Trabajo Práctico Especial
Programación Orientada a Objetos
Julio 2020

Entrega: 14/07/20

Integrantes:

- Julián Francisco **Arce Doncella** 60509
- Roberto José **Catalán** 59174
- Gian Luca **Pecile** 59235

Informe - **PaintPOO**

Funcionalidades agregadas

Cuadrado, Elipse y Línea

Para la implementación de nuevas figuras como el cuadrado, la elipse y la línea, se decidió crear los modelos correspondientes dentro del paquete `model`, localizado en el paquete `backend`.
Con el objetivo de utilizar las buenas prácticas del paradigma de objetos estas figuras fueron extendidas de la clase abstracta `Figure`. De esta forma las clases nuevas están obligadas a implementar los métodos que les dan las funciones características de los modelos deseados.

```
public class Square extends Rectangle {

    public Square(Point topLeft, Point bottomRight) {
        super(topLeft, new Point(bottomRight.getX(), topLeft.getY() + bottomRight.getX()
- topLeft.getX()));
    }

    @Override
    public String toString() {
        return String.format("Cuadrado [ %s , %s ]", topLeft, bottomRight);
    }
}
```

El cuadrado fue extendido de la clase `Rectangle` ya que tiene el mismo comportamiento y es un caso particular de su clase padre.
Por otro lado, para la clase de `Ellipse` se decidió utilizar como clase padre del `Circle` y se realizaron los cambios correspondientes para que se aproveche la herencia. Cabe destacar que en la clase `Ellipse`, se utilizaron dos métodos nuevos para cálculos intermedios necesarios para el funcionamiento de la misma.
A diferencia del cuadrado y la elipse, la clase `Line` fue únicamente extendida de `Figure` debido a que no tiene comportamiento en común con otras figuras hijas pero posee funcionalidad en común con la clase padre.

Personalización de figuras: borde y relleno

La selección de figuras fue modificada de la versión original a fin de extender su funcionalidad.

```
private final Collection<Figure> selectedFigures = new LinkedList<>();
```

En la implementación original dada por la cátedra, `SelectedFigure` es un `Figure`, ahora `SelectedFigures` es una LinkedList de tipo `Figure` localizada en el paquete `frontend`, dentro de la clase `PaintPane`.
Para la implementación de borde y relleno se agregó una interfaz en el paquete `backend`, dentro del paquete `model` implementada por las figuras que permite establecer dichas propiedades. Esta interfaz permite el uso de un color de fondo, color de borde, y ancho de borde.
También se hizo lo necesario en el paquete `frontend` para visualizar los selectores de color y borde.

```
private final Slider strokeSlider = new Slider(1,50,0);
private final ColorPicker strokeColorPicker = new ColorPicker(Color.BLACK);
private final ColorPicker fillColorPicker = new ColorPicker(Color.YELLOW);
...
strokeColorPicker.setOnAction(event -> selectedFigures.forEach(figure ->
figure.setStrokeColor(strokeColorPicker.getValue())));

fillColorPicker.setOnAction(event -> {
  selectedFigures.forEach(figure -> figure.setFillColor(fillColorPicker.getValue()));
  redrawCanvas();
});

StrokeSliderHandler sliderHandler = new StrokeSliderHandler();

strokeSlider.setOnMouseClicked(sliderHandler);

strokeSlider.setOnMouseDragged(sliderHandler);
...
private class StrokeSliderHandler implements EventHandler<MouseEvent>{
  @Override
  public void handle(MouseEvent event) {
     selectedFigures.forEach(figure -> figure.setStrokeWidth(strokeSlider.getValue()));
     redrawCanvas();
  }
}
```

La clase privada `StrokeSliderHandler` fue creada a fin de no repetir código con expresiones lambda para las invocaciones del `strokeSlider`, el cual se acciona al ser presionado o deslizado y permite cambiar el borde de la figura seleccionada.

Borrado y selección múltiple

Para la implementación del borrado de figuras en primer lugar se hizo lo necesario en el paquete `frontend` a fin de visualizar el botón y garantizar su funcionamiento.

```
private final  ToggleButton deleteButton = new ToggleButton("Borrar");
...
```

```
deleteButton.setOnAction(event -> {
  canvasState.removeSelectedFigures(selectedFigures);
  selectedFigures.clear();
  deleteButton.setSelected(false);
  redrawCanvas();
});
```

Al accionar el botón se llama al método `removeSelectedFigures()` localizado en el paquete `backend`, en la clase `CanvasState` que se usa para visualizar las figuras presentes en el canvas. Luego se usa el método `clear()` para deseleccionar las figuras que se encuentran seleccionadas, después se deselecciona el botón para ser usado nuevamente y por último `redrawCanvas()` se encarga de eliminar la visualización de lo seleccionado.

```
public void removeSelectedFigures(Collection<Figure> selectedFigures) {
   selectedFigures.forEach(canvasFigures::remove);
}
```

En `removeSelectedFigures()` se elimina de `selectedFigures`, la estructura que contiene las figuras a ser mostradas en el canvas todas las apariciones de los elementos pertenecientes a `selectedFigures`, la colección de las figuras seleccionadas recibidas como parámetro.

La selección múltiple se implementa en el paquete `frontend`. El concepto principal de la funcionalidad agregada es el uso de una instancia de `Rectangle` invisible donde aquellas figuras que estén contenidas en su totalidad serán seleccionadas.
De esta forma, el primer paso es crear como una herramienta auxiliar un rectángulo, que se forma a partir del primer punto del “pressed” y con el punto donde se realiza el “released.” Luego, se debe preguntar a todas las figuras que se encuentran en `canvas.figures()`, un método que retorna un Iterable de tipo `Figure` con las figuras presentes en el canvas y se verifica si están contenidas dentro del rectángulo instanciado. Aquellas que se encuentren en su totalidad dentro del mismo, serán agregadas a `selectedFigures`.

Profundidad

Primero se implementó lo necesario en el paquete `frontend`, en específico `PaintPane` para poder visualizar los botones.

```
private final  ToggleButton bringForwardButton = new ToggleButton("Al fondo");
private final ToggleButton sendBackButton = new ToggleButton("Al frente");
...
bringForwardButton.setOnAction(event -> {
  canvasState.moveForward(selectedFigures);
  bringForwardButton.setSelected(false);
  redrawCanvas();
});

sendBackButton.setOnAction(event -> {
```

```
    canvasState.moveBackwards(selectedFigures);
    sendBackButton.setSelected(false);
    redrawCanvas();
});
```

Primero se llama al método `moveForward()` o `moveBackwards()` del paquete `backend`, luego se deselecciona el botón después de ser activado y por último se vuelve a dibujar el canvas para actualizar las figuras dibujadas.

```
public void moveForward(Collection<Figure> selectedFigures) {
    removeSelectedFigures(selectedFigures);
    LinkedList<Figure> aux = new LinkedList<>(selectedFigures);
    aux.descendingIterator().forEachRemaining(canvasFigures::addFirst);
}
public void moveBackwards(Collection<Figure> selectedFigures) {
    removeSelectedFigures(selectedFigures);
    selectedFigures.forEach(canvasFigures::addLast);
}
```

En ambos métodos se pasa como parámetro la colección de figuras seleccionadas y se itera para reordenar lo visualizado por `canvasFigures`.

Modificaciones realizadas

PaintPane.java

La clase `PaintPane` sufrió la mayor cantidad de cambios de las clases provistas por la cátedra dado que es la que conecta el back-end con el front-end. En primer lugar se cambiaron de public a private las variables declaradas.
Toda funcionalidad que agregada resultó en cambios a la misma y algunos de los mismos fueron mencionados en dicha sección. Los botones de las figuras fueron delegados a un Enum denominado `FigureButtons`.

```
RECTANGLE(new ToggleButton("Rectángulo")){
   @Override
   public Figure getFigure(Point startPoint, Point endPoint) {
       return new Rectangle(startPoint,endPoint);
   }
},
```

Toda figura posee en su constructor una instancia nueva de `ToggleButton` con el nombre apropiado de la figura. Cada tipo de enum sobreescribe el método abstracto `getFigure()` que retorna en su nombre una instance de `Figure` con dos parámetros de entrada de clase `Point` el cual es usado para reemplazar el uso de `instanceOf` junto con toda aparición de la misma (que viola los principios del paradigma) en el llamado de `setOnMouseDragged()`. El uso de streams es requerido a fin de recolectar las instancias de los botones, por este motivo se decidió optar por una List de `ToggleButton` en vez de un Array de los mismos.

```
List<ToggleButton> toolsList = new ArrayList<>();
...
toolsList.addAll(Arrays.stream(FigureButtons.values()).map(FigureButtons::getButton).col
lect(Collectors.toList()));
```

De esta manera se logra evitar el uso extenso de condiciones **`if()`**.

```
private void redrawCanvas() {
  gc.clearRect(0, 0, canvas.getWidth(), canvas.getHeight());
  for(Figure figure : canvasState.figures()) {
     if(selectedFigures.contains(figure)) {
        gc.setStroke(Color.RED);
     } else {
        gc.setStroke(figure.getStrokeColor());
     }
     gc.setLineWidth(figure.getStrokeWidth());
     gc.setFill(figure.getFillColor());
     figure.draw(gc);
  }
}
```

`RedrawCanvas()` fue modificado para no incluir el uso de `instanceOf` y a la vez fue reinterpretado su funcionamiento al paquete `backend` como método `draw()` presente en la clase `Figure`.

CanvasState.java

La clase `CanvasState` posee la menor cantidad de cambios entre las clases aportadas por la cátedra y la mayoría de los mismos provienen de las funcionalidades agregadas.

```
public class CanvasState {

    private final LinkedList<Figure> canvasFigures = new LinkedList<>();

    public void addFigure(Figure figure) {
        canvasFigures.add(figure);
    }

    public void removeSelectedFigures(Collection<Figure> selectedFigures) {
        selectedFigures.forEach(canvasFigures::remove);
    }

    public Iterable<Figure> figures() {
        return canvasFigures;
    }

    public void moveForward(Collection<Figure> selectedFigures) {
        removeSelectedFigures(selectedFigures);
        LinkedList<Figure> aux = new LinkedList<>(selectedFigures);
        aux.descendingIterator().forEachRemaining(canvasFigures::addFirst);
    }

    public void moveBackwards(Collection<Figure> selectedFigures) {
        removeSelectedFigures(selectedFigures);
        selectedFigures.forEach(canvasFigures::addLast);
    }
}
```

Donde es implementada una LinkedList de tipo `Figure` ya que permite el uso de métodos tales como `addFirst()` o `addLast()` para manipular el orden de inserción de los elementos a ser visualizados en el canvas.

Figure.java

La clase de `Figure` es una de las que más cambios sufrió, ya que al agregarse las funcionalidades deben seguirse los conceptos de orientación de objetos. Debe implementar los métodos con los comportamientos generales de una figura tales como: la capacidad de moverse, los colores, los tamaños de los bordes y cómo debe dibujarse.

Con el objetivo de agregar los comportamientos deseados a las figuras fue necesario agregar interfaces como `Movable`, `Drawable` y `Colorable`, las cuales se implementaron en `Figure`.

```
public interface Colorable {

   default void setColorProperties(Color strokeColor, Color fillColor, double
strokeWidth) {
        setStrokeColor(strokeColor);
        setFillColor(fillColor);
        setStrokeWidth(strokeWidth);
   }

   void setStrokeColor(Color strokeColor);

   void setFillColor(Color fillColor);

   void setStrokeWidth(double strokeWidth);

   Color getStrokeColor();

   Color getFillColor();

   double getStrokeWidth();
}
```

La interfaz `Colorable` se agregó para brindarle las propiedades de color de linea y superficie, como también el grosor de la linea. Para ello se utilizan los métodos que se encuentran en el contrato de la interfaz.

```
private Color strokeColor = Color.BLACK;
private Color fillColor = Color.BLACK;
private double strokeWidth = 1;
```

Se decidió que la clase `Figure` contenga tres variables de instancia privadas para guardar las propiedades mencionadas.

```
@FunctionalInterface
public interface Movable {
   void move(double moveInX, double moveInY);
}
```

La interfaz funcional `Movable`, cuenta con el método `move()` el cual se implementa en `Figure` y no en las clases hijas (toman la implementación del padre). Se utiliza esta interfaz para brindarle la funcionalidad de moverse a la figura.
Para no repetir código el método en Figure y sus hijas se desarrolló de la siguiente forma:

```
@Override
```

```
public void move(double moveInX, double moveInY){
   getPoints().forEach(p->p.move(moveInX,moveInY));
}
```

Donde `getPoints()` es un método de cada figura que retorna los puntos que definen a la figura.

```
public abstract boolean contains(Point p);
```

El método `contains()` permite determinar si un punto está dentro de la figura o no. Éste es un método abstracto, y cada figura lo sobreescribe.

```
public abstract boolean isInside(Rectangle container);
```

También se añade el método `isInside()`, el cual determina si la figura está dentro de un rectángulo, se utiliza para la selección múltiple.

```
public interface Drawable {
   void draw(GraphicsContext gc);
}
```

La interfaz `Drawable` contiene el método `draw()`, no es funcional para poder ser alterada si es necesario.

```
@Override
public abstract void draw(GraphicsContext gc);
```

La clase `Figure` implementa el método abstracto `draw()`, el cual recibiendo un `graphicsContext`, permite dibujar la figura. Es abstracto ya que la forma de dibujarse varía dependiendo de cada figura en particular.

```
private static int count = 0;
private final int ID = getNewID();
...
private int getNewID() { return count++; }
```

Por último, se decidió buscar una forma para poder identificar a cada figura sin importar de qué tipo es y dónde está ubicada. Para poder llevarlo a cabo se utilizó una variable de clase `count` que permite asignarle un ID único a cada instancia de `Figure`. La variable de instancia ID se utiliza para el `Equals()` y `HashCode()`, ya que una figura es igual a otra si y sólo si su ID es el mismo. Si bien en el `frontend` desarrollado no son utilizados, es necesario para poder utilizar un Map si el usuario lo desea y además brinda versatilidad al `backend`.

Point.java

La clase `Point` implementa la interfaz `Movable`, con lo cual un punto puede moverse por el plano.

```
@Override
public void move(double moveInX, double moveInY) {
   x += moveInX;
   y += moveInY;
}


public double distanceTo(Point p){
   return Math.sqrt(Math.pow(p.getX() - x, 2) + Math.pow(p.getY() - y, 2));
}

public double horizontalDistanceTo(Point p) {
   return Math.abs(p.getX() - x);
}

public double verticalDistanceTo(Point p) {
   return Math.abs(y - p.getY());
}
```

También se agregaron métodos para calcular la distancia entre puntos, permiten calcular la norma, la distancia horizontal, y la vertical. Además se agregaron `equals()` y `hashCode()`, donde dos puntos son iguales si tienen las mismas coordenadas.

Rectangle.java y Circle.java

En el caso de `Circle` se deja de extender a `Figure` y se extiende a `Ellipse`, con quien comparte propiedades en común.

Para la clase `Rectangle`, se sobreescriben los métodos abstractos de `Figure`. Dicha clase además, sirve como padre de `Square`.

Problemas encontrados

Separación Front-end y Back-end

El mayor problema fue decidir dónde se realizaría el dibujado de las figuras. Se decidió que las figuras tengan un método que reciba un `GraphicsContext`, y luego éstas se dibujen en el lugar correspondiente con sus características, utilizando un concepto análogo al `toString()` de un objeto.
Para el caso de la lista de `SelectedFigures`, se decidió tenerlo en el paquete `frontend` (en `paintPane`), debido a que es una colección de figuras que sólo le interesa al usuario y tenerla en el paquete `backend` obligaba a implementar métodos redundantes que incrementarían innecesariamente la longitud el programa, generando código repetido en `canvasState`. Además, teniendo acceso directo a esa colección, es más fácil para el usuario implementar funcionalidades nuevas.

Uso de JavaFX

Debido a la falta de experiencia en JavaFX, las principales dificultades se presentaron a la hora de configurar algunos de los objetos y el uso correcto de los eventos en los mismos.
Como por ejemplo al utilizar el Slider, éste presentaba comportamientos distintos si se lo desplaza o se clickea directamente en un valor, por lo cual hubo que contemplar estas dos opciones y programarlas.
A la hora de analizar los eventos que sufren los botones o el canvas, fue confuso decidir dónde y cómo agregar las funcionalidades con los eventos, ya que existen muchas superposiciones como con “clicked” y “released.” Creemos que logramos relacionar las funciones que deben cumplir los botones con las acciones que sufren de forma correcta.

Instanciar figuras en Botones

Uno de los problemas con los que nos encontramos fue hallar una forma práctica basada en las clases teóricas y prácticas para poder instanciar distintas Figuras, botones para las mismas y recorrerlas.
Se decidió optar por el uso de un Enum el cual contiene un parámetro de entrada-salida que instancia un botón específico para cada tipo. Luego para resolver el problema de instanciar las figuras, se delega a un método abstracto el cual se sobreescribe para cada tipo de Enum dónde se devuelve una instancia de cada figura hija. Por último se recorren los valores con un método denominado `fetchFigure()`, el cual devuelve la instancia de la figura la cual su botón se encuentra seleccionado. Dicho método está hecho para delegar un proceso previamente realizado en el `PaintPane` que posee ya muchas tareas para una clase, el método para poder ser ejecutado se debió hacer `static` ya que tiene comportamiento análogo al del método `values()`.

Selección Múltiple

En un principio se intentó que lo relacionado a la selección múltiple se maneje en setOnMouseDragged, pero esto producía problemas de funcionamiento ya que colisionaba con setOnMouseClicked. Por esta razón, se decidió implementar tanto la selección simple como la múltiple en setOnMouseClicked.
Su funcionamiento se basa en que clicked corresponde a la acción de presionar y soltar el click, por lo tanto, si se presiona y suelta en el mismo lugar se usa selección simple, y si el lugar dónde se hace el click y dónde se lo suelta son distintos, entonces se interpreta que el usuario quiere hacer una selección múltiple, por lo que se crea un rectángulo invisible, y las figuras que están dentro de ese rectángulo son añadidas a la lista de figuras seleccionadas.
Por último surgió un problema al realizar el movimiento de un grupo de figuras seleccionadas, tal como en el programa original, al soltar el botón fuera de las mismas se deseleccionan y dentro de las mismas se selecciona una única figura.

Preservar orden Profundidad

Al realizar la adición de la funcionalidad de profundidad junto con la selección múltiple se encontró que el orden de las figuras seleccionadas se perdía al presionar el botón para mover al frente las figuras, para solucionar esto se creó una LinkedList auxiliar, copia de la colección recibida como parámetro, debido que posee el método `descendingIterator()` que permite el uso de `forEachRemaining()` para iterar sobre los valores con el orden descendente a fin de preservar su orden original.