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 y 6). En este caso, incluiremos las siguientes mejoras:

- Nuevo diseño de interfaz con una única barra de herramientas (véase Fig.1)
- A las formas de dibujo ya existentes (línea, rectángulo y elipse), se incorpora una nueva figura que represente una fantasma[1].
- **Cada figura tendrá sus propios atributos** de dibujo (a diferencia de las prácticas anteriores, donde todas las figuras se dibujaban con el mismo color, relleno, etc.). Los atributos, en este caso, serán los ya trabajados anteriormente: color, relleno (práctica 4), grosor del trazo, transparencia y alisado (práctica 6). La forma de selección de estos atributos también varía respecto a las prácticas anteriores.

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. El usuario podrá elegir entre diferentes formas de dibujo; concretamente, en este ejercicio se incluirán la **línea, rectángulo, elipse y una figura que represente un fantasma**. 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 el **color** (con un único botón[2], que lance el diálogo de selección de colores[3]; el botón tendrá como color de fondo el color activo en ese momento), el **grosor** del trazo (seleccionable mediante un *deslizador*), si la forma está o no **rellena**, si se aplica o no **transparencia**[4] y si se **alisan** o no las formas en el renderizado (asociadas a estas tres últimas opciones, botones de dos posiciones).

El lienzo 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 como ya se hizo en la práctica 5). Todas las figuras se pintarán de forma interactiva, es decir, gestionando el par de eventos "*pressed*" y "*draggued*". En el caso de la figura "fantasma", se puede considerar de un tamaño fijo (en cuyo caso se crea en el *pressed*, pero no se modifica en el *draggued*). Cada figura tendrá sus **propios atributos**[5], independientes del resto de formas (es decir, no compartirán los mismos valores). Cuando se dibuje la forma por primera vez, ésta usará los atributos que estén activos en ese momento (que no tienen por qué coincidir con los de las figuras que ya estén en el lienzo).

---

[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] Puede que el botón no muestre correctamente el color de fondo si está en una barra de herramientas; en este caso, poner un panel en la barra de herramientas y dentro de éste el botón. Ponerle al panel un preferedSize (p.e. 30x30)

[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);*

[4] Entendida como semitransparencia correspondiente a un alfa 0.5

[5] Nótese que, a diferencia de la práctica 4, en la que todas las formas se mostraban con los mismos atributos, en este caso cada forma tendrá asociado un conjunto de atributos propio.

.

.

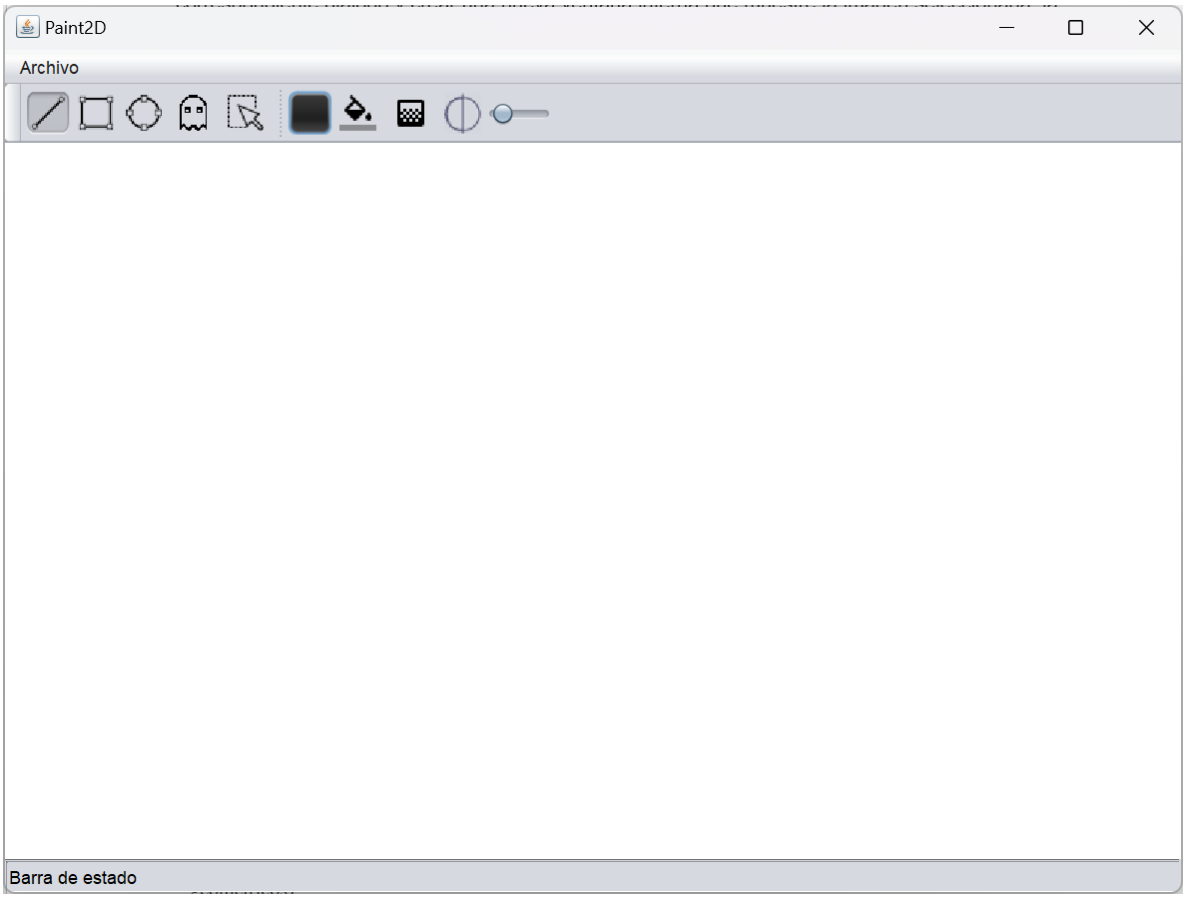


Figura 1: Aspecto de la aplicación

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).

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[6] (.jar). También deberá de incluirse la API (en formato HTML) que se genera a partir del **Javadoc**[7]. Para ello, todas las clases de diseño propio que se incluyan en la biblioteca deberán estar documentadas usando **Javadoc**. 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).

[6] Véase apéndice 1.
[7] Véase apéndice 2.

Algunas recomendaciones iniciales

1. Crear una biblioteca[8] 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 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[9] `Lienzo2D` que herede de `JPanel` (que incluiremos en la librería que hemos creado). 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, 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 seleccionada.
   - En el evento "`mouseDragged`" se modificará la figura creada

3. Crear clases propias para las diferentes figuras de dibujo. El hecho de que cada figura tenga sus propios atributos implica obligatoriamente la definición de una jerarquía de clases asociadas a las formas y sus atributos. Por este motivo, se recomienda incluir en la biblioteca, en el paquete *sm.xxx.graficos*, las nuevas clases diseñadas para abordar este ejercicio. Recordemos algunas notas vistas en teoría en relación con estas clases:
   - Hacer que todas las clases de formas (línea, rectángulo, etc.) hereden de una clase `MiShape` común (puede ser una clase abstracta o una interface, dependiendo del enfoque por el que se haya optado[10]). En dicha clase estarán definidos los métodos comunes a todas las formas (`setColor`, `setGrosor`, `draw`, etc.)
   - Delegar a cada clase el proceso de dibujar esa forma; para ello, se recomienda que cada clase tenga un método `draw` de tipo:

---

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

[9] Al trabajar ya con clases propias (con métodos unificados) y delegar a dichas clases el cómo pintar la figura, el código de la clase Lienzo de las prácticas 4-6 variará bastante; por esta razón se recomienda crear una clase nueva en lugar de modificar la ya existnte

[10] Recuérdese las diferentes alternativas discutidas en la clase de teoría.

```
public void draw(Graphics2D g2d){
  g2d.setStroke(…);
  g2d.setPaint(…);
  .
  .
  .
  g2d.draw(…);
}
```

dentro del cual se activen los atributos de dibujo asociados a la forma y ésta se pinte. Teniendo en cuenta el método *draw* anterior, el método *paint* del *Lienzo2D* quedaría de la siguiente manera:

```
public void paint(Graphics g){
  super.paint(g);
  Graphics2D g2d = (Graphics2D)g;

  for(MiShape s: listaFiguras) {
    s.draw(g2d);
  }
}
```

asumiendo que *listaFiguras* ahora estará declarado en *Lienzo2D* como *List<MiShape>*. Nótese que en el cuerpo del bucle sólo debe haber una línea de código correspondiente a la llamada al método (externo a la clase del lienzo) que pinta la forma.

- Para facilitar mover una forma a una posición dada, se aconseja definir dos métodos en cada clase propia asociada a una figura: *getLocation()* y *setLocation(Point2D pos)*; el primero devolverá la ubicación de la forma, mientras que el segundo colocará la forma en la localización indicada.

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! 😊). Algunas propuestas:

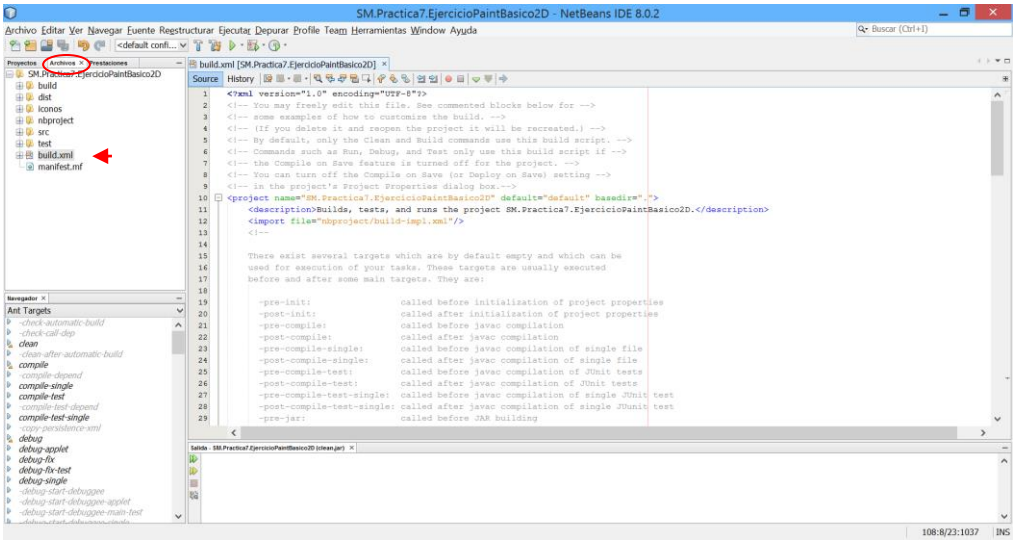
- Incluye descripciones emergentes ("*tooltips*")
- Muestra en la barra de estado las coordenadas del puntero al desplazarse sobre el lienzo[11].
- Cambiar el puntero en función de lo que se esté haciendo; por ejemplo, "punto de mira" para pintar en el lienzo, "flecha" para seleccionar, "mover" cuando 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)

[11] 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.

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[12] */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.

[12] 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:

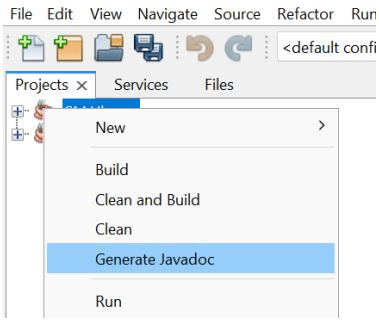
```
/**
 * Devuelve la imagen asociada a este lienzo, incluyendo las formas
 * dibujadas si así se indica. En caso de que se opte por incluir las formas
 * dibujadas, la imagen devuelta será una copia de la original sobre la que
 * se dibujará las figuras incluidas en el lienzo. En caso contrario, se
 * 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)[13].

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).



[13] 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.