Práctica 4. Entornos gráficos avanzados: Java Swing

Entornos de Usuario

5 de febrero de 2024

Objetivos

La manipulación de elementos gráficos, tales como figuras geométricas e imágenes, se realiza en Java a través de la API Java 2D. Esta API proporciona un amplio conjunto de métodos para dibujar primitivas, manipular colores, manejar figuras, etc. Uno de los objetivos de esta práctica es usar esta API.
Por otra parte, en esta práctica se introduce Java Swing y algunos de las clases avanzadas de Java Swing que permiten generar aplicaciones con elementos gráficos más complejos, tales como JSlider y JFileChooser. También se introducirán algunos interfaces que permiten la interacción avanzada con el usuario (tales como `MouseListener` y `ChangeListener`).
En esta práctica realizaremos una pequeña herramienta de dibujo gráfico que permita al usuario dibujar polígonos sobre una imagen gráfica previamente cargada (en los formatos JPG, GIF, TIFF, PNG). También se implementará la posibilidad de guardar la imagen modificada.

Índice

Índice de figuras

Índice de listados

1 Introducción

En esta práctica abordaremos el desarrollo de una sencilla aplicación de dibujo que contará con las siguientes características:

- Debe permitir al usuario cargar una imagen gráfica (en cualquiera de los formatos jpg, gif, png y tiff) desde el sistema de ficheros. Para ello deberá utilizar la clase `JFileChooser` de Swing.
- Debe permitir dibujar sobre la imagen cargada polígonos con un número de puntos indeterminado. Esta operación se simplifica mucho usando la clase `Polygon` de Java2D.
- El usuario debe poder seleccionar además el grosor del lápiz, color del lápiz y color de relleno (ver figura 1).

- La interacción del usuario con las herramientas de dibujo se hará a través del ratón del siguiente modo:
  1. El *click* del botón izquierdo indicará el principio de una nuevo polígono.
  2. Los sucesivos clicks con el botón izquierdo indicarán los puntos que delimitan la forma del polígono.
  3. El click del botón derecho indicará el final de creación del polígono.
- La aplicación contará con una zona informativa en la que se deberá mostrar el grosor del lápiz y los colores del lápiz y de relleno seleccionados.
- Será posible definir tantos polígonos como el usuario desee.
- Por último, debe permitir guardar la imagen modificada como un nuevo fichero en el sistema de archivos en el formato jpg. Para ello haremos uso de nuevo de la clase `JFileChooser`.

En la figura 1 se muestra el aspecto que deberá presentar la interfaz gráfica de la aplicación propuesta y en la figura 2 se muestra un ejemplo de uso de la aplicación.

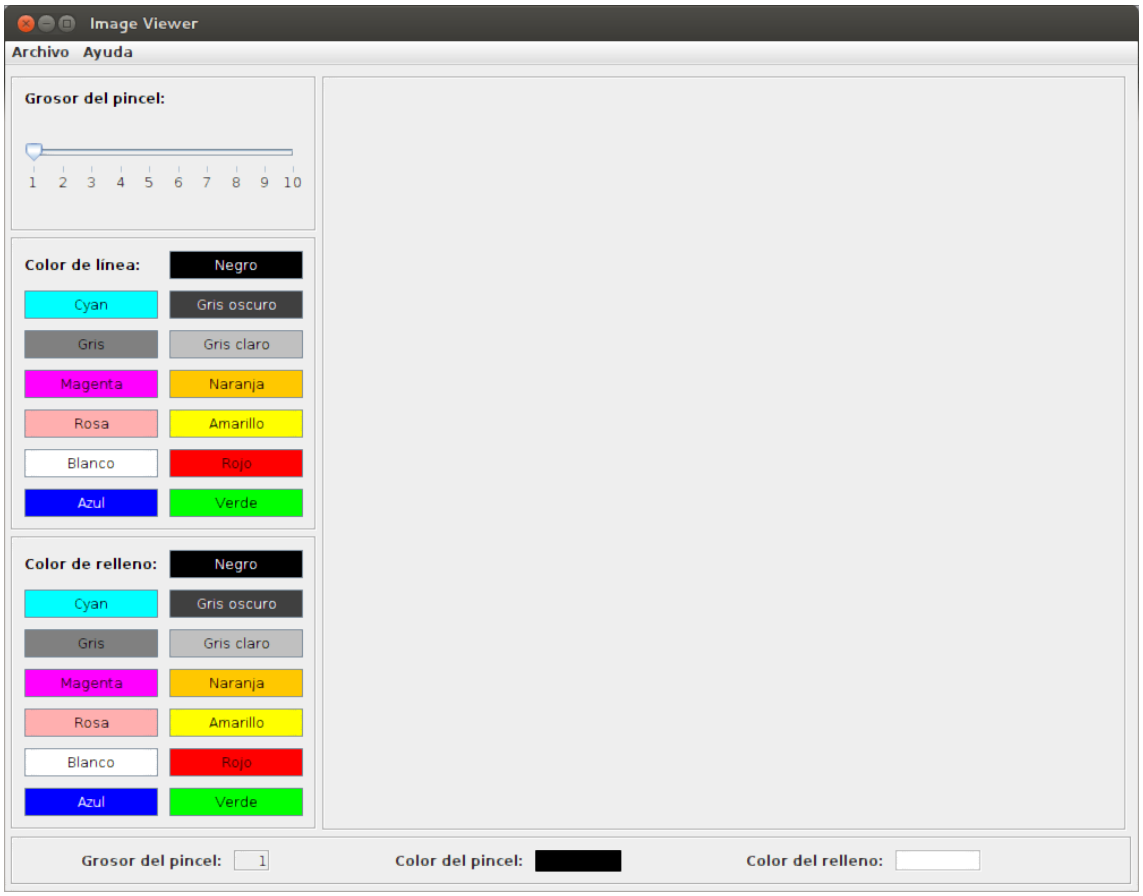

Figura 1: Aspecto general de la aplicación destacando la estructura de paneles.

2 La aplicación

Esta aplicación, como hasta ahora, se basa en la arquitectura M-V-C. Sin embargo, a diferencia de la práctica anterior, en este caso la vista necesita hacer uso del modelo parar leer su estado (la imagen gráfica que el modelo mantiene). Este es el motivo por el que el constructor de la vista recibe una referencia al modelo (ver la línea 18 en el listado 1).

En la figura 3 se proporciona la estructura completa de nuestra aplicación, compuesta por cuatro paquetes:

- **photoeditor**, que contiene la clase principal de la aplicación (`PhotoEditor`). Esta clase no cuenta con atributos y sólo dispone del método `main`, que instancia objetos de las clases del modelo, la vista y el controlador de forma similar a como hemos visto en prácticas anteriores (ver listado 1).

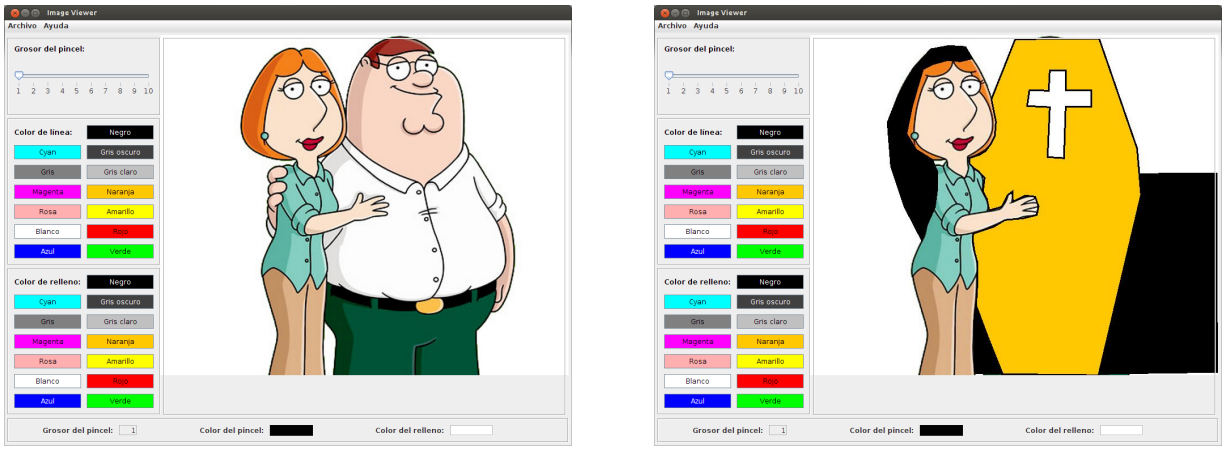

Figura 2: Ejemplo de uso de la aplicación `PhotoEditor`. En la izquierda se muestra la imagen original y en la derecha la imagen modificada mediante la adición de polígonos.

- **photoeditor.model**, que contiene el conjunto de clases que implementan los objetos que manipulará el programa y el modelo. Las clases de este paqueste se verán con más detalle en la sección 2.1.
- **photoeditor.view**, en el que se definen todas las clases que contienen la vista (figura 1). La sección 2.2 se dedicará a explicar los aspectos más relevantes de estas clases.
- **photoeditor.controller**, que contiene la clase del controlador `PhotoEditorController`.

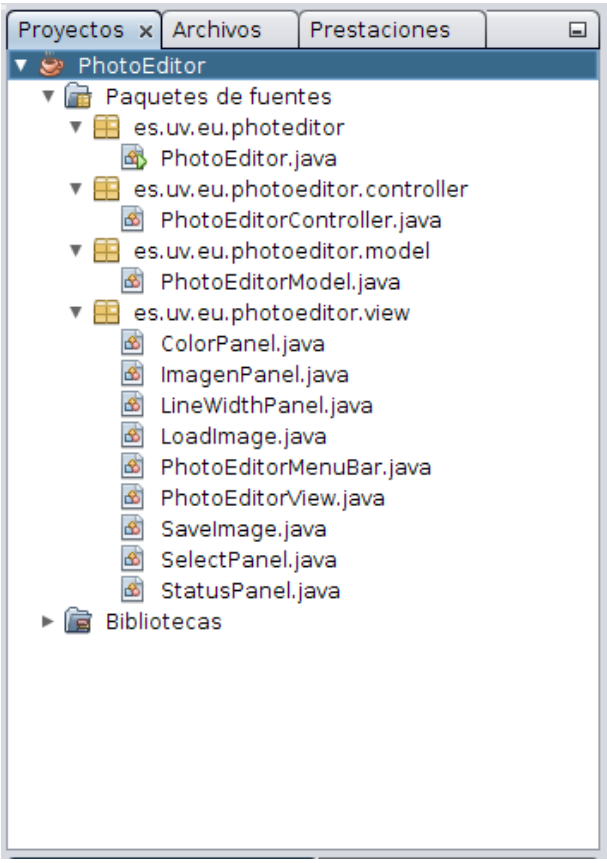

Figura 3: Estructura general del proyecto NetBeans de la aplicación `PhotEditor`.

Listado 1: Código de la clase principal `PhotoEditor.java`.

```
1 package es.uv.eu.photeditor;
2
3 import es.uv.eu.photoeditor. controller .PhotoEditorController;
```

```
4  import es.uv.eu.photoeditor.model.PhotoEditorModel;
5  import es.uv.eu.photoeditor.view.PhotoEditorView;
6
7  /**
8   *
9
10  */
11 public class PhotoEditor {
12
13    /**
14     * @param args the command line arguments
15     */
16    public static void main(String[] args) {
17       PhotoEditorModel model = new PhotoEditorModel();
18       PhotoEditorView view = new PhotoEditorView(model);
19       PhotoEditorController controlador = new PhotoEditorController(model, view);
20    }
21 }
```

Para entender mejor cómo se integran los diferentes elementos de la arquitectura MVC que vamos a desarrollar, es necesario primero tener un buen conocimiento del modelo de datos que vamos a utilizar. En el siguiente apartado se describe con detalle la implementación del modelo (que se proporciona con el código base del proyecto) y a continuación se describen las clases vista y controlador.

2.1 El modelo

El modelo se implementa en la clase `PhotoEditorModel` (ver listado 2). Este modelo mantiene una imagen gráfica almacenada en un objeto de la clase `BufferedImage`. Se utiliza un objeto de este tipo porque permite cargar inicialmente una imagen a partir de un fichero (mediante `ImageIO.read`) y posteriormente editarla dibujando polígonos mediante el método `pintaPoligono`. Los métodos principales con los que cuenta son:

- **loadImagen** y **saveImagen** que permiten leer y guardar el `BufferedImage` en fichero a partir de un objeto del tipo `File` que se puede obtener mediante un `JFileChooser`.
- **getImagen**, que devuelve el objeto imagen para que pueda ser dibujado sobre un *canvas*.
- **pintaPoligono** que dibuja el polígono `poly` con el grosor y el color de lápiz indicados por `penWitdth` y `penColor` y el color de relleno `fillColor` sobre el entorno gráfico asociado al `ImageBuffer`, lo que modifica directamente esta imagen.

Listado 2: Código de la clase del modelo `PhotoEditorModel.java`.

```
1  package es.uv.eu.photoeditor.model;
2
3  import java.awt.BasicStroke;
4  import java.awt.Color;
5  import java.awt.Graphics2D;
6  import java.awt.Polygon;
7  import java.awt.image.BufferedImage;
8  import java.io.File;
9  import java.io.IOException;
10 import javax.imageio.ImageIO;
11
12 /**
13  *
14
15  */
16 public class PhotoEditorModel {
17    private BufferedImage imagen;
```

```
18    private String imagenFileName = "";
19
20    public PhotoEditorModel() {
21      try {
22        imagenFileName = "imagenes/imagen_00.jpg";
23        imagen = ImageIO.read(new File(imagenFileName));
24      }
25      catch (IOException e) {
26        System.out.println("Problemas leyendo la imagen '" + this.imagenFileName + "'.");
27        System.out.println("Motivo: " + e.getLocalizedMessage());
28      }
29    }
30
31    public BufferedImage getImagen() {
32      return imagen;
33    }
34
35    public String getImagenFileName() {
36      return imagenFileName;
37    }
38
39    public void loadImagen(File imagenFile) {
40      if (imagenFile != null) {
41        this.imagenFileName = imagenFile.getName();
42        try {
43          imagen = ImageIO.read(imagenFile);
44        }
45        catch (IOException e) {
46          System.out.println("Problemas leyendo la imagen '" + this.imagenFileName + "'.");
47          System.out.println("Motivo: " + e.getLocalizedMessage());
48        }
49      }
50    }
51
52    public void saveImagen(File imagenFile) {
53      if (imagenFile != null) {
54        try {
55          this.imagenFileName = imagenFile.getName();
56          ImageIO.write(imagen,"jpg",imagenFile);
57        }
58        catch (IOException e) {
59          System.out.println("Problemas guardando la imagen '" + imagenFile.getName() + "'.");
60          System.out.println("Motivo: " + e.getLocalizedMessage());
61        }
62      }
63    }
64
65    public void pintaPoligono(Polygon poly, int penWidth, Color penColor, Color fillColor) {
66      Graphics2D gr = (Graphics2D)imagen.getGraphics();
67      gr.setColor( fillColor );
68      gr. fillPolygon (poly);
69      gr.setColor(penColor);
70      gr.setStroke(new BasicStroke(penWidth));
71      gr.drawPolygon(poly);
72    }
73  }
```

2.2 La Vista

En la figura 4 se describe la estructura general de la clase `PhotoEditorView` y se muestra la jerarquía de paneles que la componen. Esta clase deriva de la clase basse `JFrame` que es la que implementa el marco general de la aplicación en Swing. Los elementos principales son:

- **PhotoEditorMenuBar**. Es la barra de menú (no se muestra en la figura) y deriva de la clase `JMenuBar`. Cuenta con las operaciones "Cargar imagen...", "Guardar imagen..." y "Salir" en el menú `Archivo`. Las operaciones cargar y guardar imagen se implementarán mediante las clases `SaveImage` y `LoadImage` comentadas más abajo.

  Nota: En Swing, a diferencia de AWT, es necesario registrar el oyente para cada `JMenuItem` del `JMenu`. En prácticas anteriores sólo hemos registrado el oyente sobre el objeto `Menu` de AWT.

- **SelectPanel**. En esta clase, que deriva de `JPanel`, se integran los paneles que permiten seleccionar el grosor del pincel, el color de línea y el color de fondo. Contiene los siguientes elementos:

  - **LineWidthPanel**. También deriva de `JPanel` y contiene una etiqueta identificativa (de tipo `JLabel`) y un componente del tipo `JSlider` que permite seleccionar el grosor del pincel entre 1 y 10 de uno en uno.

    Nota: Consulta en la API javax.swing accesible desde `http://docs.oracle.com/javase/7/docs/api/index.html`, para consultar los métodos que te permiten: 1)definir el espaciado entre marcas, 2)visualizar las marcas, 3)poner etiquetas en las marcas y 4)que sólo sean posibles los valores en la marca.

  - **ColorPanel**. Este panel, que también deriva de la clase `JPanel`, muestra una matriz de `JButton` que permiten seleccionar un color. `SelectPanel` incorpora dos instancias de esta clase que permitirán seleccionar el color del lápiz y el color de fondo del polígono, por lo que será necesario registrar un oyente distinto para cada instancia de la clase.

- **ImagenPanel**. Esta clase también deriva de `JPanel`. Es el área de dibujo donde se muestra la imagen y sobre la cual se dibujarán los polígonos mediante el click del ratón tal como se ha especificado anteriormente. Esta clase además tiene que sobreescribir el método `paintComponent` para repintar la imagen cada vez que se produzca una modificación. La forma de repintar la imagen consiste simplemente en recuperar el `ImageBuffer` que mantiene el modelo y mostrarlo llamando al método `drawImage` de la clase `Graphics`.

- **StatusPanel**. Este panel deriva también de la clase `JPanel` y se limita a mostrar (mediante JLabels) la información de los valores actuales del grosor del pincel, el color de borde y el color de fondo. Esta clase debe contar con los *setter* necesarios para actualizar el valor de estos atributos.

- Las clases de utilidad **LoadImage** y **SaveImage**, que derivan de la clase `JFileChooser` y que se utilizan en el proceso de carga y guardado de la imagen. Estas clases se suministran con el código base del proyecto.

Nota: Identifica el tipo de *layout* que se utiliza en cada uno de los elementos descritos ¿Qué layout te permite apilar elementos vertical u horizontalmente sin especificar el número de elementos y sin que cada elemento ocupe exactamente el mismo espacio? ¿En qué clase o clases lo utilizarías en este caso? ¿Por qué?

Recuerda que será necesario dotar a la clase `PhotoEditorView` de los métodos que te permitan:

- Registrar los oyentes apropiados a cada componente. Por ejemplo: un listener del tipo `ChangeListener` para el componente `JSlider`, un `MouseListener` para recoger los click de ratón sobre el panel `ImagenPanel` y los `ActionListener` para cada item del menú y los `ColorPanel`.

- Acceder a los atributos privados de cada clase que el controlador pueda necesitar para realizar su labor. Por ejemplo, el controlador necesitará leer el valor seleccionado en el `JSlider`.

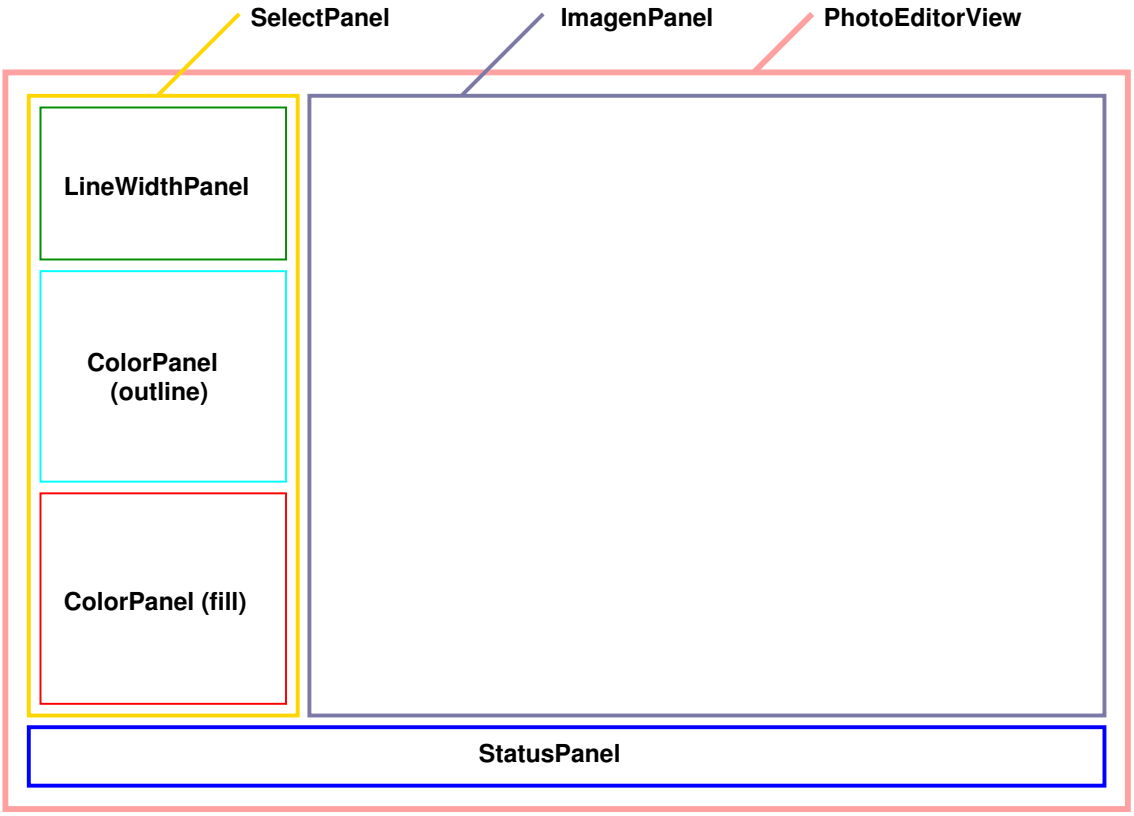

Figura 4: Estructura de la clase `PhotoEditorView` en la que se muestran los paneles que componen la vista.

2.3 El controlador

El controlador se implementa en la clase `PhotoEditorController`. Como siempre, la misión del controlador es actuar de mediador entre la vista y el modelo respondiendo a las acciones del usuario. Para llevar a cabo esta labor, proporciona varias clases y métodos empotrados. En esta aplicación será necesario contar con los siguientes tipos de clases oyentes:

- **ActionListener**. Serán necesarios varios de estos listeners:
  - Para gestionar los items del menú de la aplicación.
  - Para gestionar los botones del `ColorPanel` que selecciona el color del lápiz.
  - Para gestionar los botones del `ColorPanel` que selecciona el color de fondo.
- **ChangeListener**. Será necesario para gestionar el `JSlider` asociado al panel `LineWidthPanel` y que controla el grosor del lápiz. Este listener se invocará cada vez que el usuario modifique la barra deslizante.
- Un **MouseListener** asociado al panel `ImagenPanel` que deberá recoger los clicks del ratón (tanto botón izquierdo como derecho) y su posición para definir los puntos del polígono.

Entre las misiones del controlador, está también la de actualizar el estado de los atributos en el panel de estado `StatusPanel` cada vez que el usuario modifique alguno de estos valores.

Otra de las funciones del controlador es la añadir puntos a un objeto de tipo `Polygon` cada vez que el usuario hace un click con el botón izquierdo. Este polígono debe ser dibujado sobre el modelo (haciendo uso del método del modelo `pintaPoligono` cuando el usuario haga click con el botón derecho). Una

vez pintado el polígono (click con el botón derecho), será necesario vaciar la lista de puntos del polígono y empezar de nuevo.

Nota: ¿Qué métodos de la clase `Polygon` necesitarás para añadir los puntos y para limpiar la lista de puntos? Consulta la API de Java 2D.

Nota: ¿Qué clase oyente se ocupará de gestionar la adición de puntos al polígono y de invocar el método `pintaPoligono`?

3 Tareas

Para la realización de esta práctica se dedicarán **2 sesiones** de laboratorio. En la primera sesión se deberán entregar todas las clases que componente la vista. En la segunda sesión se deberá entregar la aplicación completa.

4 Entrega

La entrega de cada sesión se realizará en Aula Virtual antes del comienzo de la sesión de la siguiente práctica.