PRÁCTICA 7

Ejercicio "Paint Básico 2D"

Descripción del ejercicio

El objetivo de esta práctica es realizar una versión mejorada del "Paint Básico" realizado en la práctica 4 (y ampliada en la 5). En este caso, incluiremos las siguientes nuevas funcionalidades relativas a dibujo:

- Nuevas formas de dibujo: a la línea, rectángulo y elipse, se le unen el **trazo libre**, la **curva** y una figura que represente una **cara sonriente**.
- Grosor del trazo.
- Transparencia.
- Alisado de las formas.

y éstas relativas a imágenes:

- Se podrán abrir tantas imágenes como se desee, mostrándose cada una de ellas en ventanas internas independientes (una imagen por ventana).
- Las imágenes se podrán guardar (incluyendo las formas dibujadas).

Además, contaremos con las funcionalidades ya incorporadas en la práctica 5:

- El lienzo mantendrá todas las figuras que se vayan dibujando.
- Desplazamiento de las figuras previamente pintadas.

El aspecto visual de la aplicación será el mostrado en la Figura 1. En el escritorio se podrán tener tantas ventanas internas como se quiera, cada una de ellas con su propia imagen y lienzo de dibujo. El usuario podrá elegir entre diferentes formas de dibujo; concretamente, en este ejercicio se incluirán el **trazo libre, línea, rectángulo, elipse, curva con un punto de control y una figura que represente una cara sonriente**[1]. La forma a dibujar se seleccionará mediante botones situados en la barra de herramientas (la barra de estado mostrará la forma activa). Además, la barra de herramientas ofrecerá la opción de cambiar atributos; concretamente, se podrá elegir entre un conjunto de **colores** predeterminados (agrupados en un panel, como se muestra en la Fig. 1), el **grosor** del trazo (seleccionable mediante un *spinner*), si la forma está o no **rellena**, si se aplica o no **transparencia**[2] y si se **alisan** o no las formas en el renderizado (asociadas a estas tres últimas opciones, botones de dos posiciones). En el panel de colores se añadirá la opción "+" que lanzará un diálogo para seleccionar un color[3].

Todas las figuras se pintarán de forma interactiva, es decir, gestionando el par de eventos "*pressed*" y "*draggued*". En el caso de la curva con un punto de control (clase `QuadCurve`) has de tener en cuenta que el proceso de dibujo tiene dos pasos: en el primero se definen los puntos extremos de la curva, siendo el efecto visual el mismo que cuando se pinta una línea (el punto de control se iguala a uno de los puntos extremos); en el segundo paso se define el punto de control (una vez fijados los extremos en el paso anterior). Tanto un paso como otro estarán asociaos a una secuencia pressed-dragged-released. En el caso de la figura "cara sonriente", se puede considerar de un tamaño fijo (en cuyo caso se crea en el *pressed*, pero no se modifica en el *draggued*).

---

[1] Se recomienda definir esta forma usando áreas. Recuerda que para ir agrupando las distintas formas en un área existen diferentes métodos (add, subtract, etc.); es importante que uses el adecuado para obtener el resultado deseado.

[2] Entendida como semitransparencia correspondiente a un alfa 0.5

[3] El siguiente código lanza un diálogo para seleccionar colores; como resultado, devuelve el color seleccionado (o null si no se ha seleccionado ninguno): *Color color = JColorChooser.showDialog(this, "Elije un color", Color.RED);*

El lienzo de cada ventana interna mostrará **el conjunto de formas dibujadas** en ese lienzo (en este caso, y al contrario que en la práctica 4, hay que mantener las figuras dibujadas usando un vector de formas). Todas **las formas de un mismo lienzo se mostrarán con los mismos atributos** de color, grosor, relleno, transparencia y alisado (seleccionados en la barra de herramientas), de forma que un cambio en un atributo implicará el cambio de dicha propiedad en todas las formas dibujadas en dicho lienzo[4]. No obstante, dados dos lienzos distintos, correspondientes a dos ventanas internas distintas, los atributos de uno y otro lienzo pueden ser diferentes entre sí[5].

El usuario podrá **mover las figuras** que ya estén dibujadas. Para ello, se incluirá en la barra de herramientas un botón de dos posiciones, de forma que, si está seleccionada, indicará que la interacción del usuario será para mover las figuras (en caso contrario, dicha interacción implicará la incorporación de nuevas formas).

El menú “Archivo” incluirá tres opciones: “Nuevo”, “Abrir” y “Guardar”. La opción “Nuevo” deberá crear una nueva ventana interna con una imagen en blanco de un tamaño predeterminado (p.e., 300x300), incluyendo un lienzo sobre el que dibujar; la opción “Abrir” deberán lanzar el correspondiente diálogo y crear una nueva ventana interna que muestre la imagen seleccionada; la opción “Guardar” lanzará el correspondiente diálogo y guardará la imagen de la ventana interna seleccionada.

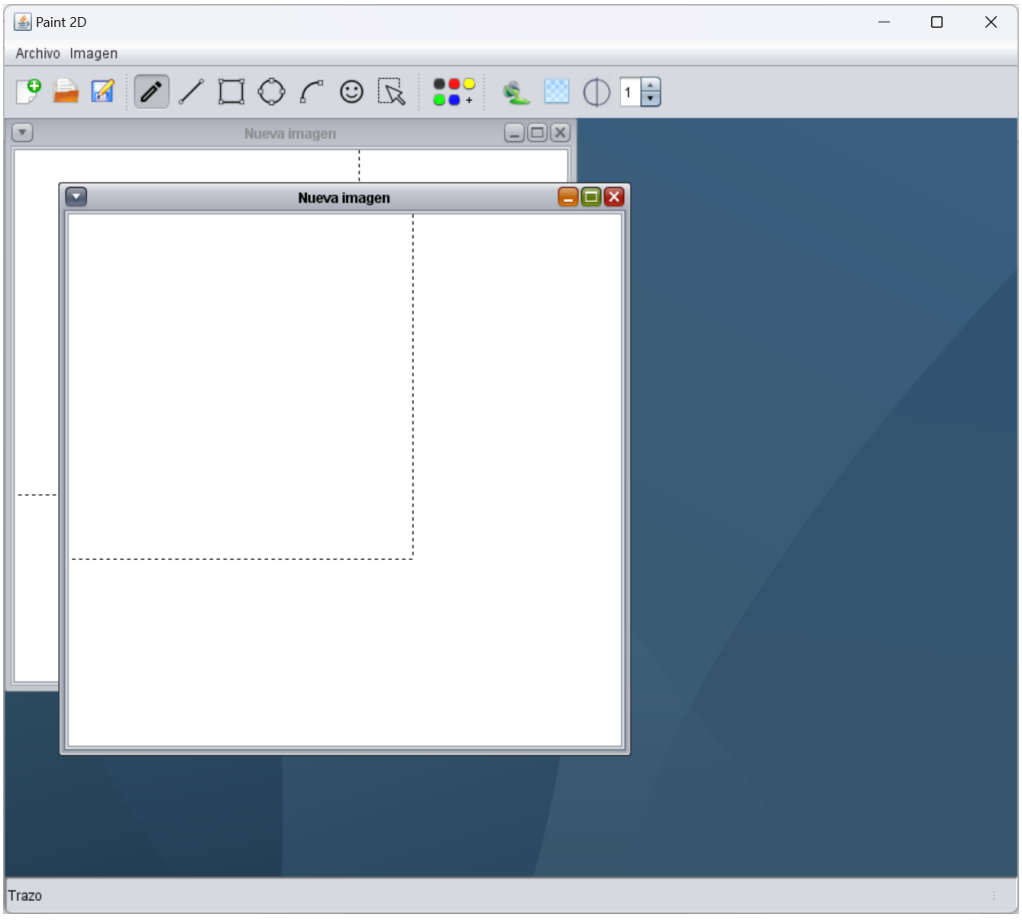


Figura 1: Aspecto de la aplicación

[4] Esto simplifica notablemente la aplicación, ya que, si se considerase como requisito que cada forma mantuviese los atributos con los que se dibujó, implicaría la necesidad de nuevas clases que agruparan información geométrica y de atributos (recordemos con los objetos *Shape* sólo contienen datos geométricos).

[5] Es decir, en una ventana interna se podría estar dibujando en color rojo, mientras que en otra en color azul. Esto será así porque se asume que los atributos con los que se dibuja en un lienzo son propiedades de la instancia (es decir, cada lienzo tiene las suyas).

Algunas recomendaciones iniciales

1. Crear una biblioteca[6] propia que contenga paquetes en los que incluir las clases de nueva creación que puedan ser necesarias en futuras prácticas. En particular, se recomienda incluir en esa biblioteca la clase correspondiente al lienzo y, si las hubiese, las clases de creación propia correspondientes a formas. Para ello:

   - Crear un proyecto en NetBeans de tipo "*Java Class Library*" de título *SM.XXX.Biblioteca*, con "XXX" las iniciales del estudiante
   - En el proyecto, dentro de la carpeta "Paquetes de fuentes", se crearán tantas subcarpetas como paquetes tenga nuestra biblioteca (en el menú contextual del proyecto, seleccionamos *Nuevo → Java Package*). En particular, se propone crear dos paquetes:
     - *sm.xxx.iu*, en la que se irán incluyendo componente y contendores de propósito general que puedan ser útiles a la hora de diseñar interfaces de usuario (por ejemplo la clase `Lienzo2D` que comentaremos posteriormente).
     - *sm.xxx.graficos*, en la que se incluirán clases de diseño propio relativas a gráficos (por ejemplo, aquellas correspondientes a clases de formas).
   - Una vez creada la biblioteca, hay que recordar incluirla en aquellos proyectos en los que vaya a utilizarse (a través de *Propiedades → Biblioteca → Añadir proyecto*). En particular, hay que añadirla en el proyecto de esta práctica 7.

2. Al igual que en la práctica 4, para el área de dibujo se recomienda crear una clase propia `Lienzo2D` que herede de `JPanel` (que incluiremos en la librería que hemos creado). En este caso (véase práctica 5), el lienzo estará situado dentro de la ventana interna (y no en la principal). Dicha clase gestionará todo lo relativo al dibujo: vector de formas, atributos, método `paint`, gestión de eventos de ratón vinculados al proceso de dibujo, etc. Recordemos las principales recomendaciones que se destacaron en las prácticas 4 y 5:

   - El método `paint` deberá contener código centrado sólo en el dibujo de formas (mediante llamadas a métodos `draw` y `fill`) y activación de atributos, pero no de creación de formas o de atributos (de no ser así, la solución se considerará errónea)
   - En el evento de "`mousePressed`" se creará el objeto correspondiente a la forma seleccionada; en el manejador de dicho evento se considerarán las distintas casuísticas para crear uno u otro objeto en función de la forma.
   - En el evento "`mouseDragged`" se modificará la figura creada; en este caso, dependiendo de la clase a la que pertenezca la forma, habrá que invocar a un método u a otro. Por ejemplo, en el caso de la línea vimos que para modificar sus puntos de inicio y final usábamos el método `setLine`; para el caso del rectángulo, usamos el método `setFrameFromDiagonal`. Por este motivo, es necesario preguntarle al objeto "a qué clase pertenece" para poder hacer el "casting" que permita invocar al método. Por ejemplo:

     ```
     if(s instanceof Line2D) ((Line2D)s).setLine(p1,p2);
     else if(s instanceof RectangularShape)
       ((RectangularShape)s).setFrameFromDiagonal(p1, p2);
     ```

     Nótese que el if/else **no se hace en base a la figura seleccionada en la barra de herramientas**, sino considerando la clase de la forma que estamos modificando.
   - Al igual que en la práctica 5, se recomienda definir un método `getFiguraSeleccionada` para gestionar la selección de una figura. En dicho

[6] Si tienes dudas, puedes consultar el vídeo tutorial que hay en PRADO.

método se comprobará si el punto donde se ha hecho el clic está contenido dentro de alguna de las formas del lienzo[7].

- Para desplazar una forma a una posición dada, ya vimos en la práctica 5 que no existe un método en la clase `Shape` común para todas las formas (tipo `setLocation`); de hecho, en la mayoría de clases no está definido un método específico que permita reubicar la forma, por lo que tendremos que programarlo nosotros[8]. El caso de la línea, rectángulo y elipse ya se analizó en la práctica 5; la mayor dificultad en esta práctica estará, por tanto, en el caso del trazo libre, ya que habrá que desplazar todos los puntos del camino[9], y en la curva[10].

  Con carácter general, se aconseja crear clases propias que hereden de las formas básicas y crear objetos de dichas clases (siguiendo el ejemplo de la Línea en la práctica 5); esto nos permitirá definir en cada clase métodos que se adapten a las particularidades de cada caso. En particular, se aconseja definir dos métodos en cada clase: `getLocation()` y `setLocation(Point2D pos)`; el primero devolverá la ubicación de la forma, mientras que el segundo colocará la forma en la localización indicada.

Introduciendo nueva funcionalidad en la clase *Lienzo2D*

Puesto que uno de los requisitos de esta práctica es mostrar una imagen en la ventana interna, es necesario introducir la llamada al método "`drawImage`" en el cuerpo de un método `paint`. La cuestión ahora es en qué clase incorporamos este nuevo código, ¿en una clase nueva específica para mostrar imágenes o, por el contrario, lo hacemos en una ya existente a la que ampliemos su funcionalidad? Ambas opciones tienen sus ventajas e inconvenientes, si bien en esta práctica vamos a optar por la más sencilla: extender[11]. la funcionalidad de la clase `Lienzo2D`.

Si recordamos, en la práctica 5 incorporamos un lienzo en el centro de la ventana interna, de forma que focalizamos todo el código para pintar en ese lienzo (en particular, es donde sobrecargábamos el método `paint`). Dado que uno de los requisitos es que se pueda dibujar

---

[7] En el caso de líneas, no sería "contenido" sino "cerca de" (véase práctica 5).

[8] Recordar que en la práctica 5 se definió como sería un posible método "`setLocation`" para el caso de la línea. En aquel ejemplo, además, fue necesario crear una clase propia heredando de *Line2D* (más concretamente, de *Line2D.Float* o de *Line2D.Double*) para poder sobrecargar el método `contains`.

[9] Para el caso del trazo, que internamente se representa como un vector de puntos, se podría aplicar algo similar a lo que hicimos en la práctica 5 con la línea: ir punto a punto del camino y desplazarlo (previo cálculo de dicho desplazamiento). No obstante, la clase Path2D nos ofrece la posibilidad de aplicarle una transformación a todos los puntos, por lo que podríamos desplazar todo el trazo de la siguiente manera:

```
AffineTransform at;
at = AffineTransform.getTranslateInstance(pos.getX()-loc.getX(),pos.getY()- loc.getY()));
gp.transform(at)
```

siendo `gp` el trazo a desplazar, `pos` la posición a la que se quiere desplazar y `loc` la posición actual (nótese que la diferencia pos-loc nos da el desplazamiento).
Recordar que a toda figura le podemos asociar una localización (por ejemplo, en el caso del rectángulo la esquina superior izquierda, en el caso de la línea el punto inicial, etc.). En el caso de un trazo (un objeto Path2D) podría optarse por asociarlo al primer punto del camino o al último; si se opta por el último, el método `getCurrentPoint()` devuelve su posición (no existe método para el primero, aunque se puede obtener como el primer elemento devuelto por el iterador).

[10] El caso de la curva es muy similar al de la línea: si entendiste cómo se desplazaba una línea, no tendrás problema en mover la curva (si no, es buen momento de repasar lo que hiciste en la práctica 5)

[11] Una solución basada en clases desacopladas y con un una jerarquía asociada, sería una opción más versátil (por ejemplo, un panel específico para imágenes). No obstante, requeriría conocer con algo más de detalle cómo funciona el método *paint* y a qué otras funcionalidades convoca. Por este motivo, en esta práctica se opta por la solución más sencilla: modificar una clase ya existente. Nótese que esto es posible, en este caso, porque se trata de una clase propia (*Lienzo2D*); si esta clase viniese dada por un tercero (p.e., en una biblioteca), no sería posible modificarla y, por tanto, sería necesario extenderla (herencia) para añadirle las nuevas funcionalidades.

además de mostrar la imagen, una posible solución sería incorporar a ese lienzo una nueva propiedad: una imagen de fondo. Concretamente, se propone ampliar la clase ya existente de la siguiente forma:

- Añadir una variable de tipo *BufferedImage* que almacenará la imagen de fondo vinculada al lienzo. Se recomienda declarar la variable como privada y definir los métodos *setImage* y *getImage* para modificar el valor de esta variable:

```
public void setImage(BufferedImage img){
  this.img = img;
}

public BufferedImage getImage(){
  return img;
}
```

- El método *paint* deberá visualizar la imagen y el vector de formas. Para ello, incorporamos la llamada al método *drawimage* antes de pintar el vector[12]:

```
public void paint(Graphics g){
  super.paint(g);
  Graphics2D g2d = (Graphics2D)g;
  if(img!=null) g2d.drawImage(img,0,0,this);
  g2d.setPaint(color);
  g2d.setStroke(stroke);
  for(Shape s:vShape) {
    if(relleno) g2d.fill(s);
    g2d.draw(s);
  }
}
```

- Para que aparezcan barras de desplazamiento en la ventana interna en caso de que la imagen sea mayor que la zona visible, añadir un *JScrollPane* usando el NetBeans (área "contenedores swing") en el centro de la ventana interna y, dentro del él, añadir el *Lienzo2D*. Además, en el *setImage* habrá que añadir el siguiente código para que el tamaño predeterminado del lienzo sea igual al tamaño de la imagen:

```
public void setImage(BufferedImage img){
  this.img = img;
  if(img!=null) {
    setPreferredSize(new Dimension(img.getWidth(),img.getHeight()));
  }
}
```

Nueva imagen

En el gestor de eventos asociado a la opción "nuevo", además de lanzar una nueva ventana interna (como hacíamos en la práctica 5), hay que crear una nueva imagen e incorporarla al lienzo:

```
private void menuNuevoActionPerformed(ActionEvent evt) {
  VentanaInterna vi = new VentanaInterna();
  escritorio.add(vi);
  vi.setVisible(true);
  BufferedImage img;
  img = new BufferedImage(300,300,BufferedImage.TYPE_INT_RGB);
  vi.getLienzo2D().setImage(img);
}
```

[12] En este ejemplo se está asumiendo que sólo se aplican los atributos de trazo y color; en la práctica serán más los atributos que haya que activar.

Con el código anterior, la imagen se verá negra al estar inicializados todos sus pixeles a [0,0,0]. En el código anterior, incluir las líneas necesarias para que la imagen se vea con color de fondo blanco[13].

Abrir imágenes

En el gestor de eventos asociado a la opción de abrir, además del código visto en clase para leer una imagen, hay que incorporar el código necesario para crear la ventana interna y asignarle la imagen recién leída en el lienzo:

```
private void menuAbrirActionPerformed(ActionEvent evt) {
  JFileChooser dlg = new JFileChooser();
  int resp = dlg.showOpenDialog(this);
  if( resp == JFileChooser.APPROVE_OPTION) {
    try{
       File f = dlg.getSelectedFile();
       BufferedImage img = ImageIO.read(f);
       VentanaInterna vi = new VentanaInterna();
       vi.getLienzo2D().setImage(img);
       this.escritorio.add(vi);
       vi.setTitle(f.getName());
       vi.setVisible(true);
    }catch(Exception ex){
      System.err.println("Error al leer la imagen");
    }
  }
}
```

Guardar imágenes

En el caso de guardar, habrá que acceder a la imagen de la ventana seleccionada y almacenarla siguiendo el código visto en clase:

```
private void menuGuardarActionPerformed(ActionEvent evt) {
  VentanaInterna vi=(VentanaInterna) escritorio.getSelectedFrame();
  if (vi != null) {
    BufferedImage img = vi.getLienzo2D().getImagen();
    if (img != null) {
      JFileChooser dlg = new JFileChooser();
      int resp = dlg.showSaveDialog(this);
      if (resp == JFileChooser.APPROVE_OPTION) {
        try {
          File f = dlg.getSelectedFile();
          ImageIO.write(img, "jpg", f);
          vi.setTitle(f.getName());
        } catch (Exception ex) {
            System.err.println("Error al guardar la imagen");
        }
      }
    }
  }
}
```

Obsérvese que con el código anterior se guardaría la imagen, pero no lo que hemos dibujado sobre ella. Es necesario, por tanto, incorporar el código que permita dibujar el vector de formas sobre la imagen (a través del `Graphics2D` asociado a la imagen). Para ello, se aconseja definir el siguiente método en la clase `Lienzo2D`:

[13] No existe un método específico en la clase `BufferedImage` para este propósito. Para poder rellenar la imagen de un color, habrá que acceder a su objeto `Graphics2D` y pintar un rectángulo relleno del color deseado (blanco), cuyas dimensiones coincidan con las de la imagen.

```
public BufferedImage getImage(boolean pintaVector){
  if (pintaVector) {
     // TODO: Código para crear una nueva imagen
     //       que contenga la imagen actual más
     //       las formas
  }
  else
    return img;
}
```

Basándose en lo visto en clase, habría que incorporar en el método anterior el código necesario para (1) crear una nueva imagen y (2) dibujar sobre ella la imagen del lienzo más las formas que éste incluye:

- En primer lugar, crearemos una nueva imagen del mismo tamaño y tipo que la original:

```
BufferedImage imgout = new BufferedImage(img.getWidth(),
                                         img.getHeight(),
                                         img.getType());
```

- En segundo lugar, crearemos el "*graphics*" asociado a la nueva imagen para poder pintar sobre ella:

```
Graphics2D g2dImagen = imgout.createGraphics();
```

- Por último, habría que usar `g2dImagen` para pintar en la nueva imagen tanto (1) la imagen de fondo como (2) las distintas formas del vector; además, las formas tendrían que pintarse usando los atributos activos en el lienzo. En principio, el código asociado sería algo como:

```
if(img!=null) g2dImagen.drawImage(img,0,0,this);
g2dImagen.setPaint(color);
g2dImagen.setStroke(stroke);
for(Shape s:vShape) {
  if(relleno) g2dImagen.fill(s);
  g2dImagen.draw(s);
}
```

  pero, si observamos, ese código corresponde a cuerpo del método `paint`, así que lo correcto sería llamar a dicho método `paint` pasándole como objeto "*graphics*" el asociado a la imagen. Por tanto, el *if* del método anterior podría implementarse como[14]:

```
if (pintarVector) {
    BufferedImage imgout = new BufferedImage( ... );
    this.paint(imgout.createGraphics());
    return(imgout);
}
```

Una vez implementado, en el método `menuGuardarActionPerformed` sustituiríamos la llamada `getImage()` por `getImage(true)`.

---

[14] De cara a futuras prácticas, en las que trabajaremos con imágenes con canal alfa, una mejora en este código sería la siguiente (donde desactiva la propiedad "opaco" del lienzo en caso de que la imagen tenga alfa):

```
if (drawVector) {
    BufferedImage imgout = new BufferedImage( … );
    boolean opacoActual = this.isOpaque();
    if (img.getColorModel().hasAlpha()) {
      this.setOpaque(false);
    }
    this.paint(imgout.createGraphics());
    this.setOpaque(opacoActual);
    return imgout;
}
```

Entrega del ejercicio

Antes del comienzo de la siguiente clase de prácticas, se deberá de entregar la versión final a través de PRADO[15]. Concretamente, al comenzar la sesión de prácticas correspondiente a esta práctica 7, se activará una entrega en PRADO; dicha **entrega estará activa hasta el comienzo de la siguiente sesión de prácticas** (siguiente semana).

En la entrega deberá incluirse un fichero comprimido (zip o rar) que contenga el proyecto (carpeta Neatbeans tanto de la biblioteca como de la aplicación) y el ejecutable[16] (.jar).

Todas las clases de diseño propio que se incluyan en la biblioteca deberán estar documentadas usando **Javadoc**[17]. Debe incluirse tanto la descripción de la clase, como la de sus variables y métodos (en este último caso, incluyendo tanto parámetros como información devuelta).

Posibles mejoras para ir trabajando poco a poco en casa…

Una vez realizada la práctica, se proponen una serie de mejoras para ir trabajándolas poco a poco y darle mayor funcionalidad y mejorar el interfaz (no tienes que hacerlas ahora, ¡eh! 😊). Para mejorar la interfaz de usuario:

- Incluye descripciones emergentes ("*tooltips*")
- Cuando se cambie de una ventana interna a otra, haz que los botones de forma y atributos de la ventana principal se activen conforme a la forma y atributos del correspondiente lienzo[18].
- Muestra en la barra de estado las coordenadas del puntero al desplazarse sobre el lienzo[19].
- Definir el área de recorte (clip) del lienzo haciéndola coincidir con el área de la imagen, de forma que no se dibuje fuera de dicha zona[20].
- Incluir un "marco" (rectángulo) que delimite la imagen.
- Cambiar el puntero en función de lo que se esté haciendo; por ejemplo, "punto de mira" para pintar en el lienzo, "flecha" para seleccionar, "mover" cuando se desplace la forma, etc.
- A la hora de mover una figura, que el "punto ancla" sea el punto donde se hace el clic al seleccionar (y no la esquina superior, i.e., evitar el "salto" de la práctica 5)

Si nos centramos en aspectos relativos a la creación, lectura y escritura de imágenes, posibles mejoras serían:

- Usar filtros en los diálogos abrir y guardar para definir tipos de archivos[21].
- A la hora de guardar, obtener el formato a partir de la extensión del fichero.
- Lanzar diálogos para informar de los errores (excepciones) producidos a la hora de abrir o guardar ficheros.
- Permitir al usuario elegir el tamaño de la imagen nueva a través de un diálogo específico (que se lance seleccionando la opción correspondiente del menú)).

---

[15] En la entrega no son obligatorias las mejoras que proponen para ir trabajando en casa. No obstante, si has trabajado en ellas, inclúyelas en la entrega.

[16] Véase apéndice.

[17] Véase apéndice 2.

[18] Para abordar esta funcionalidad habrá que gestionar el evento de "ventana interna activa" desde la ventana principal. Véase en PRADO el vídeo tutorial donde se explica cómo hacerlo.

[19] La gestión de los eventos de movimiento de ratón no se hará desde la clase *Lienzo2D*, sino desde la ventana principal (o la interna). Téngase en cuenta que en este caso el lienzo es el generador.

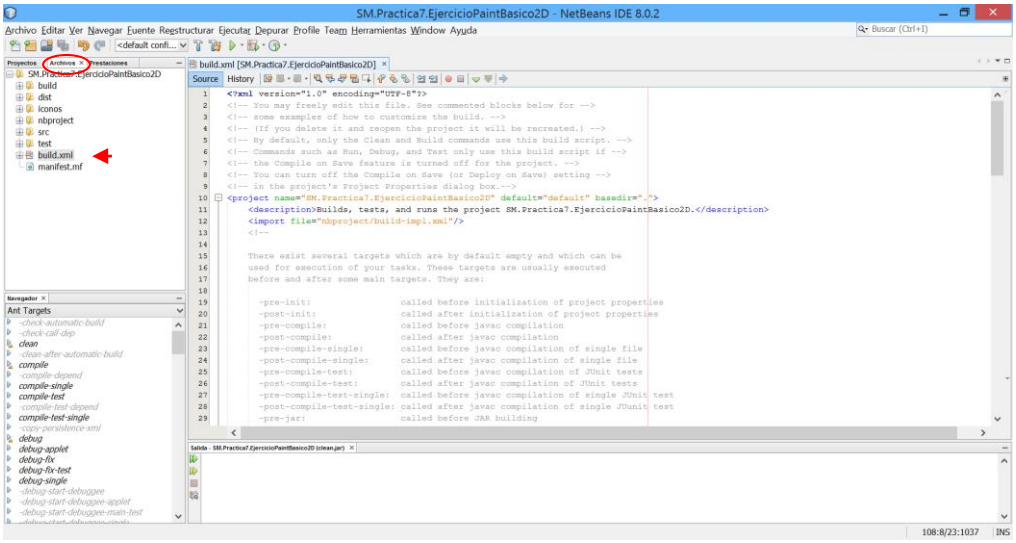
[20] Usar el método `clip` en lugar del `setClip` (el primero combinará la nueva área con la ya existente).

[21] Véase clase *FileFilter* y subclases (por ejemplo, *FileNameExtensionFilter*). Para conocer los formatos reconocibles por *ImageIO*, la clase ofrece métodos como *getReaderFormatNames* o *getWriterFormatNames*

Apéndice 1: Incluir todas las bibliotecas en un único JAR

Como sabemos, para un proyecto dado, NetBeans genera el correspondiente fichero *.jar* en la carpeta */dist*. No obstante, si el proyecto usa bibliotecas externas (como ocurre en esta práctica), éstas no se incluyen en el fichero *.jar*, sino que, en su lugar, crea una carpeta */lib* (dentro de */dist*) en la que incorpora todas estas bibliotecas. Esto implica que, para ejecutar el fichero *.jar*, tiene que estar siempre presente la carpeta[22] */lib* (haciendo más "engorrosa" la posible distribución de nuestro programa -véase el fichero *README.TXT* generado por NetBeans-). Si quisiéramos generar un *.jar* que empaquetase todas las bibliotecas en un único fichero, habría que hacer lo siguiente:

1. Nos vamos a la sección "Archivos" (junto a "Proyectos", en el panel superior izquierdo) y seleccionamos el fichero *build.xml*:



   Como vemos, hay muy poco código (la mayoría del fichero es una sección comentada que explica cómo ampliarlo)

2. Incorporamos el siguiente código XML al final del archivo *build.xml* (antes del *tag* de cierre *</project>*):

```
<target name="-post-jar">
  <property name="pack.jar" value="dist/${application.title}.pack.jar"/>
  <echo message="Packaging into a single JAR at ${pack.jar}"/>
  <jar jarfile="${pack.jar}">
    <zipfileset src="${dist.jar}" excludes="META-INF/*" />
    <zipgroupfileset dir="dist/lib" includes="*.jar" excludes="META-INF/*"/>
    <manifest>
      <attribute name="Main-Class" value="${main.class}"/>
    </manifest>
  </jar>
</target>
```

3. Ahora, al "Limpiar y construir" nuestro proyecto, además del *.jar* que se obtenía antes, se generará otro fichero *.pack.jar* que incluirá todas las bibliotecas y, por lo tanto, se podrá ejecutar de forma autónoma sin necesidad de estar junto a la carpeta */lib*. En caso de tener que distribuir vuestro programa, usad este fichero.

[22] Si no está la carpeta */lib*, no tendrá acceso a las clases de las bibliotecas y lanzará excepciones cuando trate de usarlas. Para poder ver el trazado de estas excepciones, habrá que ejecutar nuestra aplicación desde una ventana de comandos. Por este motivo, se aconseja que antes de distribuir una aplicación a un cliente final, ésta se ejecute desde una ventana de comandos para asegurarnos que no se lanzan excepciones de inicio.

Apéndice 2: Javadoc

Java ofrece una herramienta con la que ir documentando nuestro código y poder generar, de forma sencilla y automática, la API asociada a nuestra implementación (en formato HTML). Esta herramienta se llama **javadoc** y es el estándar para documentar clases de Java, estando incluida por defecto en NetBeans.

La forma de utilizarla es sencilla. En primer lugar, a la hora de ir documentando tu código, todo lo que se incluya entre `/**` y `*/` será información que irá a la documentación de tu API. El texto que incluyas entre ambas marcas puede ser texto plano, etiquetas (tags) de HTML o ciertas palabras reservadas precedidas por el carácter "@". Por ejemplo:

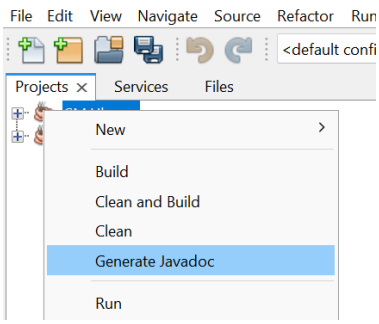
```
/**
 * Devuelve la imagen asociada a este lienzo, incluyendo las formas
 * dibujadas si así se indica. En caso de que se opte por incluir las formas
 * dibujadas, la imagen devuelta será una copia de la original sobre la que
 * se dibujará las figuras incluidas en el lienzo. En caso contrario, se
 * devolverá una referencia a la imagen actual.
 *
 * @param pintaVector establece si se dibujarán o no las figuras del lienzo
 * en la imagen devuelta. Si <code>true</code>, la imagen incluirá las
 * formas dibujadas.
 * @return la imagen asociada a este lienzo, incluyendo las formas dibujadas
 * si así se indica.
 */

public BufferedImage getImage(boolean pintaVector){
   .
   .
   .
}
```

Este tipo de comentarios, que luego se reflejarán en la documentación de la API, se incluyen (1) al **inicio de la clase**, donde se describe la clase y qué representa, (2) antes de cada **variable miembro** de la clase, donde se describe dicha variable, y (3) antes de cada método, donde se describe qué hace dicho método, qué parámetros tiene y qué devuelve.

Hay muchas palabras reservadas, que se pueden usar tanto en la descripción de la clase, como en la de variables y métodos En el caso de la documentación de un método, las más utilizadas son `@param` (para describir los parámetros del método), `@return` (para describir lo que devuelve el método) y `@throws` (para indicar la excepciones que pueden lanzar). Para más información sobre estos parámetros, y sobre la documentación con javadoc en general, pueden consultarse los siguientes enlaces de la Wikipedia o de Oracle. También, a modo de ejemplo, pueden verse las descripciones de las clases estándar Java (p.e., las relativas a gráficos, imágenes, etc. usadas a lo largo de la asignatura)[23].

Una vez incluidos los comentarios siguiendo el esquema descrito anteriormente, para generar la documentación con NetBeans bastará con seleccionar el proyecto, haciendo clic con el botón derecho, y en el menú contextual que nos sale elegir la opción “Generar Javadoc”. Si no hay ningún error, la documentación (ficheros HTML) se creará en la carpeta `dist/javadoc` de tu proyecto. Si hubiese algún error (en los comentarios), éste se indicará en el panel de salida (con enlaces a donde se localiza dicho error).



[23] Recuérdese que, además de la API oficial, desde NetBeans se puede acceder al código fuente de las clases y ver el uso de Javadoc en la documentación de las mismas.