Fecha: 08/07/2023

Informe Final Programación Orientada a Objetos

Grupo 5:

Santiago Diaz Sieiro 63058

Magdalena Cullen 63065

Agustin Alonso 63316

Tomas Becerra 63732

Introducción

El presente trabajo tiene como objetivo presentar los resultados y análisis del “Trabajo Práctico Especial Programación Orientada a Objetos “. Este proyecto se desarrolló con el propósito de implementar una aplicación de dibujo de figuras, haciendo uso del lenguaje de programación Java, junto con una interfaz visual JavaFx.

Para el desarrollo del trabajo en cuestión, se contaba con una implementación inicial de la aplicación, en la cual se podían dibujar distintos tipos de figuras, además de contar con un método (botón) para seleccionar, mover y/o borrar estas mismas.

Durante la ejecución del trabajo se plantearon muchos desafíos, los cuales demandaron tiempo, esfuerzo pero sobre todo investigación y profundización en temas de estudios, los cuales eran completamente nuevos para los cuatro integrantes del grupo. Así mismo pudimos llevar a cabo la implementación de la aplicación siguiendo el paradigma de programación visto durante la cursada de la materia.

Desarrollo:

Es sabido que el paradigma de programación orientada a objetos (POO) trae muchas ventajas a la hora de desarrollar programas informáticos con una gama de posibilidades infinita. Dentro de los pilares fundamentales del paradigma se encuentran el polimorfismo, encapsulamiento, herencia y abstracción. Es nuestro trabajo como programadores adherirnos a estos principios para desarrollar programas con un código que no solo sea fácil de leer y entender, sino que sea mantenible y abierto a la implementación de nuevas funcionalidades.

La última parte fue extremadamente relevante para el progreso de este trabajo práctico, ya que fue necesario realizar adaptaciones en el código preexistente proporcionado por la cátedra, debido a que no cumplía con los lineamientos establecidos por el paradigma orientado a objetos. Además de eso, la implementación anterior del programa no permitía la inclusión de nuevas funcionalidades de manera adecuada y sencilla.

El primer desafío que enfrentamos al encarar el trabajo fue, como dicho anteriormente, la readaptación del código, en específico la clase *PaintPane*. Dicha clase manejaba su lógica casi exclusivamente con sentencias *if/else* y usando también la palabra reservada *instanceof* para la decisión de qué figura debería dibujar el programa. Ambas son claras violaciones del paradigma orientado a objetos, inclinándose más hacia el paradigma de la programación imperativa.

Para atacar esto lo primero que hicimos fue modificar la estructura de clases de las figuras, creando una clase abstracta padre llamada *FormatFigure* la cual es vital para el correcto funcionamiento del programa ya que contiene la estructura general de todas las figuras, incluyendo el formato que se mostrará en pantalla. *FormatFigure* además implementa una interfaz creada por nosotros llamada *FigureDrawer* sobre la cual profundizaremos más adelante.

Otro cambio estructural en el programa fue la creación de la clase *EspecifiedToggleButton* y el enum *ButtonType,* para la diferenciación de qué figura se debería dibujar dependiendo de qué botón estaba seleccionado.

Siguiendo con el cambio de estructura de las figuras, se modificó también la herencia de ellas, haciendo que la clase *Square* extienda de la clase *Rectangle* y del mismo modo *Circle* de *Ellipse*. Esto

se hizo ya que ambas figuras hijas son simplemente casos particulares de las figuras padres, por lo cual tiene sentido que hereden casi totalmente su comportamiento y parámetros.

Para tratar con la impresión de las figuras en pantalla se modificó totalmente la implementación del método *redrawCanvass()*, distribuyendo su lógica entre las clases de las figuras y la clase *FrontFigureDrawer.* Esta última clase perteneciente al front-end que implementa la interfaz *FigureDrawer* del back-end se encarga del dibujo de las figuras con los métodos *drawEllipse(), drawCircle(), drawSquare() y drawRectangle().* Estos métodos utilizan funciones provistas por la biblioteca de *JavaFX* como *fillOval(), strokeOval()* para el dibujo de las elipses y círculos y *fillRect()* y *strokeRect()* para los cuadrados y rectángulos.

El segundo desafío fue implementar el comportamiento de las capas, los cual consiste en incluir un *ChoiceBox* que permite al usuario elegir una capa antes de dibujar una figura y luego asociarla para poder elegir con un *CheckBox* que capa visualizar en la pantalla. Esto trajo varios conflictos, pues implementar los controles especiales de *CheckBox* y *ChoiceBox*, lo cual no usaba un *setOnAction()* para poder agregarle comportamientos sino usaba *selectedProperty().addListener()* porque había que manejar los cambios en el estado de selección de los controles. Creamos una clase nueva que extendía *HBox* para así representar visualmente los *CheckBoxes*, como todos compartían el comportamiento de *selectedProperty()* entonces hicimos un for para asignarle las propiedades específicas que pedían en la segunda funcionalidad del trabajo. Acá vale aclarar que en trabajo pedía que en el *back-end* se implemente para infinitas capas y en el *front-end* solo para 3, por lo tanto creamos un *SortedMap* y contenía como claves los layers y valores un *List* con las figuras que pertenecen a la capa.

Dentro de los incisos con los que había que cumplir, el 1 y el 4 fueron los más fáciles de implementar en nuestro código, para los colores y grosores del borde de las figuras fue simplemente crear una clase llamada *Format*, la cual almacena en su interior el color del borde y relleno, además del grosor deseado de la figura. Esta clase fue implementada en las figuras usando composición, ya que, como fue mencionado anteriormente, la clase abstracta *FormatFigure* almacena el formato como un campo privado de tipo *Format.* Como último comentario sobre el punto 1, la funcionalidad de copiar formato fue sencillo, ya que fue simplemente agregar el botón que corresponde en la barra lateral y mediante el método *setOnAction()* le adherimos el comportamiento de copiar el formato de la figura seleccionada en una variable de instancia de *PaintPane,* la cual sería consecuentemente pegada en la próxima figura que se seleccione sobre el lienzo.

Siguiendo con el inciso 4, no hubo problemas con agregar los nuevos botones y cajas de texto necesarias ya que al haberlo hecho varias veces para las implementaciones anteriores entendíamos bien como hacerlo. La lógica que permite su funcionamiento es simple, las figuras contienen un campo que almacena sus *tags* en un *HashSet* y se muestran en pantalla si y sólo si el botón “todos” de la barra de tags está activado o alguno de sus *tags* coincide con el *tag* que se le pasa a la caja de texto en la barra inferior, el cual se almacena en *PaintPane* como una variable llamada *activeTag.*

La implementación del comportamiento de "undo" y "redo" resultó ser el desafío más importante. Al analizar los requisitos, se llegó a la conclusión de que la forma más eficiente de abordar este problema era utilizando dos *Deques*: uno para almacenar las acciones que se deshacen ("undo") y otro para almacenar las acciones que se pueden rehacer ("redo"). Estas estructuras se implementaron en la clase *ActionManager*. Sin embargo, surgieron algunos problemas al decidir qué información debía guardarse en los *Deques*.

Para resolver este inconveniente, se creó la clase *LastAction*, donde se almacenaba la figura antes de la acción y la figura después de la acción. Además, se utilizó una interfaz funcional para

implementar los métodos correspondientes para revertir las acciones. Adicionalmente, se creó un enum llamado *ActionType* que contenía las posibles acciones que se podían realizar en la aplicación.

Con esta estructura y enfoque, se logró implementar el comportamiento de "undo" y "redo", permitiendo al usuario deshacer y rehacer las acciones realizadas en la aplicación.

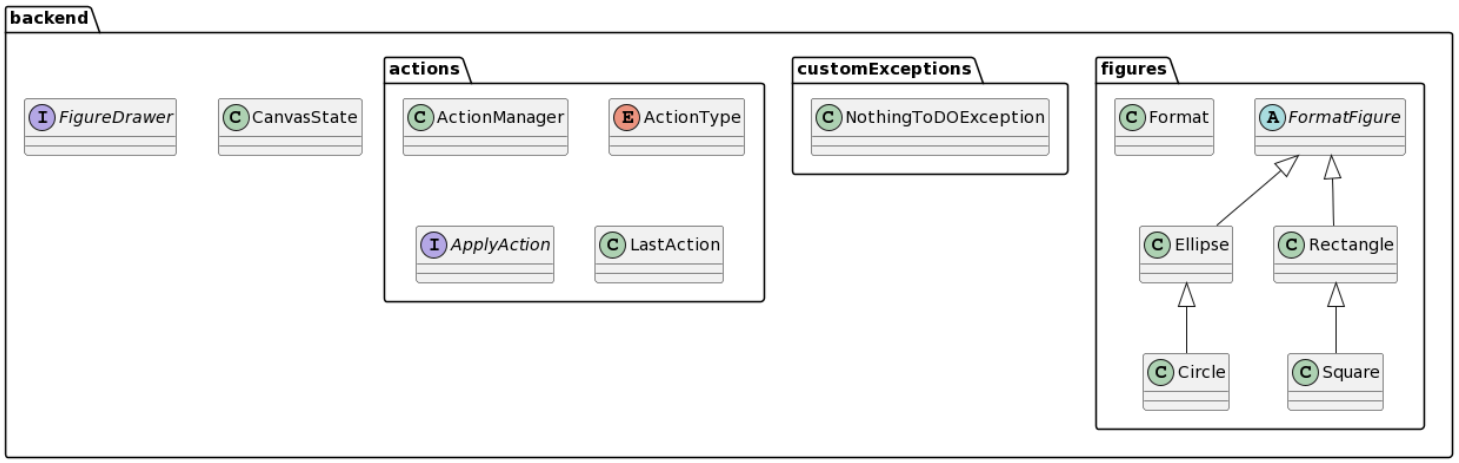

Figura 1: UML demostrando las clases, interfaz, and enum del *front-end*.

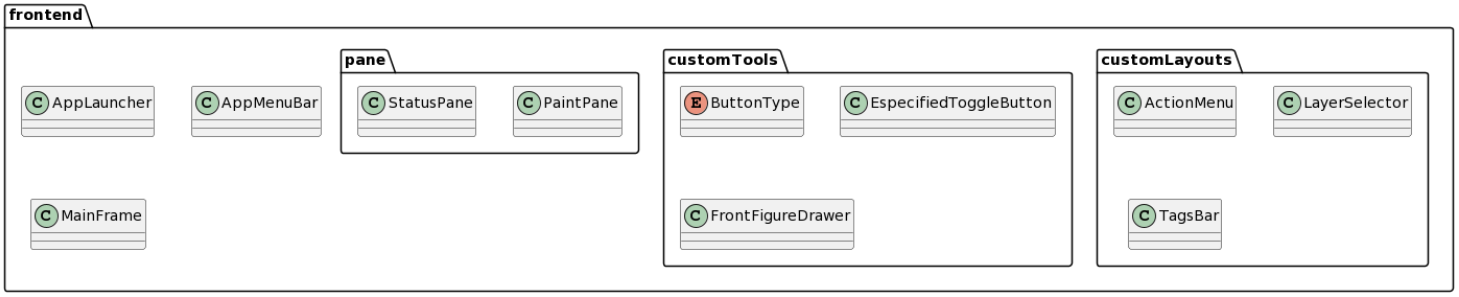

Figura 2: UML demostrando las clases, interfaz, and enum del *back-end.*

Conclusión:

En resumen, este trabajo nos ha brindado una mayor comprensión y conocimiento sobre el paradigma de Programación Orientada a Objetos y sus beneficios. La utilización de la interfaz JavaFX ha facilitado el desarrollo del proyecto al permitirnos reutilizar clases y métodos según lo propuesto en el enfoque. Sin embargo, debemos destacar que adquirir conocimientos en esta área ha presentado desafíos significativos debido a nuestra falta de experiencia previa con esta interfaz.

Afortunadamente, como equipo, hemos superado estos desafíos mediante un trabajo arduo y logramos desarrollar exitosamente la aplicación de dibujo, implementando cuatro funcionalidades adicionales y alcanzando los objetivos establecidos. Este proceso nos ha permitido crecer tanto en términos de conocimiento técnico como en habilidades de trabajo en equipo.