PRÁCTICA 1 GDDMD

* Durante el desarrollo de esta aplicación, hemos observado que el funcionamiento de un juego o aplicación software tiene su centro neurálgico en clase Controller; apoyado por los patrones ModelView-Controller, Observer y Command. Por tanto, hemos decidido explicar el funcionamiento de nuestro código siguiendo la estructura de nuestra clase Controlador.
* Lo primero que observamos en la clase Controlador, es el manejo de eventos, mediante dos métodos: actionPerformed (Para eventos de botones) y mouseClicked (Para manejo de clicks de ratón en la ventana).
* La acción principal y básica de la aplicación, sería el movimiento de piezas. Esto implica los eventos de ratón, junto a distintos eventos de botones. Al hacer click sobre la ventana, mouseClicked recibe un evento con información de la posición del ratón en pantalla. A partir de esta información, creamos un MovCommand (Que hereda de Command), cuyo constructor se encarga de llamar al método movePiece de BoardView, comprobando qué pieza ha sido pulsada (locatePiece) y si el movimiento es posible o no, devolviendo un array con la posición en la lista de posiciones de la pieza actual y la pieza blanca. El comando se encarga de, si el movimiento no es null, llamar a notifyObservers del controlador, actualizando tanto la vista como el modelo de datos. El modelo de datos es el que se encarga de comprobar si un movimiento lleva a la victoria o no. Todos estos comandos son almacenados en una pila en Controlador, permitiendo posteriormente el funcionamiento de los botones deshacer, desordenar… Junto con otra pila que almacena los movimientos deshechos, si deseamos rehacerlos uno a uno. De esta forma, estamos implementando los requisitos de movimiento de piezas, desordenar y resolver automáticamente, junto con el mensaje de victoria.
* El siguiente requisito indicado en el enunciado es el de cargar y guardar partida. Para ello, desde una nueva clase Configuracion con métodos estáticos, decidimos almacenar la información en formato Json, empleando la librería gson. Para hacer más fácil el guardado y acceso a la información, creamos una clase de apoyo: PartidaLD, que almacena el modelo y la pila de movimientos necesarios para resolver el puzzle. Así, el almacenamiento es una clase directa, y la lectura se realiza parseando a esta clase, reiniciando el modelo y la view con los parámetros del modelo de la partida guardada y volcando la pila de MovCommand de nuevo sobre la pila actual de Controlador. A estos métodos de la clase Configuracion se accedería desde el manejador de eventos de Controlador.
* Para el siguiente requisito, que abarca la utilización de un fichero de configuración, empleamos un fichero en formato xml que incluye un dtd, al tener una estructura más definida y regida por este último. Desde la misma clase Configuracion, realizaríamos la lectura y parse del fichero xml a una nueva clase Info, que almacena el path de una imagen predeterminada, un número de filas y columnas inicial y el tamaño predeterminado de imagen. De esta forma, al inicio de la aplicación, tratamos de leer este fichero. Si la lectura es posible, creamos la vista y el modelo en base a sus elementos; si la lectura se realiza con errores o el fichero no está disponible, la configuración mostrada sería similar a la configuración básica que implementaba el código base de la práctica.
* Los últimos requisitos planteados en el enunciado, se relacionan entre sí. Ya sea que el permitir la carga de otros formatos está supeditado a la carga y generación en sí de un puzzle a partir de una imagen, como el hecho de pedir los parámetros de esta imagen para así poder generar el puzzle de la manera deseada. Desde un evento de botón en Controlador, creamos un nuevo LoadCommand (Que hereda de Commad), cuyo método redo obtiene una imagen desde el método showFileSelector de PuzzleGUI. Este método se encarga de crear una ventana de navegación y selección, con un filtro de extensiones de archivo deseadas (Nosotros soportamos JPG, GIF, PNG y BMP). Si se selecciona una imagen válida, se llama al método updateBoard de PuzzleGUI. En este, la view es eliminada de la lista de observers, creándose un nuevo BoardView mediante un constructor que recibe además del resto de parámetros un fichero de imagen. Este constructor llamará a los métodos de: resizeImage, donde se creará una variable BufferedImage con las dimensiones deseadas; y al método splitImage, que calcula las dimensiones deseadas de cada imagen individual y almacena las imágenes en una ruta asignada dependiendo de su id, guardando el path en un array de string. Tras obtener el array de BufferedImage, el constructor generará una PieceView por cada posición, obteniendo su imagen del array generado previamente. Una vez generado el nuevo BoardView, se añade como observador en Controlador y acaba la ejecución del método updateBoard en PuzzleGUI. En controlador, se elimina de la lista de observadores el modelo actual y se genera uno nuevo, a partir de las variables almacenadas en BoardView, tras lo cual se añade de nuevo como observador. Al ser un puzzle nuevo, las pilas de comandos son vaciadas y se desordena de nuevo el puzzle.