Tecnología de la Programación

Curso 2016-2017

Práctica 5

Interfaz gráfico de usuario para los juegos de tablero

Antes de comenzar...

¡Importante!

- Antes de empezar la práctica 5, descarga la nueva versión del código base.
- Para evitar problemas, haz primero una copia de seguridad de tu práctica 4.
- A continuación descomprime el zip de la práctica 5 **sobre un directorio nuevo y copia después `pom.xml` y el directorio `src` sobre tu proyecto de la práctica 4.**
- Este proceso **sobrescribe** `pom.xml` y algunos archivos del código base de la práctica anterior. También añade otros nuevos.

Antes de comenzar...

- Recuerda que no está permitido modificar ningún fichero del paquete `es.ucm.fdi.tp.base`.
- Tampoco se pueden utilizar herramientas de generación automática de interfaces gráficos de usuario (como *NetBeans*).

- Esta práctica tiene dos partes:
  1. Implementar el patrón MVC, inicialmente con una vista básica en modo consola.
  2. Implementar una vista gráfica de forma que se reutilice el código entre los distintos juegos.

Descripción general de la práctica

- **Fecha de entrega: 3 de mayo de 2017 a las 9:00h.**
- **Objetivos:**
  - **GUI en Java utilizando Swing.**
  - **Patrón de diseño Modelo-Vista-Controlador.**
  - **Diseño Orientado a Objetos.**
- En esta práctica debes desarrollar e integrar varias **vistas gráficas** para los juegos *Tic-Tac-Toe* y *Lobo y Ovejas* (nos referiremos a ellas como vistas de `ventana`).
- **Debes utilizar el patrón de diseño MVC**. Además, debes utilizar un diseño que permita desarrollar **una sola vista que después puedas adaptar a cada juego concreto** utilizando **herencia** o **composición**.
- En las clases de la asignatura veremos más reglas generales de diseño que puedes utilizar.

Estructura del código

- De la práctica 4:
  - **es.ucm.fdi.tp.base**, **.ttt** y **.was**: clases base para implementar juegos y jugadores, y reglas de los juegos. No se deben modificar (solo para corregir errores de `was`).
  - **es.ucm.fdi.tp.launcher**: clases de inicio del programa.
- **es.ucm.fdi.tp.mvc**: **paquete nuevo** que debe contener las clases que implementan el **modelo** del patrón MVC. En el código base se incluyen:
  - Los interfaces que deben implementar los componentes MVC.
  - La estructura de la clase `GameTable`, **motor de juego** del modelo en el patrón MVC.
- **es.ucm.fdi.tp.view**: **paquete nuevo** que deberá contener todas las clases relacionadas con las vistas y controladores MVC.

- **Además: es.ucm.fdi.tp.extra**: ejemplos de uso de algunos componentes: `JTable`, `JColorChooser` y la forma de dibujar un tablero en un `JComponent`.

Clase `Main` para la Práctica 5

- **Antes de empezar: Copia el fichero `es.ucm.fdi.tp.launcher.Main.java`** en `MainPr4.java` y renombra la clase de este fichero a `MainPr4`.
- Modifica para esta práctica el fichero `Main.java`.
- En la práctica 5 el programa recibe **cuatro argumentos** de la línea de comandos:

`game mode player1 player2`

- **`mode`** es un argumento nuevo que puede tomar uno de los siguientes valores:
  - **`console`** para iniciar el programa en modo consola.
  - **`gui`** para iniciar el programa en modo GUI Swing.
- Los jugadores pueden tomar los valores `random`, `smart` o `manual` (este último cambia respecto a la P4 para no confundirlo con el modo de ejecución).
- **Utiliza las indicaciones del enunciado** de la práctica para completar el código de la clase `Main`.

Patrón MVC en la práctica 5: el modelo

- El **modelo** está formado por las clases de los paquetes **`mvc`**, **`base`** (excepto jugadores) y los paquetes de los juegos: **`ttt`** y **`was`**.
- La clase principal del modelo es **`GameTable`**.
  - Contiene el **motor de juego** del modelo en MVC.
  - Se proporciona la estructura básica de la clase.
  - **Debes completar el código** de esta clase para que mantenga el **estado de una partida** y ejecute las **operaciones** correspondientes.
  - Cuando se modifique el estado interno de la partida debe **notificar a los observadores registrados.**
- La comunicación entre el modelo y la vista se produce mediante notificaciones de objetos **`GameEvent`**.
  - Las vistas deben implementar el interfaz **`GameObserver`**.
  - Solo contiene un método **`notifyEvent`** que proporciona en el argumento `GameEvent` toda la información necesaria.

Patrón MVC: vista y controlador en modo consola

- Los componentes vista y controlador son específicos de cada modo.
- Todos deben estar en el paquete **`es.ucm.fdi.tp.view`.**
- **`ConsoleController`** es la clase que implementa el controlador de consola.
  - El método `run` debe hacer básicamente lo mismo que el método `playGame` de la P4, **pero no debe mostrar nada en la pantalla:** la interacción con el usuario la debe hacer la vista.
  - **Utiliza el esquema** proporcionado en el enunciado para esta clase.
- **`ConsoleView`** implementa la vista de consola.
  - La vista de consola debe añadirse a sí misma como observadora del modelo. Su objetivo fundamental consiste en mostrar al usuario los cambios producidos durante el juego.
  - Debe implementar el interfaz `GameObserver`.
  - **Utiliza el esquema** proporcionado en el enunciado para esta clase. Puedes encontrar la signatura de la constructora en la descripción del enunciado para la clase `Main`.

El modo de ventana

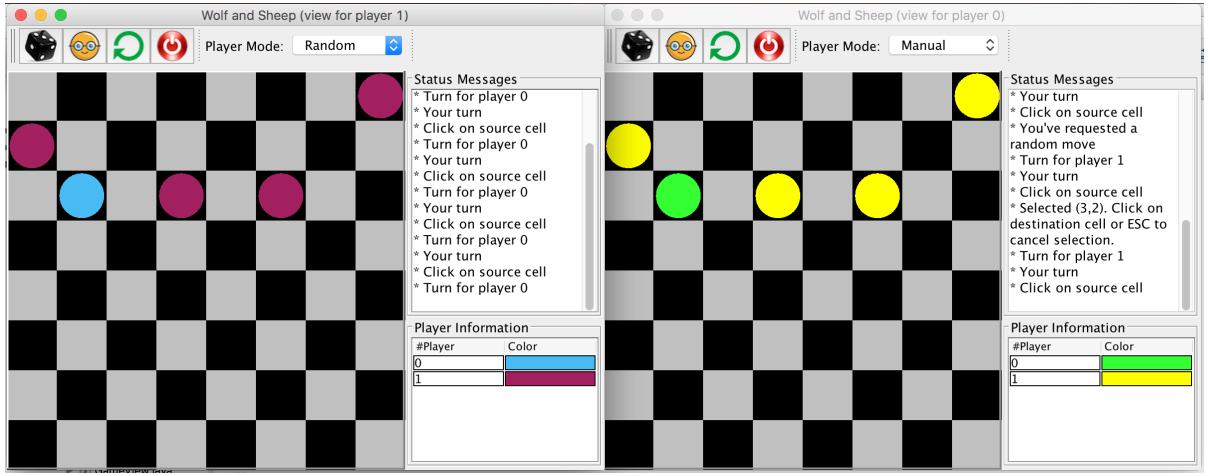
- Las clases del modo de ventana deben estar en el paquete **codees.ucm.fdi.tp.view.**
- El modo de ventana debe iniciarse con todos los jugadores de tipo manual: en este caso **los argumentos de jugadores pueden omitirse de la línea de comandos.**
- En este modo **cada jugador tiene su propia ventana** y cada ventana solo la puede utilizar un jugador.
  - Al inicio del programa se crean tantas vistas de ventana como jugadores.
  - Para crear cada vista de ventana se debe utilizar una llamada a **`SwingUtilities.invokeAndWait(...)`** (similar a `invokeLater`, pero espera a que termine de ejecutarse el argumento).

Patrón MVC: vista y controlador en modo ventana

- El **controlador** recibe peticiones de la vista de ventana para realizar distintas acciones sobre el modelo. Básicamente las siguientes:
  - **`public void makeMove(A action)`** realiza un movimiento. En la vista se obtiene el movimiento (de forma `manual`, `random` o `smart`).
  - **`public void stopGame()`** termina la partida actual.
  - **`public void startGame()`** inicia una nueva partida.
  - además de los métodos para añadir/eliminar observadores al modelo.
- En algunos casos el controlador solo debe pasar la llamada solicitada por la vista al modelo.
- La vista es específica de cada juego, pero se debe intentar reutilizar todo el código que sea posible mediante **herencia** y **composición**.
  - Además hace más fácil crear nuevos juegos...
- Sigue las indicaciones generales indicadas en el enunciado para la implementación de la vista en modo ventana.

Patrón MVC: vista en modo ventana

- El aspecto de las ventanas debe ser parecido al siguiente:



- Sigue las recomendaciones del enunciado para implementar el funcionamiento de las vistas de ventana.
- En particular, solo se puede jugar en la ventana del jugador al que le toca mover. En la otra ventana no se pueden hacer movimientos: solo se puede cambiar el modo de juego, el color de las fichas o salir.
- A continuación se muestran algunas ideas de implementación de la vista. Son **sugerencias** que te pueden ayudar a realizar la práctica.

Vista en modo ventana: Creación de la vista

- El modo **`gui`** debe iniciar siempre con dos jugadores manuales.
- Pero la vista de ventana debe tener disponibles dos jugadores para los movimientos **`random`** y **`smart`**. Se pueden pasar en la constructora.
- También puedes pasar a la ventana de la vista el componente de la vista que depende del juego y que se crea aparte (**`GameView`**).
- Puedes hacer una constructora para la vista parecida a la siguiente:

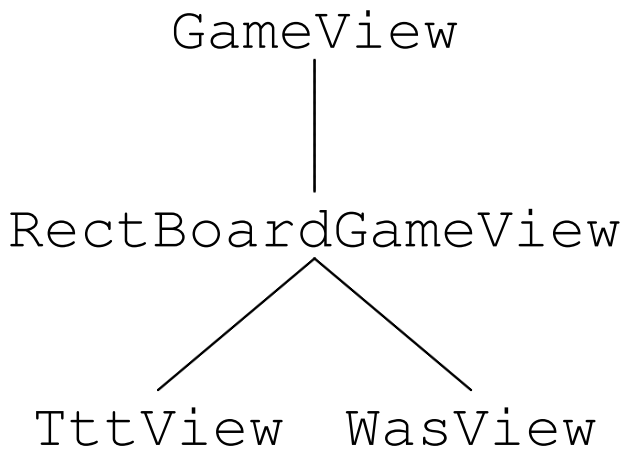
```
public class GameWindow<S extends GameState<S, A>,
                        A extends GameAction<S, A>>
  extends JFrame implements GameObserver<S, A> { // (*)
   ...
   public GameWindow(int playerId,
     GamePlayer randPlayer,
     GamePlayer smartPlayer,
     GameView<S, A> gameView,
     GameController<S, A> gameCtrl) { ...
```

- Recuerda que se debe crear una ventana por cada jugador (la constructora recibe el jugador en `playerId`).

(*) (Otros profesores proponen una implementación alternativa)

Vista en modo ventana: Estructura de clases

- **`GameWindow`** es la clase que implementa la ventana de un jugador. Puedes colocar en esta ventana la barra de botones que aparece en la parte superior (que puede ser de tipo **`JToolBar`**).
- La ventana recibe en la constructora un objeto de tipo **`GameView`**.
- Esta clase es abstracta y se debe implementar una subclase para cada juego. Se recomienda la siguiente estructura de herencia de clases:



Vista en modo ventana: Estructura de clases

- **`GameView<S, A>`** es una clase abstracta que extiende **`JComponent`** y contiene los siguientes métodos abstractos:
  - **`public abstract void enable()`** permite al usuario jugar.
  - **`public abstract void disable()`** impide jugar.
  - **`public abstract void update(S state)`** actualiza la vista.
  - Es necesario poder comunicar `GameWindow` con `GameView`. Por ejemplo puedes pasar una referencia del controlador a GameView con: **`public abstract void setController(GameController<S, A> gameCtrl)`**
- Estos métodos serán llamados desde `GameWindow` para actualizar la vista **cuando se reciban notificaciones del modelo** (`GameWindow` es el observador del modelo).
- **`RectBoardGameView`** extiende `GameView` y contiene la representación visual del tablero.
  - Implementa los métodos anteriores.
  - Incluye otros métodos abstractos para que los implementen las subclases.

Vista en modo ventana: Estructura de clases

- **`RectBoardGameView`** puede incluir los siguientes métodos abstractos:
  - **`protected abstract int getNumCols()`** devuelve el número de filas del tablero.
  - **`protected abstract int getNumRows()`** devuelve el número de columnas.
  - **`protected abstract Integer getPosition(int row, int col)`** devuelve el valor de una posición del tablero en el estado actual del juego.
  - Un método que se llame cuando se pulse sobre una casilla del tablero. Si se utiliza un `MouseListener` puede ser el siguiente:
    `protected abstract void mouseClicked(int row, int col, int clickCount, int mouseButton)`
    pero puede ser más sencillo si se utiliza un tablero de `JButton` y `ActionListener` (ver siguiente transparencia).
  - **`protected abstract void keyTyped(int keyCode)`**
- **`TttView`** y **`WasView`** extienden `RectBoardGameView` y deben implementar los métodos anteriores para que ejecuten las acciones sobre el controlador.

Vista en modo ventana: El tablero

- El tablero de juego puede representarse de varias formas.
- Como una **matriz de botones `JButton`.**
  - Se puede representar mediante un `JPanel` con un `GridLayout` con tantas filas y columnas como el tamaño del tablero.
  - Internamente hay que mantener un *array* bidimensional con las referencias a los botones para actualizar el tablero en pantalla.
  - También se debe mantener información en cada botón: puedes hacer una clase `JCasillaTablero` que extienda `JButton` con atributos para las coordenadas de la casilla a la que corresponde.
- Como un **componente en el que se dibujan las fichas.**
  - Permite más control sobre el funcionamiento del tablero, pero se programa a más bajo nivel.
  - En el código de base se incluye un ejemplo de programación: `es.ucm.fdi.tp.extra.jboard`.

Vista en modo ventana: Notificaciones del modelo

- **GameWindow** (que extiende `JFrame`) es el observador del modelo.
- Debe recibir las notificaciones del modelo y actualizar la vista.
- El código que modifica los componentes visuales debe ejecutarse con **SwingUtilities.invokeLater:**

```
public void notifyEvent(...) {
  SwingUtilities.invokeLater(new Runnable() {
    @Override
    public void run() {
      handleEvent(e); // actualiza componentes visuales.
    }
  });
}
```

- Así se actualizan los componentes visuales de forma ordenada en la hebra de Swing.
- De esta forma no se bloquea el bucle de gestión de eventos de Swing y el código ya está preparado para utilizar hebras (en la práctica 6).

Vista en modo ventana: Otros elementos visuales

- El área de mensajes informativos es un componente **`JTextArea`**.
- La **tabla de jugadores** es un **`JTable`**.
- El **selector de colores** utiliza un **`JColorChooser`**.
- En el código de base se puede encontrar un ejemplo de los dos últimos en **`es.ucm.fdi.tp.extra.jcolor.`**
- Para utilizar una imagen en un botón puedes cargar un icono (guardado en un directorio del *classpath*) de la siguiente forma:

```
miBoton = new JButton();
miBoton.setToolTipText("Pulsa y verás");
miBoton.setIcon(new ImageIcon(Utils.loadImage("dice.png")));
```