Segunda Entrega
PROP

Grupo 2.1

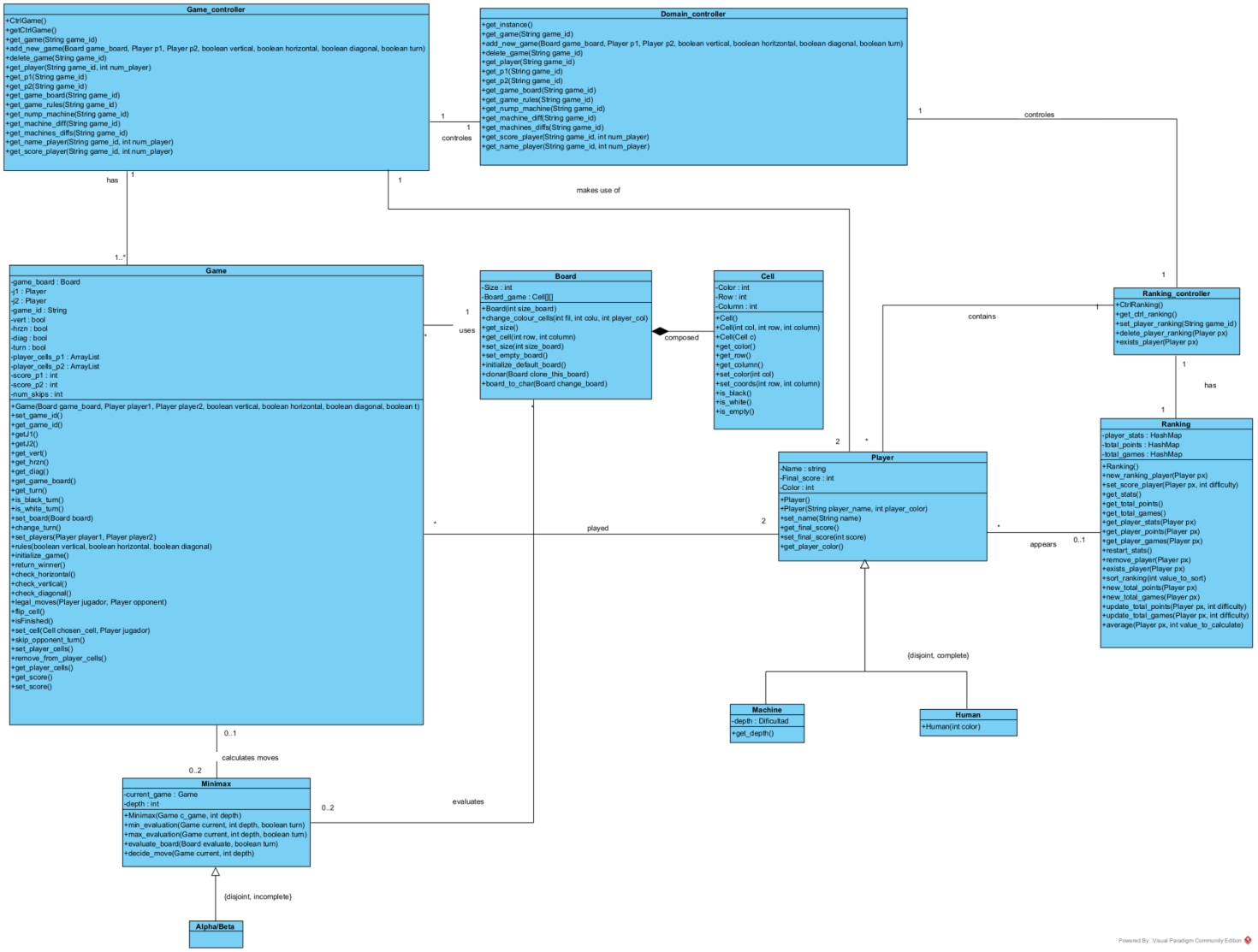
Aidan Martínez - aidan.martinez
Nicolas Orellana - nicolas.ignaci.orellana
Isis Rodríguez - isis.rodriguez

2ª Entrega 25 de Mayo de 2021

Índice

1. Capa de Dominio

1.1 Diagrama



1.2 Descripción clases

1.2.1 Cell
La clase representa cada una de las celdas de un Tablero, ya sea vacía, ocupada por una ficha del Jugador de color blanco o por el negro. Tiene dos atributos para indicar su posición en el tablero como fila y columna. Las celdas componen el Tablero.

1.2.2 Board
La clase representa el tablero sobre el que se va a jugar. Este tablero tiene un atributo dimensión para indicar su tamaño en filas y columnas, teniendo en cuenta que el tablero será siempre cuadrado. Está compuesta por la clase Cell, por lo cual es un conjunto de esta. El tablero es creado por cada instancia de Game y es jugado por dos jugadores (humanos o máquinas).

1.2.3 Game
La clase representa la asociación entre dos jugadores (humanos o máquinas) que juegan una partida de Othello. Esta clase tiene un Tablero, dos jugadores y una serie de tres booleanos para indicar si en las normas de la partida se permiten realizar movimientos verticales, horizontales y diagonales. También tiene un identificador que se asignará a cada juego y un booleano que indica el turno del jugador en cada momento del juego.
Como mejora hemos implementado dos arraylists, uno para cada jugador. Estas arraylists ya estaban anteriormente implementadas en la clase Player, pero quitándolas de ahí y poniéndolas en Game quitamos la dependencia innecesaria que esto creaba con la clase Cell.
Estos arraylists contienen las Cells que tienen cada jugador. También hemos añadido dos variables relacionadas con los arraylists que indican la puntuación de cada jugador en cualquier momento de la partida.

Otra mejora que hemos implementado ha sido añadir handicaps a los Games. Ahora Game tendrá un entero num_skips que determina cuántas veces puede saltar el turno del oponente. num_skips será 0 en caso de no seleccionar ningún handicap, pero en caso de hacerlo, puede tomar valores de 1, 2, o 3. Con este atributo incluimos una función que salte el turno del oponente skip_opponent_turn() la cual reducirá num_skips cada vez que es llamada si es que num_skips es mayor a cero y cambie el turno.

Dentro de esta clase hay tres funciones principales; check_horizontal, check_vertical, y check_diagonal. Estas funciones se usan tanto en legal_moves como en flip_cell.
En legal moves, dependiendo de las reglas que el jugador haya escogido para la partida, por cada ficha que tenga el jugador en su arraylist correspondiente, llamará

a las funciones relacionadas con las reglas escogidas. Cada una de estas funciones agregan entradas al HashMap creado en legal_moves.

Las Keys serán Cells en las que el jugador puede poner su ficha, y los Values asociados a esa Key serán las direcciones por las que han sido descubiertas dichas casillas.

KEY	**VALUES**			
Movimiento posible (Cell)	Dirección 0	Dirección 1	...	Dirección N

Gracias a este HashMap podemos buscar cuales son los movimientos posibles de cada jugador, pero tiene más de una función.
Cuando el jugador quiere colocar una ficha llamará a flip_cell que, aparte de mirar si esa ficha se encuentra dentro de las casillas encontradas con los movimientos permitidos (legal_moves), llamará a las funciones anteriormente mencionadas check_horizontal, check_vertical, y check_diagonal. Estas funciones no solo tiene la función de encontrar las casillas en las cuales un jugador puede poner una ficha si no que también, si se le indica, puede dar la vuelta a las fichas del adversario al colocar una ficha. Por lo que cuando llamemos otra vez a las funciones check pondremos a true el booleano de flip, indicando que queremos que las funciones den la vuelta a las casillas del adversario e indicaremos la dirección por la cual queremos dar la vuelta a las fichas. Pero antes de llamarlas, buscaremos la casilla seleccionada por el jugador en el HashMap y miraremos los Valores que hay para esa Key. Los valores indican las direcciones por las que ese movimiento se ha descubierto (por lo que hay una ficha del jugador si nos movemos en esa dirección y fichas del adversario de por medio). Como mínimo una casilla puede haber sido descubierta por una dirección, y como máximo por ocho. Las direcciones son valores int a los que hemos dado un valor para cada una de las direcciones.

Valor	**Dirección**	**Función check a la que llamará**
0	left	check_horizontal
1	right	check_horizontal
2	up	check_vertical
3	down	check_vertical
4	diagonal up left	check_diagonal
5	diagonal up right	check_diagonal
6	diagonal down left	check_diagonal

7	diagonal down right	check_diagonal
8	none	-

Dependiendo del valor, llamaremos a las funciones relacionadas mencionadas en la tabla anterior, poniendo en direction el valor de la dirección por la cual queremos hacer flip.
Teniendo este HashMap con las direcciones nos ahorramos tener que pasar por todas las funciones check (relacionadas con las reglas establecidas) para cada movimiento posible descubierto, lo que nos ahorra tener que entrar en todos los bucles de cada una de estas funciones.

1.2.4 Minimax

Esta clase representa el algoritmo que va a utilizar el jugador maquina para participar en una partida. Tiene una partida y un indicador de la profundidad. Esta profundidad sirve para que cuanto más baje en el árbol de recursión más posibilidades analizará y mejor será su jugada.

1.2.5 Player

La clase representa a un jugador de la partida, que puede ser un humano usuario del programa o una máquina. Dispone de un nombre para identificar, una puntuación y el color que va a tomar en la partida que esté jugando. Hemos modificado para esta segunda entrega la clase Player de manera que tenga ahora un entero con la puntuación final del jugador en vez de un array de Celdas como teníamos antes. Para hacer este cambio posible también hemos necesitado añadir dos funciones para coger este valor y actualizarlo. Se cambia cuando la partida se acaba y es para que ranking pueda acceder a la puntuación del jugador

a. Human: representa el usuario que jugará la partida.
b. Machine: representa al jugador que va a utilizar el algoritmo de Minimax. Contiene un atributo para indicar la profundidad del algoritmo y así indicar la dificultad de la máquina.

1.2.6 Ranking

La clase representa la clasificación de los jugadores que hayan finalizado las partidas.
Disponemos de un HashMap para representar el nombre de los jugadores como key, y tener un array de enteros (con 8 valores) para guardarnos los valores de la máxima puntuación de cada dificultad (Fácil, Normal, Difícil, Contra Jugadores), además del promedio de estos jugadores en el total de partidas en estos niveles de dificultad.
HashMap <String,ArrayList<Integer>> player_stats

Nombre	Puntuación Fácil	Puntuación Normal	Puntuación Difícil	Puntuación vs Jugador	Media de Puntos Fácil	Media de Puntos Normal	Media de Puntos Difícil	Media de Puntos vs Jugador

En esta clase tenemos también para calcular estos promedios dos Hashmaps, uno para ir sumando la cantidad total de puntos del jugador en cada una de las diferentes modalidades de la partida. El otro Hashmap se utiliza para calcular cuantas partidas de esa modalidad lleva jugadas este jugador.

HashMap <String,ArrayList<Integer>> total_points

Nombre	Totalidad puntos fácil	Totalidad puntos normal	Totalidad puntos difícil	Totalidad puntos vs jugador

HashMap <String,ArrayList<Integer>> total_games

Nombre	Totalidad partidas fácil	Totalidad partidas normal	Totalidad partidas difícil	Totalidad partidas vs jugador

En esta clase ranking tenemos funciones como restart_stats() que reiniciamos el ranking entero, es decir reiniciamos las partidas jugadas de los jugadores, su total de puntos y sus récords en todos los tipos de modalidades de partidas.
De este ranking podemos eliminar también a un jugador con la función remove_player(Player px), si le indicamos un Player. Esta función elimina a este jugador de los tres HashMaps nombrados anteriormente.
Además de las funciones anteriores encontramos sort_ranking(int value_to_sort), en esta función pasamos un valor por el cual podemos ordenar este ranking. Este valor para ser ordenado puede ser:

0 - Ordenar por el récord de puntos en dificultad fácil.
1 - Ordenar por el récord de puntos en dificultad normal.
2 - Ordenar por el récord de puntos en dificultad difícil.
3 - Ordenar por el récord de puntos en dificultad vs jugadores.
4 - Ordenar por el promedio de puntos en dificultad fácil.
5 - Ordenar por el promedio de puntos en dificultad normal.
6 - Ordenar por el promedio de puntos en dificultad difícil.
7 - Ordenar por el promedio de puntos en dificultad vs jugadores.

Destacar que en esta clase tenemos funciones como new_ranking_player, new_total_points, new_total_games y average que el usuario no tendrá acceso a estas funciones porque son partes de otras funciones más grandes, es decir, solo estas funciones llamarán a estas más pequeñas.

1.2.7 CtrlDominio

La controladora de dominio es la clase principal que se encarga de poder realizar la comunicación entre las capas de presentación y de datos. Dentro de la misma crearemos instancias de las controladoras de Ranking y la controladora de Game, ya que la controladora de Dominio es la que se comunicará con estas dos controladoras y la que pasará la información al resto de capas.

1.2.8 CtrlRanking

La controladora de ranking se comunica principalmente con la controladora de dominio y con la clase ranking, su uso es para generar una comunicación y poder usar las funciones que se comunican con la controladora de dominio, ya sean las de Game o por defecto su controladora.

Tenemos también una relación con Player, ya que se obtienen diferentes jugadores de esta clase para su utilización en nuestra propia comunicadora de Ranking.

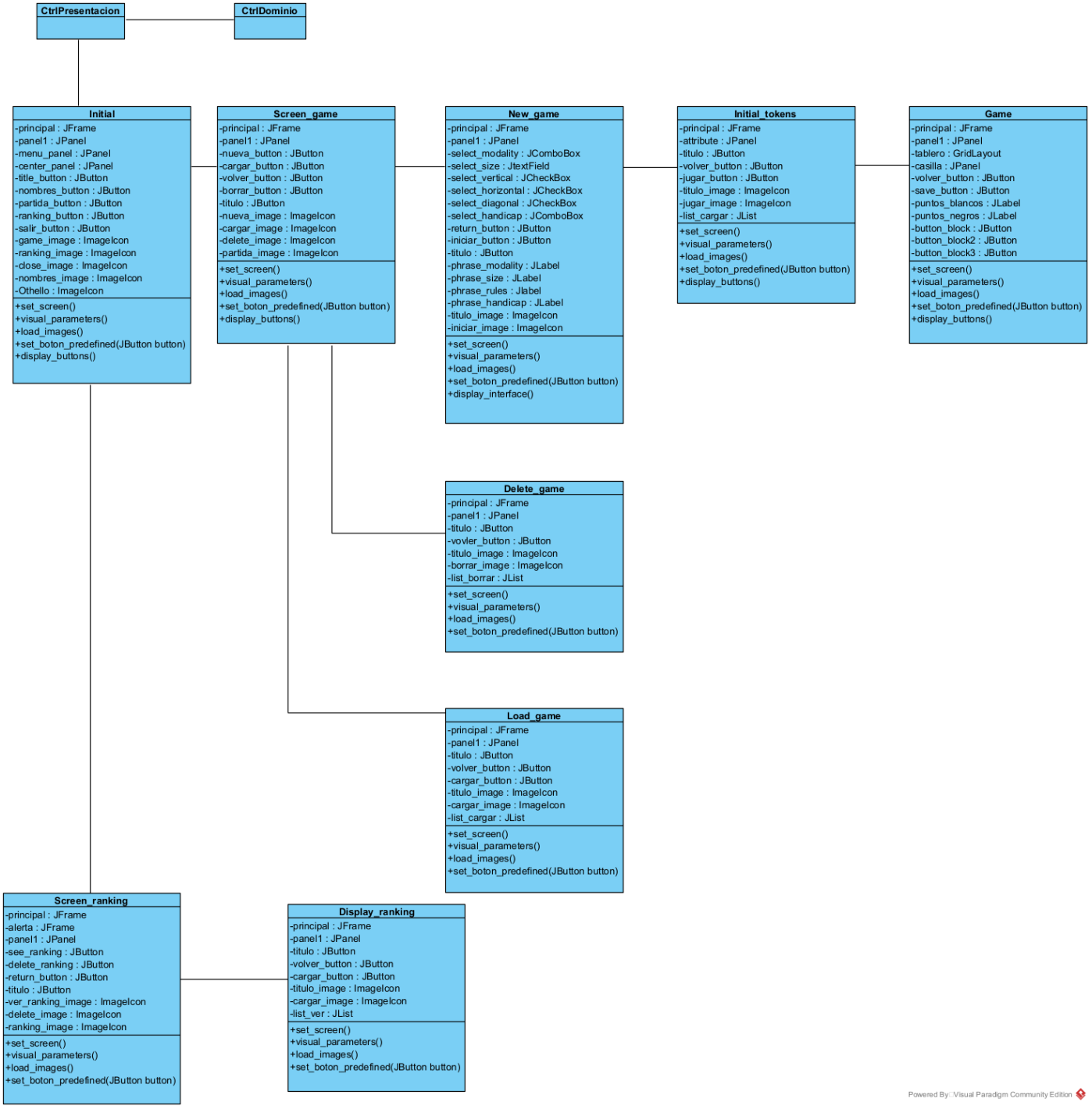
1.2.9 CtrlGame

La controladora de Game se comunica principalmente con la controladora de dominio y con la clase Game, su uso es para generar una comunicación y poder usar las funciones que se comunican con la controladora de dominio, ya sean las de Ranking o por defecto su controladora.

Tenemos también una relación con Board, Cell, y Player, ya que son necesarios para la creación de Game.

2. Capa de presentación

2.1 Diagrama



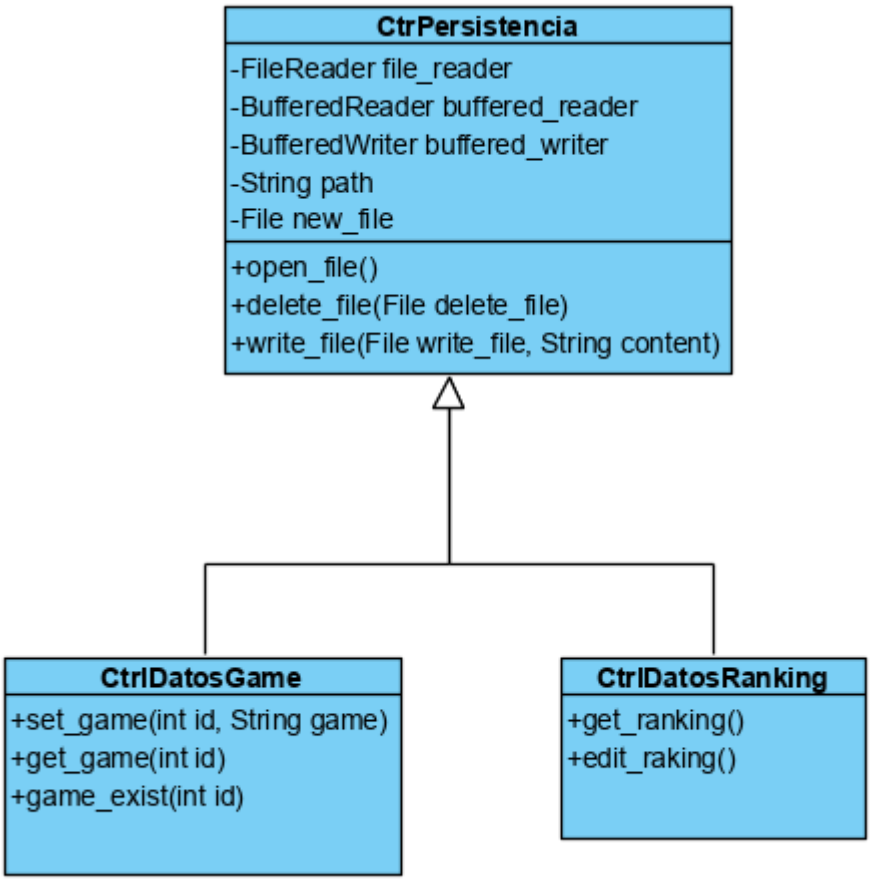
2.2 Descripción

En el diagrama anterior se representa la capa de presentación. Tenemos dos clases controladoras, la de CtrlDominio y CtrlPresentacion para gestionar la relación entre capas.
Empezamos nuestro diagrama con la relación entre la Controladora de Presentación y la clase Initial. Esta clase es nuestra primera pantalla, en la cual el objetivo es dar al usuario la posibilidad de elegir entre Ranking para ver toda la información relacionada con esta pantalla o irse a Partida donde tendremos diferentes opciones. También hay otros elementos estéticos.
Entonces vemos las relaciones de Initial con Screen_game (Partida) y Screen_ranking (Ranking), que son las clases a las que llevan los botones mencionados anteriormente.
En Screen_game el usuario puede elegir entre tres opciones de partida, si empezar una nueva, si cargar o borrar una partida guardada. Si luego seguimos las relaciones de Screen_game veremos tres clases que corresponden a los botones mencionados anteriormente.
En Delete_game (Borrar) vemos una pantalla para poder borrar partidas guardadas si elegimos una nos aparecerá una ventana emergente que nos pedirá confirmación antes de borrarlo.
En Load_game (Cargar) tendremos exactamente lo mismo pero con la opción para poder cargarlas.
En New_game (Nueva) es donde elegiremos los parámetros para comenzar una nueva partida (modalidad de jugadores, tamaño del tablero, reglas y handicap).
Esta pantalla anterior nos llevará a Initial_tokens en la que el usuario puede elegir si quiere empezar una partida con fichas diferentes de las estándar. Finalmente, esta pantalla nos redirige a la pantalla propia de jugar un Game.

En Screen_ranking se ve la opción de ver el ranking y el de borrarlo entero con dos botones. Si vamos con el botón de see_ranking llegremos a la pantalla en la que se verá todo el ranking con la posibilidad de ordenarlo de distintas maneras, explicadas anteriormente en la clase ranking del diagrama de uso.
En todas las pantallas excepto en la Initial se puede volver a la anterior con el botón return_button. En la Initial hay un botón para cerrar el programa directamente.

3. Capa de persistencia

3.1 Diagrama



3.2 Descripción

En este diagrama estamos representando la capa de persistencia. Tenemos una clase Controladora de Persistencia con las operaciones:

open_file, para poder abrir un fichero y en caso de no existir crearlo:

delete_file, para poder eliminar un fichero

write file, para poder escribir contenido dentro de un fichero

Esta clase también tiene atributos que sobre todo sirven para poder realizar el control de flujo de entrada y salida(BufferedReader y BufferedWriter), así como un atributo para el fichero File new_file, otro atributo para el path de donde esté el fichero String path y finalmente una variable de tipo FileReader para que se pueda leer del fichero.

La controladora de persistencia tiene herencia con dos clases:

Controladora de datos de Game: es la encargada de poder guardar las partidas mediante la función set_game , obtener las partidas basadas en su id mediante la función get_game y también tenemos una función auxiliar que determinará si la partida ya existe en el ranking.

Por otra parte tenemos la controladora de datos de Ranking: es la encargada de gestionar el fichero del Ranking, tiene una función de obtener ranking: get_ranking y otra función que permite editar el ranking edit_ranking

4. Mejoras en el algoritmo

El algoritmo que hemos usado de cara a esta primera entrega es el algoritmo de Minimax.
El algoritmo se basa en la idea de tener dos valores Min y Max en cada nivel del árbol de recursividad que se realizará en el algoritmo (se utiliza backtracking).

De forma que los niveles de Max serán los turnos del jugador que está aplicando el algoritmo y se encargará de maximizar las ganancias del jugador, mientras que los niveles de min serán los turnos del oponente y por ende se encargará de minimizar las ganancias de ese jugador.

Nuestro código se divide en 4 funciones principales.

- **Min_evaluation:** función utilizada para evaluar el nivel MIN. Utilidad para realizar el algoritmo Minimax.

```
FUNCTION min(partida, depth,jugador)
        IF( nodo OR partida_finalizada ) -> Lo mismo que estado_terminal
                return CALL evaluar_tablero(partida.tablero())
        lowest=infinito
        moves= CALL movimientos_posibles(jugador)
        for (moves_availables in moves )
                CALL poner_ficha(moves_availables)
                value= CALL max(partida, depth-1)
                IF (value < lowest) lowest=value
        return lowest
ENDFUNCTION
```

- **Max_evaluation:** función utilizada para evaluar el nivel MAX. Utilidad para realizar el algoritmo Minimax.

```
FUNCTION max(partida, depth,jugador)
        IF( nodo OR partida_finalizada ) -> Lo mismo que estado_terminal
                return CALL evaluar_tablero(partida.tablero())
        highest=-infinito
        moves= CALL movimientos_posibles(jugador)
        for (moves_availables in moves )
                CALL poner_ficha(moves_availables)
                value= CALL max(partida, depth-1)
                IF (value > highest) highest=value
        return highest
ENDFUNCTION
```

- **Decide_move:** Función que decidirá qué movimiento es el más óptimo basado en el recorrido del árbol. Esta función es el cerebro del algoritmo. Primero indicamos un valor -infinito para poder indicar luego cuál es la jugada óptima sobre la que realizaremos el siguiente movimiento.
  Lo siguiente es guardarnos los movimientos posibles del jugador que aplicará el algoritmo (la máquina), en nuestro caso serían los movimientos del jugador sobre el que estamos ejecutando el algoritmo, en nuestro caso nos guardamos cada movimiento válido de un jugador en una tabla de hash.
  Una vez hecho esto tenemos que recorrer todos los movimientos posibles, es decir, recorreremos la tabla de hash que contendrá todos los movimientos que puede realizar. Para cada movimiento llamaremos a la función min, que será la función que se ejecutará para realizar el turno del oponente (minimizando a la máquina), por su parte esta función min llamará a la función max que maximizará las ganancias de la máquina. En value guardaremos el valor de cada jugada que hagamos, quedándonos obviamente con el valor mayor ya que es el que nos aportará más ganancias de cara a futuras jugadas, todo esto basándonos en cuando un nodo sea terminal.
  Para que un nodo sea terminal, en nuestro caso hemos tenido en cuenta que este caso se da cuando ninguno de los dos jugadores tiene movimientos disponibles (la partida se ha acabado) o cuando se ha alcanzado la profundidad deseada (depth==0)
  Finalmente devolveremos el valor de la mejor jugada.
  No hemos incluido una descripción de las funciones Min y Max ya que el código es prácticamente el mismo que el de decide.

```
FUNCTION decide(partida, depth, jugador)
        value=-infinito
        moves= CALL movimientos_posibles(jugador)
        for (moves_availables in moves )
                CALL poner_ficha(moves_availables)
                value= CALL min(partida, depth-1)
                IF (value > highest)
                        highest=value
                        best_move=moves_availables
        return best_move
ENDFUNCTION
```

- **Evaluate_board:** Función que dado un nodo terminal, se encarga de darle un valor al tablero escogido (heuristica). Es la que se encarga de darle una "puntuación" a los nodos hijos cuando llegamos a la profundidad indicada. En nuestro caso hemos tenido en cuenta el número de fichas de cada jugador, pero también tenemos en cuenta que las esquinas son lugares estratégicos muy valiosos, por lo que les damos un valor elevado para que el algoritmo priorice estos elementos. También observando diversas estrategias de Othello hemos descubierto que los sitios antes de las esquinas (p.e i=1 j=1) son casillas donde no queremos poner nuestras fichas ya que dejamos descubierta la esquina para el oponente, por lo que le hemos puesto un valor negativo.
  Esta función de cara a futuras entregas intentaremos mejorarla teniendo en cuenta otros aspectos como puede ser la movilidad o la estabilidad de las fichas, dos aspectos muy importantes en el juego.

```
FUNCTION evaluar_tablero(tablero)
        num_fichas_blancas = CALL get_fichas_blancas()
        num_fichas_negras = CALL get_fichas_negras()
        comprobar_esquinas
        comprobar_anteesquinas
        return num_fichas_blancas-num_fichas_negras
ENDFUNCTION
```

En cuanto a mejoras, hemos añadido la poda alfa-beta.
Esta poda permite mejorar el tiempo de ejecución del algoritmo, funciona de forma que ahora mantenemos dos enteros, el valor alfa y el valor beta, que representan la puntuación mínima que aseguramos al max y la puntuación máxima que aseguramos al min.
En un principio, tanto el valor alfa como el valor beta toman un valor infinito negativo e infinito positivo respectivamente, por lo tanto, tanto el jugador max como el min tendrán su peor puntuación. Una vez entremos en las funciones max y min, cunado la puntuación máxima que se otorga al jugador que minimiza (min) se vuelve menor que la puntuación mínima que se otorga al jugador max (el jugador alfa) (beta<alfa). Podemos podar el árbol ya que no necesitamos explorar lo que hay debajo del nodo.

En cuanto a la heurística, hemos añadido ahora también que tenga en cuenta la movilidad de las fichas, de esta forma también tenemos en cuenta los movimientos disponibles al realizar una jugada. Esto nos interesa ya que después de leer distintas estrategias, creemos que es muy importante intentar que la máquina tenga el mayor número de movimientos disponibles. Esta estrategia se basa en la idea de que cuantas más opciones tenga la máquina a su disposición, más fuerte será la jugada que puede realizar.

5. Mejoras en estructuras de datos

Como mejora en Game hemos implementado dos arraylists, uno para cada jugador. Estas arraylists ya estaban anteriormente implementadas en la clase Player, pero quitándolas de ahí y poniéndolas en Game quitamos la dependencia innecesaria que esto creaba con la clase Cell.