Entorno de resolución de problemas de ajedrez

Mayra Pastor Valdivia mayra.pastor@est.fib.upc.edu
Roger Guasch Ibarra roger.guasch.ibarra@est.fib.upc.edu

Versión entrega 1.0

Grup 9.5

Índice

1. Diagrama de casos de uso **3**
1.1 Gestión de usuarios 3
1.2 Gestión de problemas 4
1.3 Gestión Juego 6
1.4 Gestión de estadísticas 6

2. Diagrama del modelo conceptual **7**
2.1 Clase Controlador de dominio 7
2.2 Clase Jugador 7
2.2.1 Clase Persona 7
2.2.2 Clase Maquina 7
2.3 Clase Estrategia Simple 8
2.4 Clase Evaluacion 8
2.5 Clase Movimiento 8
2.6 Clase MovimientoPrueba 8
2.7 Clase Tablero 8
2.8 Clase Pieza 9
2.8.1 Clase Alfil 9
2.8.2 Clase Caballo 9
2.8.3 Clase Peon 9
2.8.4 Clase Reina 9
2.8.5 Clase Rey 9
2.8.6 Clase Torre 10
2.9 Clase Problema 10

3. Descripción de las funciones de cada clase **11**
3.1 Clase Controlador de Dominio **11**
3.2 Clase Pieza **12**
3.2.1 Clase Alfil 12
3.2.2 Clase Caballo 12
3.2.3 Clase Peón 13
3.2.4 Clase Reina 13
3.2.5 Clase Rey 13
3.2.6 Clase Torre 13
3.3 Clase Problema **14**
3.4 Clase Jugador **15**
3.5 Clase Persona 15
3.6 Clase Maquina 16

3.7 Clase Movimiento **16**
3.8 Clase Movimiento Prueba **16**
3.9 Clase Estrategia Simple **16**
3.10 Clase Evaluacion **17**
3.11 Clase Tablero **17**
3.12 Clase Controlador de Dominio **18**
3.13 Clase Pieza **19**
3.13.1 Clase Alfil 20
3.13.2 Clase Caballo 20
3.13.3 Clase Peón 20
3.13.4 Clase Reina 21
3.13.5 Clase Rey 21
3.13.6 Clase Torre 21
3.14 Clase Problema **21**
3.15 Clase Jugador **22**
3.15.1 Clase Persona 23
3.15.2 Clase Maquina 23
3.16 Clase Movimiento **23**
3.17 Clase Movimiento Prueba **24**
3.18 Clase Estrategia Simple **24**
3.19 Clase Evaluacion **25**
3.20 Clase Tablero **25**

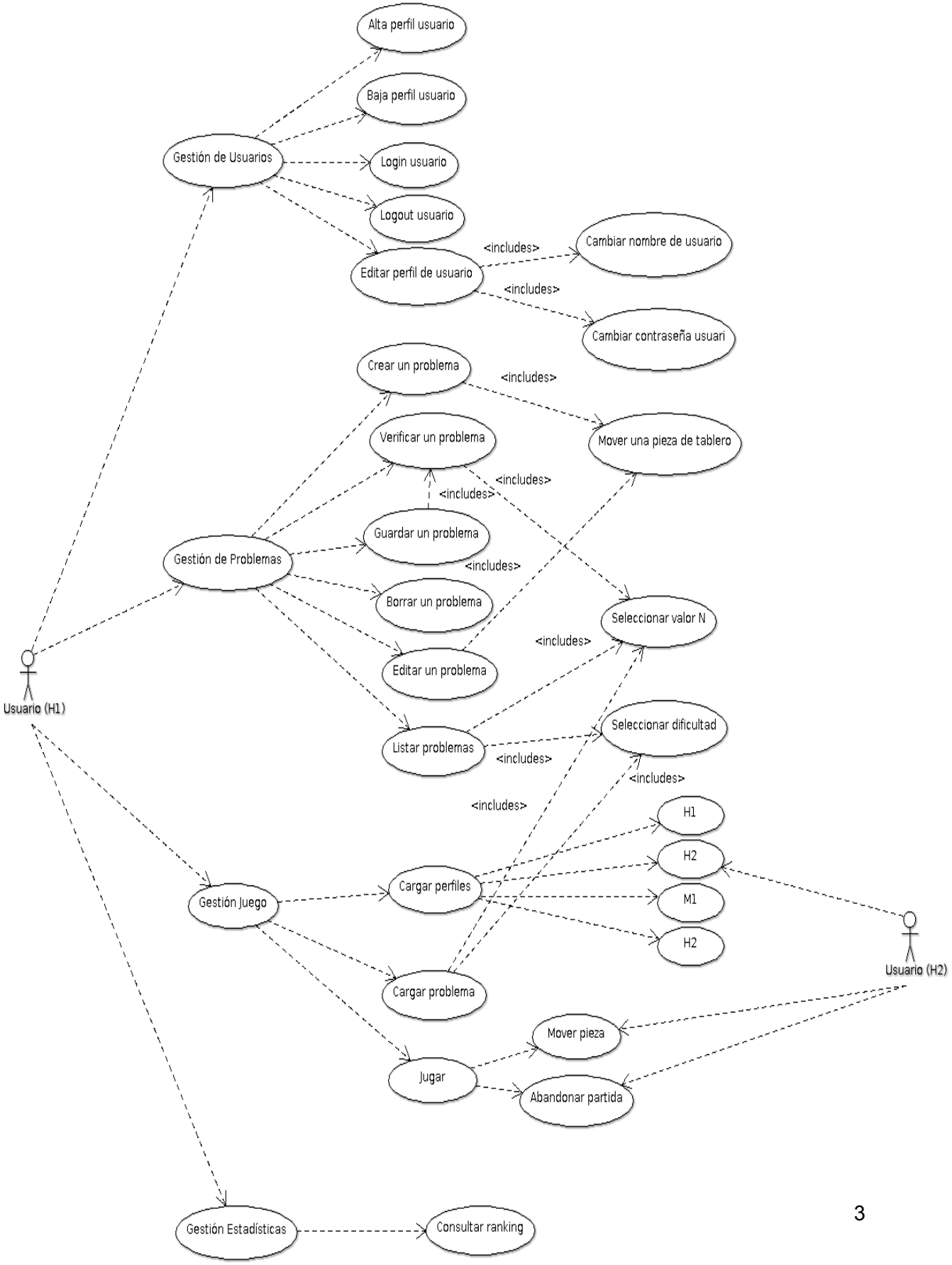
4. Relación de clases implementadas por cada miembro del grupo **27**

5. Estructuras de datos y algoritmos **27**

6. Tests **28**
Clase Pieza 28
Clase Problema 29
Clase Movimiento 30
Clase Movimiento Prueba 30
Clase Tablero 30
Clase Estrategia Simple 31
Clase Evaluación 31
Clase Jugador 31
Clase Controlador de Dominio 31

1. Diagrama de casos de uso

1.1 Gestión de usuarios



Caso de uso UC001: Alta perfil de usuario.
El usuario del sistema introduce un nombre de usuario y una contraseña para darse de alta en el sistema. Se crea un nuevo usuario con el nombre y la contraseña introducida. En caso que el nombre de usuario ya existiera en el sistema, el sistema muestra un mensaje de error al usuario.

Caso de uso UC002: Baja perfil de usuario.
El usuario se da de baja del sistema. Introduce su nombre de usuario y contraseña como confirmación. El sistema borra todos los datos relacionados con el usuario.

Caso de uso UC003: Login usuario.
El usuario introduce su nombre de usuario y su contraseña para acceder al sistema. Si el usuario no estaba registrado previamente se le muestra un mensaje de error, informándolo que no está presente en el sistema. Si existe el nombre de usuario pero la contraseña no es correcta, el sistema muestra un mensaje de error al usuario, informándolo que la contraseña introducida no es la correcta.

Caso de uso UC004: Logout usuario.
El usuario selecciona salir del sistema. El sistema cierra la sesión actual del usuario.

Caso de uso UC005: Modificar nombre de usuario.
El usuario introduce el nuevo nombre de usuario dos veces y su contraseña. Si el nuevo nombre de usuario doblemente introducido coincide, no existe en el sistema otro usuario con el mismo nombre y la contraseña es correcta, el usuario pasa a tener el nuevo nombre.

Caso de uso UC006: Modificar contraseña.
El usuario introduce la contraseña actual y, seguidamente, introduce dos veces la nueva contraseña. Si la contraseña actual es la correcta y la nueva contraseña, introducida dos veces, coincide, el usuario pasa a usar la nueva contraseña.

1.2 Gestión de problemas

Caso de uso UC007: Crear un problema.
El usuario crea un nuevo problema.

Caso de uso UC008: Introducir valor N.
El usuario introduce en el sistema el valor de N. Esto es: el número de movimientos máximos que puede hacer las piezas atacantes para conseguir el mate.

Caso de uso UC009: Introducir valor k.
En el caso de que los dos jugadores que jueguen la partida sean M1 y M2, el usuario podrá introducir sobre cuántos problemas quiere que se evalúen los algoritmos de M1 y M2.

Caso de uso UC0010: Verificar un problema.
El usuario selecciona verificar el problema que él ha creado. Si el sistema considera que el problema tiene solución se informará al usuario que dicho problema se puede solucionar en, como mínimo, N o menos movimientos que previamente ha introducido. En caso contrario, el sistema informa que el problema no tiene solución.

Caso de uso UC011: Guardar un problema.
El usuario guarda un problema que él ha creado. Si el problema ha sido verificado el sistema guarda el problema para que este pueda ser jugado. Si el problema no ha sido verificado el sistema también guarda el problema pero no lo mostrará como un candidato para ser jugado durante la selección de problema en el caso de uso de Jugar un problema.

Caso de uso UC012: Borrar un problema.
El usuario selecciona, de una lista de problemas que él ha creado, el problema que desea borrar y lo borra del sistema. Si el problema no había sido previamente jugado por nadie el sistema borra dicho problema. En caso contrario, el sistema informa al usuario que el problema seleccionado no puede ser borrado.

Caso de uso UC013: Editar un problema.
El usuario selecciona, de una lista de problemas donde o bien él es el creador o de unas plantillas predefinidas en el sistema, el problema que quiere editar. Si el problema no ha sido jugado previamente, el usuario puede editar la composición de las piezas en el tablero de dicho problema. En caso contrario, el sistema le informa que no se puede editar el problema seleccionado.

Caso de uso UC014: Mover pieza en edición del tablero.
El usuario coloca una pieza en el tablero en una posición donde no exista ninguna otra pieza. En caso contrario, el sistema muestra un mensaje de error.

Caso de uso UC015: Borrar pieza en edición del tablero.
El usuario borra una pieza en el tablero en una posición donde existía una pieza.

Caso de uso UC016: Listar problemas.
El usuario introduce el tema del problema, esto es el número de N o número de jugadas para conseguir el mate, y/o introduce la dificultad. El sistema le muestra los problemas existentes, creados por el usuario y plantillas predefinidas, que cumplen con las condiciones detalladas. En caso que no exista ningún problema, el sistema le muestra un listado vacío. Si no introduce ningún elemento por el cual filtrar, el sistema entonces le mostrará todos los problemas que ha creado el usuario y las plantillas que no hayan sido jugadas.

Caso de uso UC017: Seleccionar dificultad.
El usuario introduce la dificultad de los problemas a mostrar por el sistema.

1.3 Gestión Juego

Caso de uso UC018: Cargar perfil Persona
El usuario selecciona cargar, como uno de los jugadores de la partida, al jugador H1 o H2.

Caso de uso UC019: Cargar perfil M1.
El usuario selecciona cargar, como uno de los jugadores de la partida, a la máquina M1.

Caso de uso UC020: Cargar perfil M2.
El usuario selecciona cargar, como uno de los jugadores de la partida, a la máquina M2.

Caso de uso UC021: Cargar problema a jugar.
El usuario selecciona, de una lista de problemas ya verificados en el sistema, que problema desea que se vaya a jugar. Si no introduce ni el tema del problema a jugar ni la dificultad, el sistema le mostrará todos los problemas presentes en el sistema. En caso contrario el sistema le muestra solo los problemas que cumplan con las condiciones que él ha establecido.

Caso de uso UC022: Mover pieza.
En caso que le toque jugar al usuario H1 o al usuario H2, dicho usuario podrá mover una pieza que pertenezca al color de su conjunto de piezas -blancas o negras-. El sistema no permitirá movimientos incorrectos ni infracciones de las normas del juego.

Caso de uso UC023: Abandonar partida.
En caso que le toque jugar al usuario H1 o al usuario H2, dicho usuario podrá elegir abandonar la partida. Si lo hace, automáticamente su oponente pasará a ser el ganador de la partida y el jugador que ha abandonado será el perdedor de la misma.

1.4 Gestión de estadísticas

Caso de uso UC024: Consultar ranking.
El usuario podrá consultar el ránking general de jugadores. El sistema le mostrará una lista de jugadores y sus puntuaciones, ordenada de mayor a menor puntuación.

2. Diagrama del modelo conceptual

Dado que el modelo conceptual es demasiado grande para ponerlo en este documento, se podrá visualizar en la imagen Diagramadeclase.png
A continuación detallaremos una breve descripción de cada clase.

2.1 Clase Controlador de dominio

La clase controlador de dominio contiene todas las funciones necesarias para ejecutar las funcionalidades principales de la primera entrega. Estas son: todas las funciones para gestionar una partida y todas las necesarias para poder validar un problema. Básicamente, la función principal del controlador es la de actuar como nexo entre las diferentes clases implementadas para que el programa funcione correctamente. Todos los posibles errores que puedan ocurrir durante la ejecución del código, así como también los resultados de las partidas (si un jugador está en jaque mate, si ha ganado, etc) se gestionan a través de excepciones que se propagan por las clases.

2.2 Clase Jugador

La clase Jugador es una clase abstracta que, como métodos más importantes, tiene aquellos que permiten a cada tipo de jugador, ya sea persona o máquina, hacer sus respectivos movimiento. Hemos decidido implementarlo así ya que pensamos que, de esta manera, y aprovechando las características de la herencia y la abstracción, nuevos jugadores podrían ser implementados sin mucho esfuerzo y pudiendo reutilizar código.

2.2.1 Clase Persona

Clase concreta que extiende la clase Jugador y implementa los métodos de la clase. Contiene los métodos necesarios para que, dado una persona humana (a partir de ahora, H1), pueda hacer los movimientos necesarios para mover una pieza de una posición a otra en el tablero. En caso que el movimiento introducido no sea correcto, la función gestionará los posibles errores usando excepciones de Java.

2.2.2 Clase Maquina

Clase concreta que extiende la clase Jugador y implementa los métodos necesarios para que la máquina (M1, a partir de ahora) pueda efectuar sus movimientos. Al igual que la clase Persona, Maquina también gestiona sus errores y lanza excepciones para informar al controlador.

2.3 Clase Estrategia Simple

La clase estrategia simple implementa el algoritmo minimax. Básicamente, y a grandes rasgos, el algoritmo se encarga de calcular el mejor movimiento para el jugador M1 teniendo en consideración que, el oponente, hará siempre el mejor movimiento. Así pues, la principal funcionalidad de la clase es devolver el movimiento que M1 deberá ejecutar y que, según nuestra heurística y los posibles movimientos del oponente, el movimiento retornado, es el mejor entre todos los movimientos posibles.

2.4 Clase Evaluacion

Clase que implementa la heurística, necesaria para ejecutar el algoritmo minimax. Para esta entrega, y teniendo en consideración que M1 implementa un algoritmo con heurística simple, solo consideramos que cada pieza tiene un valor determinado.

2.5 Clase Movimiento

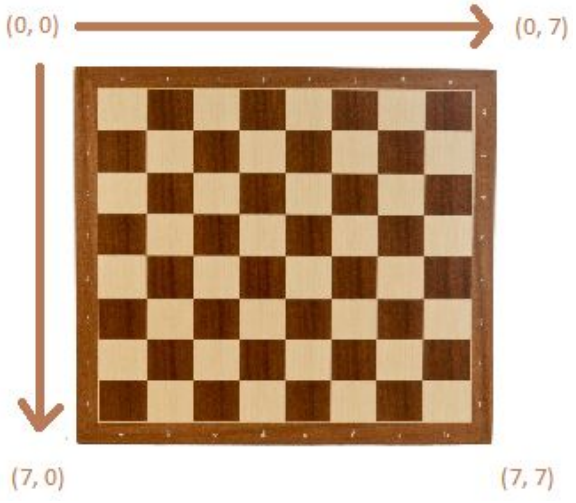
Clase que contiene los atributos necesarios para efectuar un movimiento y ejecuta una función para verificar si, efectivamente, ese movimiento es posible. Usamos este objeto para simplificar las cabeceras de las funciones, ya que contiene toda la información que necesitamos para efectuar los movimientos de las piezas en el tablero.

2.6 Clase MovimientoPrueba

La clase MovimientoPrueba, si tuviéramos que buscar un símil con alguna otra clase, sin duda esa sería la clase Pair de C++. MovimientoPrueba simplemente contiene dos tableros distintos: el primer tablero es el tablero en el cual se quiere hacer un movimiento, el segundo tablero es el mismo tablero que el anterior, pero con el movimiento ejecutado. La clase contiene también un objeto movimiento, que indica de forma explícita que movimiento se ha realizado entre los dos tableros que contiene la clase.
MovimientoPrueba se usa en el algoritmo del minimax.

2.7 Clase Tablero

Representamos un tablero como una matriz de caracteres de tamaño 8x8. Si una casilla no contiene ninguna pieza, en la posición vacía, la matriz contendrá el carácter ‘0’.Si la casilla contiene una pieza, entonces en esa posición habrá el char que identifica a una pieza en concreto, usando el formato FEN. El tablero, para nosotros, se representa de la siguiente forma:



Así pues, si quiero mover una pieza des del extremo superior izquierdo al extremo inferior izquierdo del tablero (típico movimiento de una Torre o una Reina) debería escribir, por terminal: 0 0 7 0.

El tablero también contiene una Array de Piezas Blancas y Piezas Negras y un boolean que indica de quién es el turno en cada caso. Esta clase tiene todas las funciones necesarias para mover las piezas por el tablero y para poblar dicho tablero acorde a un FEN concreto, previamente tratado.

2.8 Clase Pieza

Clase abstracta, contiene las funciones necesarias para verificar que un movimiento es correcto y para listar todos los posibles movimientos que una pieza podría hacer. Esta última función es usada por el minimax, para probar todos los posibles movimientos que la pieza puede hacer.

2.8.1 Clase Alfil

La clase Alfil extiende la clase Pieza y implementa los métodos abstractos de la clase padre. Verifica que el movimiento particular de la pieza Alfil, hecha por un jugador H1, se hace correctamente y se asegura de listar todos los posibles movimientos legales que el minimax podría hacer con la pieza.

2.8.2 Clase Caballo

La clase Caballo extiende la clase Pieza y implementa los métodos abstractos de la clase padre. Verifica que el movimiento particular de la pieza Caballo, hecha por un jugador H1, se hace correctamente y se asegura de listar todos los posibles movimientos legales que el minimax podría hacer con la pieza.

2.8.3 Clase Peon

La clase Peon extiende la clase Pieza y implementa los métodos abstractos de la clase padre. Verifica que el movimiento particular de la pieza Peon, hecha por un jugador H1, se hace correctamente y se asegura de listar todos los posibles movimientos legales que el minimax podría hacer con la pieza.

2.8.4 Clase Reina

La clase Reina extiende la clase Pieza y implementa los métodos abstractos de la clase padre. Verifica que el movimiento particular de la pieza Reina, hecha por un jugador H1, se hace correctamente y se asegura de listar todos los posibles movimientos legales que el minimax podría hacer con la pieza.

2.8.5 Clase Rey

La clase Rey extiende la clase Pieza y implementa los métodos abstractos de la clase padre. Verifica que el movimiento particular de la pieza Rey, hecha por un jugador H1, se

hace correctamente y se asegura de listar todos los posibles movimientos legales que el minimax podría hacer con la pieza.

2.8.6 Clase Torre

La clase Torre extiende la clase Pieza y implementa los métodos abstractos de la clase padre. Verifica que el movimiento particular de la pieza Torre, hecha por un jugador H1, se hace correctamente y se asegura de listar todos los posibles movimientos legales que el minimax podría hacer con la pieza.

2.9 Clase Problema

La clase Problema contiene el FEN, tratado previamente para que solo sea la situación de las piezas en el tablero, y un boolean que indica quien empieza el juego y otro para definir que el problema ha sido o no validado.
La funcionalidad principal de la clase problema es implementar el minimax, pero ligeramente modificado, para que quede o no validado el problema. Básicamente, buscamos una hoja en el árbol generado por el minimax que contenga un jaque mate hecho por el jugador que abre el juego.

3. Descripción de las funciones de cada clase

3.1 Clase Controlador de Dominio

El controlador de dominio implementa un patrón singleton. Dado que solo puede existir una y solo una instancia del objeto del controlador, se ha decidido implementar este patrón. Como atributos contiene:

- Dos objetos jugadores
- Un objeto problema
- Un objeto tablero
- Un objeto privado controlador de dominio, para implementar el patrón singleton

Getters y Setters de los atributos

public static ctrl_dominio getInstance()
Esta función es usada para crear una instancia del controlador. En caso que el atributo privado de controlador de dominio sea null, getInstance permite llamar la creadora privada del controlador. Si detecta que ya existe una instancia de controlador, no permite crear más objetos.

public static void seleccionarJugadores(int jug1, int jug2, boolean j1EsBlanco)
Función pensada para crear los dos jugadores que formarán parte de la partida de ajedrez. Sigue la lógica siguiente:

- El atributo j1EsBlanco indica, si es true, que el jugador 1, que es el que abre el juego, lleva las piezas blancas. En caso que j1EsBlanco sea false, indicará que el jugador 1, que abre el juego, llevará las piezas negras.
- El atributo jug1 y jug2 siguen la misma lógica:
    - Si jug1 tiene valor 1 entonces se crea un objeto Persona
    - Si jug1 tiene valor 2 entonces se crea un objeto Maquina

public static void jugar(int n) throws Exception
La función será llamada solo si le toca hacer un movimiento a una Máquina. La función jugar recibe, como parámetro, la profundidad máxima que el minimax puede llegar a explorar. La función se encarga de verificar que efectivamente ninguno de los dos jugadores se encuentre en jaque mate y, entonces, llama a la respectiva función de minimax, implementada en los jugadores de tipo Máquina. Si no se ha podido ejecutar el movimiento por alguna razón, la función se encarga de gestionar los distintos problemas usando excepciones personalizadas.

public static void jugar(int posX, int posY, int movX, int movY) throws Exception
Usamos *method overloading*, que permite el lenguaje Java, para implementar la función jugar con el mismo nombre pero distintos parámetros de entrada. Está función será llamada en caso que sea el jugador Humano el que deba hacer hacer un movimiento. La función verifica a quién le corresponde el movimiento, teniendo en cuenta quien debería tirar en

cada turno, y efectúa dicho movimiento. Si se produjera algún error, informaría de ello a través de las Excepciones.

public static void crearPartida(String FEN, int N, int jug1, int jug2)
Función que recibe, como parámetro, un string FEN, una N

3.2 Clase Pieza

La clase Pieza es una clase abstracta. Como atributos contiene:

- El color de la pieza
- La posición de la pieza dentro de tablero
- El tipo de pieza, identificado como un char y siguiendo la nomenclatura FEN
- El valor de la pieza en puntos

Getters y Setters de los atributos

abstract boolean esMovimientoOk(int movX, int movY, int estadoTablero[][])
La función recibe, como parámetros, la posición donde se desea mover la pieza y el estado del tablero, previo al movimiento de la pieza.
La función devuelve true si el movimiento, pasado por parámetro, se puede efectuar, teniendo en cuenta los distintos movimientos que puede hacer cada tipo de pieza. Si el movimiento es incorrecto, devuelve false.
abstract ArrayList<Movimiento> movimientosPosibles(char estadoTablero[][])
La función recibe, como parámetro, una matriz que representa el tablero, con las piezas contenidas en él en formato FEN.
La función devuelve una lista de todos los movimientos que puede hacer dicha pieza, teniendo en cuenta el estado actual del tablero.

3.2.1 Clase Alfil

Clase que extiende de Pieza y implementa sus métodos abstractos.
Para verificar el movimiento del Alfil hemos tenido en cuenta que se mueva estrictamente en diagonal, que pueda ir a una posición donde haya una pieza de distinto color, que no se pueda mover por la diagonal si hay otra pieza obstaculizando su camino y, por último, que el movimiento esté dentro de los límites del tablero.
Para listar los movimientos posibles de la pieza hemos tenido en cuenta lo mismo que para la verificación de los movimientos.

3.2.2 Clase Caballo

Clase que extiende de Pieza y implementa sus métodos abstractos.
Para verificar el movimiento del Caballo nos aseguramos que haga uno de los ocho posibles movimientos que puede hacer la pieza, siempre considerando que dicho movimiento no mueva la pieza fuera de los límites del tablero o en otra posición ocupada por una pieza del mismo color.
Aplicamos el mismo criterio para devolver todos los posibles movimientos de la pieza.

3.2.3 Clase Peón

Clase que extiende de Pieza y implementa sus métodos abstractos.
Para verificar el movimiento del Peón nos aseguramos que pueda moverse dos posiciones, en caso que se encuentre en la posición inicial del tablero, y limitamos a un movimiento en horizontal en el caso que no se encontrara en su posición inicial cuando se creó el tablero. También verificamos que, para matar otra pieza enemiga, sea en diagonal y que no pueda seguir avanzando si se encuentra con otra pieza por el camino. Como en las otras piezas, nos aseguramos que no pueda ser colocada fuera de los límites del tablero.
Aplicamos el mismo criterio para devolver todos los posibles movimientos de la pieza.

3.2.4 Clase Reina

Clase que extiende de Pieza y implementa sus métodos abstractos.
Contiene dos métodos privados, exclusivos de este objeto. La función esMovimientoOk llama a los métodos privados movimientoHorizontalVerticalOK, en caso que detecte que se está realizando un movimiento horizontal o vertical, o a la función movimientoDiagonalOK, en caso que detecte que se está realizando un movimiento en diagonal. Se verifica lo mismo que se verifican en las piezas Alfil y Torre ya que, la Reina, es la unión de los dos tipos de movimientos.
Para listar los movimientos posibles, también lo dividimos en dos funciones: movimientosPosiblesDiagonales y movimientosPosiblesHorVert.

3.2.5 Clase Rey

Clase que extiende de Pieza y implementa sus métodos abstractos.
Tenemos en cuenta de no validar un movimiento que haga que la pieza salga del tablero, que se sitúe en una posición donde ya hay otra pieza del mismo color y, en definitiva, que se pueda mover en todas las direcciones pero solo una casilla de distancia.
Aplicamos el mismo criterio para devolver todos los posibles movimientos de la pieza.

3.2.6 Clase Torre

Clase que extiende de Pieza y implementa sus métodos abstractos.
Verificamos que no se valide ningún movimiento ilegal: consideramos movimientos ilegales todo aquel movimiento que sitúe la pieza fuera de los límites del tablero o ponga la pieza en una casilla ocupada por otra pieza del mismo color. También verificamos que los movimientos sean estrictamente horizontales o verticales y que no se pase por encima de otra pieza durante la trayectoria por el tablero hacia la posición de destino.
Aplicamos el mismo criterio para devolver todos los posibles movimientos de la pieza.

3.3 Clase Problema

Como atributos, la clase tiene:

- Un String, llamado problema, que contiene, solamente, la situación de las piezas en el tablero
- Un integer, que indica el valor de N
- un Boolean que indica si el juego es iniciado por las blancas o no
- Un objeto evaluación, para poder validar un problema
- Un boolean que indica si el problema ha sido o no verificado

Getters y Setters de los atributos

public void setFEN(String FEN)
La función recibe, como parámetro, un FEN completo (por ejemplo: 1k6/4n3/2r5/8/8/8/4N3/2K5 w - - 0 1). La función elimina los últimos elementos del FEN y se queda solo con la situación del tablero para asignarlo en el atributo problema, de la clase (en nuestro caso asignaría al atributo problema: 1k6/4n3/2r5/8/8/8/4N3/2K5).
public boolean verificarProblema(String FEN)
Como precondición, los atributos de la clase problema (N, y el String Problema) deben estar asignados. El objetivo principal de la función es preparar todos los parámetros que necesita el minimax para poder ejecutarse.

private void verificar(Tablero tablero, boolean maxAttackingPlayer)
Verificar es la función del minimax. La función ejecuta el algoritmo de minimax però, en esta ocasión, no busca el mejor movimiento sino que, simplemente, busca una hoja donde las piezas que abren el juego hagan mate. Si la encuentra, el atributo de la clase verificado se pone a true, en caso contrario el problema no está verificado.

private int min(Tablero tablero, int depth)
Función auxiliar para el minimax. Como parámetros recibe un tablero donde se ha ejecutado un movimiento y un nivel de profundidad que debe explorar. La función retorna un valor que será el mínimo de todos los movimientos posibles, después de que el movimiento inicial se ejecutara.

private int min(Tablero tablero, int depth)
Función auxiliar para el minimax. Como parámetros recibe un tablero donde se ha ejecutado un movimiento y un nivel de profundidad que debe explorar. La función retorna un valor que será el máximo de todos los movimientos posibles, después de que el movimiento inicial se ejecutara.

private static boolean gameOver(final Tablero tablero)
La función recibe, como parámetro, un tablero. En función de este tablero, devuelve true si el Jugador está en jaque mate o si no le he es posible hacer ningún movimiento.

3.4 Clase Jugador

La clase Jugador es una clase abstracta. Como atributos contiene:

- Booleans que indican si el Jugador es una Máquina, está en Mate, tiene piezas negras, y si está Atacando.
- Una pieza de su Rey
- Unos arrays donde guarda sus piezas del tablero y otra donde están todos los posibles movimientos de sus piezas.
- Una copia del tablero en el que se está jugando en diferentes puntos del juego.

private boolean tieneEscape()
Cuando la función es llamada, esta crea una nueva instancia de tablero y revisa que el parámetro implícito tenga algún movimiento que se pueda hacer sin quedar expuesto a una ataque a su Rey. La función devuelve un boolean que es true de encontrar al menos un movimiento posible, y false de no encontrarlo.

public static ArrayList<Movimiento> hayAtaquesPendientes(final int theX, final int theY, final ArrayList <Movimiento> movimientos)
La función recibe como parámetros la ubicación del Rey y una lista de movimientos posibles del oponente. Devuelve un array de movimientos que podría hacer el oponente al Rey del parámetro implícito.

public MovimientoPrueba hacerMovimiento(final Tablero tablero, final Movimiento movimiento)
La función recibe el estado del tablero y un movimiento a probar dentro de este. La función devuelve una instancia de Movimiento Prueba en la cual guardamos el tablero de donde se vino, el tablero a donde se va, el movimiento que ha sido probado y un boolean que nos indica si dejaría al Rey en una posición de Mate (false) o no (true).

public abstract Tablero jugar(Tablero tablero, Movimiento movimiento)
La función recibe el estado del tablero y un movimiento a probar que ha hecho un jugador Persona. La función devuelve de tener éxito un Tablero modificado con el movimiento que ha mandado la Persona.

public abstract Tablero jugar(final Tablero tablero, final int N)
La función recibe el estado del tablero y un número de movimientos que será el nivel de profundidad a explorar por parte del jugador Maquina. La función devuelve de tener éxito un Tablero modificado con el movimiento que ha mandado la Máquina.

3.5 Clase Persona

Clase que extiende de Jugador y implementa sus métodos abstractos.

3.6 Clase Maquina

Clase que extiende de Jugador y implementa sus métodos abstractos.

Como atributo tiene una instancia de la clase Estrategia Simple. Desde este objeto el jugador Máquina puede acceder a su algoritmo de decisión para que le devuelva el mejor movimiento que puede hacer dependiendo de si tiene una posición de ataque o de defensa.

3.7 Clase Movimiento

- Integers que nos indican por un lado la posición de origen y por otro la posición de destino.
- Un char que nos indica que pieza se encuentra en la posición de destino de haber una pieza.

public Tablero intentar(final Tablero iniTablero)
La función recibe como parámetro un tablero, se crea un nuevo tablero de él para poder intentar ejecutar el parámetro implícito en este. La función devuelve dicha instancia nueva de tablero que ha sido modificada.

public boolean esNada()
La función evalúa si el parámetro implícito es una posición no existente. La función devuelve un boolean de true si es así y false si es un movimiento válido.

3.8 Clase Movimiento Prueba

- Dos tableros de origen y destino
- El movimiento que ha sido hecho
- Un boolean que nos indica si se ha podido hacer o no el movimiento

Esta clase solo tiene getters y setters. Objetos de esta clase son usados como una forma de probar movimientos dentro de la llamada recursiva de la clase de Estrategia simple y obtiene información del movimiento hecho para que en un futuro podamos expandir a otros tipos de estrategias o algoritmos.

3.9 Clase Estrategia Simple

- Un objeto Evaluación
- Un integer que nos dice la profundidad dada por el jugador

public Movimiento estrategiaSimple(final Tablero tablero)
La función recibe como parámetro el estado actual del tablero donde el siguiente movimiento le toca al Jugador que ha llamado la función. La función devuelve una instancia de Movimiento que será o el mejor movimiento que pueda hacer ese jugador o será un

movimiento nulo en caso de no haber ninguna opción viable ya que el jugador está o estaría en mate.

private int min(final Tablero tablero, final int depth)
La función recibe como parámetro el estado del tablero después de un movimiento hecho en este y el nivel de profundidad a explorar. La función devuelve un valor que será el menor valor de todos los movimientos posibles después de que el movimiento inicial ha sido ejecutado.

private int max(final Tablero tablero, final int depth)
La función recibe como parámetro el estado del tablero después de un movimiento hecho en este y el nivel de profundidad a explorar. La función devuelve un valor que será el mayor valor de todos los movimientos posibles después de que el movimiento inicial ha sido ejecutado.

private static boolean gameOver(final Tablero tablero)
La función recibe como parámetro el estado del tablero y evalúa si el jugador está en jaque mate o está estancado. La función devuelve un boolean que será true si el jugador se encuentra en cualquiera de estos estados, y false si no se encuentra en ninguno de estos estados.

3.10 Clase Evaluacion

public int evaluar(Tablero tablero, int depth)
La función recibe como parámetro el estado del tablero después de un movimiento hecho en este y el nivel de profundidad a explorar. La función devuelve un valor que será la diferencia de las puntuaciones evaluadas del jugador que ataca y la del jugador que se defiende.

private static int puntuacion(Jugador jugador, int depth)
La función recibe por parámetro un jugador a evaluar y la profundidad en la que estamos dentro del árbol de movimientos. La función devuelve la suma de las diferentes evaluaciones a las que se llama para el jugador del parámetro.

private static int evaluarPuntosPiezas(Jugador jugador)
La función recibe como parámetro un jugador. La función devuelve un integer que es la suma de los valores individuales de cada pieza que el jugador todavía tiene en el tablero en su estado actual.

private static int elOtroJugadorEnJaque(Jugador jugador)
La función recibe como parámetro un jugador. La función devuelve si el oponente del jugador pasado por parámetro se encuentra en mate un bonus, sino devuelve un 0.

3.11 Clase Tablero

- Un 2D array/matriz de chars donde guardamos el tipo de pieza que hay o un valor de ‘0’ de no haber ninguna.
- Un array de piezas blancas y otro de piezas negras.
- Un booleano que lleva de quien (piezas blancas o negras) es el turno.
- Dos objetos de Jugador.

public void FENToTablero(String FEN, boolean turnoBlancas)
La función recibe como parámetro un string donde se encuentra el problema a resolver y cuál de los dos colores de piezas comienza primero. La función no devuelve nada pero modifica el atributo de matriz y actualiza todos los otros atributos del parámetro implícito.

private Pieza getPiezadeChar(char piezasCode, boolean esNegra, int posX, int posY)
La función recibe un char que indica el tipo de pieza, un boolean que nos dice si es negra o blanca, y su posición en la matriz/tablero (posX,posY). La función devuelve un objeto Pieza de la subclase indicada por el char.

private ArrayList<Movimiento> todosLosMovimientosPosibles(final ArrayList<Pieza> piezasJugador)
La función recibe como parámetro un array de piezas de un jugador. La función devuelve en un array todos los posibles movimientos para cada una de las piezas del array del parámetro.

public void ejecutarMovimiento(final Movimiento movimiento)
La función recibe como parámetro un movimiento del jugador. La función no devuelve nada pero modifica el parámetro implícito y actualiza completamente, incluyendo a sus jugadores.

public boolean movimientoHumano(Movimiento movimiento)
La función recibe como parámetro un movimiento del jugador Persona. La función devuelve un boolean que indica si ha sido posible hacer el movimiento del parámetro nada pero modifica el parámetro implícito y actualiza completamente, incluyendo a sus jugadores.

public Jugador getAttackPlayer(boolean estaAtacando)
La función recibe como parámetro un boolean que nos indica si queremos el jugador que está atacado (true) o no (false). La función devuelve el jugador del parámetro implícito que está atacando o no dependiendo del parámetro pasado.

public Jugador esSuTurno()
La función devuelve el jugador del parámetro implícito del cual es su turno de hacer un movimiento.

public Jugador miOponenteEs(final Jugador jugador)
La función recibe una instancia de jugador. La función devuelve el oponente del jugador del parámetro.

3.12 Clase Controlador de Dominio

El controlador de dominio implementa un patrón singleton. Dado que solo puede existir una y solo una instancia del objeto del controlador, se ha decidido implementar este patrón.
Como atributos contiene:

- Dos objetos jugadores
- Un objeto problema
- Un objeto tablero
- Un objeto privado controlador de dominio, para implementar el patrón singleton

Getters y Setters de los atributos
public static ctrl_dominio getInstance()
Esta función es usada para crear una instancia del controlador. En caso que el atributo privado de controlador de dominio sea null, getInstance permite llamar la creadora privada del controlador. Si detecta que ya existe una instancia de controlador, no permite crear más objetos.

public static void seleccionarJugadores(int jug1, int jug2, boolean j1EsBlanco)
Función pensada para crear los dos jugadores que formarán parte de la partida de ajedrez. Sigue la lógica siguiente:

- El atributo j1EsBlanco indica, si es true, que el jugador 1, que es el que abre el juego, lleva las piezas blancas. En caso que j1EsBlanco sea false, indicará que el jugador 1, que abre el juego, llevará las piezas negras.
- El atributo jug1 y jug2 siguen la misma lógica:
    - Si jug1 tiene valor 1 entonces se crea un objeto Persona
    - Si jug1 tiene valor 2 entonces se crea un objeto Maquina

public static void jugar(int n) throws Exception
La función será llamada solo si le toca hacer un movimiento a una Máquina. La función jugar recibe, como parámetro, la profundidad máxima que el minimax puede llegar a explorar. La función se encarga de verificar que efectivamente ninguno de los dos jugadores se encuentre en jaque mate y, entonces, llama a la respectiva función de minimax, implementada en los jugadores de tipo Máquina. Si no se ha podido ejecutar el movimiento por alguna razón, la función se encarga de gestionar los distintos problemas usando excepciones personalizadas.

public static void jugar(int posX, int posY, int movX, int movY) throws Exception
Usamos *method overloading*, que permite el lenguaje Java, para implementar la función jugar con el mismo nombre pero distintos parámetros de entrada. Está función será llamada en caso que sea el jugador Humano el que deba hacer hacer un movimiento. La función verifica a quién le corresponde el movimiento, teniendo en cuenta quien debería tirar en cada turno, y efectúa dicho movimiento. Si se produjera algún error, informaría de ello a través de las Excepciones.

public static void crearPartida(String FEN, int N, int jug1, int jug2)

Función que recibe, como parámetro, un string FEN, una N

3.13 Clase Pieza

La clase Pieza es una clase abstracta. Como atributos contiene:

- El color de la pieza
- La posición de la pieza dentro de tablero
- El tipo de pieza, identificado como un char y siguiendo la nomenclatura FEN
- El valor de la pieza en puntos

Getters y Setters de los atributos

abstract boolean esMovimientoOk(int movX, int movY, int estadoTablero[][])
La función recibe, como parámetros, la posición donde se desea mover la pieza, el estado del tablero, previo al movimiento de la pieza, y una tabla de identificaciones internas de piezas que referencian a objetos Pieza.
La función devuelve true si el movimiento, pasado por parámetro, se puede efectuar, teniendo en cuenta los distintos movimientos que puede hacer cada tipo de pieza. Si el movimiento es incorrecto, devuelve false.

abstract ArrayList<Movimiento> movimientosPosibles(char estadoTablero[][])
La función recibe, como parámetro, una matriz que representa el tablero, con las piezas contenidas en él en formato FEN.
La función devuelve una lista de todos los movimientos que puede hacer dicha pieza, teniendo en cuenta el estado actual del tablero.

3.13.1 Clase Alfil

Clase que extiende de Pieza y implementa sus métodos abstractos.
Para verificar el movimiento del Alfil hemos tenido en cuenta que se mueva estrictamente en diagonal, que pueda ir a una posición donde haya una pieza de distinto color, que no se pueda mover por la diagonal si hay otra pieza obstaculizando su camino y, por último, que el movimiento esté dentro de los límites del tablero.
Para listar los movimientos posibles de la pieza hemos tenido en cuenta lo mismo que para la verificación de los movimientos.

3.13.2 Clase Caballo

Clase que extiende de Pieza y implementa sus métodos abstractos.
Para verificar el movimiento del Caballo nos aseguramos que haga uno de los ocho posibles movimientos que puede hacer la pieza, siempre considerando que dicho movimiento no mueva la pieza fuera de los límites del tablero o en otra posición ocupada por una pieza del mismo color.
Aplicamos el mismo criterio para devolver todos los posibles movimientos de la pieza.

3.13.3 Clase Peón

Clase que extiende de Pieza y implementa sus métodos abstractos.

Para verificar el movimiento del Peón nos aseguramos que pueda moverse dos posiciones, en caso que se encuentre en la posición inicial del tablero, y limitamos a un movimiento en horizontal en el caso que no se encontrara en su posición inicial cuando se creó el tablero. También verificamos que, para matar otra pieza enemiga, sea en diagonal y que no pueda seguir avanzando si se encuentra con otra pieza por el camino. Como en las otras piezas, nos aseguramos que no pueda ser colocada fuera de los límites del tablero.
Aplicamos el mismo criterio para devolver todos los posibles movimientos de la pieza.

3.13.4 Clase Reina

Clase que extiende de Pieza y implementa sus métodos abstractos.
Contiene dos métodos privados, exclusivos de este objeto. La función esMovimientoOk llama a los métodos privados movimientoHorizontalVerticalOK, en caso que detecte que se está realizando un movimiento horizontal o vertical, o a la función movimientoDiagonalOK, en caso que detecte que se está realizando un movimiento en diagonal. Se verifica lo mismo que se verifican en las piezas Alfil y Torre ya que, la Reina, es la unión de los dos tipos de movimientos.
Para listar los movimientos posibles, también lo dividimos en dos funciones: movimientosPosiblesDiagonales y movimientosPosiblesHorVert.

3.13.5 Clase Rey

Clase que extiende de Pieza y implementa sus métodos abstractos.
Tenemos en cuenta de no validar un movimiento que haga que la pieza salga del tablero, que se sitúe en una posición donde ya hay otra pieza del mismo color y, en definitiva, que se pueda mover en todas las direcciones pero solo una casilla de distancia.
Aplicamos el mismo criterio para devolver todos los posibles movimientos de la pieza.

3.13.6 Clase Torre

Clase que extiende de Pieza y implementa sus métodos abstractos.
Verificamos que no se valide ningún movimiento ilegal: consideramos movimientos ilegales todo aquel movimiento que sitúe la pieza fuera de los límites del tablero o ponga la pieza en una casilla ocupada por otra pieza del mismo color. También verificamos que los movimientos sean estrictamente horizontales o verticales y que no se pase por encima de otra pieza durante la trayectoria por el tablero hacia la posición de destino.
Aplicamos el mismo criterio para devolver todos los posibles movimientos de la pieza.

3.14 Clase Problema

Como atributos, la clase tiene:

- Un String, llamado problema, que contiene, solamente, la situación de las piezas en el tablero
- Un integer, que indica el valor de N
- un Boolean que indica si el juego es iniciado por las blancas o no
- Un objeto evaluación, para poder validar un problema
- Un boolean que indica si el problema ha sido o no verificado

Getters y Setters de los atributos

public void setFEN(String FEN)
La función recibe, como parámetro, un FEN completo (por ejemplo: 1k6/4n3/2r5/8/8/8/4N3/2K5 w - -). La función elimina los últimos elementos del FEN y se queda solo con la situación del tablero para asignarlo en el atributo problema, de la clase (en nuestro caso asignaría al atributo problema: 1k6/4n3/2r5/8/8/8/4N3/2K5).
public boolean verificarProblema(String FEN)
Como precondición, los atributos de la clase problema (N, y el String Problema) deben estar asignados. El objetivo principal de la función es preparar todos los parámetros que necesita el minimax para poder ejecutarse.

private void verificar(Tablero tablero, boolean maxAttackingPlayer)
Verificar es la función del minimax. La función ejecuta el algoritmo de minimax però, en esta ocasión, no busca el mejor movimiento sino que, simplemente, busca una hoja donde las piezas que abren el juego hagan mate. Si la encuentra, el atributo de la clase verificado se pone a true, en caso contrario el problema no está verificado.

private int min(Tablero tablero, int depth)
Función auxiliar para el minimax. Como parámetros recibe un tablero donde se ha ejecutado un movimiento y un nivel de profundidad que debe explorar. La función retorna un valor que será el mínimo de todos los movimientos posibles, después de que el movimiento inicial se ejecutara.

private int min(Tablero tablero, int depth)
Función auxiliar para el minimax. Como parámetros recibe un tablero donde se ha ejecutado un movimiento y un nivel de profundidad que debe explorar. La función retorna un valor que será el máximo de todos los movimientos posibles, después de que el movimiento inicial se ejecutara.

private static boolean gameOver(final Tablero tablero)
La función recibe, como parámetro, un tablero. En función de este tablero, devuelve true si el Jugador está en jaque mate o si no le he es posible hacer ningún movimiento.

3.15 Clase Jugador

La clase Jugador es una clase abstracta. Como atributos contiene:

- Booleans que indican si el Jugador es una Máquina, está en Mate, tiene piezas negras, y si está Atacando.

- Una pieza de su Rey
- Unos arrays donde guarda sus piezas del tablero y otra donde están todos los posibles movimientos de sus piezas.
- Una copia del tablero en el que se está jugando en diferentes puntos del juego.

private boolean tieneEscape()
Cuando la función es llamada, esta crea una nueva instancia de tablero y revisa que el parámetro implícito tenga algún movimiento que se pueda hacer sin quedar expuesto a una ataque a su Rey. La función devuelve un boolean que es true de encontrar al menos un movimiento posible, y false de no encontrarlo.

public static ArrayList<Movimiento> hayAtaquesPendientes(final int theX, final int theY, final ArrayList <Movimiento> movimientos)
La función recibe como parámetros la ubicación del Rey y una lista de movimientos posibles del oponente. Devuelve un array de movimientos que podría hacer el oponente al Rey del parámetro implícito.

public MovimientoPrueba hacerMovimiento(final Tablero tablero, final Movimiento movimiento)
La función recibe el estado del tablero y un movimiento a probar dentro de este. La función devuelve una instancia de Movimiento Prueba en la cual guardamos el tablero de donde se vino, el tablero a donde se va, el movimiento que ha sido probado y un boolean que nos indica si dejaría al Rey en una posición de Mate (false) o no (true).

public abstract Tablero jugar(Tablero tablero, Movimiento movimiento)
La función recibe el estado del tablero y un movimiento a probar que ha hecho un jugador Persona. La función devuelve de tener éxito un Tablero modificado con el movimiento que ha mandado la Persona.
public abstract Tablero jugar(final Tablero tablero, final int N)
La función recibe el estado del tablero y un número de movimientos que será el nivel de profundidad a explorar por parte del jugador Maquina. La función devuelve de tener éxito un Tablero modificado con el movimiento que ha mandado la Máquina.

3.15.1 Clase Persona

Clase que extiende de Jugador y implementa sus métodos abstractos.

3.15.2 Clase Maquina

Clase que extiende de Jugador y implementa sus métodos abstractos.

Como atributo tiene una instancia de la clase Estrategia Simple. Desde este objeto el jugador Máquina puede acceder a su algoritmo de decisión para que le devuelva el mejor movimiento que puede hacer dependiendo de si tiene una posición de ataque o de defensa.

3.16 Clase Movimiento

- Integers que nos indican por un lado la posición de origen y por otro la posición de destino.
- Un char que nos indica que pieza se encuentra en la posición de destino de haber una pieza.

public Tablero intentar(final Tablero iniTablero)
La función recibe como parámetro un tablero, se crea un nuevo tablero de él para poder intentar ejecutar el parámetro implícito en este. La función devuelve dicha instancia nueva de tablero que ha sido modificada.

public boolean esNada()
La función evalúa si el parámetro implícito es una posición no existente. La función devuelve un boolean de true si es así y false si es un movimiento válido.

3.17 Clase Movimiento Prueba

- Dos tableros de origen y destino
- El movimiento que ha sido hecho
- Un boolean que nos indica si se ha podido hacer o no el movimiento

Esta clase solo tiene getters y setters. Objetos de esta clase son usados como una forma de probar movimientos dentro de la llamada recursiva de la clase de Estrategia simple y obtiene información del movimiento hecho para que en un futuro podamos expandir a otros tipos de estrategias o algoritmos.

3.18 Clase Estrategia Simple

- Un objeto Evaluación
- Un integer que nos dice la profundidad dada por el jugador

public Movimiento estrategiaSimple(final Tablero tablero)
La función recibe como parámetro el estado actual del tablero donde el siguiente movimiento le toca al Jugador que ha llamado la función. La función devuelve una instancia de Movimiento que será o el mejor movimiento que pueda hacer ese jugador o será un movimiento nulo en caso de no haber ninguna opción viable ya que el jugador está o estaría en mate.

private int min(final Tablero tablero, final int depth)
La función recibe como parámetro el estado del tablero después de un movimiento hecho en este y el nivel de profundidad a explorar. La función devuelve un valor que será el menor

valor de todos los movimientos posibles después de que el movimiento inicial ha sido ejecutado.

private int max(final Tablero tablero, final int depth)
La función recibe como parámetro el estado del tablero después de un movimiento hecho en este y el nivel de profundidad a explorar. La función devuelve un valor que será el mayor valor de todos los movimientos posibles después de que el movimiento inicial ha sido ejecutado.

private static boolean gameOver(final Tablero tablero)
La función recibe como parámetro el estado del tablero y evalúa si el jugador está en jaque mate o está estancado. La función devuelve un boolean que será true si el jugador se encuentra en cualquiera de estos estados, y false si no se encuentra en ninguno de estos estados.

3.19 Clase Evaluacion

public int evaluar(Tablero tablero, int depth)
La función recibe como parámetro el estado del tablero después de un movimiento hecho en este y el nivel de profundidad a explorar. La función devuelve un valor que será la diferencia de las puntuaciones evaluadas del jugador que ataca y la del jugador que se defiende.

private static int puntuacion(Jugador jugador, int depth)
La función recibe por parámetro un jugador a evaluar y la profundidad en la que estamos dentro del árbol de movimientos. La función devuelve la suma de las diferentes evaluaciones a las que se llama para el jugador del parámetro.

private static int evaluarPuntosPiezas(Jugador jugador)
La función recibe como parámetro un jugador. La función devuelve un integer que es la suma de los valores individuales de cada pieza que el jugador todavía tiene en el tablero en su estado actual.

private static int elOtroJugadorEnJaque(Jugador jugador)
La función recibe como parámetro un jugador. La función devuelve si el oponente del jugador pasado por parámetro se encuentra en mate un bonus, sino devuelve un 0.

3.20 Clase Tablero

- Un 2D array/matriz de chars donde guardamos el tipo de pieza que hay o un valor de ‘0’ de no haber ninguna.
- Un array de piezas blancas y otro de piezas negras.
- Un booleano que lleva de quien (piezas blancas o negras) es el turno.
- Dos objetos de Jugador.

public void FENToTablero(String FEN, boolean turnoBlancas)
La función recibe como parámetro un string donde se encuentra el problema a resolver y cuál de los dos colores de piezas comienza primero. La función no devuelve nada pero modifica el atributo de matriz y actualiza todos los otros atributos del parámetro implícito.

private Pieza getPiezadeChar(char piezasCode, boolean esNegra, int posX, int posY)
La función recibe un char que indica el tipo de pieza, un boolean que nos dice si es negra o blanca, y su posición en la matriz/tablero (posX,posY). La función devuelve un objeto Pieza de la subclase indicada por el char.

private ArrayList<Movimiento> todosLosMovimientosPosibles(final ArrayList<Pieza> piezasJugador)
La función recibe como parámetro un array de piezas de un jugador. La función devuelve en un array todos los posibles movimientos para cada una de las piezas del array del parámetro.

public void ejecutarMovimiento(final Movimiento movimiento)
La función recibe como parámetro un movimiento del jugador. La función no devuelve nada pero modifica el parámetro implícito y actualiza completamente, incluyendo a sus jugadores.

public boolean movimientoHumano(Movimiento movimiento)
La función recibe como parámetro un movimiento del jugador Persona. La función devuelve un boolean que indica si ha sido posible hacer el movimiento del parámetro nada pero modifica el parámetro implícito y actualiza completamente, incluyendo a sus jugadores.

public Jugador getAttackPlayer(boolean estaAtacando)
La función recibe como parámetro un boolean que nos indica si queremos el jugador que está atacado (true) o no (false). La función devuelve el jugador del parámetro implícito que está atacando o no dependiendo del parámetro pasado.

public Jugador esSuTurno()
La función devuelve el jugador del parámetro implícito del cual es su turno de hacer un movimiento.

public Jugador miOponenteEs(final Jugador jugador)
La función recibe una instancia de jugador. La función devuelve el oponente del jugador del parámetro.

4. Relación de clases implementadas por cada miembro del grupo

Mayra Pastor Valdivia	Roger Guasch Ibarra
Jugador	Controlador de dominio
Persona	Problema
Máquina	Pieza
Movimiento	Pieza: Torre
Movimiento prueba	Pieza: Alfil
Estrategia Simple	Pieza: Reina
Evaluación	
Tablero	
Pieza: Peón	
Pieza: Caballo	
Pieza: Rey	

5. Estructuras de datos y algoritmos

Para esta primera entrega hemos implementado el algoritmo de Minimax que se inicia cuando el Jugador de tipo Máquina, ya sea en modo de ataque o defensa tiene que decidir qué movimiento de todos los movimientos posibles de su conjunto de piezas es el “mejor” que puede hacer. Esta funcionalidad se llamará en los escenarios que un Jugador humano juegue contra una Máquina y si dejamos que dos Máquinas jueguen una contra la otra. Para ello se llama desde la función de un jugador Maquina jugar() a un objeto de la clase EstrategiaSimple al que se le pasa el tablero que contiene la ubicación de las piezas de ambos jugadores a la función que comienza a implementar el algoritmo y devolverá una instancia de la clase Movimiento que contiene la posición de inicio y final de la pieza a mover. En esta función se iterarán todos los movimientos del array de piezas del jugador en cuestión y por cada uno de ellos se creará un árbol de posibles movimientos que al llegar a la hoja pasarlos por una evaluación que les da un valor dependiendo del estado de cada

jugador en ese momento después de haber intentado una serie de movimientos por cada uno. Por ahora evaluamos el número de Piezas de cada jugador y valoramos aún más que el otro jugador quede en jaque al final de estos nodos de movimientos.

Cada nodo de estos posibles movimientos por ambos jugadores ocurren por la función de min() o por la función de max() si el jugador se está defendiendo. En cada llamada se crea una instancia de la clase MovimientoPrueba en la cual se guarda el tablero del que se viene y al que se va después de ejecutar el movimiento y que nos dice mediante un boolean si el movimiento ha sido posible de completar porque no deja a su Rey en una posición vulnerable a ataques del oponente. De no pasar la prueba, no se seguirán creando más nodos por esa rama. Por cada nodo/movimiento explorado se tiene en cuenta una profundidad N, cada Jugador tiene N movimientos disponibles para, o hacer jaque o no dejar al otro jugador hacer mate.

También hemos adaptado el Minimax para la verificación de que un Problema puede ser resuelto en N movimientos y devuelve un true en el caso de encontrar un posible escenario de movimientos en el cual el oponente del jugador en cuestión queda en jaque. La mayor diferencia entre esta modificación y la anterior es que esta no devuelve el mejor movimiento posible, sino que explora que en N o en menos de N movimientos hemos podido evaluar que el oponente queda en mate.

6. Tests

Decidimos aplicar la estrategia de testing de testeo incremental, asumiendo el riesgo que, en caso de detectar algún error, nos sería más difícil determinar de dónde procede dicho error.
A continuación, se describe brevemente todos los casos que se han probado para los drivers hechos.

Clase Pieza

Dado que Pieza es una clase abstracta, en ningún momento hacemos un driver para ella, pues nunca la podríamos instanciar. De todos modos, cuando hablamos de Pieza, nos referimos a todas las subclases, hijas de Pieza, que definen el comportamiento de cada pieza en particular. Para todas hemos probado los mismos casos de prueba que a continuación detallamos:

1. Que se pueda crear un objeto con los atributos dados por terminal
2. Que la pieza se situe correctamente en el tablero, según se le ha indicado

3. Para el caso específico de la función esMovimientoOk(..):
    a. Que no permita un movimiento que situe la pieza fuera del tablero
    b. Que no permita un movimiento ilegal, teniendo en cuenta los distintos movimientos que cada pieza puede hacer
    c. Que no permita un movimiento si en la posición de destino hay otra pieza del mismo color
    d. Que permita el movimiento si en la posición de destino hay una pieza de distinto color
    e. En general, que permita hacer cualquier movimiento que una Pieza de su tipo podría hacer
4. Para el caso específico de la función movimientosPosibles(..):
    a. Que devuelva todos los movimientos que dicha pieza podría hacer, teniendo en cuenta que, la función, solo devuelve los movimientos legales teniendo en cuenta cada tipo de pieza y no considera movimientos ilegales.

Todos los Drivers que hemos implementado que testean las distintas clases de nuestro proyecto siguen una estructura muy similar. Tienen un menú inicial que permiten al usuario indicar qué quiere testear y, por cada opción concreta, el driver le informa al usuario qué debe introducir por terminal para que el test se ejecute de forma correcta. Para el caso particular del testeo de la función esMovimientoOk de las Piezas (en otras no lo hacemos), el driver imprime por terminal *Test completado con éxito*, en color rojo. Si la terminal no soporta el uso de ANSI scape codes, el color no aparecerá de forma correcta.

Los Drivers para probar las Piezas (Alfil, Caballo, Peón) tienen nombre de Driver[nombre_de_la_pieza].java. Por ejemplo, el dirver de la clase Caballo tiene el nombre de DriverCaballo.java

Las clases que extienden Pieza no dependen de otras, por lo que no es necesario haber probado ninguna antes, para verificar el correcto funcionamiento de estas.

Para ejecutar los tests es necesario usar el fichero testDriver[nombre_de_la_pieza].txt Por ejemplo, si quiero ejecutar el test de la pieza Caballo ejecutaré, por terminal, el siguiente comando: $ java -jar DriverCaballo.jar < testDriverCaballo.txt

Clase Problema

Para la clase Problema verificamos que se pueda instanciar correctamente así como que valide un problema dado un FEN determinado. La verificación del problema, como sabemos, busca una hoja del árbol del minimax en el cual se encuentre que, el jugador que abre el juego, hace mate al otro jugador. Así pues, para verificar que dicha función hace lo que se espera, probamos dándole distintos FENs que, previamente, nosotros sabemos que son o no validables. La función, como resultado, nos indica si ha podido encontrar un jaque mate en el que el jugador que abría el juego se convertía en el ganador de la partida.
En todos los tests probados la función ha retornado el valor esperado, por lo que estamos satisfechos con su testeo. De todos modos, somos conscientes que el minimax es un

algoritmo de fuerza bruta y, como prueba todos los posibles movimientos de todas las piezas y predice también el mejor movimiento del rival, se vuelve un algoritmo algo lento si, como parámetro, se le pasa un valor de N superior a 5. Por este motivo, hemos testeado dicho algoritmo con una N más pequeña o igual a 5.

Como problema ejecuta un minimax y usa un objeto Evaluación tenemos que estar seguros que se ha testeado previamente.

El Driver que testea la clase Problema se llama DriverProblema.java y, las pruebas que hemos ejecutado en este driver, están detalladas en el fichero testDriverProblema.txt

Clase Movimiento

La clase Movimiento contiene un movimiento que una pieza puede hacer en el tablero y una función que permite construir un tablero directamente con el movimiento. En el driver que tenemos hecho prueba que se pueda crear un objeto movimiento y que la función implementada en Movimiento cree un tablero a partir del movimiento que tiene como atributo y un tablero que se le pasa como parámetro.

El Driver que testea la clase Movimiento se llama DriverMovimiento.java Como la función que implementa usa una instancia de Tablero hay que estar seguros que la clase Tablero ha sido verificada.

Clase Movimiento Prueba

Probamos que se pueda crear un objeto de MovimientoPrueba y que contenga los parámetros que le hemos asignado por terminal. No es necesario testear ninguna otra función en dicha clase. Como la clase Movimiento Prueba contiene dos tableros y un objeto movimiento, es necesario que se haya verificado que las clases tablero y movimiento pueden ser correctamente instanciadas, previamente al testeo de la clase Movimiento Prueba.

Clase Tablero

Para esta clase, y siguiendo las indicaciones del profesor, la hemos testeado usando el JUnit. Probamos cada función de forma aislada y, usando el *assert* de JUnit, verificamos que nos devuelve los valores que nosotros esperamos.
Esta es, sin duda, una de las clases con más funciones de toda la capa de dominio. Entre otras cosas, nos aseguramos que se cree un tablero correctamente dado un FEN concreto, que se asignen a los jugadores las piezas de cada jugador, que un movimiento en el tablero de una pieza en particular se haya ejecutado correctamente, etc. También se prueban todos los getters y setters que tiene la clase.

Clase Estrategia Simple

La clase Estrategia Simple implementa el algoritmo del minimax, por lo que nuestro Driver lo que hace es verificar que la función estrategiaSimple(..) nos devuelve un movimiento concreto, que sería el movimiento que M1 debería realizar en su turno, ya que el minimax considera que ese es la mejor opción, teniendo en cuenta siempre los movimientos del oponente.

Clase Evaluación

La clase evaluación testeamos que, dado un problema en formato FEN y una N concreta, nos devuelve puntos para los dos jugadores que están jugando la partida, indistintamente de si están atacando o defendiendo. Usamos la clase evaluación como la heurística de nuestro programa, así que verificamos con el driver que, efectivamente, nos devuelve los valores esperados.

Clase Jugador

Para la clase jugador pasa el mismo caso que con la clase Pieza: al ser una clase abstracta no se puede instanciar pero, como Persona y Máquina extienden Jugador y implementan sus métodos abstractos, podemos hablar genéricamente de las dos clases, dado que, tanto en el Driver Persona como en el Driver Máquina probamos lo mismo:

1. Que se pueda crear una instancia del objeto de forma correcta (tanto de Persona como de Máquina)
2. Que la función jugar, que implementan ambas clases, ejecute el movimiento deseado.
    a. Para la el objeto Persona el usuario deberá introducir, por terminal, la posición de la pieza y la posición dónde quiere moverla y la función realizará el movimiento en función de la validez de los datos introducidos por terminal. Así pues, como la función hace un movimiento en el Tablero, es necesario asumir que, previamente al testeo de la clase Persona, la clase Tablero ha sido verificada.
    b. Para el objeto Máquina el usuario deberá introducir un FEN, que será el estado del tablero, y un valor de N. La función jugar se encargará de hacer el mejor movimiento en función de nuestra heurística y el algoritmo minimax. Dado que esta función ejecuta el minimax, es necesario que dichas funciones hayan sido testeadas previamente, así como también es necesario que el tablero haya sido también probado.

Clase Controlador de Dominio

El Controlador de Dominio, su función principal, es la de ejecutar las dos funcionalidades principales que debemos tener implementadas para esta entrega: por un lado verifica un problema y, por el otro, permite jugar una partida que tenga, como uno de los dos

participantes, a M1. Así pues, en el DriverCtrl_Dominio se testea que se pueda jugar una partida y se verifique un problema concreto.

En nuestros tests que ejecutamos verificamos que, efectivamente, solo se permite tener una instancia de controlador de dominio, ya que lo hemos implementado siguiendo un patrón singleton. También comprobamos que, cuando la máquina juega contra H1, tanto si gana como si pierde, el controlador gestiona de forma correcta no permitir hacer más movimientos. Por último verificamos que, efectivamente, la verificación de un problema es correcta. Lo comprobamos ejecutando el programa y pasándole FENs que sabemos que son correctos (hay mate en N o menos jugadas) y FENs que no lo son y, en ambos casos, las respuestas del programa son las esperadas.
Los tests de este Driver se podrán encontrar en el fichero testCtrl_Dominio.txt