Práctica Final. Sistemas Multimedia.

Mario Ruiz Calvo

---

1. Análisis de requisitos.

Requisitos de Gráficos:

- Crear jerarquía de clases que cumpla con las necesidades de la práctica y permita dibujar las siguientes figuras:
    - Línea
    - Rectángulo
    - Rectángulo Redondeado
    - Elipse
    - Arco
    - Curva con un punto de control
    - Trazo Libre
    - Formas personalizadas
        - Figura “M”
        - Figura “Sello M”
- Asociar a la forma una serie de atributos de estilo:
    - Color de fondo
    - Color de frente
    - Contorno
        - Sin contorno, o con contorno
        - Color de contorno
        - Grosor
        - Tipo de trazo
            - Continuo
            - Discontinuo
    - Relleno
        - Color de fondo
        - Color de frente (para degradado)
        - Tipo de relleno
        - Tipo de degradado
            - Horizontal
            - Vertical
            - Diagonal Izquierda
            - Diagonal Derecha
            - Radial

    - Transparencia
        - Grado de transparencia
    - Alisado

Requisitos de Imagen:

- Permitir dibujar formas sobre la imagen.
- Permitir guardar una imagen y las formas que haya dibujada en ella.
- Se podrán aplicar las siguientes operaciones:
    - Brillo
    - Contraste
        - Normal
        - Iluminado
        - Oscurecido
    - Filtros
        - Media
        - Binomial
        - Enfoque
        - Relieve
        - Laplaciano
        - Sobel X
        - Sobel Y
    - Seno
    - Sepia
    - Negativo
    - Bandas
    - Cambio de espacio de color
    - Contraste mejorado
    - Viñeta
    - Espejo horizontal
    - Espejo vertical
    - Tintado
    - Aumentar escala
    - Disminuir escala
    - Rotación
- Las operaciones son concatenables.

Requisitos de la Interfaz Gráfica:

Asociados a gráficos:

- El panel mantiene todas las formas que se van dibujando, cada una con su estilo propio.
- Se puede seleccionar una forma haciendo click en ella.
- Mostrar un bounding-box al seleccionar una figura.
- Se puede mover una forma seleccionada.
- Se puede modificar el estilo tanto del lienzo, como de las formas dibujadas.
- Se puede eliminar una forma seleccionada.
- Se puede copiar (y pegar tantas veces como se quiera) una forma seleccionada.
- Se puede cambiar la ordenación de las formas. Para ello se añaden las siguientes opciones:
    - Enviar al fondo
    - Enviar atrás
    - Traer adelante
    - Traer al frente
- Se puede modificar el tamaño de una forma.
- Mostrar una barra de estado, en la que se indica:
    - El tipo de herramienta seleccionada (de dibujo de forma, de edición o de pegar).

    - Las coordenadas del píxel sobre el que se sitúa el ratón.
    - Los valores RGB del píxel sobre el que se sitúa el ratón.
- Al alternar entre el modo edición y de dibujo o a cambiar la forma seleccionada, los componentes asociados a los atributos (botones, sliders, etc) cambian a los valores que tenga asociado el lienzo o la forma seleccionada.
- Al alternar entre ventanas de imágenes, los componentes asociados a los atributos (botones, sliders, etc) cambian a los valores que tenga asociado el lienzo de la ventana seleccionada o la forma seleccionada de la ventana.
- Cambiar el tipo de cursor dependiendo de si está activado el modo dibujo, edición o pegar.

Asociados a sonidos:

- Añadir una barra de reproducción que cuente con:
    - Botón play/pause.
    - Botón stop.
    - Cronómetro que indica el progreso del audio.
    - Barra de progreso.
    - Tiempo de duración total del audio.
    - Lista de reproducción.
- Al abrir un sonido se añade a la lista de reproducción.
- No añadir sonidos repetidos.
- Al cambiar el sonido en la lista de reproducción, si hay un sonido reproduciendo se detiene (con stop).
- Cambiar el icono de play/pause en función de si se está reproduciendo o no.
- Cuando se pausa un sonido, se detiene el cronómetro (y la barra de progreso).
- Al volver a reproducir después de pausar, el cronómetro (y la barra de progreso) continúa por donde iba.
- Tanto al hacer un stop como cuando la reproducción finaliza de manera natural, el cronómetro (y la barra de progreso) vuelven a los valores iniciales.
- Añadir un botón de grabación de sonido.
- Cambiar el icono del botón de grabación cuando se inicia y se para la grabación.
- Añadir un cronómetro que indique el tiempo de grabación y que aparezca y se oculte cuando se inicie y se pare la grabación.
- Almacenar la grabación en un fichero temporal cuando se inicia la grabación y una vez terminada solicitar la ubicación del archivo y borrar el fichero temporal.
- Añadir la grabación a la lista de reproducción cuando se ha guardado el fichero.
- Permitir cancelar el guardado de la grabación (borrando el fichero temporal).

Asociados a video:

- Utilizar la barra de reproducción de sonido como barra de reproducción de video. (Los botones, cronómetro y barra de progreso funcionan de la misma manera).
- Al abrir un video, se abre una nueva ventana de tipo video.
- Al cambiar entre ventanas de tipo video, si el video se estaba reproduciendo, se pausa.
- Al volver a una ventana de video que estaba pausado, el cronómetro y la barra de progreso se actualizan al tiempo en el que estaba.

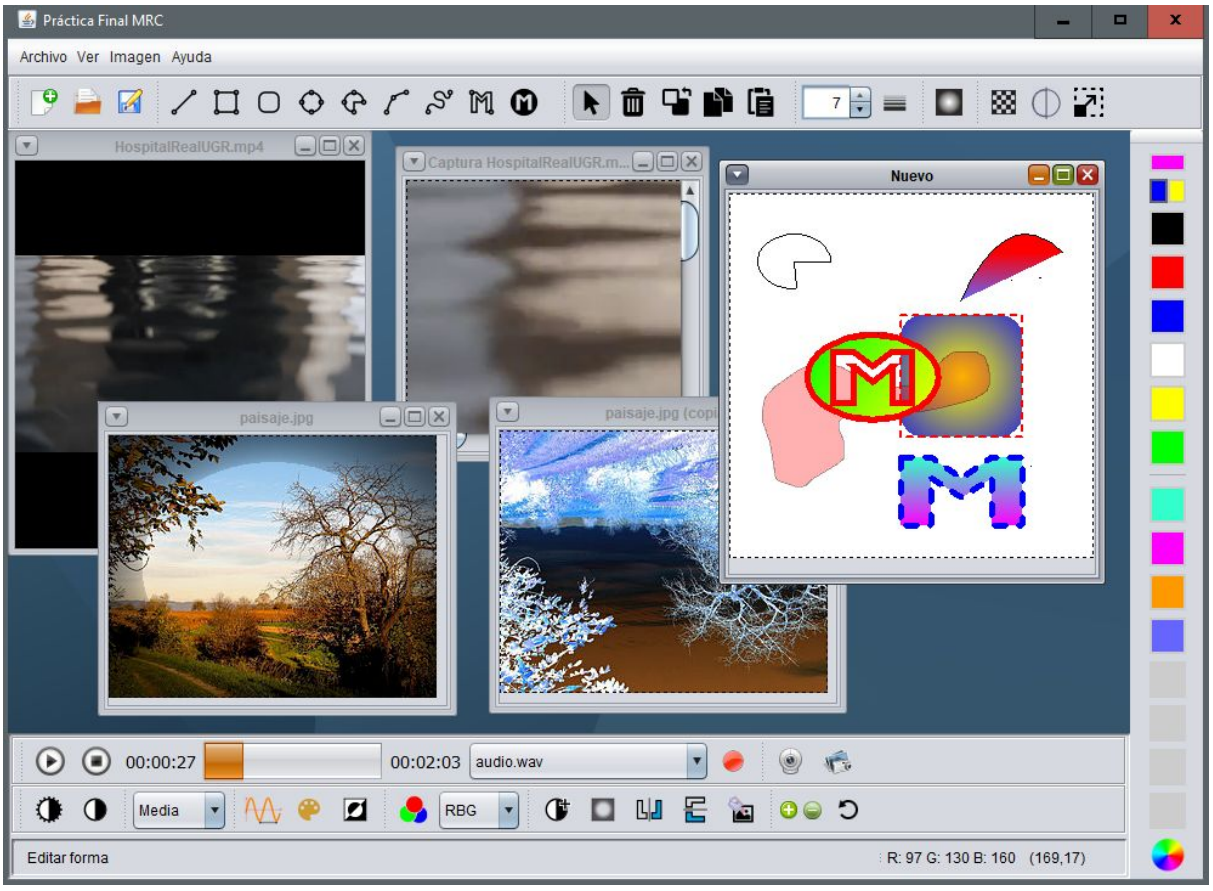
- Si hay una ventana de video activa, la barra de reproducción se utiliza para el video, si no se utiliza para el audio.
- Permitir hacer capturas de imagen de la ventana de vídeo seleccionada. Se abrirá una nueva ventana de tipo imagen que se podrá guardar y a la que podremos aplicarle las operaciones ya comentadas y dibujar formas.
- Añadir un botón que inicie la webcam en una ventana de tipo webcam.
- Permitir hacer capturas de imagen de la ventana de webcam activa. Se abrirá una nueva ventana de tipo imagen que se podrá guardar y a la que podremos aplicarle las operaciones ya comentadas y dibujar formas.
- Utilizar el mismo botón para las capturas de imagen de video y de webcam.

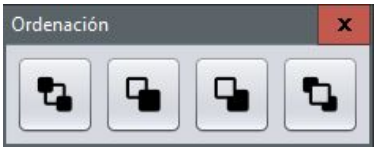
Asociados a la interfaz gráfica:

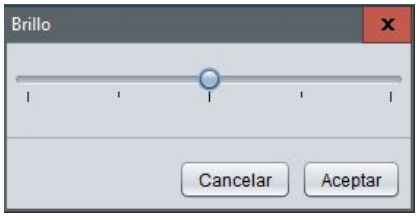
- Crear un panel paleta de colores que permita:
    - Cambiar los colores de contorno, de fondo y de frente.
    - Elegir entre seis colores básicos.
    - Elegir entre ocho colores personalizables, que se pueden añadir mediante un botón que muestra una paleta avanzada y que se van renovando de manera circular. (El color nueve se muestra en el primer botón personalizable).
- Crear un panel para introducir tamaños.
- Controlar que en los TextField del panel de tamaño solo se puedan introducir números y que el número siempre sea mayor de 0 y menor de 9999. Si se introduce algo que no cumpla con esas condiciones, el TextField ignora la pulsación de tecla.
- Crear un diálogo para escalar y redimensionar una imagen.
- Crear diálogos para modificar los atributos del panel y de las formas.
    - Diálogo para seleccionar el tipo de ordenación de la forma.
    - Diálogo para el contorno.
    - Diálogo para el relleno.
- Deshabilitar los botones para cambiar el tipo de relleno, en caso de que la forma no sea rellenable.
- Deshabilitar los botones para cambiar la ordenación si la ventana seleccionada no corresponde a una imagen.
- Crear un diálogo general que muestre un slider y nos permita modificar propiedades de las imágenes y formas:
    - Transparencia de una forma.
    - Nivel de brillo.
    - Parámetro k, de la operación viñeta.
    - Grado de tintado.
    - Rotación.
- Mostrar los cambios en las imágenes y formas a medida que se cambian en los diálogos (antes de cerrar el diálogo), en lugar de cuando se aceptan; y permitir volver al estado inicial si se cancelan los cambios.
- Cambiar el spinner de grosor de la ventana principal a la vez que se cambia en el diálogo de contorno.
- Permitir cerrar los diálogos mediante la tecla *ESC*.
- No permitir acceder a la ventana principal mientras haya un diálogo abierto.

- Cambiar el icono del tipo de relleno de la ventana principal atendiendo al tipo de relleno de la forma seleccionada (o del lienzo, si no hay forma seleccionada).
- Al abrir cualquier diálogo, mostrar los valores asociados a la forma que lo ha abierto (o al lienzo, si no hay seleccionada ninguna forma).
- Cambiar los valores de los componentes, a medida que se cambia de forma seleccionada o el lienzo.
- Mostrar mensajes de error cuando se intenta abrir un diálogo y no hay ninguna ventana seleccionada.
- Mostrar mensajes de error cuando se intenta abrir un diálogo teniendo una ventana seleccionada que corresponde a un tipo de recurso distinto al que maneja el diálogo. (Por ejemplo, intentar cambiar el brillo teniendo una ventana de vídeo seleccionada).
- Añadir un botón “Nuevo” que muestre un diálogo al panel de tamaño y cree un nuevo lienzo vacío.
- Añadir un botón “Abrir” que permita seleccionar ficheros de imágenes, sonido y vídeo (solo los formatos permitidos).
- Añadir un botón “Guardar” que permita almacenar imágenes.
- Cuando se abre un diálogo de selección de fichero (para guardar o abrir), agregar filtros para que solo se muestren los formatos permitidos.
    - Añadir un filtro general que muestre todos los formatos permitidos.
    - Añadir un filtro general que muestre todos los formatos relativos a imágenes.
    - Añadir un filtro general que muestre todos los formatos relativos a sonidos.
    - Añadir un filtro general que muestre todos los formatos relativos a vídeos.
    - Añadir tantos filtros individuales como tipos de formatos permitidos haya.
- Añadir un menú Archivo que incluya las opciones “Nuevo”, “Abrir” y “Guardar”.
- Añadir un menú Ver que permita ocultar o visualizar:
    - Borde de la imagen.
    - Paleta de colores.
    - Barra de formas.
    - Barra de imagen.
    - Barra de sonido.
    - Barra de vídeo.
- Añadir un menú Imagen que incluya las opciones:
    - Escalar/Redimensionar imagen.
    - Duplicar imagen.
- Añadir un menú Ayuda que muestre un diálogo “Acerca de”.
- Al abrir cualquier tipo de ventana (imagen, video, webcam) se ubicará en forma de cascada, tomando como referencia la ventana que esté activa en ese momento.

2. Aspecto Visual







3. Análisis y Diseño

Gráficos:

Java ofrece el paquete *Graphics2D* para dibujar gráficos y atributos. Tiene clases para dibujar formas (*Line2D*, *Rectangle2D*, etc) y clases para cambiar el estilo (*Stroke*, *Composite*, Paint, etc) que se cambian utilizando métodos sobre el graphics. Pero esta manera de trabajar presenta varios problemas.

En lo relativo a las formas, no existe demasiada uniformidad en los métodos que proporciona cada clase. Métodos que se esperarían comunes a todas las formas, sólo los encontramos en algunas de ellas, como el *getWidth*. Muchas veces, a formas concretas les faltan métodos que se esperarían encontrar de manera intuitiva. Es el caso del *Rectangle2D*, en el que si se necesita cambiar el punto que corresponde a la esquina superior izquierda, forzosamente se debe utilizar un método que modifique el rectangle completo. Tampoco se van a localizar métodos para mover la figura o cambiar el tamaño.

Problemas parecidos se van a encontrar en las clases que definen atributos. Hay atributos en los que, para cambiar una propiedad concreta, se deben cambiar todas las propiedades. Es más, se tendrá que crear un nuevo objeto, pues tampoco tienen métodos para cambiar propiedades y hay que hacer uso del constructor.

Sin embargo, el problema más importante es cómo se gestionan los atributos. No es la forma quien define el estilo, como se podría suponer, sino que es el propio graphics quien se encarga de hacerlo. De esta manera, si se quiere cambiar el color de trazo, se hace con el método *setPaint* del graphics, pero a partir de ahí todo se dibujará con ese color. Si se quiere que una forma tenga un color de relleno distinto al de contorno, hay que utilizar dos veces el método *setPaint* (una tercera vez si se quiere volver al color original). Para dibujar una forma se utiliza el método *draw*, pero si se quiere que tenga relleno hay que utilizar el método *fill*, por lo que a la hora de dibujar una forma se tiene que saber como se quiere dibujar. Si ademas de relleno, también se quiere dibujar el contorno hay que utilizar una combinación de ambos métodos, lo que lo hace más tedioso. Todo esto solo para dibujar una forma. Si se quiere dibujar más de una forma y que además cada una tenga su propio estilo la cosa se complica mucho más.

Todos estos problemas indican un mal diseño. Se parte de la idea de que es el graphics el que dibuja, pero debería ser al revés. Es la forma la que tiene el estilo y le debe decir al graphics como quiere que lo dibuje.

Para solventar estos problemas, se hace necesaria la definición de una jerarquía de clases propia en la que se traslade el problema de los atributos a las formas, se trate de unificar el uso de las clases y se proporcionen métodos intuitivos y coherentes para modificar sus propiedades.

Naturalmente, el diseño se puede plantear de muchas maneras y para realizar este proyecto se han ideado varias. En un primer momento, se pensó crear dos superclases: una clase *Forma* general, y una clase *FormaDiagonal* que heredara de esta. El motivo de hacer esto era que en la clase *FormaDiagonal* heredaran formas que se dibujaran mediante una diagonal, pero sobretodo por la idea de la diagonal. Entendiendo que cualquier forma se puede ver como un rectángulo (y que por tanto se puede dibujar con una diagonal). El problema de este planteamiento fue que se hizo pensando solo en la línea, el rectángulo y

la elipse. Precisamente porque cualquier forma se puede ver como un rectángulo, cualquier forma heredaría de *FormaDiagonal*, lo que hace que la clase sea innecesaria. También, a pesar de que todas las formas se pueden ver como un rectángulo, no todas las formas se pueden dibujar mediante una diagonal.

Un segundo planteamiento (que se ha mantenido hasta el final), ha sido separar las formas atendiendo al tipo de atributos que manejan. De esta manera, se han creado dos superclases (abstractas): la clase *Forma* y la clase *FormaRellena*, que hereda de esta y añade el relleno al estilo de la *Forma*. La clase línea, al no tener relleno hereda de *Forma* y todas las demás heredan de *FormaRellena*.

Pero todavía seguía con la idea de la diagonal. Yo ahí veía un patrón. Con la clase *FormaDiagonal* había conseguido unificar el método de dibujo en las primeras formas. Aunque me había dado cuenta de que el planteamiento de la clase no era el más adecuado, el patrón seguía ahí y por eso mantuve esos métodos, dejándolos como vacíos en la clase *Forma* para poder seguir utilizándolos. Pero esto no me convencía en absoluto, pues no todas las clases que heredaban de *Forma* utilizaban esos métodos y fue entonces cuando me di cuenta que debía utilizar una interfaz. Una interfaz define un patrón de comportamiento a una clase y eso es precisamente lo que estaba buscando. La separación no está en que una forma se dibuja de una manera. Las formas son formas y el método de dibujo no es lo que la define, pues un rectángulo se puede dibujar con la diagonal, pero también con un alto y un ancho. De hecho, siguiendo este planteamiento surgieron dos interfaces. Extendiendo la idea de que hay formas que se puede dibujar mediante una diagonal (esto sería definir dos puntos), también hay formas que se dibujan definiendo tres puntos. Se han creado las interfaces *PatronDibujoUnPaso* que gestiona los dos puntos con los que se dibuja una forma, y la interfaz *PatronDibujoDosPasos*, que hereda de la primera y añade un tercer punto. Las formas línea, rectángulo, elipse, arco, M y sello M implementan la primera interfaz y las formas rectángulo redondeado, arco y curva cuadrada implementan la segunda. Las interfaces solo definen el comportamiento, son las formas las responsables de utilizar los puntos para dibujarse de manera adecuada. La forma trazo libre, al tener su propio método de dibujo no implementa ninguna interfaz. Hay que destacar que si se quisiera cambiar el método de dibujo del rectángulo para dibujarlo dando un punto origen, un punto que defina el alto y otro punto que defina el ancho, bastaría con cambiar la interfaz que implementa.

Lo comentado hasta ahora hace referencia a la estructura. Internamente también se podrían plantear diversas situaciones. Se podría pensar que la superclase *Forma* debería heredar de la *Shape* de Java. Esto no parece adecuado, pues si lo que se quiere hacer es una jerarquía propia lo lógico sería que los objetos propios solo invoquen métodos propios. Haciendo un buen diseño, no se van a necesitar más. Es por esto que cada forma concreta contiene una variable del *Shape* correspondiente y es la clase propia la que gestiona la variable.

Además de los métodos concretos de cada forma, la clase *Forma* tiene algunos métodos comunes a todas, como son el *getShape* (método abstracto con visibilidad de paquete que se utilizará para dibujar la forma), el *get* y *setLocation* (abstracto), *get* y *setWidth*, *get* y *setHeight*. La definición de las interfaces ha permitido que se pueda generalizar su implementación en la superclase (exceptuando el caso de la curva cuadrada) y no haya que implementarlas en cada forma concreta. En el caso del trazo libre,

al ser un caso particular que no implementa ninguna interfaz, necesita sobreescribirlas todas.

En cuanto al estilo, aunque se podrían haber puesto un montón de variables en la clase *Forma* para cada atributo, me ha parecido más adecuado y se ha decidido crear una clase concreta para cada tipo de atributo y dos clases para agruparlos. De esta manera se definen las clases *Contorno*, *Relleno*, *Transparencia* y *Alisado*, además de las clases *Estilo* y *EstiloRelleno*(análogas a las clases de formas). Cada clase atributo tiene métodos para cambiar cada una de sus propiedades (las propiedades concretas se mencionan en el análisis de requisitos). Las clases *Estilo* y *EstiloRelleno* tienen métodos para acceder o modificar las variables de los atributos y las clases *Forma* y *FormaRelleno* sólo contienen un objeto de tipo estilo y métodos para modificar sus propiedades, no sus atributos. La razón de hacer esto así, es que a un usuario que utilice la biblioteca y quiera dibujar una forma no le interesa saber que internamente el grosor pertenece al atributo contorno, simplemente quiere modificar el grosor de una forma. Por tanto, entiende que una línea no tiene un contorno, tiene un grosor de contorno.

En cuanto a la manera de dibujar la forma se hace mediante el método *draw* de la clase *Forma*. Este a su vez, llama a los métodos *drawEstilo*, drawForma y *drawBoundignBox* (que dibuja el boundignbox de la forma en caso de que la forma este seleccionada). De manera que en la clase *FormaRelleno* solo hay que sobreescribir el método *drawForma* que será el encargado de gestionar la parte del relleno. Todos estos métodos tienen como parámetro el *Graphics2D* sobre el que se van a pintar (siguiendo con la filosofía de que es la forma la que se pinta en el graphics y no el graphics el que pinta la forma).

Para el estilo, en un primer momento se pensó crear un método con visibilidad de paquete en cada clase atributo al estilo de *getShape* para utilizarlo en los métodos *draw* de la forma. Se tendría entonces métodos como *getStroke*, *getPaint*, etc. Sin embargo, pensando de nuevo en el usuario que no tiene porqué entender de ese tipo de objetos, se han creado métodos *paint* en cada clase, al estilo de los métodos *draw* de las formas. De esta manera, solo se tiene que modificar las propiedades del atributo (sin tener en cuenta la estructura interna) y hacer uso del método *paint* para aplicarlo, ya que es el atributo el que se aplica en el graphics y no al revés. Con el método *paint* se consigue además unificar el método de aplicación de todos los atributos y se hace más intuitivo.

El esquema final de la aplicación quedaría así:

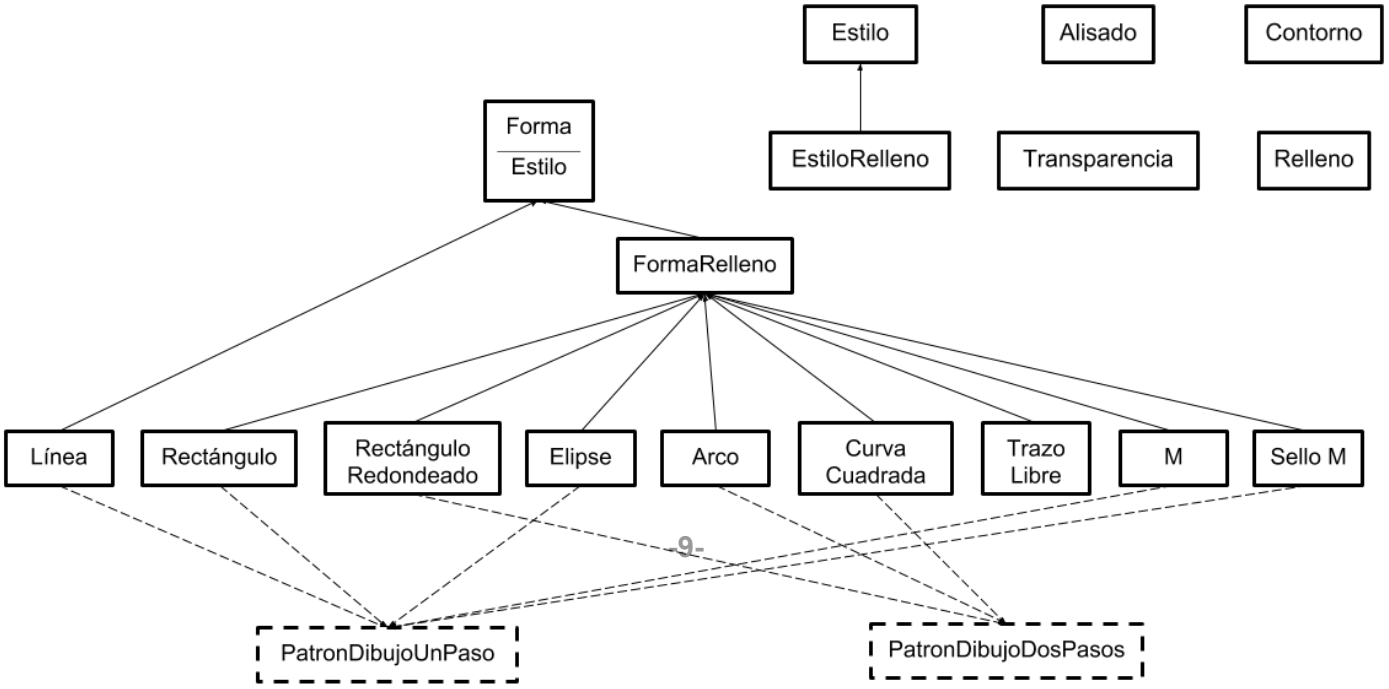


Imagen:

Para gestionar imágenes Java cuenta con un modelo (inmediato) formado principalmente por las clases *BufferedImage*, *BufferedImageOp* e *ImageIO* que permiten tratar con imágenes, procesarlas, leerlas y escribirlas. La clase *BufferedImage* está compuesta (a un nivel de abstracción alto) por un objeto *Raster*, y un *ColorModel*. Para procesamiento de imágenes ofrece la clase *BufferedImageOp* que implementa operaciones punto a punto sobre la imagen y que cuenta con una serie de clases "estándar" para aplicar transformaciones (*ResolveOp*), máscaras (*ConvolveOp*), transformaciones afines (*AffineTransformOp*), funciones (*LookupOp*), entre otras. Además permite definir operaciones propias sin más que extender de la clase *BufferedImageOp* y sobreescribiendo los métodos que se necesiten, por lo que las posibilidades de mejora son muy grandes.

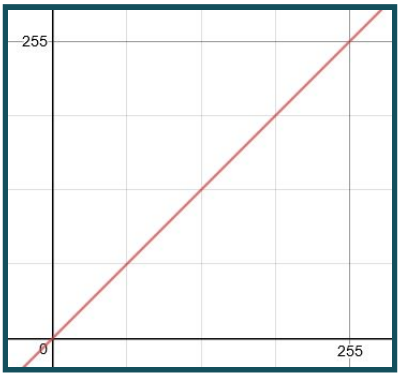
Por todo ello, los medios que ofrece Java para imágenes son suficientes para abordar los requisitos que plantea este proyecto. Se han utilizado objetos de la clase *ConvovleOp* para aplicar los filtros de media, binomial, enfoque, relieve, laplaciano y sobel. Objetos de la clase *LookupOp* para implementar los filtros de seno, negativo y contraste (utilizando también la función seno). Se han creado clases propias heredando de *BufferedImageOp* para definir operaciones nuevas como la operación sepia, viñeta, tintado, "espejo horizontal" y "espejo vertical". A continuación se detallan como se han implementado estas últimas (dado que las operaciones sepia y tintado se han visto en prácticas y teoría no se explicarán) y la operación propia utilizando *LookupOp*:

Contraste utilizando seno:

Para poder aplicar una operación mediante un objeto de la clase *LookupOp* es necesario crear un objeto *ByteLookupTable* al que a su vez hay que pasarle como parámetro un vector de tamaño 255 que representa una función con los nuevos valores que va a tomar cada componente del pixel.

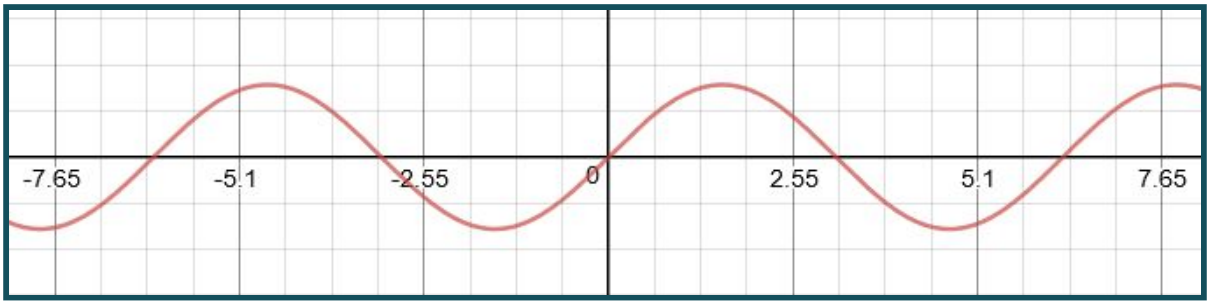
Se busca mejorar el contraste, encontrando una función que nos lo permita.

Si tuviéramos la función identidad, esto es $f(x) = x$, la imágen se quedaría como estaba. La representación gráfica de esta función sería:

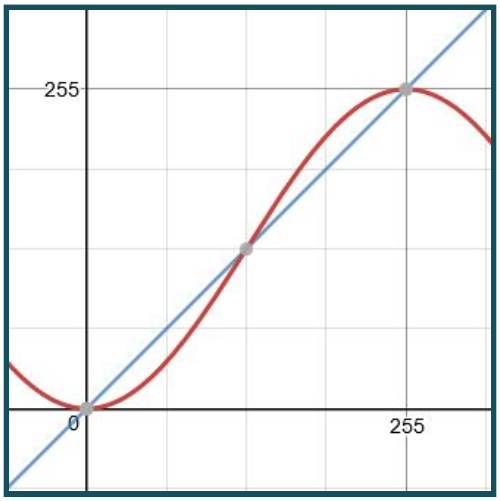


La operación contraste consiste en acentuar la diferencia entre colores claros y oscuros haciendo los colores oscuros más oscuros, y los claros más claros. Teniendo en cuenta esto y observando la gráfica de la función identidad, se necesita encontrar una función que en los valores medios se acerque a la función identidad, en los bajos sea menor, y en los altos sea mayor.

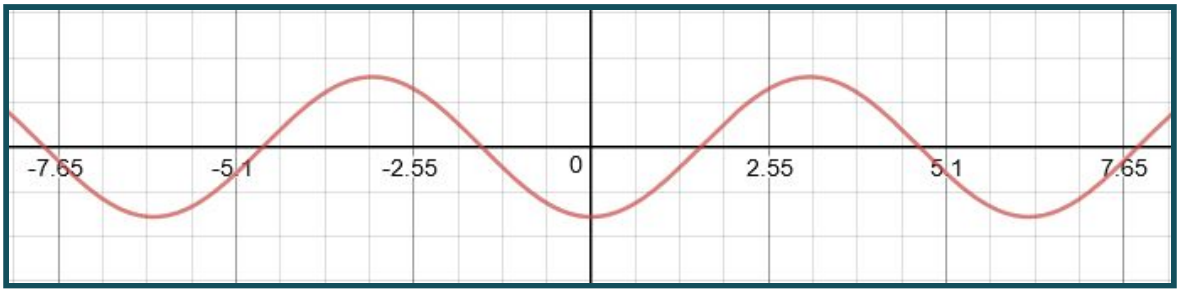
Dicho esto, la función seno presenta la siguiente gráfica:



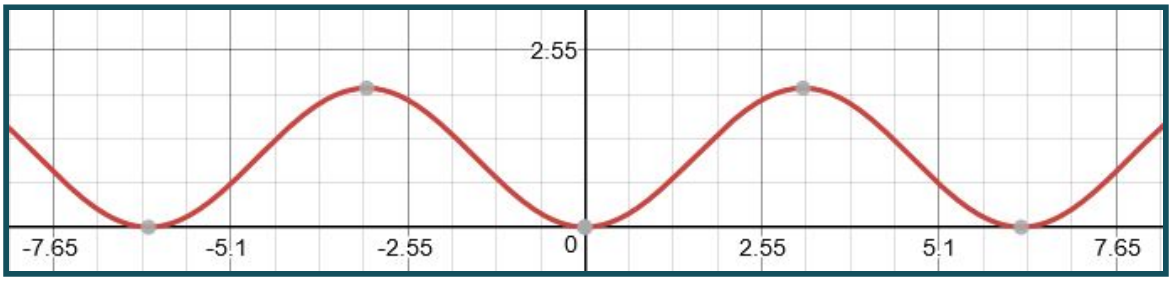
A simple vista podría parecer que no es lo que se busca, pero si se observa entre los valores -π/2 y π/2 se aprecia algo que podría cumplir con las condiciones. Sólo hay que adaptar un poco la función para que cumpla con los requisitos:



En primer lugar hay que trasladar la función en el eje x para situar el valor de -π/2 en el 0. Esto se consigue haciendo que $f(x) = sin(x - \pi/2)$:



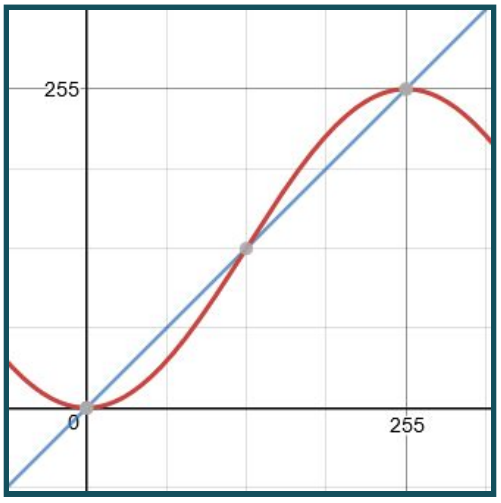
En segundo lugar trasladar la función en el eje y para que $f(0) = 0$, es decir, $f(x) = sin(x - \pi/2) + 1$.



Ahora hay que “estirar” la función para llevar el valor de π/2 al 255 multiplicando la x por $\frac{\pi/2}{255}$ y queda $f(x) = sin(\frac{\pi/2}{255}x - \pi/2) + 1.$

Por último, hay que normalizar la función resultante para obtener valores en el intervalo [0,255] que es el que domina la imagen. Para normalizar el intervalo se tiene que $k = (b-a)\frac{x-min\,x}{max\,x - minx} + a$, donde [a,b] es el intervalo. En este caso $a = 0$, $b = 255$, el máximo corresponde al valor de π/2 en la función $f(x) = sin(x - \pi/2) + 1$, en este caso 2, y el mínimo al valor de la función en el 0, que es 0.

Al final se tiene que $k = \frac{255}{2}$ y $f(x) = k\left(sin(\frac{\pi/2}{255}x - \pi/2) + 1\right)$ y esta es la función que hay construir para pasarlo al *LookupTable*.



Me ha sido un poco difícil encontrar una función para el *LookupOp* y no repetirme. Al final me he repetido por partida doble, pues ya hay operaciones de contraste en el proyecto y se utiliza la función seno para el *LookupOp*. Pero yo quería hacer algo coherente. Podría haber utilizado una función cualquiera de las que no se hayan visto en teoría y adaptarla ajustando el parámetro k, pero no se me ocurría ninguna que hiciera algo con sentido. Por ejemplo, la operación seno que hicimos en prácticas, aunque no queda mal, produce un efecto algo aleatorio y temía que pudiera pasarme esto. Es por eso que decidí hacer una operación de contraste utilizando la función seno de manera diferente (cogiendo “otra parte”).


La imagen de la izquierda corresponde a la imagen a la que se le ha aplicado el contraste normal que se ha visto en prácticas. La imagen de la derecha corresponde a la imagen a la que se aplica mi operación de contraste por seno. Producen un resultado parecido, pero si se observa con atención (bajo mi percepción), mi operación de contraste mantiene un poco mejor el color dejándolo más natural (por ejemplo en los árboles).

Viñeta:

Se ha creado una clase *ViñetaOp* que hereda de *BufferedImageOp* y que implementa la operación "viñeta". Se trata de una operación componente a componente.

Esta operación consiste en aplicar un contraste gradual que será más fuerte en los bordes y se irá atenuando cuanto más cerca se esté del centro de la imagen. Además define una elipse, en la que dentro no se aplicará el contraste y la distancia de un punto a la elipse será la que marque (además de un parámetro k) como de fuerte es el contraste.


La clase tiene dos parámetros: el radio de la elipse, que será un valor entre 0 y 1 y se corresponde con el porcentaje de alto y ancho de la imagen que va a tomar como ancho y alto la elipse; tiene también un parámetro k, con valores entre 0 y 1 que dará más o menos fuerza al contraste.

Para realizar la operación se necesita una función que calcule la distancia de un punto a una elipse, que se define de la siguiente manera:

$$d = \frac{x^2}{cx^2} + \frac{y^2}{cy^2}$$

Donde (x,y) es el punto, y (cx,cy) es el centro de la elipse. Un punto, estará dentro de la elipse si la distancia es menor que 1. De esta manera el valor de cada componente del pixel se obtiene de la siguiente forma:

$$g(x,y,b) = \begin{cases} f(x,y,b), & d < 1 \\ \dfrac{f(x,y,b)}{d + \dfrac{d}{2k}}, & d \geq 1 \end{cases}$$

Se entiende que $f(x,y,b)$ es el valor del píxel (x,y) en la banda b de la imagen original y $g(x,y,b)$ es el valor del píxel (x,y) en la banda b de la imagen resultado.

En cuanto a la manera en la que se ha llegado a $d + \frac{d}{2k}$, ha sido mediante ensayo-error. La distancia, sea cual fuera la imagen, siempre daba valores no más altos de 4. Tenía claro que debía dividir el valor original del píxel por “algo” o multiplicarlo por “0.algo” para que bajara el valor del píxel, pero hacerlo de manera que no diera bajadas muy bruscas pues enseguida el contraste se iba a valores demasiado oscuros o producía valores que se salían del rango [0,255] y quedaban resultados muy raros. Al final, después de probar mucho quedó así, que si bien no me convence del todo produce un resultado aceptable. Cuando k es igual a 1, la viñeta se oscurece pero no es negra del todo.

Algo que ha quedado pendiente, es difuminar un poco la frontera de la elipse, pues se nota mucho con valores altos de k.

Un aspecto positivo que tiene la operación es que se puede aplicar todas las veces que se quiera (y queda un poco mejor que modificando demasiado el parámetro k).

Espejo horizontal y Espejo vertical:

Se ha creado una clase *EspejoHorizontalOp* que hereda de *BufferedImageOp* y que implementa la operación “espejo horizontal”. Se trata de una operación píxel a píxel.

Es una operación muy sencilla que traslada los píxeles de la imagen en el eje horizontal, invirtiendo las columnas. De manera que los píxeles de la primera columna, se trasladan a la última y viceversa:

$$g(x,y) = f(ancho - x - 1,\ y)$$

Donde $f(x,y)$ es el pixel de la imagen original, $g(x,y)$ el píxel de la imagen resultado y $ancho$ el ancho de la imagen original (y resultado).

De manera análoga se define la clase *EspejoVerticalOp*, en la que en lugar de invertir los píxeles por columna lo hace por filas.

$$g(x,y) = f(x,\ alto - y - 1)$$

Donde $f(x,y)$ es el pixel de la imagen original, $g(x,y)$ el píxel de la imagen resultado y $alto$ el alto de la imagen original (y resultado).

Sonido:

La API de Java para sonido, *Java Sound API,* puede resultar un tanto confusa pues tiene una estructura compleja, formada por muchos elementos. Hay que manejar formatos, mezcladores y líneas. Es por ello que se ha proporcionado el paquete *SM.Sound* que añade una capa de abstracción a esta API ofreciendo métodos más simples.

No obstante, se han echado en falta algunos métodos que se veían necesarios para la gestión de la barra de reproducción y se ha decidido crear una clase, *Player*, que contenga una variable *SMClipPlayer* y que añada los métodos nuevos.

La clase *SMClipPlayer* cuenta con métodos para reproducir, pausar y parar (play, pause y stop). Sin embargo, se necesitaban métodos (y variables para gestionarlos) para saber cuando se estaba reproduciendo un sonido, o cuando estaba pausado (diferenciándolo de cuando se había parado mediante stop). También se han añadido métodos para obtener la duración del sonido en segundos y en formato String.

Se ha asociado al player a un fichero, de manera que si se quiere reproducir un sonido distinto, en lugar de crear un nuevo objeto *Player*, hay que hacer un *setPlayer* indicando el nuevo fichero de sonido.

Vídeo:

La API de Java para medios continuos, *JMF*, aunque completa no está debidamente actualizada y faltan muchos codecs recientes. Se ha utilizado entonces la librería *VLCj* que es suficiente para los requisitos del proyecto y la *WebCam Capture API* para la webcam.

Interfaz Gráfica:

Se ha intentado que la interfaz gráfica sea visualmente sencilla, evitando que en la ventana principal aparezcan sliders, ni componentes excesivos. Por ello se ha decidido crear diálogos que agrupen algunas propiedades y no sobrecargen la ventana principal. Esto ha hecho necesaria la definición de dos nuevas jerarquías de clases simples: una relativa a los propios diálogos y otra a los listener que los manejan.

En la jerarquía de diálogos, tenemos una superclase abstracta *Dialogo* (hereda de *JDialog*) que implementa algunas características que van a tener en común todos los diálogos: que el diálogo no sea resizable, que la propiedad *modalityType* sea *APPLICATION_MODAL* (para no permitir acceder a la ventana principal mientras está abierto el diálogo), que el diálogo se pueda cerrar mediante la tecla *ESC*. Además define una variable para controlar si se ha cerrado la ventana mediante el botón aceptar, o se ha hecho mediante el botón cancelar o cerrar.

De la clase *Dialogo* heredan las clases *DialogoContorno*, *DialogoContraste*, *DialogoEscalar*, *DialogoOrdenacion*, *DialogoRelleno* y *DialogoSlider* (los diálogos se pueden ver en el apartado de aspecto visual). Mientras que las primeras son diálogos para cambiar propiedades concretas, la clase *DialogoSlider* se puede utilizar para cualquier propiedad, de manera que en la ventana principal hay distintos objetos *DiaologoSlider* para cambiar propiedades como el brillo de una imagen, la transparencia de una forma, la rotación, etc.

El esquema de estas clases es sencillo. Están formadas por una serie de componentes (botones, *TextField*, spinner…) y una serie de métodos *get* para que los listener y las ventanas puedan acceder a ellos.

En el caso de *DialogoContorno* y *DialogoSlider* se pretende que a la vez que cambian los componentes del *DialogoContorno* y el slider del *DialogoSlider* cambien también los objetos a los que están modificando sus propiedades de manera que, por ejemplo, si se cambia el tipo de contorno a punteado antes de cerrar el diálogo se pueda apreciar el cambio en la forma. Sin embargo la clase *Dialogo* pertenece a la biblioteca y se necesita acceder a objetos que son propios del proyecto (como las ventanas) y no de la biblioteca, por lo que se tiene que hacer uso de clases anónimas. Para hacer esto primero se han añadido los listener correspondientes en la clase *Dialogo* correspondiente (*ActionPerformed*, *StateChange*, etc de los componentes) y en ellos se ha añadido un método que habrá que sobrecargar para añadir la funcionalidad que se desea. En segundo lugar se han utilizado las clases anónimas en la ventana principal que sobrecargan los métodos que se han creado, de manera que ya sí se puede acceder a las ventanas, a los paneles de imagen que contienen y sus formas; y para el caso del grosor de contorno también se puede modificar el spinner de grosor que hay en la ventana principal.

Realmente para *DialgoSlider* no se ha utilizado una clase anónima sino que se ha definido una nueva clase en el proyecto, *DialogoSliderListenerPracticaFinal*, ya que al haber muchos objetos *DialogoSlider* en la ventana principal resultaba más cómodo definir una nueva clase.

La jerarquía de listener pertenece al proyecto y no a la biblioteca. Puesto que utiliza objetos que pertenecen al proyecto no se puede generalizar. En ella se tiene una superclase abstracta *DialogoListener* (hereda de *WindowAdapter*) que, como en la clase *Dialogo*, implementa algunas características comunes en todos los *DialogoListener*. Concretamente, gestiona la variable de la clase *Dialogo* que indica si el diálogo se ha cerrado mediante el botón aceptar además de mostrar un mensaje de error en caso de que se intente abrir el diálogo sin haber ninguna ventana seleccionada.

De la clase *DialogoListener* heredan las clases *DialogoContornoListener*, *DialogoContrasteListener*, *DialogoEscalarListener*, *DialogoOrdenaciónListener*, *DialogoRellenoListener* y *DialogoSliderListener*. Como en el caso de los diálogos, la clase *DialogoSliderListener* se puede utilizar para cualquier propiedad modificable con un slider.

El esquema de estas clases es sencillo. Sobrecargan los métodos *windowOpened* y *windowClosed* de la clase *WindowAdapter*. En el primero se accede a los valores de las propiedades de la forma que haya seleccionada (o del lienzo, si no hay forma seleccionada) y se modifican los componentes del diálogo. En el segundo hacen la operación inversa, es decir, una vez se han modificado los valores de las componentes se trasladan a la forma seleccionada o al lienzo. Pero esto solo se hace si el diálogo se ha cerrado mediante el botón aceptar, si no es así no se hace nada con lo que la forma o el lienzo se queda como estaba.

Panel paleta de colores:

Se ha creado una paleta de colores, *PanelPaletaColores*, que es un panel y por tanto hereda de *JPanel*. Este panel está compuesto por *JTextField* que hacen la labor de botones. Tres de estos botones corresponden a los colores de contorno, de fondo, y de frente. Hay otros seis botones que corresponden a los seis colores básicos y otros ocho botones que corresponden a colores personalizados. Hay un último botón inferior (que si es un *JButton*) que se utiliza para abrir un diálogo de paleta de colores avanzado.

Esta clase sólo contiene métodos *get* a los botones básicos, al botón activo (de los tres posibles: contorno, de fondo o de frente) y al botón i del array de botones personalizables y un método *set* para cambiar el botón activo y desactivar los dos restantes.

La funcionalidad de la paleta se desarrolla en una clase *ColorListener* ubicada en el paquete de listeners del proyecto. Al hacer clic en uno de los tres posibles botones activos (contorno, fondo o frente) se activa el color correspondiente, cambiando el color de borde para que se aprecie que está activo y los otros dos se desactivan. Cuando se hace clic en uno de los botones de color (se puede elegir uno de los seis básicos o de los ocho personalizables), el color del botón activo cambia y se aplica en el estilo del lienzo o de la forma que pueda haber seleccionada. Si se hace click en el botón de paleta avanzada, se muestra el diálogo con la correspondiente paleta y cambia el color del siguiente botón al último botón del array de botones personalizables que se haya modificado. En caso de que el último botón personalizable que ha cambiado el color sea el octavo

(último), se cambiará el color del primer botón personalizable y se continuará con los sucesivos.

Panel tamaño:

Se ha creado un panel tamaño que hereda de *JPanel* que está compuesto principalmente por dos *JFormattedField*, y métodos *get* para ellos y para obtener el ancho y alto como enteros en lugar de cadenas.

Además se ha desarrollado una clase *TextFieldsTamañoListener* que hereda de *KeyAdapter* y se utiliza para controlar que sólo se introducen números y que el número que se va construyendo a medida que se pulsan teclas está en el rango [0,9999]. Para esto se sobrecargan los métodos *KeyPressed* y *KeyTyped*. En el primero se consume la pulsación (se ignora) si se pulsa la tecla de retroceso o suprimir y el número que hay en el *TextField* es menor que 10 (con lo que siempre debe haber al menos un número) . En el *KeyTyped* se ignora la pulsación si el número que se va a generar después de la pulsación es 0, si el número es mayor que 9999, o si se ha pulsado una tecla distinta a un número (como una letra).

Panel Lienzo:

Se ha creado un panel lienzo que hereda de *JPanel* y será el encargado de dibujar las formas. Principalmente cuenta con un array de formas (las que se van a dibujar), una forma seleccionada, una forma para copiar, variables para controlar si se está en modo edición, o copiar/pegar, una herramienta (que indica el modo) y un contador de pasos para las diferentes maneras de dibujar una forma. Tiene una variable de tipo *EstiloRelleno* que es la que se va modificando a medida que se utilizan los componentes de la ventana principal y los diálogos y es el estilo que se le pasará a la forma en el momento de su creación. También es el estilo al que se recurre cuando no hay formas seleccionadas y se cambian los componentes a los valores del lienzo.

Para dibujar, editar o pegar formas se sobrecargan los métodos *mousePressed*, *mouseDragged*, *mouseReleased* y *mouseMoved*.

- *MousePressed*:
    - En el modo edición obtiene la figura seleccionada, buscando en el array de formas (desde la última introducida hacia la primera) la forma más cercana al punto donde se ha hecho el pressed.
    - En el modo pegar, pega la forma que contenga la variable copia en el punto donde se ha hecho el pressed, la añade al array de formas y crea una nueva copia en la variable copia para poder pegar la misma figura tantas veces como se quiera hasta que se salga del modo pegar.
    - En el modo de dibujo, crea una nueva forma del tipo de herramienta que esté activa en la ventana.

- *MouseDragged*:
    - En el modo edición, mueve la figura al punto en el que se hace el dragged utilizando el *setLocation* de la forma. Como el *setLocation* es un método de abstracto de la clase *Forma*, y por tanto todas las formas lo deben tener implementado, no es necesario saber que tipo de forma se está moviendo. Destacar que mueve la forma de manera fluida, sin movimientos extraños. También cambia el cursor al cursor de movimiento.
    - En el modo dibujo se redimensiona la forma que se está creando utilizando el punto en el que se hace el *mouseDragged*. Aquí es donde las interfaces que se han creado tienen su razón de ser. Todas las formas excepto el trazo libre implementan la interfaz *PatronDibujoUnPaso* (pues las que heredan directamente de *PatronDibujoDosPasos* lo hacen indirectamente de la de un paso, al extender una de otra). Por tanto, solo es necesario controlar que la figura que se está dibujando no es un trazo libre, hacer un casting de la forma a la interfaz *PatronDibujoUnPaso* y utilizar el método que modifica el segundo punto del paso para redimensionar la imagen. De esta manera tan sencilla, gracias a la interfaz, se evita el tener que comprobar de qué tipo es la forma y se traslada el problema a la propia figura, que será la encargada de decidir cómo utiliza el punto que se le pasa.

      En caso de que la figura sea trazo libre simplemente se añade un punto al trazo (cada cuatro pasos, para no almacenar demasiados puntos) y se incrementa el contador de pasos.
- *MouseReleased*: Este método solo se utiliza en el modo dibujo. Es el encargado de finalizar el trazo (y poner el contador de pasos a 0), o de incrementar el paso para las figuras que se dibujan en dos pasos (arco, rectángulo redondeado y curva cuadrada). También vuelve a poner el cursor por defecto.
- *MouseMoved*: Cambia el tipo de cursor dependiendo de si se está en modo edición o no. En el modo de dibujo, se encarga de hacer el segundo y último paso de las figuras arco, rectángulo redondeado y curva cuadrada. En este caso se comprueba que la forma es la correcta, se hace el casting a la clase *PatronDibujoDosPuntos* y se utiliza el método para modificar el tercer punto. De nuevo, una sola llamada es suficiente para las tres formas y cada forma se encargará de utilizar el punto como necesite. En el caso del arco utilizará el punto para calcular el ángulo final del arco (el ángulo se define respecto a la distancia del punto del moved al centro del arco). El rectángulo redondeado utiliza el punto para calcular la anchura y altura del arco de las esquinas (que se define con respecto a la distancia del punto del moved a la esquina inferior derecha del rectángulo). La curva cuadrada utiliza el punto para modificar el punto p2 de la curva (el punto de control se modifica en el primer paso, en el *dragged*).

Otros métodos de interés del panel lienzo son *eliminar*, que elimina una forma dibujada (eliminandola del array de formas), *copiarForma* (llamando al constructor de

copia de la figura que se va a copiar) y *camibarOrden* (intercambiando las posiciones de dos formas) que se utiliza para las operaciones enviar al fondo, traer al frente, etc.

Para dibujar las formas, sobrecarga el método *paint* que simplemente hace una llamada al método *draw* de cada forma. También se sobrecarga el método *paintBorder* para dibujar un borde al lienzo.

Para seleccionar el tipo de herramienta, en la ventana principal se define una pequeña clase interna que hereda de *ActionListener*. Esta clase se encarga de recorrer el grupo de botones que forman las herramientas del panel, modifica la herramienta y las variables que controlan el modo. Si se sale del modo edición se quita la posible selección de la forma y si se sale del modo pegar se borra el portapapeles. Se encarga también de indicar la herramienta en la barra de estado de la ventana principal. Sin esta clase, se tendría que sobrecargar el *ActionPerformed* de cada botón del grupo.

Panel Imagen:

Se ha creado un panel imagen que hereda de *PanelLienzo*. Se entiende por tanto, que un panel imagen es un panel lienzo que contiene una imagen. Así que hace todo lo que hace el panel lienzo y la única diferencia está en que sobrecarga el método *paintComponent* (para que se pinte la imagen antes que las formas) para dibujar la imagen.

Panel Imagen Práctica Final:

Esta clase hereda del panel imagen y se ubica en el proyecto (no en la biblioteca). El motivo de crear esta clase es parecido al que se utilizó para las clases anónimas de los diálogos.

Cuando se está en el modo edición y se selecciona una forma quería que automáticamente se cambiarán los componentes de la ventana principal (el spinner de grosor, los colores activos de la paleta de colores, seleccionar o no el botón de alisado y cambiar el icono de relleno al tipo de relleno de la forma seleccionada). Como con las diálogos, tenía el problema de que el panel pertenece a la biblioteca y no puede acceder a los componentes de una ventana principal cuando se selecciona la forma en el *getSelectedShape* del panel lienzo.

La solución natural y la más adecuada sería crear un evento que se dispare en el *getSelectedShape* y que la ventana interna se encargue de capturar, pero en la asignatura no se ha visto cómo se crean eventos así que tuve que buscar soluciones alternativas. No se podía tener una referencia a la ventana interna (como lo hace la ventana interna con la principal) pues la biblioteca no conoce a la ventana interna. Llevar las ventanas a un paquete de la biblioteca tampoco era solución pues las ventanas forman parte de un proyecto concreto que no se van a utilizar en otro contexto, como si se podría utilizar la biblioteca.

La solución fue crear una nueva clase (*PanelImagenPracticaFinal*) en el proyecto, cuyo único propósito es el de sobrecargar el método *getSelectedShape* (y *eliminarSeleccion*) para que además de hacer la selección modifique los componentes en la ventana principal.

Por este motivo, en la ventana interna en lugar de un panel imagen lo que se tiene es un *PanelImagenPracticaFinal*.

Ventanas Internas:

Dado que hay diferentes tipos de ventanas internas, se ha definido una jerarquía sencilla de clases. Se tiene una superclase abstracta *VentanaInterna* que implementa algunas características comunes: *closable*, *inconfiable*, *maximizable* y *resizable*. Tiene una variable que indica el tipo de recurso que contiene (que cada clase se encargará de inicializar) y un contador estático de las capturas de pantalla que se han hecho (que se utilizará en el título de las ventanas internas que generen las capturas). Es en el constructor de esta clase donde se posiciona la ventana en cascada, tomando como referencia la posición de la última ventana interna activa.

De la clase *VentanaInterna* heredan *VentanaInternaImagen*, *VentanaInternaVideo* y *VentanaInternaCamara*.

La clase *VentanaInternaImagen* contiene el *PanelImagenPracticaFinal*. Sobrecarga el método *InternalFrameActived* para modificar los componentes de la ventana principal a los valores del estilo del panel imagen cuando se activa la venta. Sobrecarga el método *InternalFrameClosed* para modificar los componentes de la ventana principal a los valores por defecto cuando se cierra una ventana, de manera que si se cierran todas las ventanas los componentes se quedaran con sus valores iniciales. Sobrecarga los métodos *mouseEntered*, *mouseExited*, *mouseMoved* y *mouseDragged* del panel imagen que contiene para mostrar los componentes RGB de la imagen en la posición del mouse.

La clase *VentanaInternaVideo* es la que se encarga de la lógica del reproductor de vídeo. Es la que contiene el reproductor de *VLCj* y el fichero que tendrá asociado. Tiene métodos para hacer *play*, *pause* y *stop* del video., un método que indica si se está reproduciendo el vídeo, un método para obtener la duración en segundos y un método para hacer una captura de pantalla (que creará una ventana interna de tipo imagen). El constructor se encarga de iniciar el *vlcplayer* y de reiniciar el cronómetro. Sobrecarga el metodo *InternalFrameClosed* para parar la reproducción cuando se cierra la ventana. Sobrecarga el método *InternalFrameDeactivated* para pausar el video si se desactiva la ventana (activando otra) y sobrecarga el método *InternalFrameActivated* para, en el caso de que se haya pausado la reproducción activando otra ventana, actualizar los segundos del cronómetro para volver por donde iba, actualizar el label de duración y el valor máximo de la barra de progreso (con la duración en segundos del vídeo).

La clase *VentanaInternaCamara* inicia la webcam. Contiene un método para hacer una captura de pantalla.

En la ventana principal se definen los métodos *getVentanaInternaCamaraActiva*, *getVentanaInternaVideoActiva* y *getVentanaInternaImagenActiva*. Estos métodos se encargan de obtener la ventana interna que está actualmente activa en la ventana principal. Si la ventana interna activa es del tipo que maneja el método la devuelve con el tipo de ventana activa concreto. Si no hay ventana interna activa, muestran un mensaje de error avisando que no hay ventana interna activa y si la ventana interna activa no se corresponde al tipo del recurso que maneja el método se mostrará un mensaje de error indicando que la ventana activa no es del tipo solicitado. Tienen la opción de indicarle que no muestre los mensajes de error para los casos en que solo se necesita comprobar el tipo de ventana sin informar al usuario. La razón de separar este método según el

tipo de recurso, es que normalmente los botones solo tienen asociados un tipo de recurso, por lo que separarlo en tres es una manera muy fácil de asegurarse que la ventana es del tipo que se quiere.

Interfaz gráfica asociada a sonido:

En la ventana principal se incluye un *combobox* que hace la función de lista de reproducción. Cuando se abre un sonido se añade a la lista de reproducción, se pone como seleccionado y ya se puede empezar a utilizar.

Para controlar la reproducción se ha incluido una barra de reproducción que cuenta con un botón *play*/*pause*, un botón *stop*, un cronómetro, una barra de progreso y un *label* para la duración del sonido.

Para controlar el comportamiento de la barra de reproducción se tiene la clase *PlayListener* que implementa la interfaz *LineListener* y sobrecarga el método *udtape*. Esta clase únicamente se encarga de cambiar el icono de *play*/*pause*, según se esté reproduciendo o no y se encarga de parar y reiniciar el cronómetro (del que se hablará más adelante) cuando se haga un *stop*.

A la vez que en los *ActionPerformed* de los botones *play*/*pause* y *stop* se hacen los *play*, *pause* y *stop* del reproductor, también se hace los *play*, *pause* y *stop* del cronómetro y será el cronómetro el encargado de modificar el *label* del tiempo y la barra de progreso. Cuando se hace el *play*, se actualiza el *label* de duración y el valor máximo de la barra de progreso (con la duración en segundos del sonido).

Se incluye también una pequeña clase interna en la ventana principal que implementa la clase *ItemListener* y se encarga de parar la reproducción y el cronómetro cuando se selecciona un sonido de la lista de reproducción y de asociar el fichero del nuevo sonido seleccionado al reproductor.

En cuanto a la grabación de sonido, cuando se inicia la grabación el sonido se almacena en un fichero temporal. Cuando finaliza se muestra el diálogo de selección de fichero para guardarlo, se elimina el fichero temporal y se añade a la lista de reproducción. En caso de que se cancele la operación se elimina el fichero temporal y no se almacena nada.

El comportamiento del botón de grabación se controla en la clase *RecorderListener* que implementa *LineListener* y cambia el icono del botón cuando se inicia y para la grabación.

También se controla en el *ActionPerformed* del botón de grabación el tamaño de la barra de reproducción, de manera que mientras se está grabando, el tamaño de la barra crece para mostrar un cronómetro que indica el tiempo y cuando acaba vuelve al tamaño original, ocultándose(iniciando y parando el cronómetro a la vez que lo hace la grabación).

Interfaz gráfica asociada a video:

Se utiliza la barra de reproducción de sonido para video. De manera que en el *ActionPerformed* de los botones *play*/*pause* y *stop* se comprueba si hay una ventana de vídeo seleccionada. Si la hay actúa como reproductor de vídeo, sino de sonido.

En caso de que la barra de reproducción esté asociada al vídeo, el comportamiento de la barra se controla mediante la clase *VideoListener* que hereda

de *MediaPlayerEventAdapter*. Esta clase sobrecarga los métodos *playing*, *paused*, *stopped* y *finished* (para el caso en que la reproducción finalice de manera natural) para cambiar el icono del botón *play*/*pause* según corresponda. También sobrecarga el método *lengthChanged* para que, en el momento en que se inicie el vídeo por primera vez, se actualice el label de duración y el valor máximo de la barra de progreso.

Cronómetro:

Se ha creado una clase *Cronometro* que hereda de *Thread* y que se utiliza para el *label* de tiempo de la barra de reproducción, la barra de progreso, y el cronómetro que indica el tiempo de grabación.

Está compuesta por un *label*, donde se mostrará el cronómetro, un *JProgressBar* que será la barra de progreso del reproductor. Tiene variables para controlar si está activo, pausado o parado y un contador de segundos.

Cuando se inicia el cronómetro, se lanza una nueva hebra que inicia el contador de segundos. La hebra entra en un bucle controlado por la variable que indica si está activado o no, en el que hace un sleep de un segundo tras el que se actualiza el *label* y la barra de progreso. La barra de progreso se actualiza con los segundos, por lo que son los reproductores de sonido y video los que se deben encargar de modificar el valor máximo a los segundos totales de duración del recurso (como ya se ha comentado) para que funcione bien.

Cuando en el reproductor se pausa o se para la reproducción, la variable que controla el bucle se pone a false con lo que se sale del bucle y finaliza la hebra. Si se ha salido por un stop, se resetean los valores del label y la barra de progreso. Al volver a iniciar el cronómetro se vuelve a lanzar la hebra. Se comprueba que la última vez no se ha parado por un pause. Si se ha parado por un pause, no se reinician los segundos, con lo que el cronómetro continúa por donde iba. Si se ha parado por un stop, quiere decir que la reproducción empieza otra vez desde cero, con lo que se ponen a cero los segundos.

Como en el caso del vídeo se puede pausar el cronómetro, iniciar uno nuevo y más tarde volver al antiguo se necesita un método *setSegundos* para modificar los segundos en el *InternalFrameActivated* de la ventana de video para que al volver a iniciar la hebra pueda continuar por donde iba.