El juego del Connect 4

El Conecta 4 (o 4 en línea) es un juego estratégico de mesa para dos jugadores, tal y como se presenta en el documento del CV: *Presentacion-Entrega1Pract.pdf*, y que también podéis consultar en el siguiente artículo de la Wikipedia: https://es.wikipedia.org/wiki/Conecta_4

Como sabéis, el objetivo del juego es alinear cuatro fichas consecutivas –conectadas- del color asignado al jugador (e.g. rojo para el jugador humano) antes que tu oponente. Gana la partida el jugador que primero consigue alinear cuatro fichas consecutivas de su color en horizontal, vertical o diagonal, teniendo en cuenta que es un tablero vertical.

Se trata de construir un conjunto de clases que implementen el juego Connect 4 entre el jugador humano y el sistema, de manera que:

- mostrará el tablero de juego, de tal forma que el jugador humano pueda interaccionar para seleccionar su movimiento (la columna por la que dejar caer su ficha)
- gestionará el cambio de turno
- detectará el final de la partida para las cuatro situaciones siguientes: 1) victoria del sistema; 2) victoria del jugador humano; 3) situación de empate (todas les columnas están llenas y nadie ha conseguido alinear cuatro fichas); 4) finalización por tiempo agotado
- gestionará el movimiento del jugador no humano, así como el tiempo transcurrido hasta un cierto límite (si así lo desea el jugador humano) aplicando tan sólo jugadas válidas (para ocupar un hueco es necesario haber ocupado anteriormente todos los huecos inferiores en esa columna o, dicho de otra forma, las fichas van cayendo por la columna hasta la primera posición libre –la posición válida)

A continuación, se sugiere una posible implementación, caracterizada por la división de la solución en clases pequeñas que, combinadas, resuelven el problema.

Lógica del juego: clases a implementar

Dado que el juego se desarrolla en un tablero bidimensional y, dado que para detectar el final de partida se requiere explorar todas las direcciones posibles, se recomienda construir dos clases, una para representar direcciones de movimiento y otra para representar posiciones. De esta manera, el código que realice recorridos por el tablero será sencillo de programar.

La clase Direction

Esta clase representa las direcciones de movimiento que se utilizarán para explorar el número de fichas consecutivas del mismo color, para así detectar si alguno de los jugadores ha ganado la partida.

Las direcciones están expresadas como modificaciones sobre filas y columnas (primero la fila y luego la columna); las filas aumentan hacia abajo y las columnas hacia la derecha. Se definen 4 direcciones. Mediante la definición de constantes se puede facilitar el acceso a ellas. El array ALL las contiene todas. El método *invert*() se encarga de invertir el sentido de estas 4 direcciones.

Su código sería:

```
public class Direction {

   public static final Direction DOWN
                                          = new Direction(1, 0);
   public static final Direction RIGHT
                                          = new Direction(0, 1);
   public static final Direction MAIN_DIAGONAL
                                          = new Direction(1, 1);
   public static final Direction CONTRA_DIAGONAL
                                          = new Direction(1, -1);

   public static final Direction[] ALL = new Direction[] {
         RIGHT, DOWN, MAIN_DIAGONAL, CONTRA_DIAGONAL
   };

   private final int changeInRow;
   private final int changeInColumn;

   private Direction(int changeInRow, int changeInColumn) { . . . }

   public int getChangeInRow() { . . . }

   public int getChangeInColumn() { . . . }

   public Direction invert() { . . . }
}
```

Uno de los usos más comunes que haréis del array `ALL` será para considerar todos los movimientos posibles:

```
for (int i=0; i < Direction.ALL.length; i ++) {
    Direction dir = Direction.ALL[i];
     // buscar en la dirección dir
}
```

La clase Position

Esta clase representa una posición en el tablero, identificada por la fila y la columna.

```
public class Position {

    private final int row;
```

```
    private final int column;

    Position(int row, int column) { ... }

    int getRow() { ... }

    int getColumn() { ... }

    Position move(Direction direction) { ... }

    boolean isEqualTo(Position other) { ... }

    static int pathLength(Position pos1, Position pos2) {
       // pos1 and pos2 are aligned horizontally, vertically or diagonally???
    }
}
```

Fijaros que se ha definido como una clase inmutable. Es por ello que el método *move*() retorna una posición nueva, en lugar de modificar la posición sobre la que se aplica.

Detalle de algunos métodos:

- ➔ `public Position move(Direction direction)`
  - devuelve la posición correspondiente a moverse desde la posición del objeto receptor en la dirección indicada por direction
- ➔ `public static int pathLength(Position pos1, Position pos2)`
  - indica la longitud del camino que une ambas posiciones

La clase Board

Esta clase representa el tablero de juego. A continuación algunos detalles de parte de la clase (métodos de acceso privado al paquete):

```
public class Board {

    private final int size;
    private final Cell[1][][] cells;

    Board(int size) { ... }

    //  getters and setters

    Position occupyCell (int column, Player player) { ... }

    boolean hasValidMoves() { ... }

    int firstEmptyRow(int column) {
       // Assume column is playable
       ...
    }
```

[1] Se ha dejado sin concretar la clase *Cell*, así como también la clase *Player*.

```
        int maxConnected(Position position) { ... }
           // obtains the maximum number of connected positions in any direction
        }
```

La funcionalidad de algunos de los métodos es la siguiente:

- ➔ `boolean hasValidMoves()`
  - ◆ indica si existe alguna columna en la que poder jugar.
- ➔ `int firstEmptyRow(int column)`
  - ◆ obtiene la primera posición libre (la fila a ocupar) de la columna.
  - ◆ retorna -1 si columna llena
- ➔ `int maxConnected(Position position)`
  - ◆ devuelve el número de celdas conectadas (en alguna de las cuatro direcciones a considerar) a partir de la posición dada.

La clase Game

Es la clase que representa la partida que se está jugando. Por lo tanto, además de conocer el tablero, conoce el estado actual del juego. El estado del juego se puede modelar con un tipo enumerado (*Status* en este caso).

A continuación se presenta parte de la clase Game (tan sólo métodos del primer nivel de descomposición):

```
public class Game {

    private final Board board;
    private final int toWin;
    private Status status;
    private boolean hasWinner;
    private Player turn;

    // state members related with time
    ( . . .)

    public Game(int size, int toWin) { . . . }

    //  getters and setters

    Position playOpponent () {  . . . }

    void toggleTurn() {  . . . }

    void manageTime() {  . . . }

    boolean checkForFinish () {

    ??? drop(int col)
}
```

- Position playOpponent()
  - gestiona el movimiento del jugador no humano. Esta parte puede variar entre lo más básico (selección de la posición de forma aleatoria entre las posiciones actualmente válidas), o bien aplicar algún tipo de heurística, simulando un juego más cercano a la realidad. En este caso se pueden establecer diversos niveles de dificultad (parte opcional de la práctica).
- void toggleTurn() { . . . }
  - cambia el turno
- void manageTime() { . . . }
  - se encarga de la gestión del tiempo. Ello incluye detectar si se ha agotado el tiempo establecido y por tanto de dar los avisos pertinentes para proceder a finalizar la partida.
- boolean checkForFinish ()
  - devuelve si la partida ha finalizado: ha ganado uno de los dos jugadores o ya no quedan movimientos posibles (empate).
- ??? drop(int col)
  - podéis asumir que la partida no está acabada
  - anota el movimiento del jugador actual (deducible a partir del estado de la partida)
  - advierte en caso de columna llena
  - actualiza el estado de la partida y gestiona el tiempo (si así se requiere)
  - se encarga ella misma de refrescar el tablero (caso de tener acceso a la interfaz gráfica), o bien retorna información sobre la posición en la que ha acabado la ficha (esa información servirá para que la interfaz gráfica muestre el movimiento de forma adecuada)