Práctica 5: Interfaz Gráfica

Fecha de entrega: 24 de abril de 2017, a las 09:00

Objetivo: Componentes visuales usando Swing, el patrón Modelo-Vista-Controlador

1. Descripción de la Práctica

En esta práctica desarrollarás interfaces de consola y gráficas para los juegos de la práctica anterior, usando el patrón de diseño modelo-vista-controlador (MVC).

Para ello, deberás completar las siguientes tareas:

1. Desarrollar un motor de juego (el modelo) que pueda gestionar un juego a partir de un `GameState` inicial. El motor debe permitir a sus usuarios empezar y finalizar la partida, aplicar acciones al estado actual, y solicitar este estado. Además, el motor enviará notificaciones a los observadores que en él se hayan registrado. Las notificaciones informarán de los eventos que sucenden durante el juego: cambios en el tablero, comienzo de juego, etcétera.

2. Desarrollar una interfaz de consola usando MVC. Esto implicará desarrollar el observador correspondiente, y refactorizando algo del código de la P4 para escribir un controlador de consola.

3. Desarrollar una interfaz gráfica que permita jugar al *tres en raya* y al *lobo y ovejas*. La interfaz debe estar implementada usando Swing, y la implementación debe tener un controlador que permita gestionar el juego en este modo.

4. Crear una versión modificada del `Main` de la P4 que permita a los jugadores elegir entre la interfaz de consola y la interfaz gráfica.

En las próximas secciones se proporcionan más detalles sobre cada una de las anteriores tareas.

Envío de la práctica Debes entregar un único zip (y no .rar, ni .7z, ni .tgz: un **.zip**) por grupo vía campus virtual, conteniendo exclusivamente los siguientes archivos y directorios: pom.xml, src/ (y todo lo que hay debajo), y un fichero llamado alumnos.txt. En alumnos.txt debes incluir el nombre de integrantes del grupo, así como cualquier comentario que quieras que tu profesor tenga en cuenta durante la corrección de la práctica.

2. El Modelo

En esta parte implementarás las partes requeridas para completar el modelo. En este práctica completarás el modelo existente con un motor de juego que gestiona un juego y envía notificaciones a los observadores registrados cuando los eventos correspondientes suceden durante la partida.

Todas las clases de este apartado deben estar situadas en un nuevo paquete, `es.ucm.fdi.tp.mvc`. Parte del código de este paquete lo publicarán los profesores en Campus Virtual.

2.1. GameEvent

Esta clase (proporcionada por los profesores) se usa para notificar cambios en el juego a sus observadores. Cuando el motor de juego quiere enviarles notificaciones, crea una instancia de `GameEvent` con la información correspondiente, y se la envía a todos los observadores registrados. Un `GameEvent` contiene su tipo, la acción que lo ocasionó (en caso de que haya alguna), el estado resultante (si lo hay), el error producido (si se produjo), y una descripción textual del evento. El tipo será uno de los siguientes:

- `Start`: el juego ha comenzado
- `Change`: el estado del juego ha cambiado (por ejemplo, tras una acción)
- `Error`: se ha producido un error (por ejemplo, al intentar aplicar una acción no válida)
- `Stop`: el juego ha finalizado
- `Info`: ninguno de los anteriores

Aunque un `GameEvent` siempre debe incluir un tipo válido, es posible dejar a null los atributos que no tengan sentido para cada caso. Así, en un evento de tipo `Start`, no tendría sentido indicar error alguno, y por tanto se pasaría como `null` al constructor.

2.2. Observadores y observables

Estas dos interfaces (proporcionadas por los profesores) permitirán a tus componentes de juego estar débilmente acoplados, y facilitarán mucho el desarrollar pruebas unitarias para ellos.

Si una clase debe responder a `GameEvents`, debería implementar `GameObserver`, que sólo contiene un métodos:

```
public void notifyEvent(GameEvent<S,A> e);
```

Los `GameObservers` deben registrarse con un `GameObservable`, que mantendrá una lista interna de observadores y proporcionará métodos para que éstos se puedan suscribir y des-suscribir:

```
public void addObserver(GameObserver<S,A> o);
public void removeObserver(GameObserver<S,A> o);
```

Es muy recomendable escribir un método privado en cualquier clase que implemente `GameObservable` para enviar un `GameEvent` a toda la lista de suscriptores; por ejemplo, en `GameTable`, necesitaréis hacer esto mismo desde al menos 3 puntos del código.

2.3. GameTable

Esta clase (parcialmente proporcionado por los profesores) implementará un motor de juego que gestionará el juego partiendo de un `GameState` inicial. Permitirá iniciar la partida, pararla, ejecutar acciones, y en todos estos casos, notificar a los observadores registrados del resultado.

A continuación se explica el comportamiento esperado de cada método:

- `GameTable` (constructor): recibe un `GameState` inicial y lo almacena en un atributo. Inicializa el resto de la instancia.
- `start`: lanza la partida, o la reinicia si ya había sido lanzada anteriormente. Notifica a todos los observadores del hecho. Para lanzar el juego, basta con copiar el estado inicial (pasado en el constructor) a otro atributo que contiene el estado actual.
- `stop`: para el juego, y notifica a los observadores de tal hecho. Si el juego ya había sido parado, notifica a los observadores con un evento que contenga el `GameError` correspondiente, y lanza una excepción con el mismo objeto `GameError`. Parar una partida sólo requiere modificar un atributo booleano apropiado.
- `execute`: ejecuta la acción dada sobre el estado actual, y encaso de éxito, usa el resultado como nuevo estado actual, notificando un evento de tipo `Change` a todos los observadores. Si el juego estaba parado, no iniciado, o se produce un error ejecutando la acción, no modifica el estado actual y envía un evento de tipo error con el `GameError` correspondiente a todos los suscritos, y lanza una excepción con el mismo objeto `GameError`.
- `getState`: devuelve el estado actual.
- `addObserver` / `removeObserver`: permiten añadir y borrar observadores de la lista de observadores.

Importante: NO añadas o modifiques los métodos de esta clase, a no ser que cuentes con autorización explícita de tus profesores.

3. Modo consola

En este apartado implementarás las clases que permiten jugar en modo consola. Todas las clases de este apartado deben estar en un nuevo paquete `es.ucm.fdi.tp.view`. Debes crear las siguientes dos clases:

```
public class ConsoleController<S extends GameState<S,A>, A extends
    GameAction<S,A>>
   implements Runnable {
  // add fields  here
```

```
  public ConsoleController(List<GamePlayer> players, GameTable<S,A>
      game) {
    // add code here
  }

  public void run() {
    // add code here
  }

  // add more private methods if needed
  // ...
}

public class ConsoleView<S extends GameState<S,A>, A extends
    GameAction<S,A>>
    implements GameObserver<S,A>{
  //
  public ConsoleView(Observable<S,A> gameTable) {
    // add code here
  }

  // add missing methods
  // ...
}
```

El `ConsoleController` recibe, en el constructor, un `GameTable` y una lista de jugadores, del mismo tipo que los de la P4 El juego se inicia llamando a `run`, que debe ser bastante similar al `playGame` de la P4. En un juego con `ConsoleController`, el método `requestAction` de los jugadores se llama desde dentro del bucle de juego contenido en `run`. El controlador no debe mostrar nada en la pantalla (exceptuando los mensajes mostrados por `ConsolePlayer`): mostrar mensajes es tarea de la vista. La clase `ConsoleView` recibe un `GameObservable` como entrada, y se registra como observador. Cada vez que recibe una notificación, muestra un mensaje apropiado. Por ejempl: comienzo de juego, a quién le toca ahora, fin de juego, etcétera.

4. Modo Swing

En este apartado, debes implementar las clases que permiten jugar a al *Tres-en-Raya* y al *Lobo y Ovejas* en modo gráfico (pero debes hacerlo de forma que sea fácil añadir otros juegos; piensa en el examen...) Todas las clases de este apartado deben estar en el paquete `es.ucm.fdi.tp.view`. En los próximos laboratorios tus profesores te proporcionarán más detalles sobre el diseño a seguir.

Cuando lances el programa en este modo, sólo podrá haber jugadores de tipo `manual`, y por tanto no es necesario especificar esta información desde la línea de comando. Recuerda que puedes consultar el número de jugadores esperado usando el método `getPlayersCount` de `GameState`. Ten en cuenta, en cuanto a la generación de jugadas, que:

- Sólo debe ser posible usar el tablero gráfico para hacer jugadas en el turno del jugador correspondiente a ese tablero. Los detalles sobre cómo se hacen las jugadas dependerán del juego activo: así, en el `ttt`, bastará indicar una casilla vacía; mientras que jugando al `was`, tras elegir una ficha propia, el jugador debe elegir una

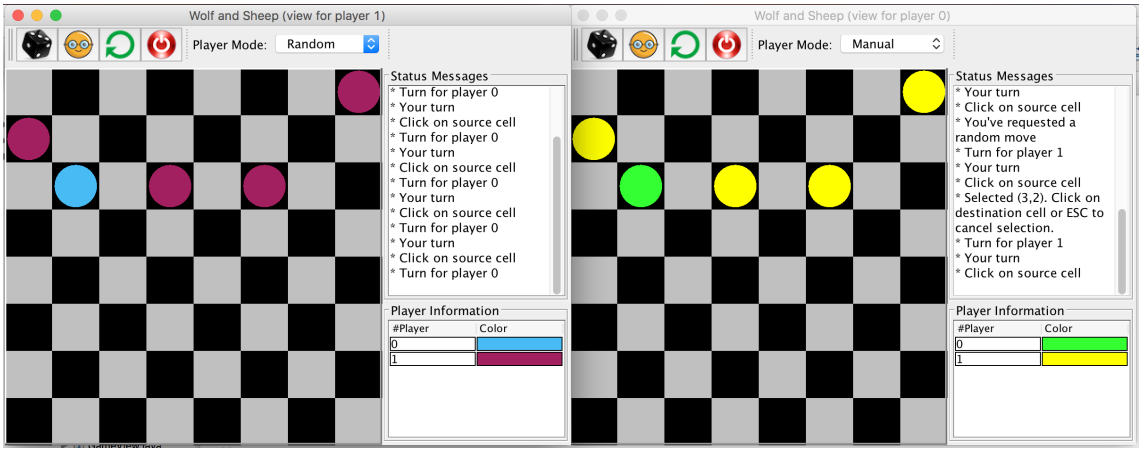

Figura 1: La interfaz gráfica

casilla vacía que además constituya una jugada válida (por ejemplo, las ovejas no pueden retroceder). Debe ser posible cancelar una jugada que está a medias; en otras palabras, debe ser posible seleccionar una oveja, y cancelar la selección para poder seleccionar a continuación otra distinta. Se recomienda usar bordes de celda coloreados para indicar el estado de la selección, y también para facilitar la selección de destinos válidos para la pieza actual.

- Debe ser posible habilitar la generación de jugadas aleatorias e inteligentes usando el combo-box de la barra de herramientas superior: haciendo click en los botones, se realizará una única jugada; y cambiando la selección en el combobox se usará el jugador seleccionado para todas las jugadas futuras (o hasta que se modifique la selección). Este tipo de jugadas deben ser completamente silenciosas, y no deben mostrar ningún mensaje que no se habría mostrado si la jugada la hubiese realizado un humano interaccionando con el tablero.

Además, desde el punto de vista de la intefaz en su conjunto, debes tener en cuenta que:

- Sólo la ventana del jugador al que le toca mover debe estar activa. Otras ventanas no deben permitir interacción alguna (excepto cambiar el modo del jugador, lo cual es útil para poder volver a pasar al modo manual).

- La barra de botones incluye un botón de “salir”. Presionarlo debe mostrar un diálogo de confirmación; y en caso de confirmarse la acción, debe cerrarse toda la aplicación (usando, por ejemplo, `System.exit(0)`).

- El botón de “reiniciar” debe reiniciar la partida. Debe ser posible usarlo en cualquier momento, y no sólo una vez finalizada la partida.

- El área de textos con mensajes informativos debe mostrar, entre otros, mensajes informando de que te toca mover, o (si no te toca), a quién le toc a ahora; y mensajes destinados a facilitar la creación de jugadas, del tipo “elige celda origen” y “elige celda destino”.

- La vista debe incluir una tabla que permita elegir el color con el que se muestran las fichas de los distintos jugadores. Los colores iniciales se generarán al azar usando,

por ejemplo, el iterador que devuelve `Utils.colorsGenerator()`. Haciendo click en la i-ésima fila debe mostrar un diálogo que permita modificar (aunque sólo en esta ventana) el color usado para las fichas de ese jugador.

- El título de la ventana debe incluir el nombre del juego (en esta práctica, `GameState` dispone de un método `getDescription` para obtener esta información) y el número del jugador.

5. Main

En este apartado modificarás el `Main.java` de tu P4 para que tenga en cuenta los cambios que introduce la P5. Empieza por copiar first tu `Main.java` actual a `MainPr4.java` (cambia también el nombre de la clase), sólo para no perder funcionalidad si necesitas consultarla en el futuro. La sintaxis de la línea de comandos se modifica en la P5: ahora hay un argumento más para permitir elegir el modo de juego. La nueva sintaxis es:

Main game mode player_1 player_2 ...

donde:

- game: ttt (para *Tres-en-Raya*) ó was (para *Wolf and Sheep*)
- mode: gui (para usar Swing) ó console (para usar la interfaz de consola)
- player_i: manual (para un jugador manual player), random (para uno aleatorio) ó smart (para uno inteligente).

AVISO: el significado de `console` varía con respecto al usado en la P4. Anteriormente, representaba un jugador por consola; y ahora para indicar lo mismo usaremos `manual` (en modo consola). Ahora, `console` sólo se refiere al "modo consola". Recomendamos seguir el siguiente esqueleto para implementar la nueva clase `Main.java`:

```
public class Main {
  private static GameTable<?, ?> createGame(String gType) {
    // create  a game with a GameState depending on the value of gType
  }

  private static <S extends GameState<S, A>, A extends GameAction<S,
      A>>
       void startConsoleMode(
          String gType, GameTable<S, A> game, String playerModes[]) {
    // create  the lis  of players  as in  assignemnt 4
    // ...

    new ConsoleView<S,A>(game);
    new ConsoleControler<S,A>(players,game).run();

  }

  private static <S extends GameState<S, A>, A extends GameAction<S,
      A>>
       void startGUIMode(
          String gType, GameTable<S, A> game, String playerModes[]) {
    // add code here
  }
```

```
  private static void usage() {
    // print  usage of  main
  }

  public static void main(String[] args) {
    if (args.length < 2) {
      usage();
      System.exit(1);
    }

    GameTable<?, ?> game = createGame(args[0]);

    if (game == null) {
      System.err.println("Invalid game");
      usage();
      System.exit(1);
    }

    String[] otherArgs = Arrays.copyOfRange(args, 2, args.length);

    switch (args[1]) {
      case "console":
        startConsoleMode(game,otherArgs);
        break;
      case "gui":
        startGUIMode(args[0],game,otherArgs);
        break;
      default:
        System.err.println("Invalid view mode: "+args[1]);
        usage();
        System.exit(1);
    }
  }
}
```

Es importante notar que el método `playGame` proporcionado en la P4 se debe mover al método `run` del nuevo `ConsoleController`.