Grado en Ingeniería Informática

Metodología de la Programación

Práctica Obligatoria 2 – Primera Convocatoria

Atari-Go 2.0

Docentes:

Raúl Marticorena

David Caubilla

Índice de contenidos

1. Introducción

El objetivo fundamental es realizar variaciones al juego de **Atari-Go,** implementado en la práctica anterior, incluyendo las propiedades avanzadas vistas en los temas de **herencia, genericidad y tratamiento de excepciones.**

Recordemos que es un juego abstracto de tablero para dos jugadores, a los que se asignan **piedras** de un determinado color a cada uno de los jugadores (negras y blancas respectivamente).

Durante el juego, los jugadores se alternan colocando las piedras, de una en una, sobre las intersecciones vacías de un tablero de 19×19 líneas en su versión estándar. Otros tamaños típicos de tablero son 9x9 y 13x13 líneas para principiantes.

Al inicio de la partida el tablero está vacío. Las negras juegan primero. Después ambos jugadores colocan piedras alternativamente en las intersecciones vacías. Una vez colocada una piedra, no se mueve en el resto de la partida.

En la versión simplificada del Atari-Go de la práctica anterior, el objetivo era realizar la **primera captura** de un grupo de piedras del contrario. **Un "grupo de piedras" está formado por una o varias piedras del mismo color** en celdas adyacentes (sólo en sentido **horizontal o vertical**, **nunca en diagonal**).

Un grupo es **capturado** en el juego **si después de una jugada no posee intersecciones vacías adyacentes**, esto es, si se encuentra **completamente rodeada por piedras del color contrario en todas sus intersecciones adyacentes**.

Un principio básico del juego es que los grupos de piedras deben tener al menos una "libertad" para quedarse en el tablero. Una "libertad" es un "punto" abierto (intersección) adyacente a una piedra**.**

En la Ilustración 1, podemos ver un ejemplo: en el estado A, la piedra negra tiene 4 libertades inicialmente. En B y C se van reduciendo sus libertades, hasta que en el estado D, su libertad es sólo de 1, y está amenazada (se denomina *atari*). Si se coloca una piedra blanca en la posición 1, su libertad es cero y pasa a ser capturada, eliminándose del tablero la piedra negra.

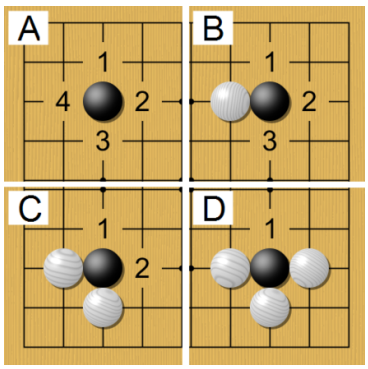

Ilustración 1: Ejemplos de libertades

En este juego, no se permite hacer una jugada ocupando una posición libre en el interior de una formación enemiga (provocando la propia captura o **suicidio**), a no ser que esta jugada **capture piedras enemigas** (ver Ilustración 2). Si la formación que uno quiere capturar, está a su vez rodeada, entonces sí se puede jugar en su interior.

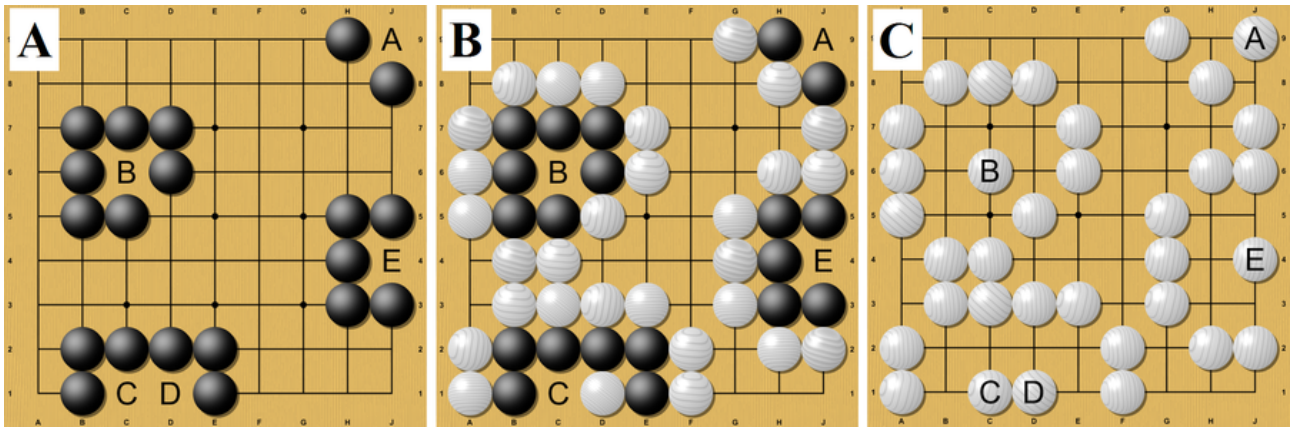

Ilustración 2: Ejemplos de suicidios prohibidos (gráfico A) y permitidos (gráfico B) junto con la situación posterior a la captura (C) para el jugador con blancas

En esta segunda versión se implementarán además **dos variantes del juego:**

- **Árbitro básico** sigue la regla de que se finaliza la partida en la **primera captura,** con independencia del número de piedras capturadas.
- **Árbitro avanzado**: se finaliza la captura cuando se captura un valor mínimo fijado de piedras del contrario. Por ejemplo si la cota es 3, cuando se capturen 3 o más piedras de un jugador de cualquier color, se finaliza.

Para desarrollar este juego se establece la siguiente estructura de paquetes y módulos/clases (ver Tabla 1):

Paquete	Módulos / Clases	Nº	Descripción
juego.modelo	Celda Color Grupo Jugador Piedra Tablero	5 clases 1 enumeración	Modelo.
juego.control	Arbitro ArbitroAtariGo ArbitroAtariGoBasico ArbitroAtariGoAvanzado ConfiguracionAtariGo	1 interfaz 1 clase abstracta 3 clases	Lógica de negocio.
juego.util	**ConversorJugada** **Sentido** **CoordenadasIncorrectasException**	1 clase 1 enumeración 1 excepción	Utilidad.
juego.textui	**AtariGo**	1 clase	Interfaz de usuario en modo texto. Se proporciona parcialmente el código a completar
juego.gui	Sin determinar	Sin determinar	Interfaz gráfica.
juego.gui.images	Sin determinar	Sin determinar	Imágenes en la aplicación.

Tabla 1: Resumen de paquetes y módulos

La interfaz gráfica se ha subcontratado a la empresa ECMA (filas en color verde en Tabla 1), siguiendo los diagramas e indicaciones que se detallan en el presente enunciado y se proporciona en el formato `.jar` en la plataforma UBUVirtual.

Es labor de los alumnos implementar los ficheros restantes necesarios[1] para el correcto cierre y ensamblaje del sistema, tanto en modo texto y gráfico, junto con el resto de productos indicados en el apartado 3 Descripción.

Para ello se deben seguir las indicaciones dadas y los diagramas de clases disponibles, respetando los diseños y signaturas de los métodos, con sus correspondientes modificadores de acceso, **, quedando a decisión del alumno la inclusión de atributos y/o métodos protegidos (`protected`), privados (`private`) o amigables, siempre de manera justificada. No se pueden añadir atributos ni métodos públicos adicionales. Tampoco se puede modificar la signatura de los métodos públicos, respecto a las excepciones comprobables indicadas en el enunciado.**

2. Objetivos

- Construir siguiendo los diagramas e indicaciones dadas la implementación del juego en Java.
  - Completando una aplicación en modo texto.
  - Completando una aplicación en modo gráfico.
- Generar la documentación correspondiente al código en formato HTML.
- Comprobar la completitud de la documentación generada previamente.
- Aportar los *scripts* correspondientes para realizar el proceso completo de compilación, documentación y ejecución (tanto en modo texto como gráfico).

3. Descripción

A continuación se desglosan los distintos diagramas UML, y comentarios, a tener en cuenta de cara a la implementación de la práctica. Aquellos métodos de consulta (Ej: `obtener`, `consultar`, etc.) y asignación (Ej: `establecer`, `colocar`, etc.) que no conllevan ningún proceso adicional, salvo la lectura o escritura de atributos, no se comentan por motivos de brevedad.

3.1 Paquete `juego.modelo`

El paquete está formado por cinco clases y una enumeración. En la Ilustración 3 se muestra el diagrama de clases correspondiente.

Comentarios respecto a `Celda`:

- Inicialmente toda celda estará vacía y sólo contiene las coordenadas con su posición en el tablero.
- A lo largo del tiempo una celda podrá estar vacía o contener una piedra, se colocará una piedra en una celda a través del método `establecerPiedra`.
- Una celda vacía no tiene piedra (valor **`null`**).
- El método `estaVacia` permite consultar si la celda contiene una piedra o no.

1 Las clases del paquete `juego.util` y el código parcial `juego.textui.AtariGo` se proporcionan en UBUVirtual.

- El método `tieneIgualesCoordenadas` comprueba si la celda actual y la pasada como argumento tienen igual fila y columna.
- Una celda puede ser vaciada, a través del método `eliminarPiedra`. La piedra pasa a estar también desvinculado de la celda.
- El método `toString` devuelve el estado de la celda en formato cadena de caracteres, solo sus coordenadas. Ej: "(0 / 0)" , "(1 / 1)", "(2 / 0)".

Comentarios respecto a `Color`:

- Tipo enumerado que contiene valores `BLANCO` y `NEGRO`.
- Permite almacenar y consultar el correspondiente carácter asociado a cada color (`'O'`, `'X'`) respectivamente.

Comentarios respecto a `Jugador`:

- Describe a un jugador, con su nombre y color asociado.
- El método `generarPiedra` permite generar nuevas piedras para el jugador (para su posterior colocación sobre el tablero). Toda piedra que se vaya a colocar sobre el tablero debe ser obtenida a través de este método.
- El método `toString` devuelve el estado del jugador en formato cadena de caracteres con su nombre y color asociado. Ej: `"Pepe/BLANCO"` , `"Juan/NEGRO"`.

Comentarios respecto a `Piedra`:

- Una piedra tiene asociado siempre un color.
- Una vez creada una piedra, se podrá asociar a una celda, a través del método `colocarEn`.
- El método `toString` devuelve el estado de la piedra en formato cadena de caracteres con su celda y color asociado. Ej: `"null/BLANCO"` , `"(0 / 0)/NEGRO"`.

Comentarios respecto a `Tablero`:

- Un tablero se considera como un conjunto de celdas, cada una en una posición (fila,columna). Suponiendo que el tablero es de 9 filas x 9 columnas, entonces tenemos: (0,0) las coordenadas de la esquina superior izquierda, (0,8) las coordenadas de la esquina superior derecha, (8,0) las coordenadas de la esquina inferior izquierda y (8,8) las coordenadas de la esquina inferior derecha. Se numera de izquierda a derecha y en sentido descendente.
- El conjunto de celdas de un tablero debe implementarse con una **lista de listas de celdas genérica** (usando la interfaz `List` y la clase `ArrayList` del paquete `java.util`). Al instanciar un tablero se crean y asignan las correspondientes celdas vacías, con sus correspondientes coordenadas.
- El método `colocar` coloca una piedra en una determinada celda. Cuando se coloca una piedra en una celda, ambos objetos (piedra y celda) **deben quedar ligados uno a otro** (la piedra está en la celda / la celda contiene una piedra). La celda debe pertenecer al tablero (debe haberse obtenido de él), para ello se debe utilizar el método `obtenerCeldaConMismasCoordenadas`. No se comprueba si la celda está vacía o no (se delega esta comprobación defensiva en el árbitro). **Lanza una excepción comprobable** **`CoordenadasIncorrectasException`**, si las coordenadas de la celda no pertenecen al tablero.
- El método `estaEnTablero` comprueba que unas determinadas coordenadas están en los límites de tablero.
- El método `estaCompleto` informa si un tablero tiene alguna celda vacía o no.
- El método `generarCopia` genera un **nuevo tablero** con una **copia** de las **piedras** y **grupos**. El tablero copia **no comparte objetos** con el original por lo que se puede trabajar sobre él, sin riesgo a modificar el tablero original. Para simplificar su implementación se usará el método `generarCopiaEnOtroTablero` de la clase `Grupo`.

- El método `obtenerCelda` devuelve la celda en las coordenadas indicadas. **Lanza una excepción comprobable** **`CoordenadasIncorrectasException`**, si las coordenadas de la celda no pertenecen al tablero.
- El método `obtenerCeldasAdyacentes` **devuelve la lista** (tipo `java.util.ArrayList`) con la celdas adyacentes (en los cuatro sentidos) a una celda dada. Como mínimo tendremos dos celdas y como máximo cuatro. La clase `ArrayList` se describe con mayor detalle en el anexo a este enunciado (ver Error: no se encontró el origen de la referencia). **Lanza una excepción comprobable** **`CoordenadasIncorrectasException`**, si las coordenadas de la celda, no pertenecen al tablero.
- El método `obtenerGradosDeLibertad` devuelve los grados de libertad de una celda concreta del tablera comprobando si están libres las celdas adyacentes. **Lanza una excepción comprobable** **`CoordenadasIncorrectasException`**, si las coordenadas de la celda, no pertenecen al tablero.
- El método `obtenerCeldaConMismasCoordenadas` devuelve la celda del tablero con las mismas coordenadas de la celda pasada como argumento. **Lanza una excepción comprobable** **`CoordenadasIncorrectasException`**, si las coordenadas de la celda, no pertenecen al tablero.
- El método `obtenerGruposDelJugador` **devuelve una lista** de grupos que tienen piedras del color del jugador indicado.
- El método `obtenerNumeroPiedras` cuenta el número de piedras de un determinado color, que hay sobre el tablero actualmente.

Comentarios respecto a `Grupo`: representa un conjunto de celdas adyacentes con piedras del mismo color (o ausencia del mismo). Un grupo está formado por una o varias celdas.

- El método `obtenerId` devuelve el identificador numérico del grupo. Cada vez que se instancia un grupo dicho contador debe incrementarse de tal forma que cada grupo tiene un único valor (Nota: utilizar un atributo estático).
- El método `obtenerColor` devuelve el color de las piedras contenidas en el grupo. Si el grupo ha sido capturado debería devolver `null`.
- El método `estaVivo` devuelve true si los grados de libertad del grupo es mayor que cero. En caso contrario está completamente rodeado y está capturado.
- El método `obtenerTamaño` devuelve el número de celdas que componen un grupo.
- El método `contiene` comprueba si la celda está contenida en el grupo, sólo comprobando sus coordenadas, sin comprobar si tiene o no piedra.
- El método `añadirCeldas` agrega al grupo el conjunto de celdas contenido en el grupo pasado como argumento.
- El método `eliminarPiedras` vacía de piedras el grupo, pasando a estar todas vacías.
- El método `generarCopiaEnOtroTablero`, genera un nuevo grupo, cuyas celdas son las equivalentes en coordenadas a las del tablero que se pasa. El grupo actual no cambia su estado.
- **Ninguno de sus métodos lanza o propaga excepciones comprobables. Pero puede lanzar o propagar excepciones NO comprobables.**

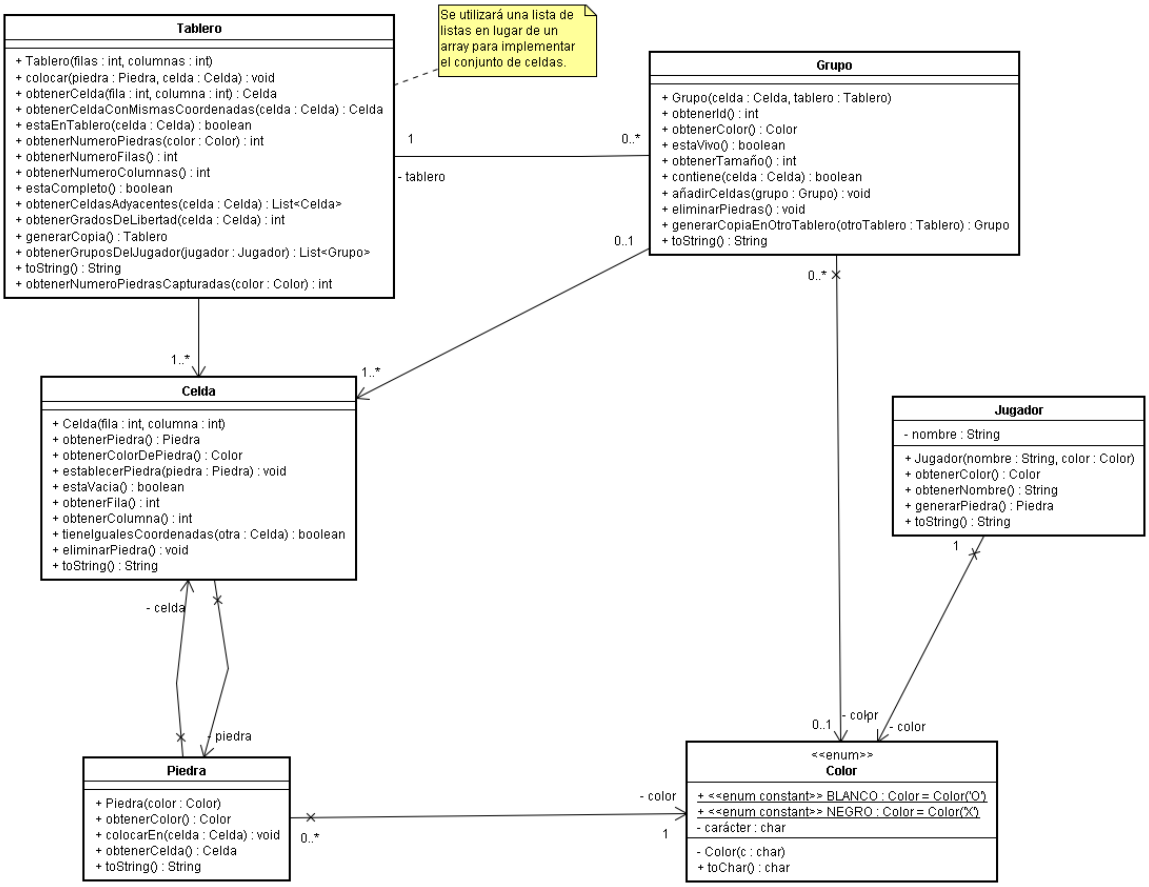

Ilustración 3: Diagrama de clases de `juego.modelo`

3.2 Paquete `juego.control`

El paquete contiene una interfaz, una clase abstracta y tres clases. Definen la lógica de negocio a implementar en soluciones concretas del juego y la configuración general del juego (ver Ilustración 4).

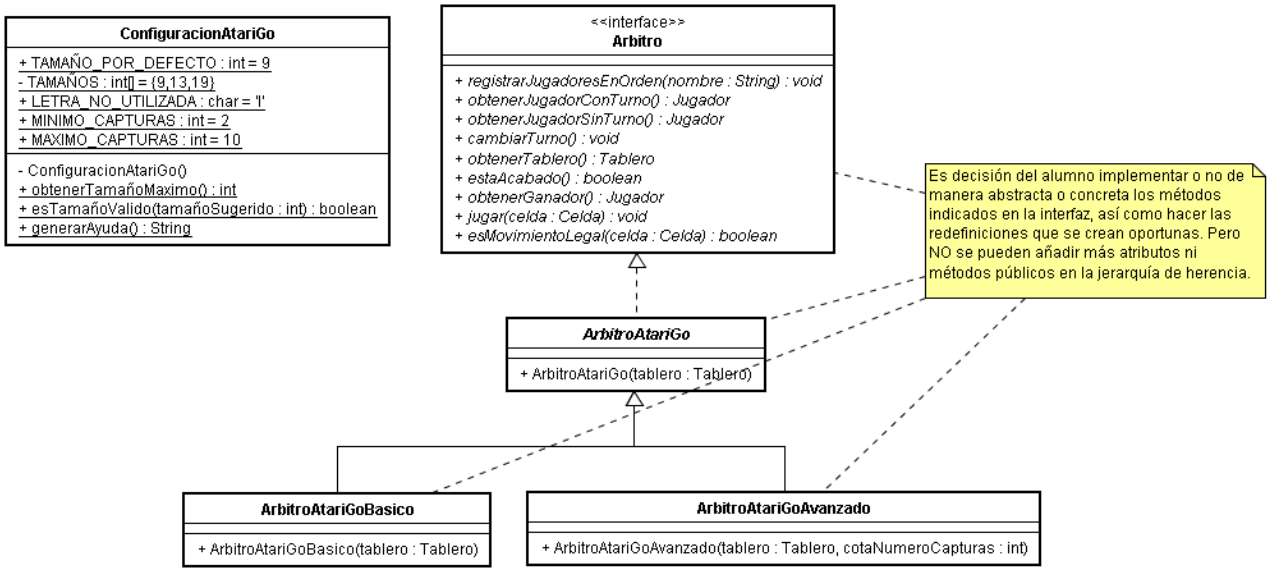

Ilustración 4: Diagrama de clases de `juego.control`

Comentarios respecto a la interfaz `Arbitro`:

- El método `registarJugadorEnOrden`, registra en primer lugar al jugador con piedras negras, en segunda invocación al jugador con piedras blancas y en sucesivas invocaciones nada. Recordad que el turno inicial lo tiene el jugador con negras.
- El método `cambiarTurno` avanza el turno según qué jugador ha realizado el último movimiento.
- El método `esMovimientoLegal` comprueba si la jugada es legal en base a las reglas del juego: la celda seleccionada, o bien está vacía, o bien el movimiento no genera una situación de suicidio. Para comprobar el suicidio, se debe generar una copia del tablero actual, y realizar el movimiento sobre dicha copia, comprobando si realmente se produce suicidio o captura sobre la copia. De esta forma el tablero original nunca se ve afectado.
- El método `estaAcabado` permite consultar si el juego está acabado.
- El método `jugar` realiza la colocación de la piedra del turno actual en la celda. Al colocar una piedra se tienen que actualizar los grupos actuales para dicho jugador. Si la piedra nueva no es adyacente a otros grupos del mismo color, formara un grupo inicial de una piedra. Pero si es adyacente a otros grupos, comprobando los cuatro sentidos, se deben unir. Para ello se puede formar un nuevo grupo con la unión de las celdas de ambos, y eliminar de la lista de grupos el grupo absorbido. Al inicio de la ejecución de este método **no se debe invocar al método** `esMovimientoLegal` puesto que se supone que siempre ha sido consultada previamente la legalidad de la jugada. Finalmente, gestiona el cambio de turno, si procede, o bien determina el estado acabado. **Este método lanza una excepción comprobable CoordenadasIncorrectasException si las coordenadas de la celda no están dentro del tablero.**
- El método `obtenerGanador` informa del jugador ganador o **`null`** en caso de que no haya ganador todavía.
- El método `obtenerTablero` permite obtener una referencia al tablero actual que se está manejando.
- El método `obtenerJugadorConTurno` y `obtenerJugadorSinTurno` obtienen los jugadores correspondientes.

Dada la complejidad de algún método, en especial del método `esMovimientoLegal` y del método `jugar`, y de que quizás se pueda generar código duplicado, se recomienda dividir el código en métodos privados más pequeños y reutilizarlos siempre que sea posible.

Comentarios respecto a la clase abstracta `ArbitroAtariGo`:

- Ver diagrama.

Comentarios respecto a la clase `ArbitroAtariGoBasico`:

- Ver diagrama.
- El juego finaliza con la primera captura.

Comentarios respecto a la clase `ArbitroAtariGoAvanzado`:

- Ver diagrama.
- El juego finaliza cuando se captura un número mínimo de piedras de cualquiera de los dos jugadores. El número mínimo o cota, se pasa en el constructor del árbitro.

Tal y como se indica en el diagrama, es decisión del alumno implementar o no de manera abstracta/concreta los métodos indicados en la interfaz en las clases descendientes, así como hacer las redefiniciones que se crean oportunas. Pero **NO se pueden añadir más atributos ni métodos públicos** en la jerarquía de herencia.

Comentarios respecto a la clase `ConfiguracionAtariGo`: la clase es meramente funcional (sin estado asociado) con atributos estáticos constantes y métodos estáticos (**`static`**):

- Incluye los valores constantes que definen el juego.
- El método `obtenerTamañoMaximo` devuelve el támaño máximo permitido en el juego.
- El método `esTamañoValido` comprueba si el número pasado está entre los tamaños permitidos para un tablero (9, 13 o 19).
- El método `generarAyuda` genera un texto de ayuda con la información de la configuración.

3.3 Paquete `juego.util`

Contiene una clase y una enumeración como se puede ver en Ilustración 5.

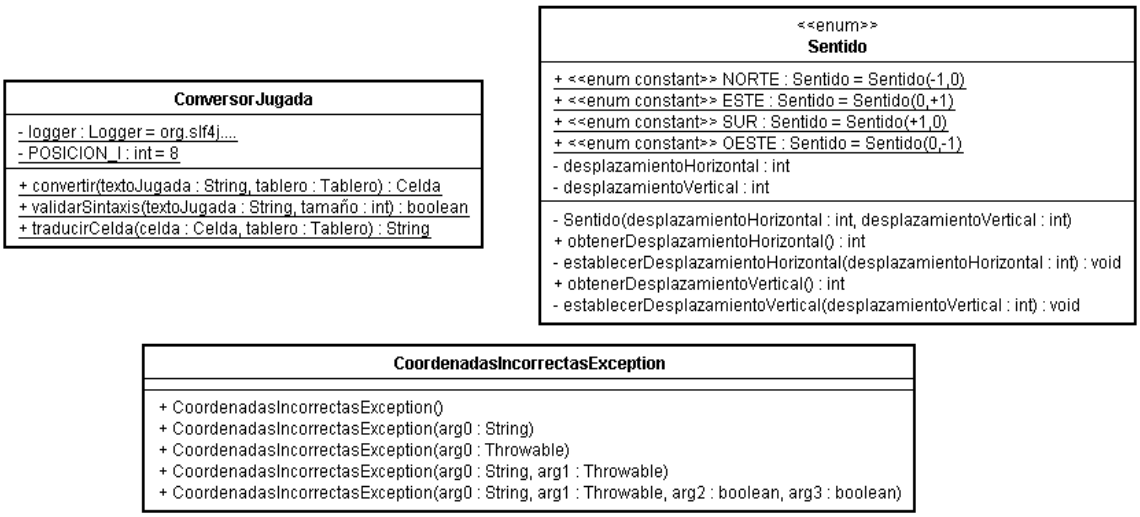

Ilustración 5: Diagrama de clases `juego.util`

La enumeración `Sentido` contiene los cuatro sentidos que se tienen en cuenta para considerar celdas contiguas, junto con su desplazamiento en filas y columnas correspondiente a cada caso. Se recuerda el uso del método `values` para obtener un *array* con todos los valores definidos en el tipo enumerado para simplificar el código.

Comentarios respecto a la clase `ConversorJugada`: la clase es meramente funcional (sin estado asociado) con tres métodos estáticos (**`static`**):

- El método `convertir` obtiene la celda correspondiente del tablero. Si los valores no son válidos devuelve un valor **`null`**.
- El método `validar` comprueba el texto introducido como jugada, indicando si puede ser traducido o no a una celda en el tablero. **No garantiza que el movimiento sea legal**, sólo que las coordenadas son válidas dentro del tablero.
- El método `traducirCelda` convierte las coordenadas de una celda (fila, columna) a su correspondiente texto a introducir por el usuario, según se muestra el tablero en pantalla. Ej: para (0,0) devuelve `9A`, para (8,8) devuelve `1J` en un tablero de 9x9 celdas.

Comentarios respecto a la excepción `CoordenadasIncorrectasException`: es una excepción **comprobable**, que incluye los constructores clásicos a la hora de construir una clase de tipo excepción.

3.4 Paquete `juego.textui`

En este paquete se implementa, la interfaz en modo texto, que reutiliza las paquetes anteriores. El código debe incluir la validación de los argumentos de invocación en línea de comandos (`args` en el método `main`) . Las clase raíz del sistema es `juego.textui.AtariGo`.

La sintaxis de invocación (sin detallar cómo debe configurarse el `classpath`) para un tablero de 9 filas x 9 columnas sería:

```
$> java juego.textui.AtariGo Pepe Juan 9
```

O bien se introducen tres o cuatro argumentos (nombre jugador negras nombre jugador blancas, tamaño y opcionalmente la cota de piedras para el modo avanzado, o bien ninguno (con valores por defecto "Abel", "Caín" y 9x9 celdas en modo básico). Si no se introducen correctamente los argumentos (bien cero, tres o cuatro argumentos) se muestra un mensaje de ayuda con el formato de invocación y se interrumpe la ejecución. El tamaño del tablero debe estar entre los valores establecidos como válidos

en la clase `ConfiguracionAtariGo` (ver método `esTamañoValido`). Si se introduce una cota para trabajar en modo avanzado el valor debe estar entre 2 a 10 (incluidos).

Una particularidad a la hora de nombrar las columnas con letras, es que no se utiliza la letra I, saltando directamente a la letra J. Por otro lado, las filas se numeran de manera decreciente.

Ejemplo de invocación (sin detallar cómo debe configurarse el `classpath y` con valores por defecto):

Jugar con un tablero de 9 filas x 9 columnas en modo básico:

```
$> java juego.textui.AtariGo
```

La salida en pantalla debe ser similar a la siguiente:

```
9       -  -  -  -  -  -  -  -  -
8       -  -  -  -  -  -  -  -  -
7       -  -  -  -  -  -  -  -  -
6       -  -  -  -  -  -  -  -  -
5       -  -  -  -  -  -  -  -  -
4       -  -  -  -  -  -  -  -  -
3       -  -  -  -  -  -  -  -  -
2       -  -  -  -  -  -  -  -  -
1       -  -  -  -  -  -  -  -  -

        A  B  C  D  E  F  G  H  J

El turno es de: Abel con piedras X de color NEGRO
Introduce jugada: 2C
Piedras capturadas de color NEGRO:     0
Piedras capturadas de color BLANCO:    0

9       -  -  -  -  -  -  -  -  -
8       -  -  -  -  -  -  -  -  -
7       -  -  -  -  -  -  -  -  -
6       -  -  -  -  -  -  -  -  -
5       -  -  -  -  -  -  -  -  -
4       -  -  -  -  -  -  -  -  -
3       -  -  -  -  -  -  -  -  -
2       -  -  X  -  -  -  -  -  -
1       -  -  -  -  -  -  -  -  -

        A  B  C  D  E  F  G  H  J
```


```
El turno es de: Cain con piedras O de color BLANCO

Introduce jugada:
```

El formato de jugada a introducir es *númeroFilaLetraColumna*. Ejemplo: `2c`. La entrada del usuario se debe convertir a mayúsculas siempre (con el método `String.toUpperCase`).

Se informa tras cada jugada legal del nuevo estado del tablero y del jugador que tiene el turno actual. Si la jugada introducida no es legal, se informa del error al usuario, solicitando de nuevo que introduzca la jugada. No hay límite en el número de reintentos.

Además de informa del número de piedras capturas en cada tirada de ambos jugadores (ya sea en modo básico o avanzado).

En cada celda se muestra la letra correspondiente al color ('0' o 'X' o bien '-' si la celda está vacía).

Si en algún momento se alcanza la situación de finalización, por victoria, se acaba la partida informando del ganador.

Otros ejemplos de invocación, en este caso con argumentos y cota 4 en modo avanzado:

1) Jugar con un tablero de 13 x 13.

```
$> java juego.textui.AtariGo Pepe Juan 13 4


12      -  -  -  -  -  -  -  -  -  -  -  -  -
11      -  -  -  -  -  -  -  -  -  -  -  -  -
10      -  -  -  -  -  -  -  -  -  -  -  -  -
9       -  -  -  -  -  -  -  -  -  -  -  -  -
8       -  -  -  -  -  -  -  -  -  -  -  -  -
7       -  -  -  -  -  -  -  -  -  -  -  -  -
6       -  -  -  -  -  -  -  -  -  -  -  -  -
5       -  -  -  -  -  -  -  -  -  -  -  -  -
4       -  -  -  -  -  -  -  -  -  -  -  -  -
3       -  -  -  -  -  -  -  -  -  -  -  -  -
2       -  -  -  -  -  -  -  -  -  -  -  -  -
1       -  -  -  -  -  -  -  -  -  -  -  -  -

        A  B  C  D  E  F  G  H  J  K  L  M  N


El turno es de: Pepe con piedras X de color NEGRO
Introduce jugada:
```

Suponiendo que se realizan los siguientes movimientos: `12A`, `11B`, `5C`, `2N`, `1N` tendríamos la partida en un estado como el siguiente:

```
Piedras capturadas de color NEGRO:      0
Piedras capturadas de color BLANCO:     0


13      -  -  -  -  -  -  -  -  -  -  -  -  -
12      X  -  -  -  -  -  -  -  -  -  -  -  -
```


```
11      -  O  -  -  -  -  -  -  -  -  -  -  -
10      -  -  -  -  -  -  -  -  -  -  -  -  -
9       -  -  -  -  -  -  -  -  -  -  -  -  -
8       -  -  -  -  -  -  -  -  -  -  -  -  -
7       -  -  -  -  -  -  -  -  -  -  -  -  -
6       -  -  -  -  -  -  -  -  -  -  -  -  -
5       -  -  X  -  -  -  -  -  -  -  -  -  -
4       -  -  -  -  -  -  -  -  -  -  -  -  -
3       -  -  -  -  -  -  -  -  -  -  -  -  -
2       -  -  -  -  -  -  -  -  -  -  -  -  O
1       -  -  -  -  -  -  -  -  -  -  -  -  X

        A  B  C  D  E  F  G  H  J  K  L  M  N

El turno es de: Juan con piedras O de color BLANCO
Introduce jugada:
```

3.5 Paquete `juego.gui/juego.gui.images`

Estos paquetes implementan la interfaz gráfica del juego. La clase raíz del sistema es `juego.gui.AtariGo`.

Ejemplo de invocación (sin detallar cómo debe configurarse el `classpath`) para un tablero de 9 filas x 9 columnas en modo básico:

```
$> java juego.gui.AtariGo
```

La interfaz inicial será similar a la mostrada en la Ilustración 6. Para realizar un movimiento se selecciona la celda de destino en una de las intersecciones de las líneas.

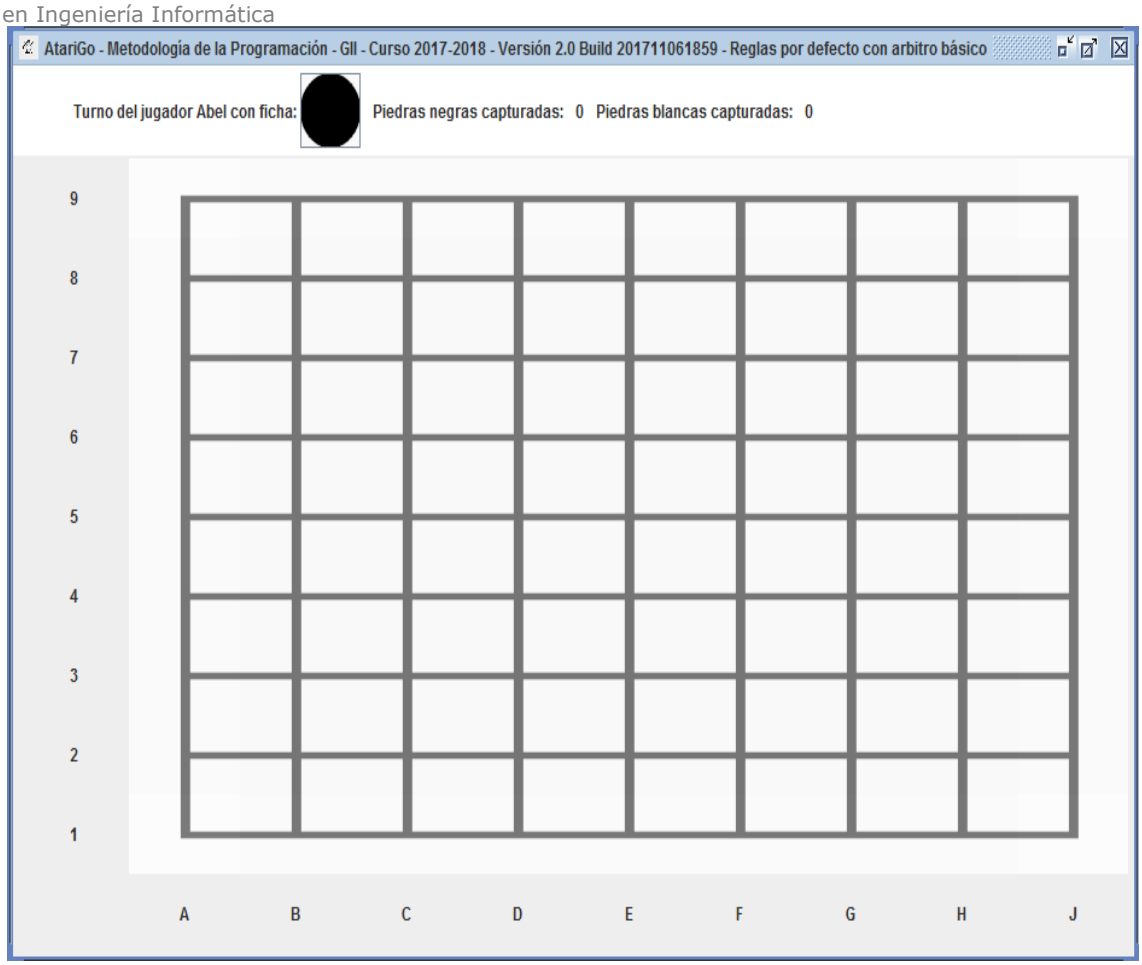

Ilustración 6: Interfaz gráfico inicial

La interfaz gráfica generada, tras una serie de movimientos, puede tener un aspecto similar al mostrado en la Ilustración 7. En dicha captura de pantalla, podemos observar que hay:

- 6 grupos de color NEGRO:
  - (9A-9B) con grado de libertad 1 en situación de atari.
  - (7A) con grado de libertad 1 en situación de atari.
  - (6C) con grado de libertad 2.
  - (5B) con grado de libertad 2.
  - (4D) con grado de libertad 3.
  - (5F-5G-5H-4F) con grado de libertad 5.
- 5 grupos de color BLANCO:
  - (8A-8B) con grado de libertad 2.
  - (6A-6B) con grado de libertad 2.
  - (5C) con grado de libertad 2.
  - (6F-6G) con grado de libertad 4.
  - (5E-4E) con grado de libertad 3.

Tras cada movimiento, se actualiza la información del turno y de piedras capturadas de cada color.

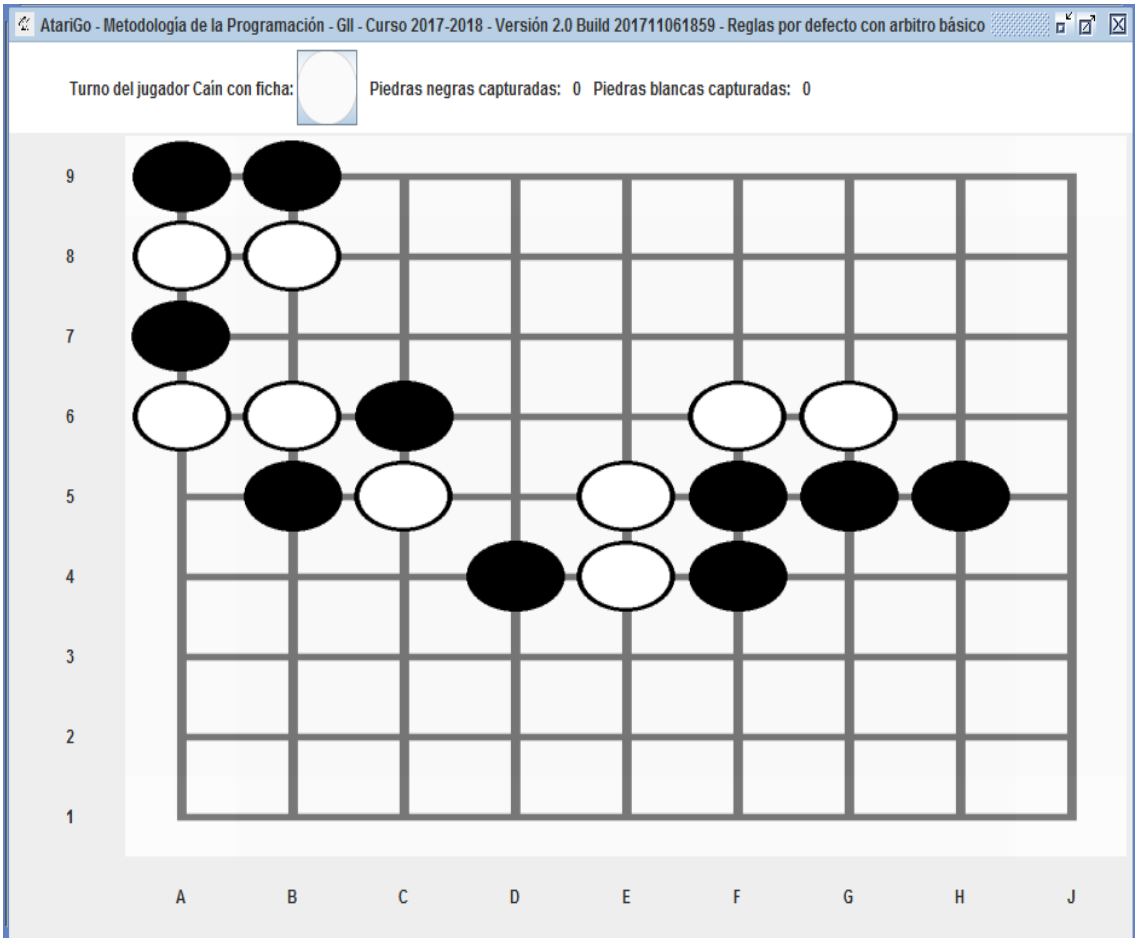

Ilustración 7: Evolución del juego en modo gráfico

Las condiciones en los argumentos de entrada son las mismas que en el modo texto (cero, tres o cuatro argumentos), pudiendo ejecutarse con valores por defecto (9 x 9) o con valores numéricos según están definidos en `ConfiguracionAtariGo`.

El cuarto valor con la cota entre [2,10] determina que jugamos con el árbitro avanzado.

Las clases correspondientes a estos paquetes se proporcionan en formato binario en un fichero `.jar` con nombre `atarig-go-gui-lib-2.0.jar` y se debe usar con los módulos construidos previamente configurando el `classpath` y **sin descomprimir en ningún caso el fichero .jar**.

3.6 Bibliotecas adicionales

Para facilitar la depuración del código al utilizar la interfaz gráfica se ha incluido el uso de un mecanismo de *log*, implementado en la biblioteca de Log4J (Log for Java) y accedido a través de una fachada con SL-F4J (Simple Logging Façade for Java). Para ello en la carpeta `lib` se deben añadir los ficheros `jar` indicados a continuación y disponibles en UBUVirtual:

- `log4j-1.2.17.jar`
- `slf4j-api-1.7.1.jar`
- `slf4j-log4j12-1.7.1.jar`

Para que el mecanismo de *log* funcione adecuadamente, se utiliza un fichero de configuración llamado `log4j.properties` que debe estar en el directorio raíz desde donde se ejecuta la aplicación (si se coloca en el directorio `./src` de nuestro proyecto, Eclipse realiza una copia al compilar en el directorio `./bin`).

A continuación se muestra un fichero de ejemplo con todas las salidas a pantalla desactivadas:

```
log4j.rootLogger = OFF, Console
log4j.appender.Console=org.apache.log4j.ConsoleAppender
log4j.appender.Console.layout=org.apache.log4j.PatternLayout
log4j.appender.Console.layout.conversionPattern=%p %c      %M     %m %n
```

El uso de estas bibliotecas queda fuera del ámbito de la asignatura (**NO es materia de evaluación**) y se incluye sólo para facilitar la depuración de errores por parte de los profesores. **Pero es obligatorio incluirlas en el `CLASSPATH` cuando se ejecuta la aplicación en modo gráfico.**

Para más información adicional, consultar la documentación en línea (Log4J - http://logging.apache.org/log4j/1.2/ y SLF4J - http://www.slf4j.org /)

4. Normas de entrega

Fecha límite de entrega

- Ver fecha indicada en UBUVirtual de la tarea **[MP] Entrega de Práctica Obligatoria 2 - EPO2.**

Formato de entrega

- Se enviará un fichero `.zip` o `.tar.gz` a través de la plataforma **UBUVirtual** completando la tarea **[MP] Entrega de Práctica Obligatoria 2 - EPO2.**
- Los ficheros fuente estarán codificados **OBLIGATORIAMENTE** en formato **UTF-8** (Tip: comprobar en Eclipse, en *File/Properties* del proyecto que el valor *Text file encoding* está configurado a dicho valor).
- **TODOS** los ficheros fuente `.java` deben incluir los **nombres y apellidos de los autores** en su cabecera y deben estar **correctamente indentados.**
- El fichero comprimido seguirá el siguiente formato de nombre **sin utilizar tildes**:
  - **`Nombre PrimerApellido-Nombre PrimerApellido.zip` o**
  - **`Nombre PrimerApellido-Nombre PrimerApellido.tar.gz`**
- Ej: si los alumnos son David Caubilla y Raúl Marticorena, su fichero `.zip` se llamará **`David Caubilla-Raul Marticorena.zip`.**
- **Se puede realizar la práctica individualmente o por parejas. Se calificará con los mismos criterios en ambos casos. Si se hace por parejas, la nota de la práctica es la misma para ambos miembros.**
- **Aunque la práctica se haga por parejas, se enviará individualmente por parte de cada uno de los dos integrantes a través de UBUVirtual. Verificar que ambas entregas son iguales en contenido. En caso de NO coincidencia, se penalizará un 20% a ambos.**
- **NO se admiten envíos posteriores a la fecha y hora límite, ni a través de otro medio que no sea la entrega de la tarea en UBUVirtual. Si no se respetan las anteriores normas de envío la calificación es directamente cero.**
- **Cualquier situación de plagio detectado en las prácticas, conlleva la aplicación del reglamento de exámenes.**

Se debe crear la siguiente estructura de directorios y ficheros en el **disco y entregar un único fichero con dicha estructura comprimida. Es obligatorio entregar un único fichero:**

- `/leeme.txt`: fichero de texto, que contendrá los nombres y apellidos de los integrantes y las aclaraciones que los alumnos crean oportunas poner en conocimiento de los profesores.

- 🗀 `lib`: contiene las siguientes bibliotecas (descargar desde UBUVirtual)
    - `doccheck.jar` para poder generar el chequeo de la documentación
    - `atari-go-gui-lib-2.0.jar`*: biblioteca con la interfaz gráfica proporcionada para poder ejecutar la aplicación en modo gráfico.
    - `log4j-1.2.17.jar` : biblioteca de terceros para activar el mecanismo de log (más información en http://logging.apache.org/log4j/1.2/).
    - `slf4j-api-1.7.1.jar` : biblioteca de tercero para activar la fachada de log (más información en http://www.slf4j.org/).
    - `slf4j-log4j12-1.7.1.jar` : biblioteca de terceros para activar la fachada de log (más información en http://www.slf4j.org/).
- 🗀 `src`: ficheros fuentes (`.java`) y ficheros de datos necesarios para poder compilar el producto completo.
- 🗀 `bin`: ficheros binarios (`.class`) y ficheros de datos necesarios.
- 🗀 `doc`: documentación HTML generada con **`javadoc`** de todos los ficheros fuentes.
- 🗀 `doccheck`: documentación HTML generada con el doclet **`DocCheck`** a partir de todos los ficheros fuentes.
- 🗀 `rsc`: fichero `log4j.properties` (descargar desde UBUVirtual) y en el caso de que se haya utilizado para la generación de la documentación el fichero `package-list`.
- `/compilar.bat o /compilar.sh`: fichero de comandos con la invocación al compilador `javac` para generar el contenido de la carpeta `bin` a partir de los ficheros fuente en la carpeta `src`.
- `/documentar.bat o /documentar.sh`: fichero de comandos con la invocación al generador de documentación `javadoc` para generar el contenido del directorio `doc` a partir de los ficheros fuente en la carpeta `src`.
- `/chequear.bat o /chequear.sh`: fichero de comandos con la invocación al chequeo de documentación para generar el contenido del directorio `doccheck` a partir de los ficheros fuente en la carpeta `src` y del fichero `doccheck.jar` en `lib`.
- `/ejecutar_textui.bat o /ejecutar_textui.sh`: fichero de comandos con la invocación a la máquina virtual java para ejecutar la clase raíz del sistema en modo texto con cualquiera de las configuraciones descritas en el presente enunciado, utilizando las clases en el directorio `bin` y las necesarias en el directorio `lib`.
- `/ejecutar_gui.bat o /ejecutar_gui.sh`: fichero de comandos con la invocación a la máquina virtual `java` para ejecutar la clase raíz del sistema en modo gráfico, con cualquiera de las configuraciones descritas en el presente enunciado utilizando las clases en el directorio `bin` y las necesarias en el directorio `lib`.

Los *scripts* se pueden entregar bien para Windows (`.bat`) o bien para GNU/Linux o Mac OS X (`.sh`), pero se debe elegir sólo una plataforma. Nota: tener en cuenta que en algunos *scripts* es necesario crear la carpeta de destino al principio. **Probar previamente siempre antes de la entrega el correcto despliegue del producto entregado.**

Comentarios adicionales

- **No se deben modificar los ficheros binarios proporcionados.** En caso de ser necesario, por errores en el diseño/implementación de los mismos, se notificará para que sea el responsable de la asignatura quien publique en UBUVirtual la corrección y/o modificación. Se modificará el número de versión del fichero en correspondencia con la fecha de modificación y se publicará un listado de erratas.
- Se realizará un cuestionario *online* en el laboratorio para probar la autoría de la misma. El peso es de 5% sobre la nota final con nota de corte 4 sobre 10 para superar la asignatura.

* Si se publica alguna corrección, el fichero cambiará en número de versión

Criterios de valoración

- La práctica es obligatoria, entendiendo que su no presentación en la fecha marcada, supone una calificación de cero sobre el total de 10 que se valora la práctica. La nota de corte en esta práctica es de 5 sobre 10 para poder superar la asignatura. Se recuerda que la práctica tiene un peso del 15% de la nota final de la asignatura.
- Se valorará negativamente métodos con un número de líneas grande (>20 líneas) sin contar comentarios ni líneas en blanco, ni llaves de apertura o cierre, indentando el código con el formateo por defecto de Eclipse. En tales casos, se debe dividir el método en métodos privados más pequeños. Siempre se debe evitar en la medida de lo posible la repetición de código.
- No se admiten ficheros cuya estructura y contenido no se adapte a lo indicado, con una valoración de cero.
- No se corrigen prácticas que no compilen, ni que contengan errores graves en ejecución con una valoración de cero.
- No se admite código no comentado con la especificación para **javadoc** con una valoración de cero.
- No se admite no entregar la documentación **HTML** generada con **javadoc** con el doclet por defecto y con el doclet **DocCheck**, con una valoración de cero.
- Se valorará negativamente el porcentaje de fallos como resultado de filtrar la documentación con el doclet **DocCheck**.
    - **NO se valorarán negativamente** los errores de tipo *Category 1: Package Error * No documentation for package. Need a package.html file*, los errores de tipo *Category 4: Text/link Error * Html Error in First Sentence --Missing a period -- {@inheritDoc}*, así como los errores en el tipo enumerado de los métodos generados automáticamente
- Es obligatorio seguir las convenciones de nombres vistas en teoría en cuanto a los nombres de paquetes, clases, atributos y métodos.
- Porcentajes/pesos[2] aproximados en la valoración de la práctica:

Apartado	Cuestiones a valorar	Porcentaje (Peso)
Cuestiones generales de funcionamiento	Integrada la interfaz gráfica (10%). Documentada sin errores DocCheck (7%). Scripts correctos (2%). Utiliza package-info. Extras de documentación (1%)	20,00%
AtariGo (textui)	Corrección del algoritmo propuesto. Uso de excepciones correcto.	5,00%
Jerarquía de herencia	Uso de modificadores de acceso. Atributos correctos. Código no repetido en las implementaciones. Código no excesivamente largo en los métodos. Gestión de turno correcta. Correcta validación de movimiento legal con detección de suicidios Jugar correcto con algoritmo adecuado. Uso de constantes simbólicas. Consulta ganador bien. estaAcabado correcto. Interfaz correcta. Clases abstracta con atributos y métodos adecuados. Sin ocultación de atributos. Clases descendientes con redefiniciones adecuadas. Uso de excecpciones correcto.	25,00%
Grupo	Atributos correctos. Uso de modificadores de acceso. Gestiona correctamente el grupo. Código no repetido en las implementaciones. Código no excesivamente largo en los métodos. Correcta copia de grupos. Uso de interfaces y clases genéricas adecuado. Uso correcto de excepciones.	20,00%
Tablero	Uso de modificadores de acceso. Atributos correctos. Código no repetido en las implementaciones. Código no excesivamente largo en los métodos. colocar correcto. Métodos correctos. Uso de constantes simbólicas. Correcta copia de tablero. Uso de interfaces y clases genéricas adecuado. Uso correcto de excepciones.	20,00%
Resto de clases del modelo	Correcta implementación de piedra. Correcta implementación de celda. Correcta implementación de color. Correcta implementación de jugador.	10,00%

Recursos

2 Los porcentajes pueden varias ligeramente al haber redondeado decimales.

Bibliografía complementaria:

[Oracle, 2017] Lesson: Language Basics. The Java Tutorials. (2017). Disponible en http://docs.oracle.com/javase/tutorial/java/.

Anexo 1. Recomendaciones en el uso de la interfaz `List` y la clase `java.util.ArrayList`

La interfaz `java.util.List` es implementada por la clase `ArrayList`. Dicha clase implementa un *array* que **dinámicamente** puede cambiar sus elementos, aumentando o disminuyendo su tamaño. Recordad que en contraposición, los *arrays* en Java son estáticos.

En esta segunda práctica utilizaremos la versión **GENÉRICA tanto de la interfaz como de la clase concreta.**

Una vez instanciado un `ArrayList` (con cualquiera de los constructores que aporta), tendremos una lista vacía. Cuando se añaden elementos se pueden utilizar los métodos:

`public boolean` `add(E e)`

Añade el elemento al final de la lista.

O bien:

`public void` `add(int index, E element)`

Inserta el elemento en la posición especificada. Desplaza el elemento en esa posición (si hay alguno) y todos los demás elementos a su derecha (añade 1 a sus índices)

Lanza una excepción IndexOutOfBoundsException – si el índice está fuera del rango (`index < 0 || index > size()`)

Mientras que el primer método añade siempre al final incrementando a su vez el tamaño (`size()`), el segundo permite añadir de forma indexada siempre que el índice **NO** esté fuera del rango (`index < 0 || index > size()`).

Si se quiere inicializar dimensionando de manera adecuada el número de elementos y se desconoce el valor a colocar en ese momento, se permite la inicialización con valores nulos (`null`) como se muestra en el ejemplo.

Ej:

```
// En este ejemplo se utiliza una variable de tipo interfaz para manejar el ArrayList.
// Siempre que sea posible se deben utilizar interfaces para manejar objetos concretos
List<Integer> array = new ArrayList<Integer>(10); // capacidad 10 pero tamaño 0
System.out.println("Tamaño del array list:" + array.size()); // se muestra 0 en pantalla

// cualquier intento de realizar una invocación a add(index,element) con index != 0
// provocaría una excepción de tipo IndexOutOfBoundsException
for (int i = 0; i < 10; i++){
     array.add(null);
}
// tamaño de array (size()) es 10 a la finalización del bucle
```

Para modificar una posición determinada se utiliza el método:

`public E` `set(int index,` **`E`** `element)`

Lanza una excepción IndexOutOfBoundsException – si el índice está fuera del rango (`index < 0 || index >= size()`)

Para consultar el elemento en una posición determinada se utiliza el método:

```
public E get(int index)
```

Lanza una excepción IndexOutOfBoundsException – si el índice está fuera del rango (`index < 0 || index >= size()`)

Para añadir todos los elementos de otro `ArrayList`, al final del `ArrayList` actual se puede utilizar el método:

```
public boolean addAll(Collection<? Extends E> c)
```

teniendo en cuenta que un `ArrayList` es una `Collection`.

Para extraer eliminando además el elemento del `ArrayList` se puede utilizar el método:

```
public E remove(int index)
```

Si queremos recorrer todos los elementos de un `List` con genericidad se puede utilizar un bucle `for-each`. Por ejemplo parar recorrer una lista de celdas y mostrar sus valores de fila y columna en pantalla:

```
public void mostrarCeldas(List<Celda> celdas) {
	for (Celda celda : celdas) {
		System.out.println(celda.obtenerFila() + "-" + celda.obtenerColumna());
	}
}
```

Para utilizar el resto de métodos (vaciar, consultar el número de elementos, etc.) se recomienda consultar la documentación en línea, aunque el conjunto de métodos indicados debería ser suficiente para la resolución de la práctica.

Licencia

Autor: Raúl Marticorena & David Caubilla

Área de Lenguajes y Sistemas Informáticos

Departamento de Ingeniería Civil

Escuela Politécnica Superior

UNIVERSIDAD DE BURGOS

2017