(Versión 25-03-2015)

Anexo práctica 4: detalles de implementación

4.1. Adaptación del programa de consola al patrón MVC

Argumentos por línea de comandos

La aplicación permite indicar si se desea jugar mediante el interfaz de consola (`-u console`) o el interfaz de ventana (`-u window`).

Comandos y Jugadores

Las clases relativas a comandos y jugadores sólo son relativas a la modalidad del juego por consola. No tienen efecto en la aplicaicón gráfica.

Observadores y Observables

- Las clases de comando sólo se tendrán en cuenta para la modalidad de `consola`.
- Es habitual que los **modelos** dispongan de los métodos necesarios para el manejo de observadores.; Para ello se hace que implementen el interfaz `Observable`.

```
public interface Observable {
void addObservador(Observador o);
void removeObservador(Observador o);
}
```

  En nuestro caso el **modelo** es `Partida`, por lo que

```
public class Partida implements Observable {
  // Nuevos atributos para gestionar observadores
  private ArrayList obs ;
  public Partida(ReglasJuego reglas) {
    obs = new ArrayList<Observador>();
    reset(reglas);
  }

  void addObservador(Observador o){
    if (!obs.contains(o)) {
      obs.add(o);
    }
    void removeObservador(Observador o){
      obs.remove(o)
    }
  }

}
```

- Las vistas `VistaConsola,VistaGUI` implementan la interfaz `Observador` que contiene los métodos que se invocarán cuando la partida (que es el **modelo** de nuestra aplicación[2]) modifique su estado interno.

[2]La partida es el punto de entrada del modelo, aunque en el modelo hay más clases.

```
public interface  Observador {
// gestión de partida

void onReset(TableroSoloLectura tab, Ficha turno);
void onPartidaTerminada(TableroSoloLectura tablero, Ficha ganador);
void onCambioJuego(TableroSoloLectura tab,  Ficha turno);

//Gestión de movimientos
void onUndoNotPossible();
void onUndo(TableroSoloLectura tablero, Ficha turno, boolean hayMas);

void onMovimientoEnd(TableroSoloLectura tablero,
                                  Ficha jugador, Ficha turno);
void onMovimientoIncorrecto(MovimientoInvalido movimientoException);

}
```

Descripción de los métodos anteriores:

- `void onReset(TableroInmutable tab, Ficha turno)`:
  Método invocado por la clase Partida que permite notificar a sus observadores(las vistas) que se ha reiniciado la partida. Proporciona información del estado inicial del tablero y el turno (que será una ficha blanca o negra).
- `onPartidaTerminada(TableroInmutable tabFin, Ficha ganador)`:
  La partida notifica a los observadores que ha terminado la partida llamando a este método. Además proporciona al observador una vista del tablero de sólo lectura y el ganador.
- `void onMovimientoEnd(TableroInmutable tab, Ficha jugador, Ficha turno)`:
  La partida notifica a los observadores que se ha terminado de realizar un movimiento. Se proporciona además una vista del tablero de sólo lectura, el jugador que ha jugado, y el turno del siguiente jugador.
- `void onUndo(TableroSoloLectura tablero, Ficha turno, boolean hayMas)`:
  La partida notifica a los observadores que se ha deshecho un movimiento. Además, proporciona el estado final del tablero, el turno del siguiente jugador y si hay más movimientos a deshacer o no.
- `void onUndoNotPossible()`:
  La partida notifica a los observadores que una operación deshacer no ha tenido éxito porque no se puede deshacer.
- `void onMovimientoIncorrecto(MovimientoInvalido movimientoException)`:
  La partida notifica que se ha producido un movimiento incorrecto proporcionando el objeto `MovimientoInvalido` con una explicación del problema que se ha producido.
- `void onCambioJuego(TableroSoloLectura tab, Ficha turno)` :
  La partida notifica a los observadores que se ha cambiado el juego. Se proporciona el estado inicial del tablero y el turno[3].

Modelos

Los modelos son los encargados de comunicar a las vistas cualquier cambio que ocurra cuando alguno de sus métodos son invocados por el controlador. Por ejemplo:

[3]Como siempre, al principio de la partida el turno es de las blancas

```
public void undo() {
   if (undoStack.empty()) {
      // Broadcast.
      for (Observador o : obs) {
         o.onUndoNotPosible();
      }
      return;
   }
   Movimiento mov = undoStack.pop();
   mov.undo(this.tablero);
   this.turno = mov.getJugador();

   // Broadcast.
   for (Observador o : obs) {
      o.onUndo(tablero, turno, !undoStack.empty());
   }
}
```

Observa que en el método `void undo()` ya no lanza excepciones, sino que el mensaje se transmite al observador. En el caso de otros métodos, como `void ejecutaMovimiento(Movimiento mv)`, la excepción se proyecta como un parámetro más del mensaje (observa los métodos del interfaz Observador)

Vistas

- La clase `VistaConsola` tiene el siguiente aspecto:

```
public class VistaConsola implements Observador {
  void OnPartidaTerminada(TableroSoloLectura tableroFinal, FICHA
ganador)
   {
   if (ganador == FICHA.BLANCA)
      System.out.println("GANAN FICHAS BLANCAS");
   else if (ganador == FICHA.NEGRA)
      System.out.println("GANAN FICHAS NEGRAS");
   else
      System.out.println("EMPATE");
    }
  }
```

  La funcionalidad que ahora está en `Controlador` debe separarse entre la vista de consola y el controlador de consola.

  Nota SUPER Importante: Todos los `system.out.println` relativos a la configuración de la partida, tablero, ganadores, etc... deben estar en VistaConsola[4]. Si queda alguno en ControladorConsola, es que algo está funcionando mal y no estáis implementando bien el MVC.

- La clase `VistaGUI` adopta el siguiente perfil.

```
public class VistaGUI extends JFrame implements Observador{
  //.....
  public void OnMovimientoIncorrecto(MovimientoInvalido  movimientoException)
  {
    JOptionPane.showMessageDialog(null, movimientoException.getMessage());
```

[4]No así los propios de mensajes relativos a sintaxis incorrecta

```
    }
  }
```

Por supuesto, en el caso de `VistaGUI` se podrán crear otras clases que aportan componentes a `VistaGUI`:

- `OuterExterno`
- `LeftPanel`
- `RightPanel`
- `ScorePanel`

En el caso de `ScorePanel`, se puede declarar como subvista que observa también al modelo.

```
public class ScorePanel extends JPannel implements Observador{
  //.....
  void onMovimientoEnd(TableroSoloLectura tablero,
                              Ficha jugador, Ficha turno){
      // ...
  }
}
```

que por supuesto, habrá que añadir como observador del modelo. Y así las otras.

- En lo relativo a la programación de eventos sobre el Tablero, dirigirse a la sección **JButton**

Controladores

- El control de la ejecución se realiza a través del controlador. Como tenemos dos formas de ejecución, hemos de crear dos controladores, uno para cada tipo de vista: `ControladorConsola` y `ControladorGUI`.

  1. Ambos deben contener como atributo privado el **modelo** (la clase Partida)
  2. `ControladorConsola` contiene la partida, el Scanner, la factoría y los jugadores (igual que el antiguo Controlador).
  3. `ControladorGUI` contiene únicamente la partida y la factoría.

  Los controladores se encargan de comunicar a la partida las acciones solicitadas por el usuario. Por tanto se necesitarán métodos como:

```
...
void reset(Factoria f);
void undo();
void poner();
void reiniciar();
void finalizar();
...
```

  Estos métodos deberíais tenerlos ya implementados en el antiguo controlador[5]. Algunos de vosotros los tenéis como privados y otros como públicos. Ahora, pasan a ser públicos, ya que en el caso de la vista gráfica, `VistaGUI` , necesitan ser invocados por ésta.

[5] Si no los tenéis como métodos tendréis que crearlos factorizando el código asociado

Métodos innecesarios en ControladorGUI Existen algunos métodos que no tienen sentido en `ControladorGUI`;

- `void run()`, ya que el encargado de gestionar eventos en la ventana es la *hebra* de Swing.
- `void setJugador()`, no tiene sentido, pues en esta modalidad no existen dos tipos de jugadores.

Métodos que cambian de prototipo en ControladorGUI Algunos métodos como

`void poner()`,

en el caso del `ControladorGUI` cambian de signatura,

`void poner(int,int)`

pues al no haber clases de Jugadores, es la vista -el valor de los TextAreas- la que determina el valor que toman estos parámetros, para pasárselos al `ControladorGUI`.

JButton

TableroSoloLectura

- Con respecto al tablero de la clase partida: Los cambios en el estado del tablero implican un cambio de estado de la partida. Esto ocurre después de poner una ficha, reiniciar, deshacer, etc.

  Los métodos anteriores necesitan el estado del tablero (del modelo) para poder actualizar el tablero (en la vista). Para evitar que las vistas puedan modificar el tablero (incluso borrarlo), hacemos uso de un tablero inmutable, esto es un tablero de solo consulta. `TableroSoloLectura` es un interfaz con el que podemos proporcionar un **tablero de solo lectura**.

```
public interface TableroSoloLectura {
  int getFilas();
  int getColumnas();
  Ficha getCasilla(int fila, int col);
  String toString();
  }
```

  Este interfaz ofrece los métodos para consultar la información del tablero. La clase `Tablero` debe implementar este interfaz.