos (imágenes, audio y vídeo de alta calidad). También se ha podido comprobar que existen algunos algoritmos muy simples e intuitivos que nos permiten reducir de forma drástica el tamaño de los datos sin comprimir. Los algoritmos estudiados se siguen usando hoy en día en formatos que incluyen compresión de datos, por eso resultan de gran utilidad para todo estudiante de informática.

En función de los objetivos planteados en la introducción de esta memoria, se pretendía lograr una aplicación con independencia de plataforma, interfaz amigable y con un contenido que se adecue a la asignatura de **Fundamentos de Sistemas Multimedia.**

Para obtener el primero de los objetivos marcados la opción de **Java** como lenguaje de programación resulta una elección adecuada en la búsqueda de una solución multiplataforma. Como reza su axioma “ser capaz de escribir un programa una vez y que pueda ejecutarse en cualquier dispositivo” nos proporciona la independencia de plataforma deseable en una comunidad tan abierta como la nuestra.

Sin una interfaz amigable el proyecto estaba exento del atractivo necesario para el posible usuario. La interfaz de ventana dota a la aplicación de sencillez y claridad en su uso. Es también notable la ayuda del “plugin-in” **Windows Builder** a la hora de facilitar el diseño de dicha interfaz.

Las codificaciones sin pérdidas ,RLE, HUFFMAN y LZW , y con pérdidas K-MEANS son los principales ejemplos mostrados en la asignatura y por eso son los que se han desarrollado.

A nivel personal, este proyecto me ha servido para profundizar en aspectos de la programación que me parecían interesantes, a la par que he aprendido conceptos de programación en los que nunca había trabajado con anterioridad, como es el caso de interfaces gráficas, en concreto las librerías Swing. Otro aspecto muy importante a destacar ha sido conocer la forma en la que Java trata las imágenes con su librería awt y concretamente sus clases Color, Graphics, Graphics2D y BufferedImage.

Con la aplicación desarrollada se consigue una herramienta multiplataforma, con una interfaz amigable y un funcionamiento sencillo que sirve de apoyo a futuros estudiantes de la asignatura **Fundamentos de Sistemas Multimedia.**

## 5 Fuentes consultadas

Estructuras de Datos en JAVA – Mark Allen Weiss

<http://chuwiki.chuidiang.org>

<http://es.wikipedia.org>

<http://docs.oracle.com/javase/7/docs/api/>

<http://nayuki.eigenstate.org/>

<https://code.google.com/>

<http://www.coderanch.com>

<http://www.eclipse.org/>

<http://stackoverflow.com>

[http://centrodeartigo.com/articulos-informativos/article\\_69978.html](http://centrodeartigo.com/articulos-informativos/article_69978.html)

<http://www.diclib.com/>

<http://rosettacode.org>

## 6 Agradecimientos

Aprovecho las presentes líneas para agradecer a Álvaro Marco Añó y Tomás Roig Martínez, alumnos de la Escuela superior de Ingeniería Informática de la Universidad Politécnica de Valencia, por su solidaridad al haberme dejado el trabajo, “*Algoritmos de compresión de información MM*” que realizaron para la asignatura Fundamentos de Sistemas Multimedia, que sirvió de fuente de inspiración para desarrollar la aplicación.

Quería también brindar mis agradecimientos a mi familia por su apoyo y comprensión durante todos estos años.

Quiero agradecer también su inestimable colaboración a Victoria Sánchez Bou, gracias por prestar tu imagen, tu voz y tus conocimientos aparte de tu inconsumible paciencia.

No quería pasar por alto la oportunidad de agradecer a todos los profesores que he tenido durante mi vida académica.

