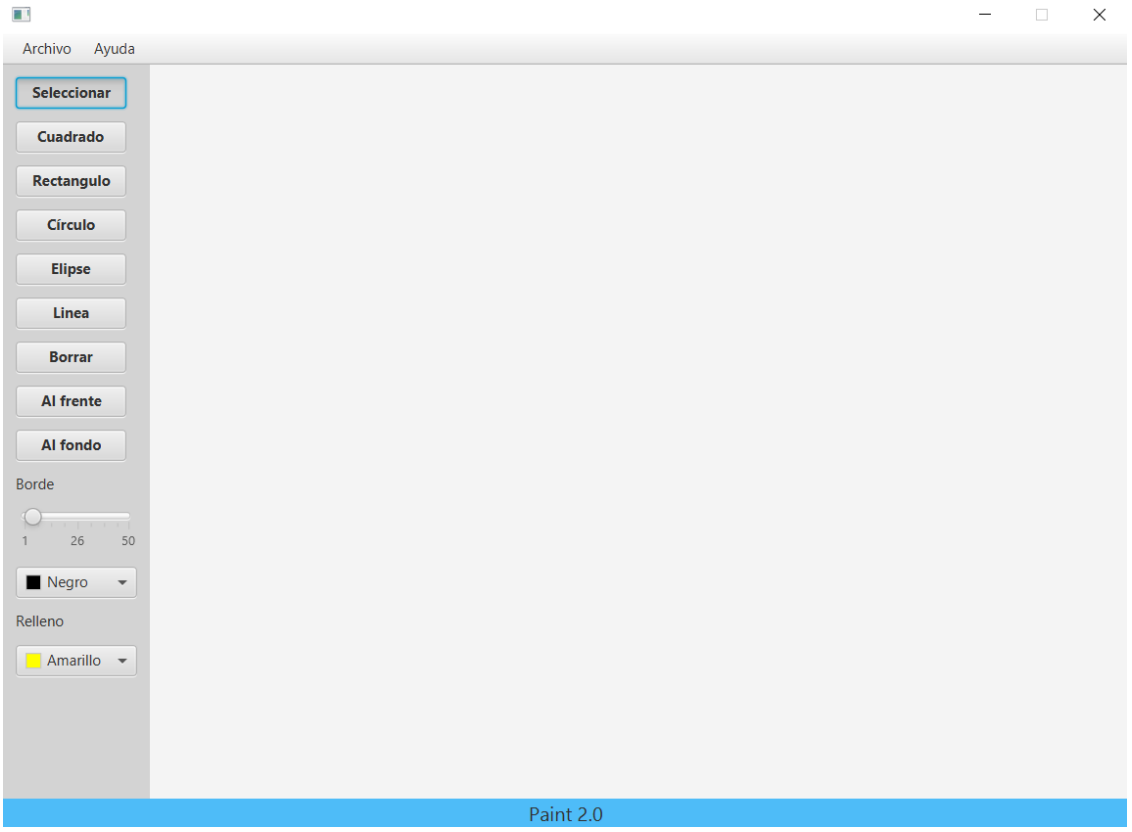
Trabajo Práctico Final - Programación Orientada a Objetos

Benvenuto, Agustin (61448)
abenvenuto@itba.edu.ar

Di Toro, Camila (62576)
cditoro@itba.edu.ar

Chayer, Iván (61360)
ichayer@itba.edu.ar

Grupo 5 - Primer cuatrimestre 2021



Introducción a Paint 2.0

El resultado de este trabajo práctico es una aplicación donde se pueden dibujar figuras como rectángulos, círculos, elipses y cuadrados, y líneas. A lo largo de este documento, nos referiremos al conjunto formado por las figuras y la línea bajo el nombre de *dibujos*. Esto se debe a que las líneas tienen la particularidad de no tener el mismo comportamiento que las figuras mencionadas anteriormente. Esto último lo desarrollaremos más adelante.

A modo de resumen, detallamos las funcionalidad de la aplicación:

- Personalizar el borde de las figuras y las líneas.
- Personalizar el relleno de las figuras.
- Seleccionar una figura para desplazarla sobre la ventana o para borrarla.
- Seleccionar múltiples dibujos para desplazarlos sobre la ventana o para borrarlos.
- Seleccionar un(múltiples) dibujo(s) para traerlo(s) al frente o al fondo de la ventana.

Funcionamiento de Paint 2.0

En esta sección explicaremos en detalle cómo llevar a cabo los bulletpoints desarrollados en la sección anterior, describiremos las funciones agregadas, las modificaciones hechas al código recibido inicialmente y los problemas encontrados.

Descripción de las funcionalidades de Paint 2.0

1. **Ejemplo para obtener un dibujo en la pantalla:**
   Se debe presionar el botón con el label correspondiente de la barra lateral izquierda, hacer click en el área de dibujo para definir el punto superior izquierdo del mismo, mantener presionado el mouse y arrastrarlo hasta el punto inferior derecho. Una vez que se deje de presionar el mouse, la figura aparecerá dibujada en la pantalla. A diferencia de las demás figuras, en el caso del círculo, el punto en el que se presiona el mouse representará el centro del círculo y el punto en el que se suelte definirá el radio del mismo.
   Importante: Las líneas pueden dibujarse en cualquier dirección mientras que las figuras se pueden dibujar únicamente como se detallo en el ejemplo descrito previamente.

2. **Mover un dibujo:**
   Para mover una figura se debe presionar el botón Seleccionar de la barra izquierda, hacer click sobre ella y luego mantener presionado el mouse para moverla en la dirección del cursor. Nótese que cuando una figura es seleccionada, su borde cambia. En la parte inferior de la ventana se ofrece una línea de texto con información relevante. Si se mueve el cursor se pueden ver las coordenadas del mismo en ese instante. Y si el cursor pasa por encima de una figura se indica información de la misma.
   Importante: Las líneas se pueden mover a través de la selección múltiple y también muestran información relevante en la parte inferior de la ventana.

3. **Personalización de dibujos: bordes y rellenos**
   En la barra lateral izquierda hay una Slider y dos colorPicker que permiten personalizar la apariencia de un dibujo seleccionado, si no hay ningún dibujo seleccionado, no se verán cambios reflejados de forma inmediata. La configuración elegida será aplicada a todos los próximos dibujo a crear, hasta que se decida seleccionar una nueva configuración. El Slider permite definir el grosor del borde y el primer colorPicker bajo el label "Border" el color del borde.El segundo colorPicker, bajo el label "Relleno" permite seleccionar el color de relleno. El color del relleno y el color y grosor del borde tienen asignados un valor por defecto (Ver clase FigureStyle).

Observación: En cualquier momento que se seleccione uno o más dibujos, el color del borde cambiará a rojo indicando que fue seleccionado.

4. **Borrado y Selección Múltiple**
   En la barra lateral izquierda hay un botón que permite eliminar una figura seleccionada: Borrar. Si se desea eliminar uno o varios dibujos estos deben ser seleccionados y luego se debe presionar el botón Borrar. Los dibujos seleccionados serán aquellos que entren por completo dentro del rectángulo imaginario que se define desde el punto inicial que se obtiene haciendo click y el punto final que se obtiene soltando el click luego de haber arrastrado el mouse. Con la selección múltiple se pueden mover múltiples dibujos al mismo tiempo o borrarlos todas de una vez. También permite cambiar el borde de todos los dibujos seleccionados o el color de relleno de todas las figuras seleccionadas a la vez. Luego de mover uno o varios dibujos los mismos se mantienen seleccionados. Para deseleccionar, es necesario clickear fuera de la región de selección.
   Importante: El rectángulo imaginario se arma desde arriba a la izquierda hasta abajo a la derecha. Para que el dibujo sea seleccionado por este rectángulo imaginario, todos sus puntos deben pertenecer al mismo.

5. **Profundidad**
   En la barra lateral izquierda hay dos botones que permiten traer al frente o mandar al fondo un dibujo seleccionado: Al frente y Al fondo. En caso de haber seleccionado más de un dibujo (con la selección múltiple) la acción se aplica para todos los dibujos de la selección aunque solo una de ellas estará efectivamente al frente o al fondo de las demás. En el caso de hacer click en un punto donde se encuentran dos o más figuras solapadas, se selecciona solamente la de más 'arriba' tal como ocurría en la implementación original.

Funcionalidades agregadas y modificaciones hechas al proyecto original

A continuación explicaremos las implementaciones, funcionalidades y modificaciones hechas en el trabajo práctico. Para ello dividiremos la explicación en dos partes: Backend y Frontend.

Backend

Analicemos las clases que componen al backend:

Carpeta models:

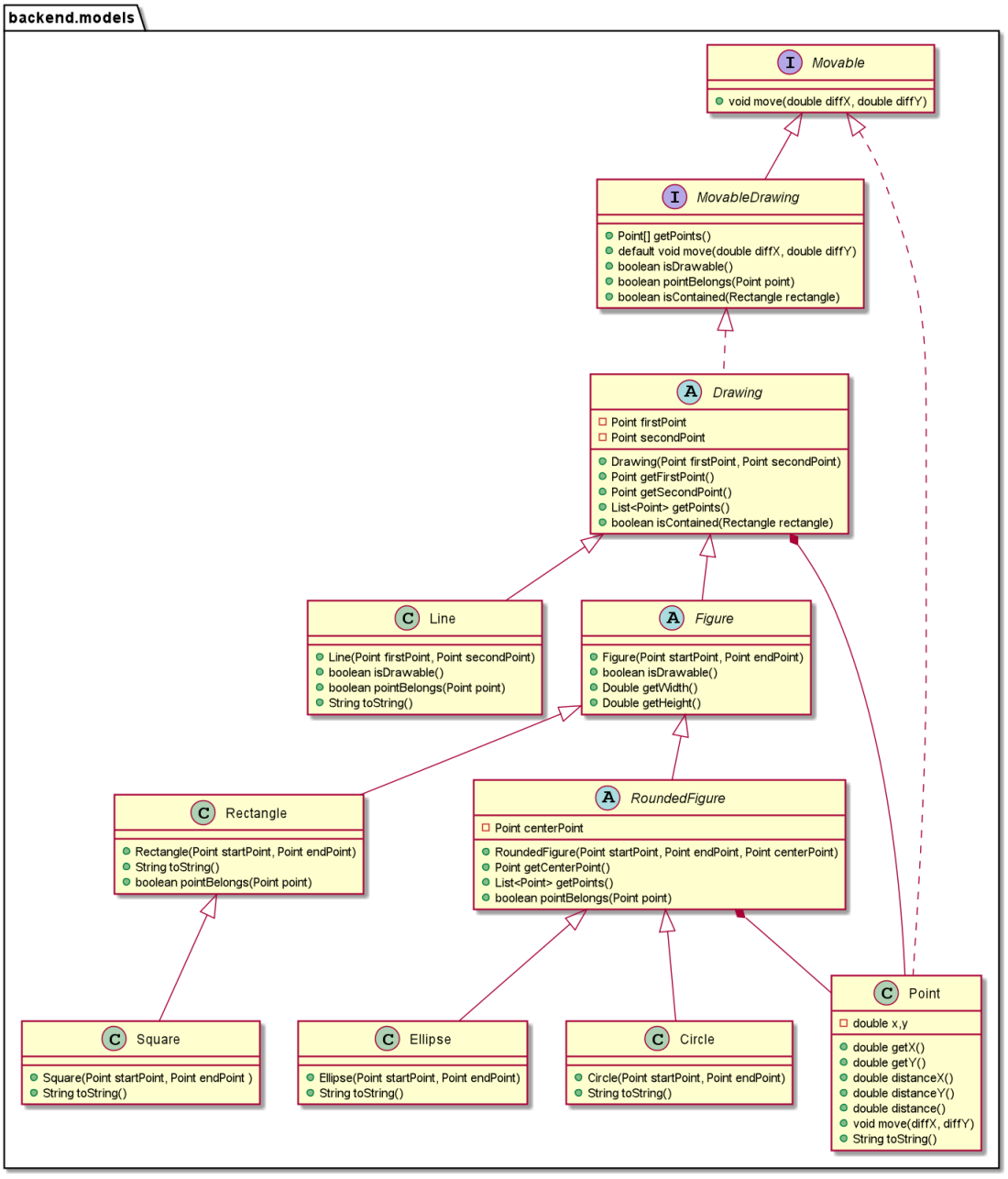


Figure 1: backend.models package

Interfaz Movable:

Esta interfaz modela el comportamiento de aquellas clases que tendrán la cualidad de poder moverse. Para eso, cuenta con un único método *move()*, el cual recibe por parámetro el diferencial en X y en Y que se desea mover al objeto correspondiente.

Clase concreta Point:

La clase Point modela un punto en el plano 2D con sus variables de instancia X e Y. Tiene los métodos necesarios para poder calcular la distancia entre dos puntos, para calcular la distancia entre las coordenadas X de dos puntos y también en Y. También permite modificar las variables de instancia mediante el método *move* que le agrega un diferencial a las coordenadas X e Y. La clase tiene los getters correspondientes a sus variables de instancia y el método *toString()*. Esta clase fue modificada en base a lo mencionado en la sección que hace referencia a los cambios hechos sobre el código provisto por la cátedra.

Interfaz MovableDrawing:

La interfaz MovableDrawing extiende de Movable. Marca el comportamiento que debe tener un dibujo para poder moverse. Definimos los dibujos a través de ciertos puntos que los identifican y nos permiten representarlos adecuadamente. De esto surge que mover un dibujo implica mover los puntos que lo representan. Por esta razón, esta interfaz cuenta con un método *getPoints()*, mediante el cual se espera obtener cada uno de los puntos que representan a un dibujo, y una implementación default del método *move()* que mueve cada uno de los puntos el diferencial deseado en las coordeandas X e Y.

Además de este comportamiento fundamental, toda instancia de una clase que implemente esta interfaz, sabrá si es dibujable, si un punto está contenido en ella, si está contenida en el rectángulo imaginario (utilizado para la selección múltiple) y también será capaz de devolver un arreglo con los puntos que la conforman.

Clase abstracta Drawing:

La clase abstracta Drawing implementa la interfaz MovableDrawing ya que en este caso todas los dibujos son capaces de moverse. Esta clase vincula los dos tipos de dibujos que podemos hacer en la aplicación: figuras y líneas. La clase se guarda el primer punto donde se hizo click para comenzar a dibujar y el último punto donde se soltó el mouse. Estos dos puntos permiten identificar a la figura y representarla en la interfaz gráfica. La clase agrega los getters de la variable de instancia e implementa los métodos *getPoints()* e *isContained()* de la interfaz MovableDrawing. Diremos que un dibujo está incluido en un rectángulo imaginario si su punto superior izquierdo y su punto inferior derecho están incluidos en él.

Clase concreta Line:

La clase Line extiende de Drawing. Modela el comportamiento de un dibujo de una línea. Implementa los métodos restantes que exige la interfaz MovableDrawing que implementa su padre, Drawing.

Clase concreta Figure:

La clase Figure extiende de Drawing. La clase modela el comportamiento de figuras geométricas. Agrega los métodos *getWidth()* y *getHeight()* e implementa el método *isDrawable()* exigido por la interfaz MovableDrawing. Esta clase fue modificada en base a lo mencionado en la sección que hace referencia a los cambios hechos sobre el código provisto por la cátedra.

Todas las figuras reciben en el constructor los dos puntos que ubica el usuario cuando busca representarlos en el canvas y se encargan de hacer los cálculos necesarios para almacenar los puntos superior izquierdo e inferior derecho que son los que nos permitirán representarlas.

Observación acerca de la distinción entre "figuras" y "líneas": decidimos representar estos objetos como clases separadas debido a las siguientes diferencias encontradas:

- A diferencia de las figuras, las líneas no cuentan con un 'ancho' y un 'alto', por lo que carecería de sentido que una línea cuente con los métodos *getWidth()* y *getHeight()*, que nunca serían utilizados.
- El uso de los métodos *getWidth()* y *getHeight()* para las figuras resulta de utilidad a la hora de dibujarlas, ya que los métodos que nos permiten dibujar figuras reciben el alto y el ancho de las mismas.
- Las líneas siempre pueden dibujarse, en cualquier dirección y sentido, mientras que las figuras deben cumplir ciertas restricciones para poder ser dibujadas. Esto útlimo se mencionó en la descripción de las funcionalidades de la aplicación.

Clase concreta Rectangle:

La clase Rectangle extiende de Figure. Modela el comportamiento de un dibujo de un rectángulo. Implementa los métodos restantes exigidos por la interfaz que implementa el padre de su padre, Drawing. Agrega el método *toString()*.

Clase concreta Square

La clase Square extiende de Rectangle. Modela el comportamiento de un dibujo de un cuadrado. Un cuadrado es un rectángulo cuyo alto es igual a su ancho. En el constructor, el punto superior izquierdo se almacena tal como es recibido y el inferior derecho se modifica según lo indicado en la consigna para que el alto y el ancho coincidan. Sobreescribe el método *toString()* de su padre, Rectangle.

Clase abstracta RoundedFigure

La clase abstracta RoundedFigure extiende de Figure. Encapsula el comportamiento que tienen las figuras redondas guardándose como variable de instancia el centro de la figura que es utilizado en reiteradas ocasiones para evaluar si un punto pertenece o no a la figura redonda. Decidimos almacenar esta variable de instancia teniendo en cuenta el costo de calcularlo y su frecuencia de uso. La clase implementa los métodos restantes exigidos por la interfaz del padre de su padre, Drawing y agrega el getter de la variable de instancia mencionada previamente.

Decidimos implementar esta clase para unificar el comportamiento de una elipse y un círculo ya que no consideramos adecuada la herencia. Esto se debe a las diferencias en los puntos recibidos para representarlas. Mientras el círculo recibe el punto del centro y otro punto que indica su radio, la elipse recibe el punto superior izquierdo y el punto inferior derecho, pero ambas necesitan del punto superior izquierdo para ser dibujadas en el frontend, por lo que los cálculos que deben realizarse difieren. La herencia implicaría una serie de cálculos extra que dificultarian la claridad del código y no tendría mucho sentido.

Clase concreta Circle

La clase Circle extiende de RoundedFigure. Modela el comportamiento de un dibujo de un círculo. Implementa el método *toString()*. Esta clase fue modificada en base a lo mencionado en la sección que hace referencia a los cambios hechos sobre el código provisto por la cátedra..

Clase concreta Ellipse:

La clase Ellipse extiende de RoundedFigure. Modela el comportamiento de un dibujo de una elipse. Implementa el método *toString()*.

Frontend

Carpeta Renders:

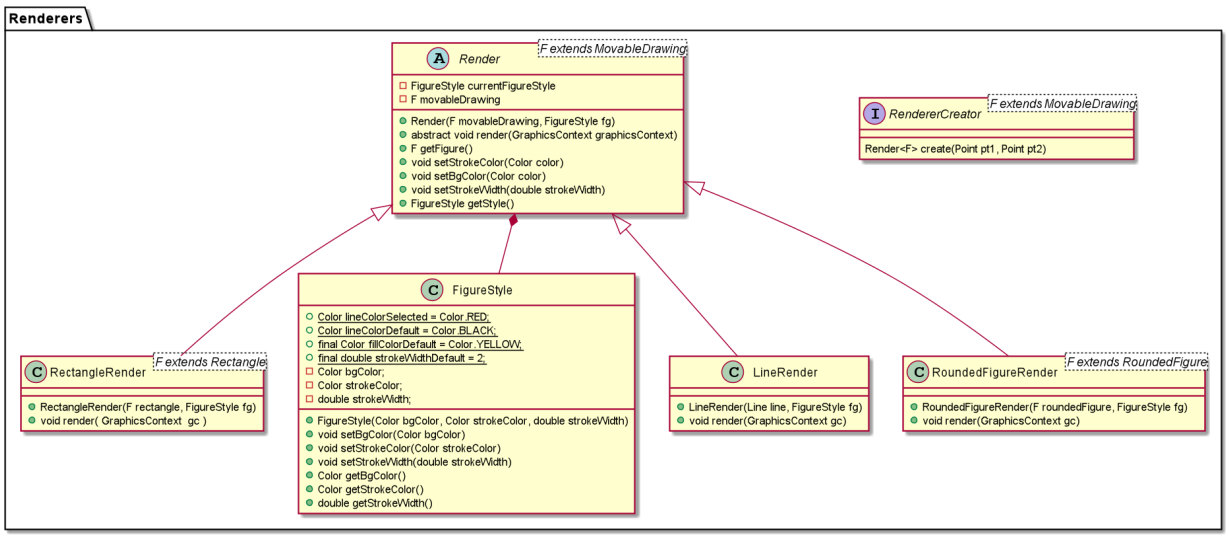


Figure 2: Renderers package

Interfaz RendererCreator:

La interfaz funcional RendererCreator permite crear un nuevo Render (o bien, dibujo) a través del método *create()* que recibe como parámetros los dos puntos que necesita un dibujo para su creación (Ver implementación de las clases concretas del backend descritas anteriormente).

Clase concreta FigureStyle:

Esta clase encapsula un conjunto de propiedades que definen el estilo de los dibujos. Es decir, el grosor y color de la línea de borde y el color de fondo.

Contiene, como variables de clase, los colores default que tienen los dibujos al inicio: el color de contorno cuando un dibujo es seleccionado y un grosor inicial por default. Tiene implementado todo lo necesario para cambiar el color del borde y del relleno y para cambiar el grosor del borde de un dibujo. Cuenta con los getters correspondientes a las variables de instancia.

Clase abstracta Render:

La clase abstracta Render encapsula el comportamiento que veremos de nuestros dibujos en pantalla. Un Render sabe que tipo de dibujo es (movableDrawing) y también conoce su estilo y lo puede modificar (currentFigureStyle).

La clase implementa métodos para acceder a las variables de instancia, para cambiar el fondo, el grueso y el color del contorno del dibujo y le exige a las clases hijas que implementen el método *render()* que permite visualizar en pantalla el dibujo.

Clase concreta RectangleRender:

La clase RectangleRender extiende de Render. Implementa el método *render()* para poder visualizar en pantalla los dibujos rectangulares.

Clase concreta LineRender:

La clase LineRender extiende de Render. Implementa el método *render()* para poder visualizar en pantalla el dibujo de una línea.

Clase concreta RoundedFigureRender:

La clase RoundedFigureRender extiende de Render. Implementa el método *render()* para poder visualizar en pantalla un dibujo de una figura circular.

Carpeta Buttons:

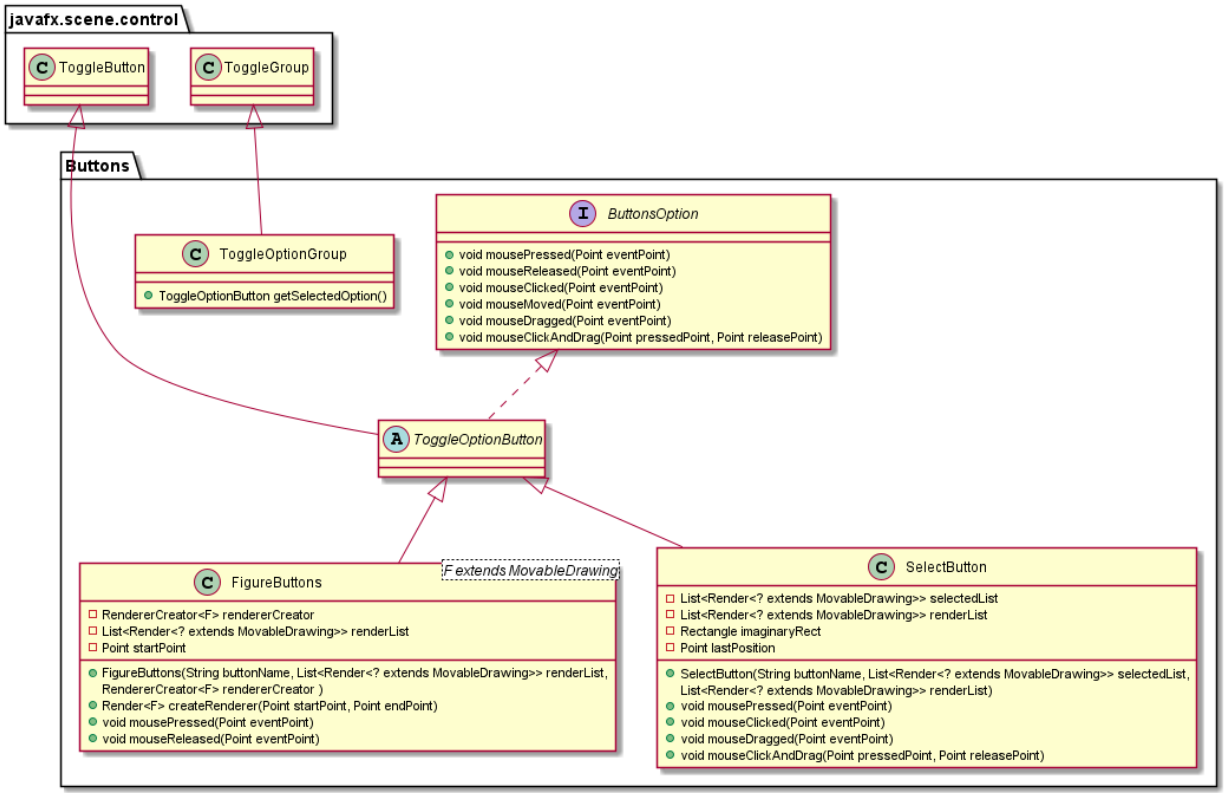


Figure 3: Buttons package

Notamos que el botón Seleccionar, como los botones de las figuras, tenían que llevar a cabo determinadas acciones al estar seleccionados en la barra izquierda. No podíamos utilizar los métodos preexistentes en los botones, ya que no queríamos que estos respondan ante un evento sobre ellos, sino sobre el canvas. Fue por eso que decidimos implementar una interfaz que permita darle este comportamiento a un botón. Con esto, cada botón llamará a los métodos cuando ocurra un determinado evento en el canvas.

Interfaz ButtonsOption:

Como se mencionó, la interfaz permite darle comportamiento a un botón. Llamaremos a estos métodos cuando ocurra un determinado evento en el canvas.

La interfaz tiene implementados todos sus métodos por default. Esto fue hecho con el fin de que cada botón pueda sobreescribir los métodos que le resulte útil para llevar a cabo su acción particular y evitar métodos en la clase concreta con cuerpo vacio.

Observación: La interfaz contiene métodos que quizás en nuestras clases concretas no se sobreescriben. Sin embargo, consideramos que es una forma adecuada de permitir futuras actualizaciones de los botones ya creados agregandoles funcionalidades o mismo, para darle comportamiento a nuevos botones mediante esta interfaz.

Clase abstracta ToggleOptionButton:

La clase ToggleOptionButton extiende de ToggleButton e implementa ButtonsOption. De esta forma, podemos englobar el comportamiento de los botones que utilizaremos en nuestro código, SelectedButton y FigureButton. Todas las clases hijas de esta clase implementaran ButtonsOption.

Clase concreta FigureButtons:

La clase FigureButtons extiende de ToggleOptionButton. Recibe en su método constructor una lista con los dibujos que se encuentran en pantalla y un RenderCreator que permite crear una figura específica mediante el método *create()*.

Si se presiona sobre algún botón de alguna figura, esta clase se guarda el punto donde se hizo click en la pantalla *(startPoint)* para luego, cuando se suelta el mouse *(endPoint)*, se crea la figura con el RenderCreator que sabe que figura crear con estos puntos y lo agrega a la lista de dibujos renderizados.

Clase concreta SelectButton:

La clase SelectButton extiende de ToggleOptionButton. Recibe en su método constructor la lista de dibujos que fueron seleccionados y una lista con los dibujos que se encuentran en pantalla.

Si se presiona el botón seleccionar, este es capaz de:

- Saber cuándo accionar la selección simple y cuando la selección múltiple.
- Saber cuándo dejar de seleccionar múltiples dibujos o uno solo.
- Mover los dibujos a lo largo y ancho de la ventana.

Para la selección múltiple, se crea un rectángulo imaginario. Todo dibujo cuyos puntos están contenidos dentro de ese rectángulo imaginario, serán obtenidos de la lista de dibujos que se encuentran en pantalla y se guardarán en la lista de dibujos seleccionados.

Observación: Las líneas se pueden seleccionar únicamente mediante la selección múltiple.

Clase concreta ToggleOptionGroup:

La clase ToggleOptionGroup extiende de ToggleGroup. En su constructor determinamos el comportamiento de los botones de forma que no exista la posibilidad de que no haya un botón presionado en ningun momento que se este ejecutando el programa. Es decir, el programa comienza con un botón por default seleccionado. Ademas contiene el metodo *getSelectedOption()* que nos devuelve el ToggleOptionButton que está presionado en ese momento. Notar que este método siempre devolverá algún boton pues, en todo momento, habrá alguno seleccionado por lo mencionado anteriormente.

Carpeta DrawingArea:

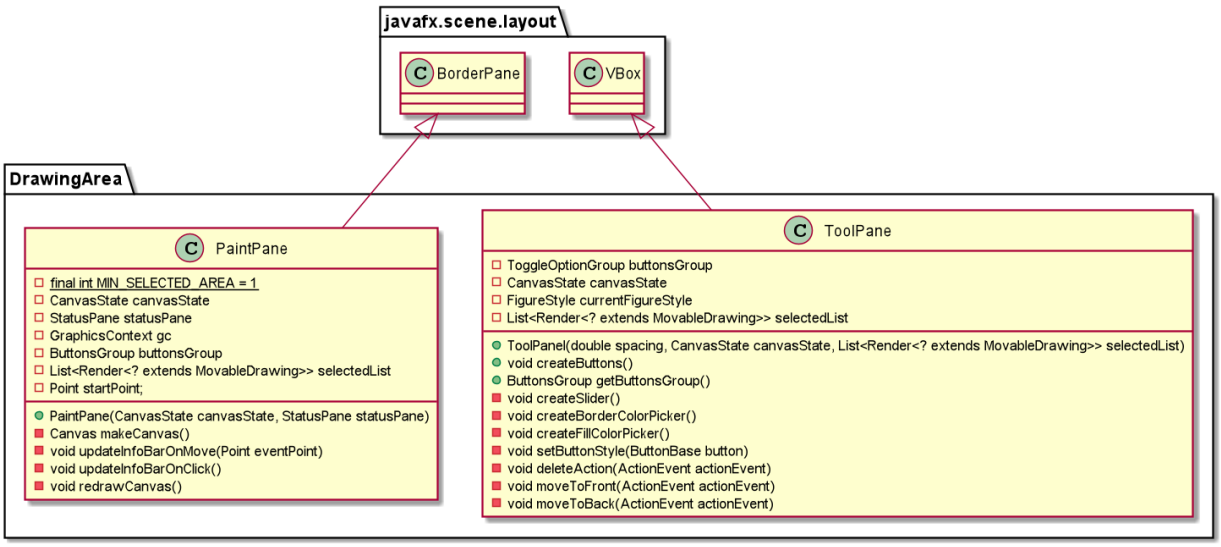


Figure 4: DrawingArea package

Clase concreta PaintPane:

La clase PaintPane extiende BorderPane. Esta clase se utiliza como nexo entre el ToolPanel y el Canvas como también de administrador del layout de los mismos en pantalla.

En el constructor se instancian el ToolPanel y el Canvas para luego ubicarlos en sus respectivas posiciones con *setLeft()* y *setRight()*. Además se configura el EventHandler de manera que siempre que se lance un evento en el ToolPanel, éste sea escuchado y se ejecute el evento *redrawCanvas()* que actualiza el canvas (lo que vemos en pantalla).

Esta clase cuenta con una variable de instancia *MIN_SELECTED_AREA* que indica el mínimo desplazamiento que reconocemos como una selección múltiple y no como un click. Este cambio fue necesario porque los métodos del canvas *setOnMouseReleased()* y *setOnMouseClicked()* se ejecutan siempre uno detrás del otro sin dejar lugar a hacer una distinción entre ambos movimientos. Esto último se explica en detalle en la sección problemas encontrados.

Por otro lado, tenemos los métodos *makeCanvas()*, *redrawCanvas()*, *updateInfoBarOnMove()* y *updateInfoBarOnClick()*. El método *makeCanvas()* se encarga de instanciar el canvas y administrar el llamado a eventos invocando a los métodos necesarios del botón seleccionado. El método *redrawCanvas()* es de los más importantes. Se encarga de setear los colores de fondo y de borde de las figuras que van a ser renderizadas en el canvas, teniendo en cuenta que si están seleccionadas el color del borde debe cambiar.

Finalmente los métodos *updateInfoBarOnMove()* y *updateInfoBarOnClick()* se encargan de realizar las acciones necesarias de actualización de la barra de información según el evento indicado.

Clase concreta ToolPanel:

La clase ToolPanel extiende de VBox. El ToolPanel representa el panel que control que tenemos a la izquierda de la aplicacion. La clase instancia todos los botones que forman parte de él, el slider para modificar el grosor del borde del dibujo y el color picker para hacer los cambios de colores en el relleno y el contorno de la figura. Cada uno lanza su respectivo ActionEvent según haya un cambio en los mismos, menos el slider que como no lanza un ActionEvent por si solo, lo accionamos manualmente para que el PaintPane

actualice el canvas cuando se cambia el borde de las figuras seleccionadas. Una vez creados e instanciados, se les aplica los estilos y todos los botones y el slider son agregados a la lista de Nodos hijos de el Toolpanel.

Carpeta App:

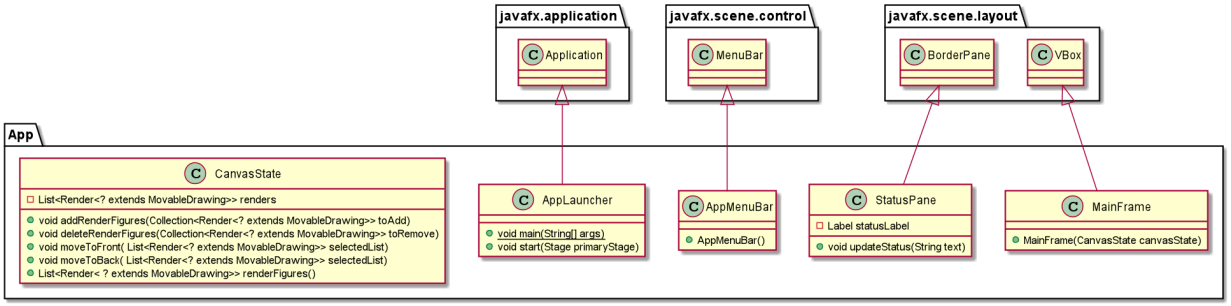


Figure 5: App package

Clase concreta CanvasState:

La clase CanvasState es la encargada de administrar los dibujos que van apareciendo y eliminandose de la pantalla. Además, cuenta con los métodos para traer al frente y al fondo uno o varios dibujos.

Para almacenar los dibujos, cuenta con una Lista. A medida que el usuario manipula los dibujos en el Canvas, los cambios se ven reflejados en esta lista para luego mostrarse en pantalla. Los dibujos se almacenan en el orden que serán mostrados, es decir, el primero en la lista podrá observarse "mas abajo" que los demás.

Métodos:

- *moveToBack():* Lleva al fondo de la pantalla uno o más dibujos.
- *moveToFront():* Lleva al frente de la pantalla uno o más dibujos.
- *addRenderFigures():* Agrega uno o más dibujos a la lista.
- *deleteRenderFigures():* Elimina uno o más dibujos de la lista.
- *renderFigures():* Retorna la lista de los dibujos almacenados hasta el momento.

Las clases AppLauncher, AppMenuBar, MainFrame y StatusPane no fueron modificadas.

Cambios sobre el código provisto por la cátedra

Tal como se mencionó en la consigna del trabajo práctico, existen ejemplos de malas prácticas que violan los principios del paradigma POO e incluso, cosas que el grupo considero que se podían modificar para lograr un mejor estilo del código. A saber:

- **Uso del operador binario instanceof**

  En varias líneas de la clase PaintPane notamos que se pregunta si una variable (local o de instancia) es de un tipo de dato en particular para ejecutar líneas de código en el caso que esto último ocurra. Para todos esos casos, el grupo decidió eliminar esa pregunta y reemplazarlo encapsulando el comportamiento en un objeto particular para que el mismo ejecute lo que deba ejecutar independientemente de la instancia que sea evitando así el uso del operador binario instanceof.

  Por ejemplo, esto se podía ver en las líneas de código del PaintPane que movían la figura:

```
    if(selectedFigure instanceof Rectangle) {
            Rectangle rectangle = (Rectangle) selectedFigure;
            rectangle.getTopLeft().x += diffX;
            rectangle.getBottomRight().x += diffX;
            rectangle.getTopLeft().y += diffY;
            rectangle.getBottomRight().y += diffY;
    } else if(selectedFigure instanceof Circle) {
            Circle circle = (Circle) selectedFigure;
            circle.getCenterPoint().x += diffX;
            circle.getCenterPoint().y += diffY;
    }
```

  Esto es una mala práctica según lo aprendido acerca del paradigma orientado a objetos. Por esta razón, decidimos encapsular este comportamiento en las clases Movable y MovableDrawing, que nos permiten mover cualquier dibujo sin consultar a qué clase pertenecen.

  Algo similar notamos en el método *redrawCanvas()*, también de PaintPane, donde antes de dibujar la figura se consultaba, haciendo uso de instanceof, a qué clase pertenecía. Esto fue mejorado haciendo uso de los Renders descritos anteriormente.

- **Variables de instancia públicos o sin modificador de visibilidad**

  Por un lado, en la clase Point, se podían acceder a las variables de instancia sin necesidad de un método getter pues estas últimas eran públicas. Esto viola el principio de Hiding pues, al igual que en los ADT's vistos en Programación Imperativa, las propiedades de la clase deberían estar ocultas y para acceder al estado de la instancia deberíamos utilizar un getter. Es por eso que el grupo decidió modificar la clase Point.

  Por otro lado, en la clase PaintPane y Circle se utilizaban variables de instancia con el modificador de visibilidad package private y de tipo protegido respectivamente. Decidimos cambiarlas a privado para mantener el ocultamiento de las propiedades de la clase (Hiding). Además, no tenía mucho sentido que fueran protegidas porque hasta ese momento, por cómo recibimos el código, estas clases no tenían ninguna subclase que utilizara estas variables de instancia declaradas.

- **El método constructor de la clase PaintPane**

  Notamos que todo el código de la clase PaintPane estaba en el método constructor. Como parte del código se podía dividir en otras funciones, el grupo decidió modularizar parte de lo que estaba en el constructor logrando de esa forma un código más legible.

- **Clase abstracta Figure que no reutiliza código y simplemente agrupa el comportamiento de una figura**

  Notamos que la clase Figure estaba vacía. Las clases hijas no reutilizaban código de su clase padre. Al ser todas figuras que se dibujaban sobre la misma pantalla, buscamos la mejor solución para que la clase Figure tenga un verdadero sentido de reutilización de código y que las clases hijas puedan aprovechar ese comportamiento.

- **Utilizar ifs para saber cuál es el botón seleccionado**

  En la clase PaintPane, en reiteradas ocasiones, podía observarse una línea de código similar a la siguiente:

```
1                 if((...)Button.isSelected())
```

  para ver si determinado botón estaba seleccionado. De seguir con esta línea de pensamiento, la cantidad de ifs hubiese aumentado considerablemente al agregar nuevos botones.

  Por esta razón, decidimos que los botones cuenten con una determinada acción que deben realizar. De esta forma, podemos obtener el botón seleccionado y ejecutar la acción correspondiente, independientemente de cuál sea el botón seleccionado. Sin embargo, crear una clase para cada botón, hubiese resultado en una gran cantidad de clases. Para solucionar este problema, concentramos el comportamiento de los botones que crean dibujos en la clase FigureButton, cuyo comportamiento varía únicamente según la figura se desea crear, por lo que recibimos este comportamiento a través de una expresión lambda.

  El grupo, en base a estas cosas y teniendo en cuenta las querys que había que implementar, llevó adelante el trabajo práctico implementando todas las funcionalidades descritas en la sección anterior y modificando estos últimos items.

- **Cambio en la clase CanvasState**

  En vez de utilizar un ArrayList como en la implementación original optamos por una LinkedList para borrar de manera más eficiente los elementos. Tambien se decidió que CanvasState forme parte del frontend dado que los elementos de la lista son los renders que forman parte del frontend.

Problemas encontrados

Debemos admitir que al principio el grupo dudó cómo encarar adecuadamente el proyecto. Estuvimos mucho tiempo intentando deducir que hacía cada línea de código de las diferentes clases intentando planear una estructura de clases que nos sea de ayuda a la hora de hacer todos las querys lo más sencillo posible.

Nos costó entender qué clases de JavaFx eran las más adecuadas para utilizar como por ejemplo: ToggleGroup, ToggleButton, RadioButton, etc. Tuvimos que investigar bastante.

Definir la estructura de clases de los botones fue un reto ya que dependiendo de qué botón esté presionado necesitábamos que pase algo distinto y al mismo tiempo teníamos el botón de selección que era completamente distinto a los de creación de figuras. Esto lo solucionamos a partir de permitile a los botones que tengan métodos que se usan en cada evento y que además, tengan los constructores de cada figura a dibujar para el caso de los FigureButtons. Por ende, cuando se clickea un botón y ocurre un nuevo evento sobre la pantalla, simplemente se llama a el método de ese evento del mismo botón y sin preguntar que tipo de botón devuelve la acción es la deseada.

Se nos dificultó buscar una manera adecuada de mantener siempre seleccionado un botón por default en el grupo de botones para no lanzar NullPointerException's si no había ningún botón seleccionado y consultabamos por el botón seleccionado en ese momento. Esta idea nos evitó líneas de código de los bloques try-catch y consultas por el botón que esté seleccionado en ese momento en algunas partes del código.

Relacionado con lo anterior, como estamos usando ToggleButtons y ToggleGroups para que no se pueda seleccionar 2 botones al mismo tiempo esto generó el inonveiente de no poder sobreescribir el método que devuelve qué botón del grupo que está seleccionado pues este es final. Esto generó que tengamos que hacer un casteo, aunque de manera segura, de Toggle a ToggleButton.

Una dificultad que surgió mientras trabajamos ocurrió el evento de *setOnClick()* que no estaba implementado como creíamos. Éste se acciona siempre que haya un press y luego un release sin importar si, en el medio hubo, un drag. Esto nos confundió bastante tiempo hasta que entendimos como funcionaba y pudimos recortar el funcionamiento del evento. En la documentación figura un método *setOnMouseDragReleased()* que creímos que podría identificar el comportamiento que buscabamos. Sin embargo, no fue así.

Para finalizar, el último problema resuelto para el correcto funcionamiento del programa fue escuchar los eventos lanzados por el ToolPanel ya que siempre que éste lanzaba un evento debemos redibujar la pantalla del canvas. Como este último no tiene acceso al método *redrawCanvas()*, decidimos que desde el PaintPane se escucharan todos los eventos lanzados por esta clase para ejecutar el método que queremos cuando ocurriera cierto evento. Sin embargo, surgio otro problema. El Slider que usamos para ajustar el grosor de los bordes no lanza un ActionEvent cuando se cambia su valor interno y por ende no se actualizaban los dibujos. Esto lo solucionamos lanzando un ActionEvent a mano cada vez que se cambia el valor interno del Slider.

Conclusión

Como grupo notamos que la combinación de JavaFX y Java permite crear y desplegar aplicaciones con un aspecto vanguardista. Es un mundo amplio para incursionar e investigar. Tuvimos la oportunidad de visualizar algunos ejemplos de aplicaciones que han sido desarrolladas con JavaFX y son alucinantes algunos de los resultados finales que se pueden obtener.

Creemos que fue super interesante y emocionante desarrollar nuestra primera aplicación con los conocimientos que fuimos aprendiendo a lo largo de este curso. Sobre todo creemos que fue una muy buena experiencia poder acercarnos a un proyecto un poco más realista y a gran escala, donde tuvimos que manejar una gran cantidad de clases y paquetes, algo que hasta el momento no habíamos hecho.