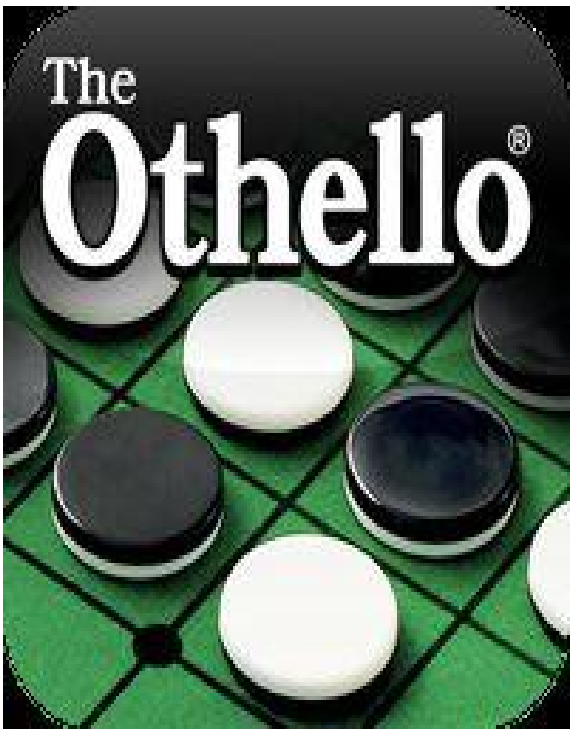


Descripción breve

En este documento desarrollamos el proyecto Othello que es un juego que se disputa entre dos personas, que comparten 64 fichas iguales, de caras distintas, que se van colocando por turnos en un tablero dividido en 64 casillas.

Autores:

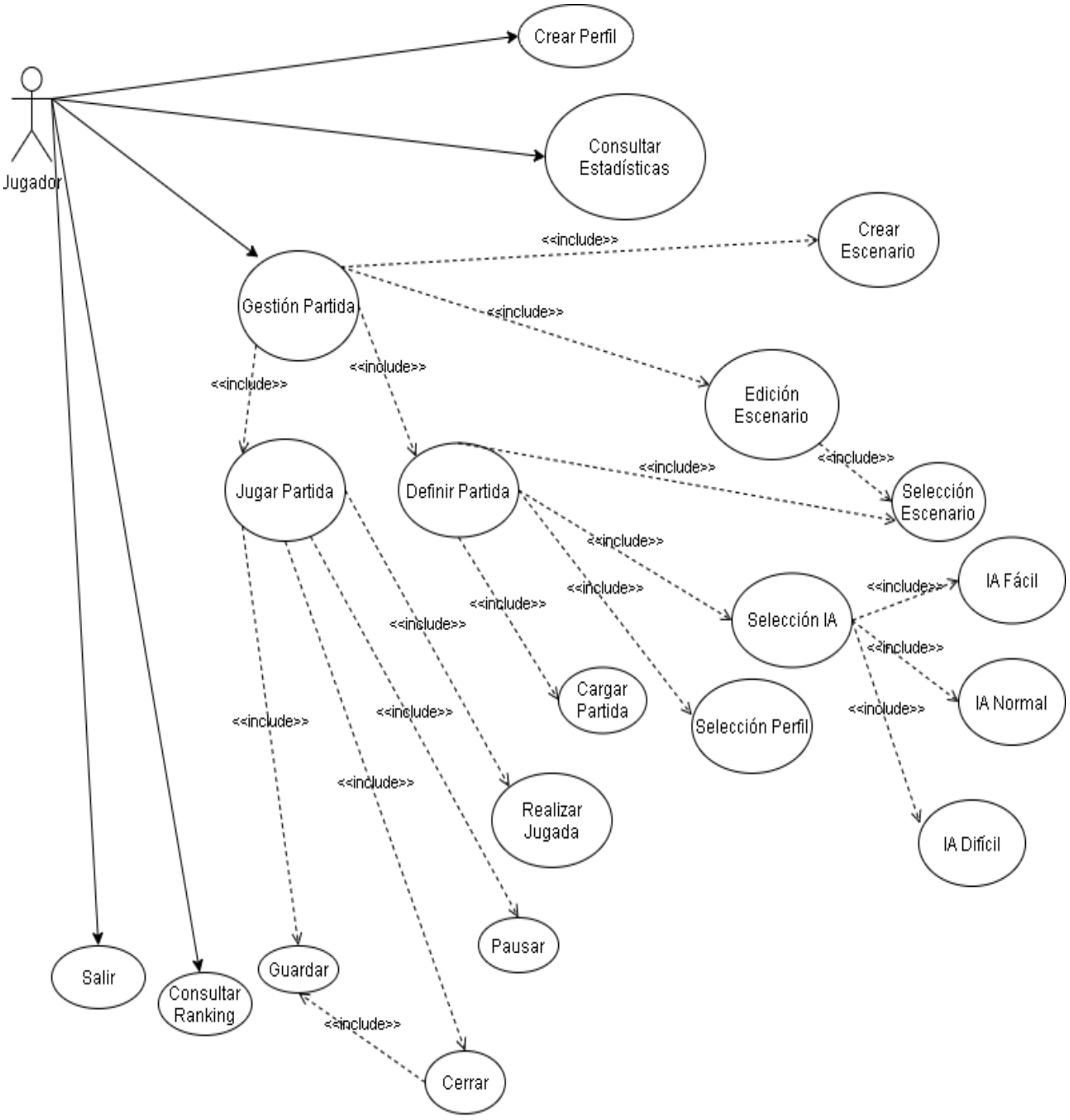
Acevedo Montañez, Franco
Baqueró Quesada, Jaume
Velasco Calvo, Aleix

Proyecto de Programación - 11

ÍNDICE

1.Casos de Uso.

1.1. Diagrama de Casos de Uso.



1.2. Descripción de casos de uso

1.2.1. *Caso:* Crear Perfil

Actor: Jugador

Comportamiento: Registra un jugador introduciendo un alias y una contraseña.

Errores posibles y cursos alternativos:

- Alias ya existe: introducir un alias diferente.
- Alias o contraseña en blanco: Se solicita nuevamente ingresar ambos campos correctamente.

1.2.2. *Caso:* Consultar Estadísticas

Actor: Jugador

Comportamiento: Permite ver todas las estadísticas de un jugador seleccionado (Juegos ganados, perdidos, empates, puntos).

Errores posibles y cursos alternativos:

- No existe ningún jugador en el sistema.

1.2.3. Caso: Consultar ranking

Actor: Jugador

Comportamiento:

El sistema muestra una lista con todos los usuarios registrados con sus respectivas puntuaciones y clasificación (ranking) en orden descendiente por puntuación.

Errores posibles y cursos alternativos:

1.2.4. *Caso:* Definir partida

Actor: Jugador

Comportamiento: Se intenta crear una partida nueva o cargar una guardada. Si se crea una partida, se elige el tipo de partida y se selecciona los participantes y sus colores y opcionalmente un escenario existente

Errores posibles y cursos alternativos:

- Si no existe una partida guardada: inicia una partida nueva.
- Si no existe un perfil o un escenario se selecciona otro o se continúa con uno por defecto.

1.2.5. *Caso:* Selección IA

Actor: Jugador

Comportamiento: Se escoge una de las 3 dificultades disponibles. En una situación de partida IA vs IA se puede seleccionar la misma dificultad para ambas.

Errores posibles y cursos alternativos:

1.2.6. *Caso:* Selección perfil

Actor: Jugador

Comportamiento: Los jugadores eligen un perfil de los disponibles. Por defecto existe un perfil “Guest”.

Errores posibles y cursos alternativos:

- Perfil de jugador no està a la llista: Crear perfil o seguir con guest.

1.2.7. *Caso:* Crear Escenario

Actor: Jugador

Comportamiento: El usuario puede crear una situación "legal" en un tablero de juego y guardarlo , el usuario asigna un identificador al escenario. Estos escenarios se pueden cargar antes de comenzar una partida.

Errores posibles y cursos alternativos:

- Identificador de escenario ya existe: cambiar identificador.

1.2.8. *Caso:* Modificar Escenario

Actor: Jugador

Comportamiento:

El usuario puede seleccionar un escenario existente, modificarlo y guardarlo como un escenario aparte o sobrescribir la actual.

Errores posibles y cursos alternativos:

1.2.9. *Caso:* Realizar jugada

Actor: Jugador

Comportamiento: El jugador activo elige donde poner una ficha.

Errores posibles y cursos alternativos:

- Colocar ficha en lugar erróneo: el sistema notifica al jugador y se solicita una nueva posición.

1.2.10. *Caso:* Jugar Partida

Actor: Jugador

Comportamiento: caso global de partida. El jugador puede pausar, guardar, hacer una jugada o salir de la partida (guardándose o no).

Errores posibles y cursos alternativos:

- Cerrar aplicación mientras se lleva a cabo el guardado de partida.

1.2.11. *Caso:* Salir

Actor: Jugador

Comportamiento: Permite al usuario abandonar por completo el sistema. Si el jugador se encuentra en una partida, el sistema pregunta si desea realmente abandonar el juego y si desea guardar la partida o no.

Errores posibles y cursos alternativos:

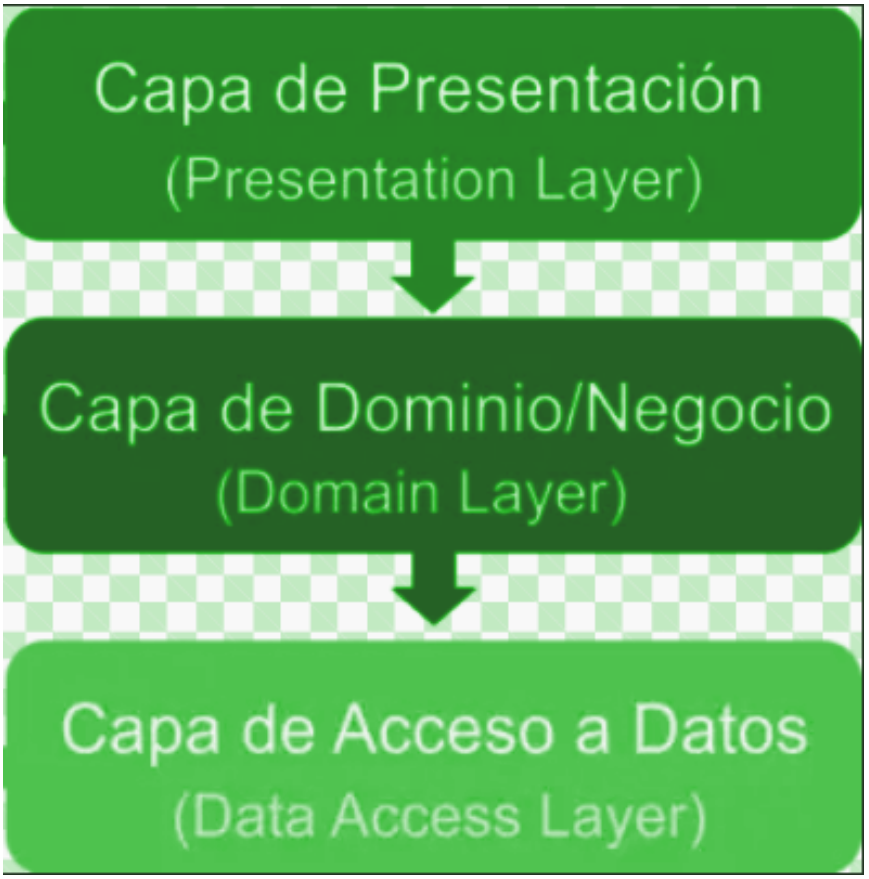
2. Arquitectura en 3 capas.

En este proyecto se aplica la arquitectura en 3 capas que consta de :

- Capa de Presentación: Responsable de la interacción con el usuario.

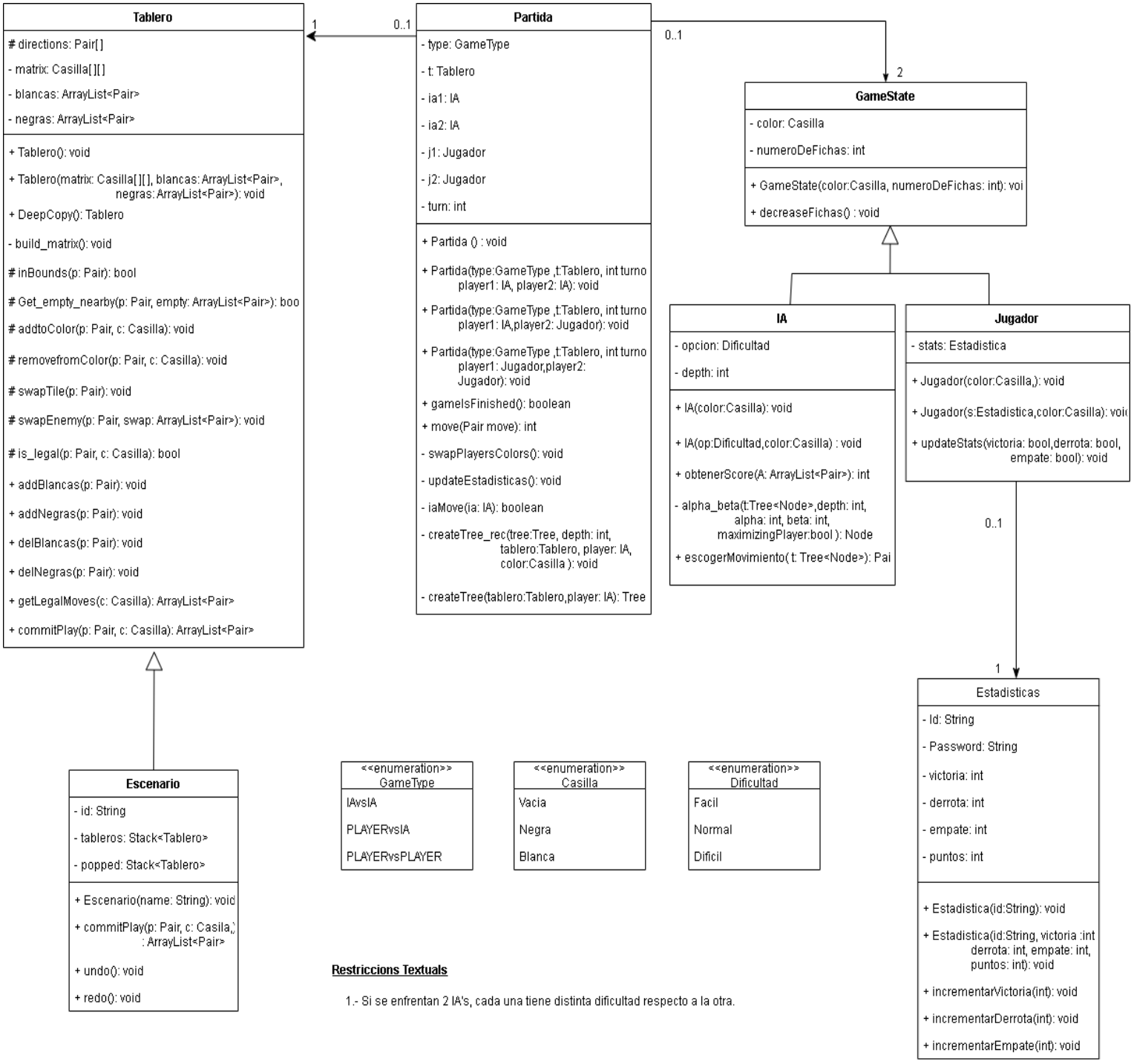
- Capa de Dominio: Contiene el núcleo del programa. Es la capa que sabe transformar y manipular los datos del usuario en los resultados que espera (la capa no "sabe", sin embargo, ni de dónde vienen estos datos ni dónde están almacenados).

- Capa de Gestión de Datos (o de Persistencia): Esta capa es la responsable de almacenar los datos. Sin embargo, ignora cómo tratarlos.



2.1. Capa de Dominio

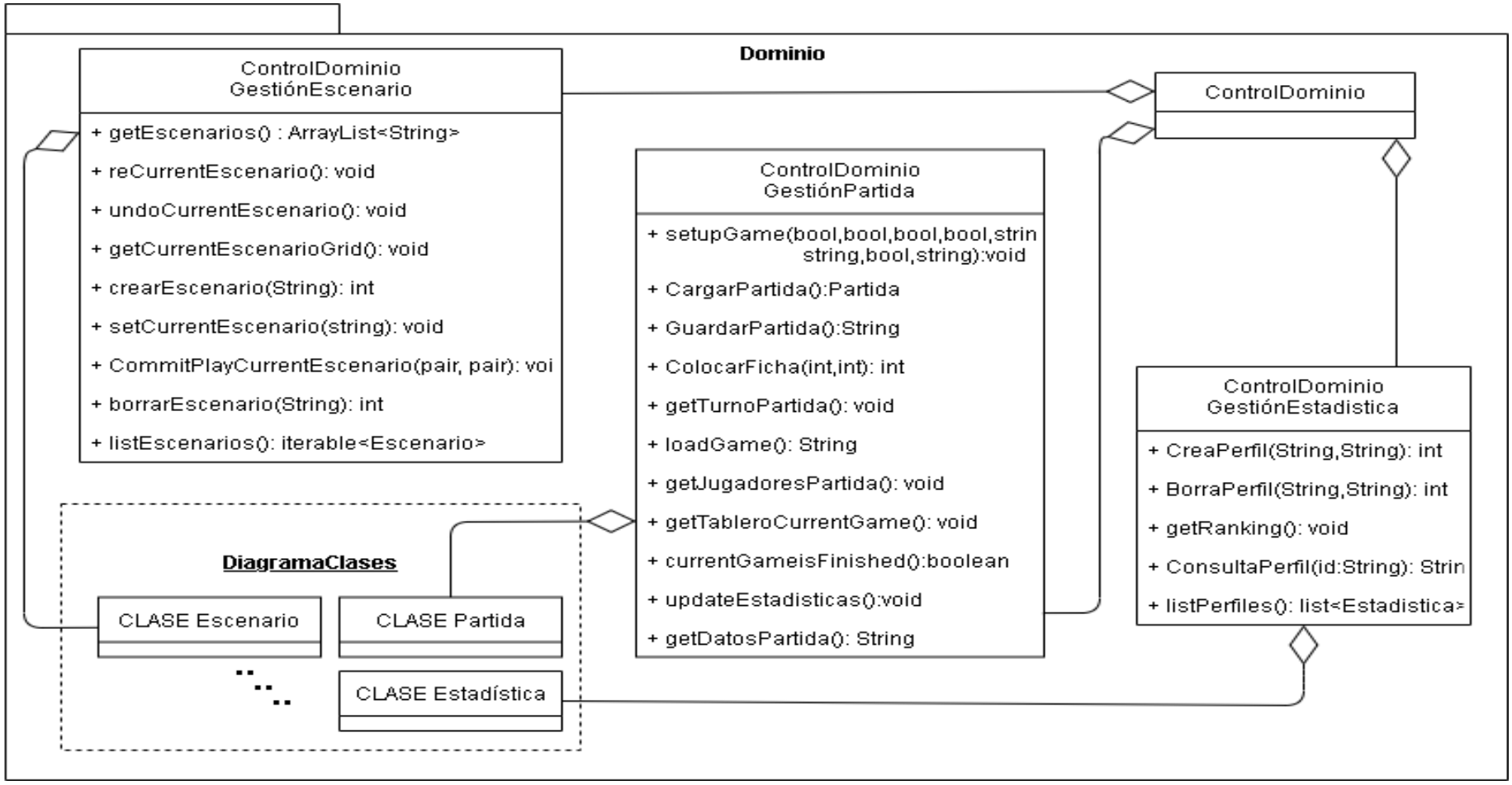
2.1.1. Diagrama de Clases



Restriccions Textuals

1.- Si se enfrentan 2 IA's, cada una tiene distinta dificultad respecto a la otra.

2.1.2. Diagrama del package de dominio con Controladores.



2.1.3. Descripción de Clases

Clase Tablero

Nombre de la clase: Tablero

Breve descripción de la clase: Simulador del tablero físico y gestor de las reglas que conciernen a la legalidad de las jugadas.

Cardinalidad: Uno por cada tablero, varios por cada escenario y uno por partida.

Descripción de los atributos:

- directions: direcciones posibles para consultar una posición del tablero.(static)
- matrix: matriz que contiene en cada casilla el color de la ficha correspondiente.(no static)
- blancas: Localización de fichas blancas con sus posiciones respectivamente.(no static)
- negras: Localización de fichas negras con sus posiciones respectivamente.(no static)

Descripción de las relaciones:

- Relación de asociación con la clase "Partida" : indica en qué partida se está utilizando el tablero.

Clase GameState

Nombre de la clase: GameState

Breve descripción de la clase: Clase abstracta que contiene las funcionalidades comunes de Jugador e IA.

Cardinalidad: Dos por cada partida.

Descripción de los atributos:

- Color: identificador de color de ficha (no static)
- NumeroDeFichas: identificador que determina el número de fichas.(no static)

Descripción de las relaciones:

- Relación de asociación con la clase "Partida": Indica cual es la partida en la que está participando.

Clase Partida

Nombre de la clase: Partida.

Breve descripción de la clase: En esta clase se simula con los atributos respectivos una partida de Othello.

Cardinalidad: Uno por cada Partida.

Descripción de los atributos:

- type: Tipo de partida que se simulará (IA vs IA, IA vs jugador, jugador vs jugador "no static").
- t: Tablero que se usará para simular el juego. (no static)
- ia1: Representa a una posible IA participante durante la partida.(no static)
- ia2: Representa a una posible IA participante durante la partida.(no static)
- j1: Representa a un posible jugador participante durante la partida.(no static)
- j2: Representa a un posible jugador participante durante la partida.(no static)
- turn: Determina la ronda actual en la partida.

Descripción de las relaciones:

- Relación de asociación con la clase "Tablero": indica qué tablero se está usando en la partida.
- Relación de asociación con la clase "GameState": indica quienes participan (IA / jugador) en la partida de Othello.

Clase Escenario

Nombre de la clase: Escenario

Breve descripción de la clase: Clase que hereda de la clase Tablero. Permite la edición de un tablero mediante un sistema de jugadas y la opción de deshacer y rehacer una acción.

Cardinalidad: Una por instancia. Tantas como se generan.

Descripción de los atributos:

- id : identificador del escenario(no static).
- tableros : pila que contiene las jugadas realizadas(no static).
- popped : pila que contiene las jugadas deshechas. (se usará para deshacer o rehacer jugadas.(no static)

Descripción de las relaciones:

- Relación de asociación con la clase Partida: Indica en qué partida se está utilizando el tablero más reciente del escenario.

Clase IA

Nombre de la clase: IA

Breve descripción de la clase: Clase que hereda de la clase GameState, y que gestiona las operaciones necesarias para que la inteligencia artificial realice sus movimientos.

Cardinalidad: Uno por cada IA, hasta dos por partida

Descripción de los atributos:

- dificultad: nivel de dificultad de la IA (fácil, normal, difícil). (no static)
- depth: valor asociado al nivel de dificultad de cada IA . (no static)

Descripción de las relaciones:

- Relación de asociación con clase Partida: Indica en qué partida participa la IA.

Clase Jugador

Nombre de la clase: Jugador

Breve descripción de la clase: Clase que hereda de la clase GameState, y que representa a una persona.

Cardinalidad: Uno por cada Jugador, hasta dos por partida

Descripción de los atributos:

- stats : contiene el id de jugador y sus estadísticas(victorias,derrotas,empates, puntos). (no static)

Descripción de las relaciones:

- Relación de asociación con clase Partida: Indica en qué partida participa el jugador.

Clase Estadística

Nombre de la clase: Estadística

Breve descripción de la clase: Permite la identificación de un usuario y almacenar sus resultados en partidas

Cardinalidad: Uno por instancia, tantos como se generan.

Descripción de los atributos:

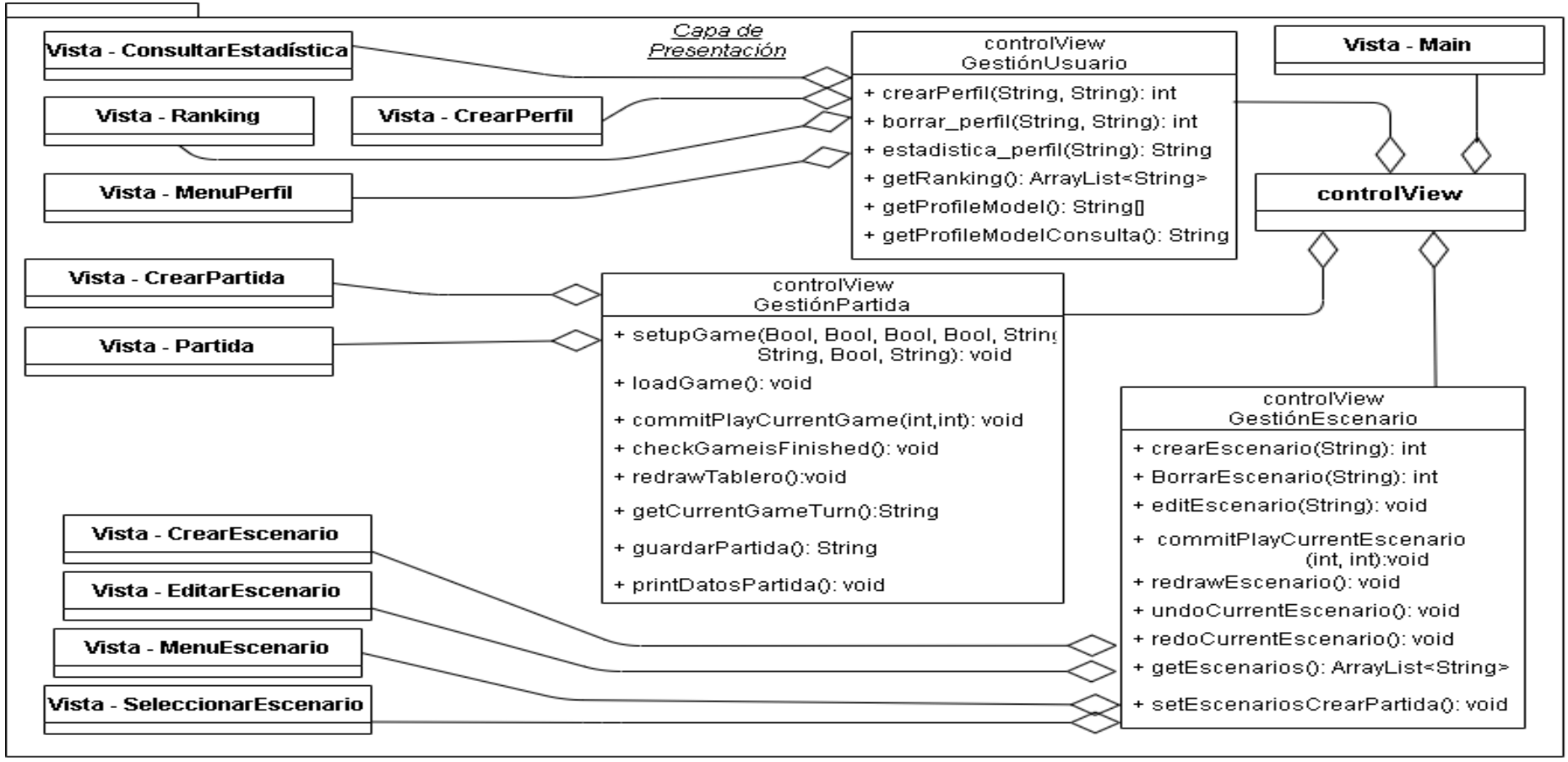
- Id: identificación del usuario.
- Password: Contraseña personal del usuario.
- Puntos: Cuantificación de los resultados en partidas realizadas por el usuario.
- Victoria: Cantidad de victorias en partidas.
- Derrotas: Cantidad de derrotas en partidas.
- Empates: Cantidad de empates en partidas.

Descripción de las relaciones:

- Relación de asociación con clase Jugador: Indica a qué jugador le pertenece la estadística.

2.2. Capa de Presentación

2.1.1. Diagrama del package de Presentación con controladores



2.1.2. Descripción de Clases

Controlador: **control View - Gestión Usuario**

Este controlador se encarga de gestionar todas las funcionalidades del usuario, tales como:

- Creación de perfil.
- Eliminación del perfil.
- Consulta de un perfil determinado.
- Obtención del ranking actualizado.
- Actualización de las estadísticas de cada perfil.

Controlador: **control View - Gestión Escenario**

Este controlador se encarga de gestionar todas las funcionalidades de un escenario, es decir, un tablero que le da la capacidad de modificarse al usuario, para futuras partidas. Las funcionalidades son las siguientes:

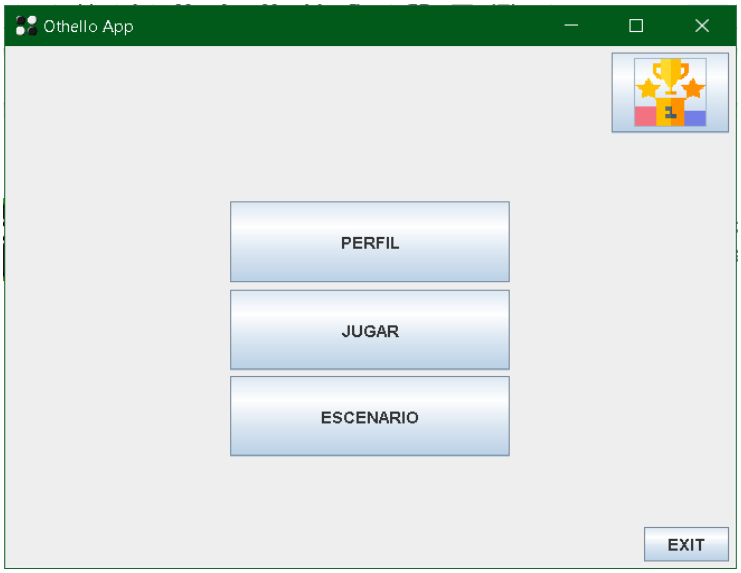
- Creación de un escenario.
- Carga de un escenario ya existente.
- Edición de un escenario.
- Confirmar movimiento del escenario actual.
- Obtener la lista de todos los escenarios.
- Deshacer un movimiento en un escenario.
- Rehacer un movimiento en un escenario.

Controlador: **control View - Gestión Partida**

Este controlador se encarga de gestionar todas las funcionalidades de una partida, es decir controlar que las reglas del juego se cumplan así como de verificar el término de esta. Las funcionalidades son las siguientes:

- Iniciar una Partida.
- Cargar una partida guardada.
- Guardar una partida.
- Escoger un escenario para iniciar una partida.
- Realizar un movimiento en una partida.
- Determinar los jugadores y el color de ficha para cada uno en la partida.
- Obtener los datos de una partida.

2.1.3. Breve descripción de las vistas.

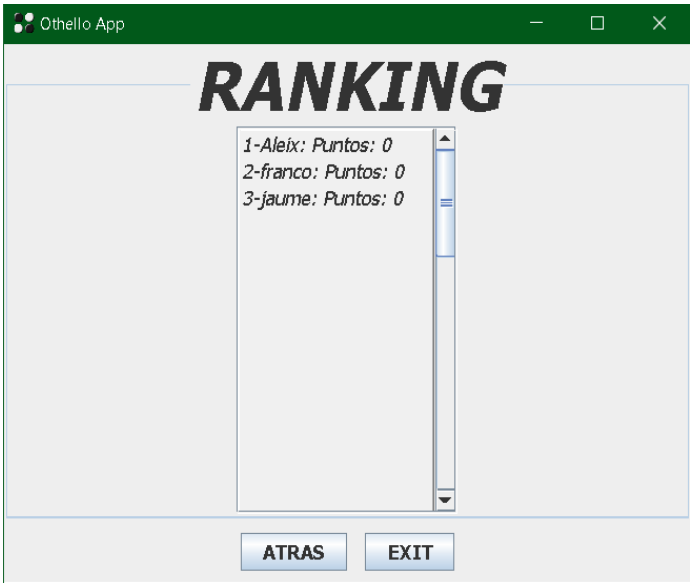


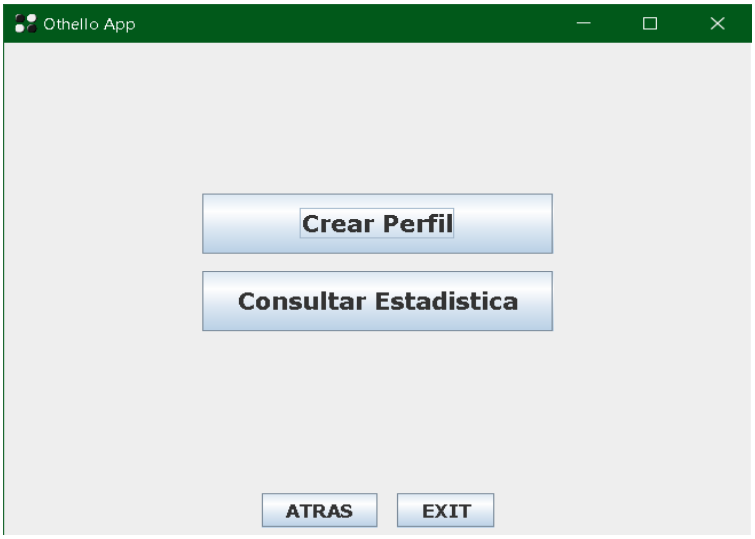
Vista Main(inicio):

El usuario tiene la opción de consultar el Ranking que está determinado por el botón de la derecha superior; también, el ingresar a un menú Perfil, iniciar una partida, ingresar a la sección de escenarios, y por último la opción de salir de la aplicación.

Vista Ranking:

El usuario puede visualizar el ranking actualizado, en la que se muestra la lista de los usuarios registrados en el sistema con sus puntos correspondientes y ordenados descendentemente respecto a sus puntos.





Vista Menú Perfil :

el usuario puede elegir la opción de crear un perfil o de consultar las estadísticas de un perfil existente en el sistema. En esta etapa el usuario también puede abandonar la aplicación.

Vista Crear Perfil:

El usuario ingresa un "nombre" y una "contraseña" para la creación de un perfil si el "nombre" ya existe en el sistema, este le emitirá un mensaje, invalidando la acción. Si no se rellena algún campo el sistema le solicitará los datos. Por último si los datos son ingresados correctamente y el "nombre" no existe en el sistema se dará de alta a un nuevo perfil.

Vista Consultar Estadística:

El usuario puede elegir de una lista desplegable cualquier usuario que está registrado en el sistema con la finalidad de obtener todas sus estadísticas desde la creación de de dicha cuenta. A su vez se puede borrar al usuario del sistema, para ello el sistema solicitará la misma contraseña que se utilizó para la creación del perfil.

Vista Crear Partida:

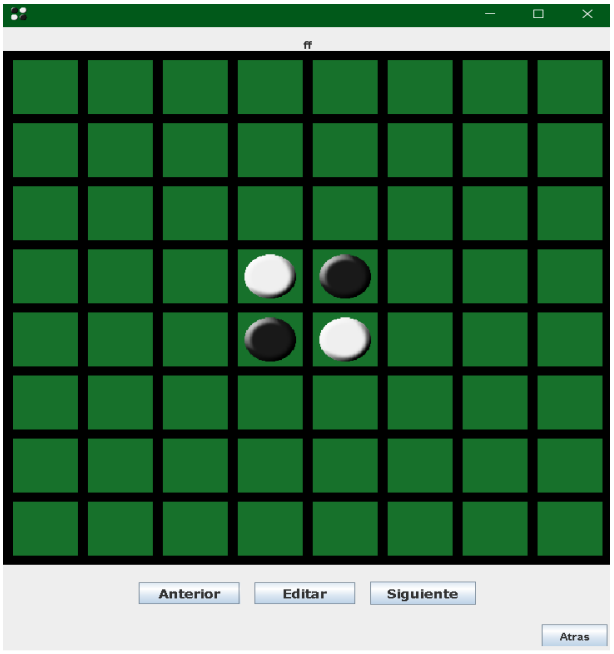
El/Los usuario(s) tienen la posibilidad de elegir un perfil previamente registrado en el sistema o caso contrario el sistema le asignará uno(GUEST). A su vez se puede elegir la opción de enfrentarse contra una IA con 3 dificultades a elegir(facil, normal,dificil), tambien esta la opción de elegir el color de ficha y determinar si se quiere iniciar una partida desde cero(standar) o iniciar una partida desde un escenario previamente creado, es decir ya existente en el sistema.

Vista Menú Escenario:

El usuario tiene la posibilidad de de crear un escenario desde cero con las reglas de juego respectivas, o de modificar un escenario ya existente

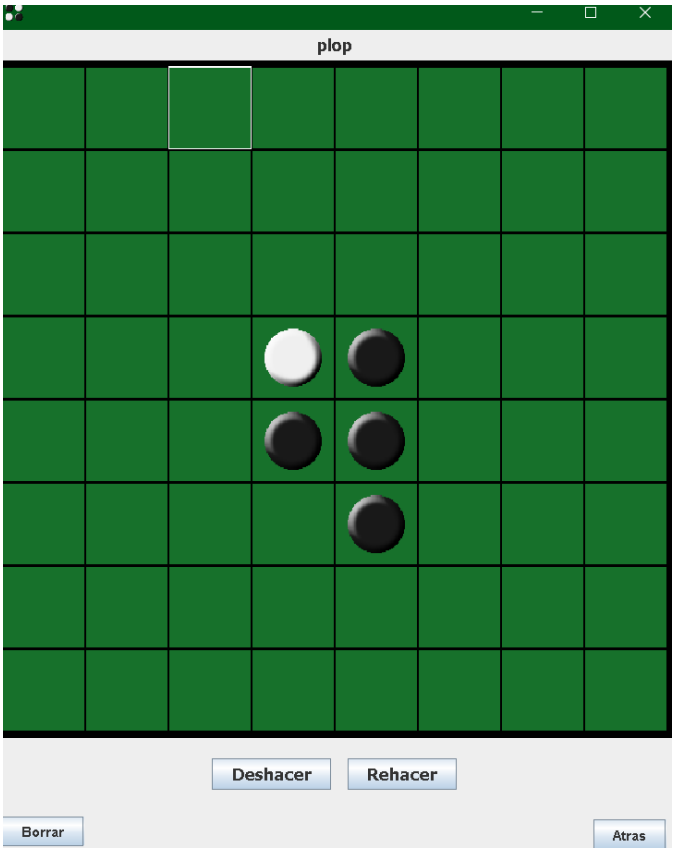
Vista Crear Escenario:

El sistema le pedirá al usuario un nombre que servirá para identificar al escenario a crear.



Vista Seleccionar Escenario:

El usuario puede escoger uno de los todos los escenarios disponibles en el sistema, para a continuación poder editarlos.

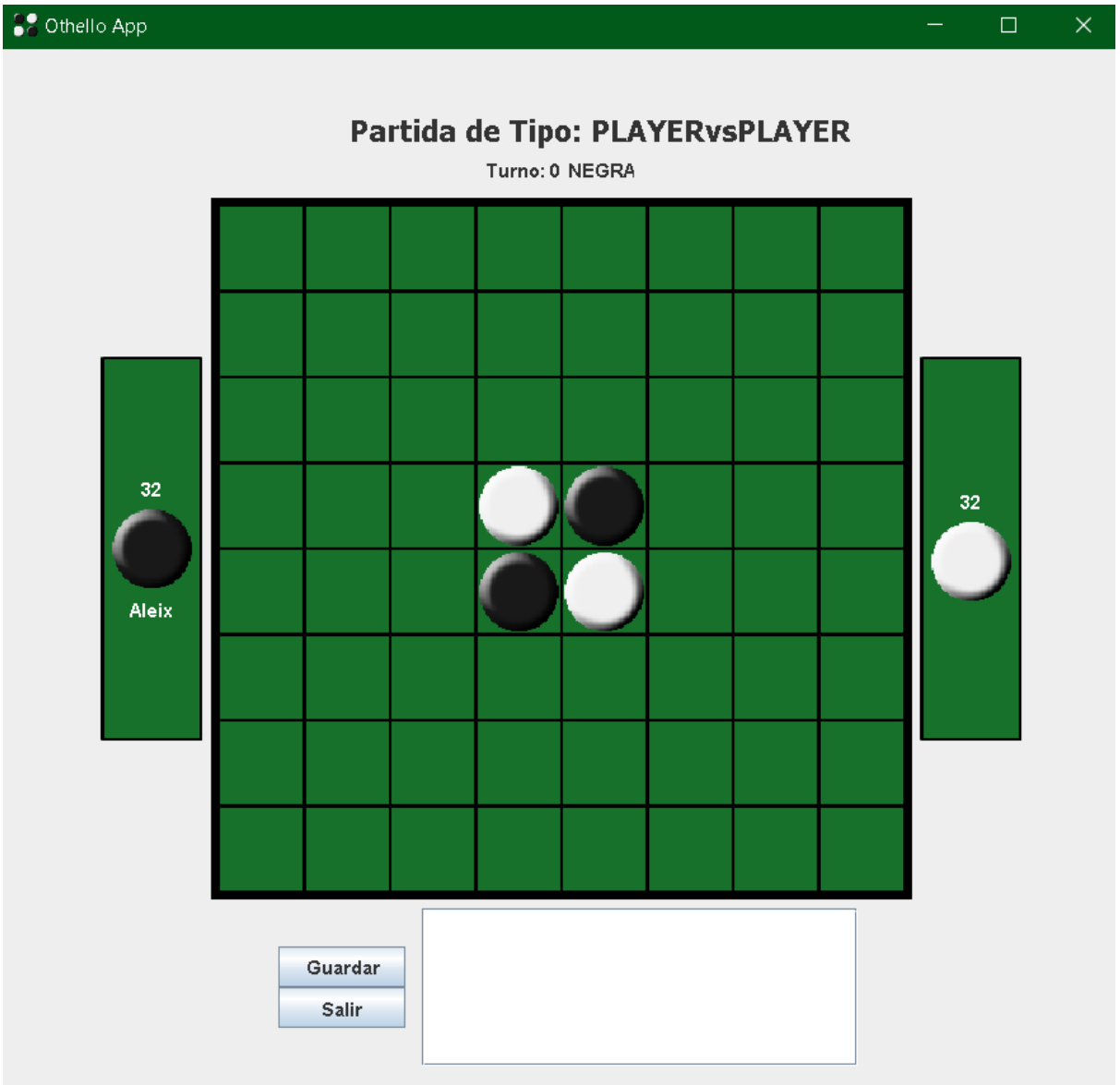


Vista Editar Escenario:

El usuario puede ir modificando el tablero colocando las fichas blancas y negras intercaladamente, siempre cumpliendo con las normas del juego. En esta parte de la edición se puede deshacer un movimiento como a su vez se puede rehacer dicho movimiento.

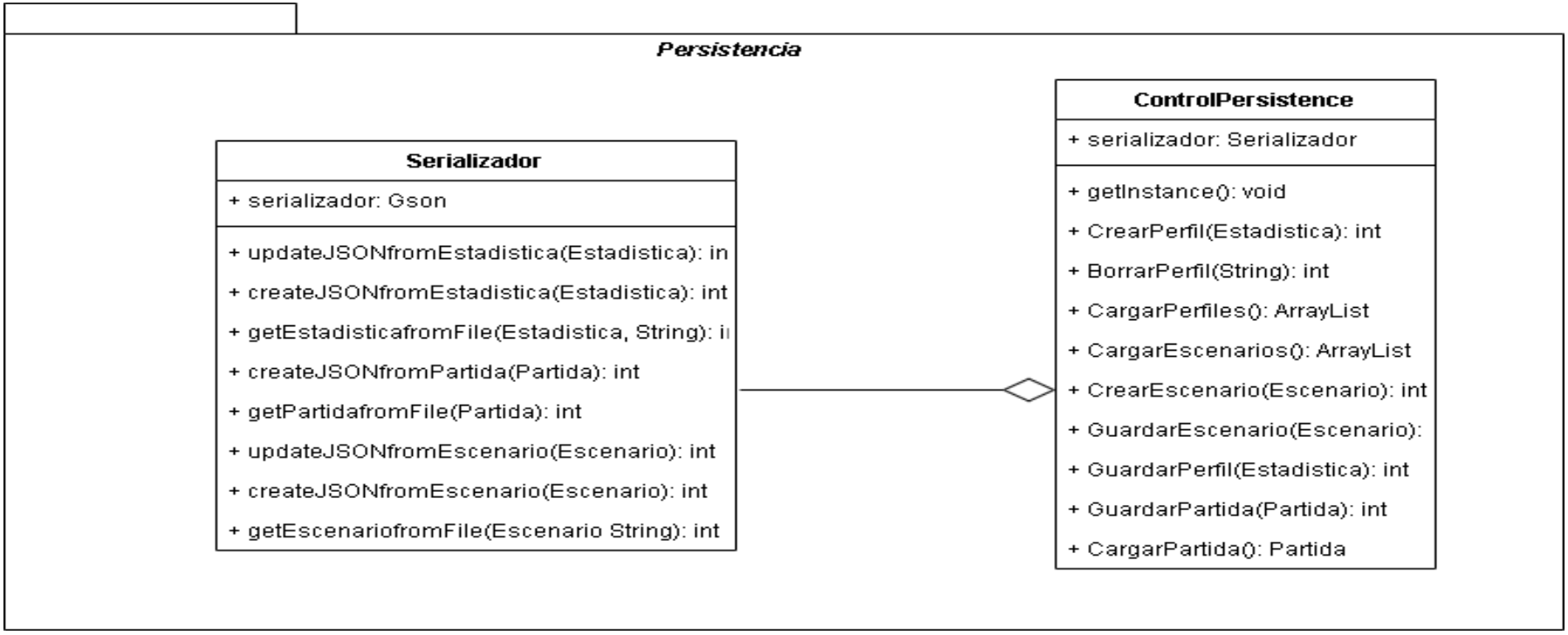
Vista Partida:

En esta vista el/los usuario(s) o la IA , van colocando las fichas en el tablero cumpliendo las normas del juego. Cuando el tablero esté completo o ambos jugadores no tengan algún movimiento disponible se determinará como ganador al jugador que posea la mayor cantidad de fichas de su color sobre el tablero. Posteriormente las estadísticas se actualizarán en el caso de los usuarios.



2.3. Capa de Persistencia.

2.3.1. Diagrama del package de persistencia con controladores.



2.3.2. Descripción de clases

Controlador: **Persistence**

Se relaciona con la capa de dominio recibiendo las operaciones de consulta y actualización de los datos, y retornando respuestas y resultados de estas peticiones. Se relaciona con el sistema de gestión de ficheros pasando las operaciones de consulta y/o actualización de los datos en el formato y lenguaje adecuados y recibiendo las respuestas y resultados. Además permite que determinados objetos del dominio sean persistentes y que el dominio ignore dónde se encuentran los datos. Contiene las siguientes funcionalidades:

- Creación de perfil.
- Guardar perfil.
- Borrar perfil.
- Cargar todos los perfiles.
- Crear escenario.
- Cargar escenario.
- Guardar escenario.
- Cargar partida.
- Guardar partida.

Clase: **Serializador**

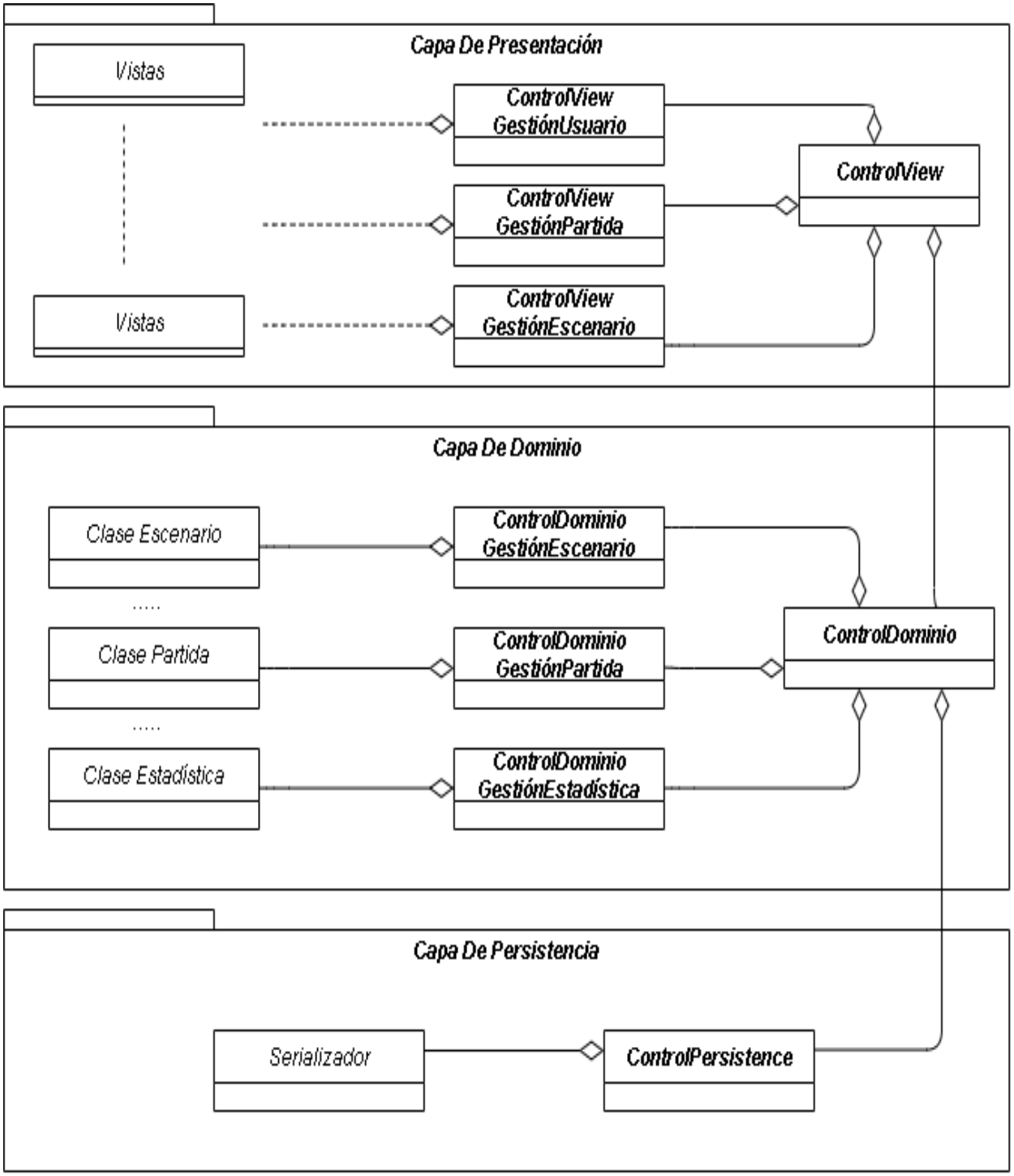
En este proyecto usamos Gson (también conocido como Google Gson) que es una biblioteca de código abierto para el lenguaje de programación Java que permite la serialización y deserialización entre objetos Java y su representación en notación JSON con las siguientes características:

- ➢ Permite la conversión entre objetos Java y JSON de una manera sencilla, simplemente invocando los métodos `toJson()` o `fromJson()`.
- ➢ Permite la conversión de objetos inmutables ya existentes.
- ➢ Soporte para tipos genéricos de Java.
- ➢ Permite la representación personalizada de objetos.
- ➢ Soporte para "Objetos arbitrariamente complejos".

Esta clase contiene las siguientes funcionalidades:

- Actualizar estadística de perfil.
- Crear perfil
- Obtener los datos del perfil
- Guardar la partida
- Cargar la partida
- Crear escenario
- Actualizar escenario
- Cargar escenario

2.4. Diagrama completo 3 capas con controladores.



3. Descripción de estructuras de datos y algoritmos.

3.1. Clase Pair

Nombre de la clase: Pair

Breve descripción de la clase: Utilizada para gestionar las coordenadas de la matriz (posiciones en el tablero).

Descripción de los atributos:

- First: Coordenada x.
- Second: Coordenada y.

Descripción de funcionalidades principales:

Sum(Pair p)/Sub(Pair p): Permite sumar/restar una coordenada con otra.

3.2. Enum Casilla

Nombre: Casilla.

Breve descripción de la clase: La matriz del tablero está formada por estos. Simulan el estado de una casilla (ocupada por blanco, negro, o sin ocupar).

Descripción funcionalidades principales:

this.contrary(): devuelve el color contrario al de la instancia: Negro -> Blanco y viceversa.

3.3. Clase Node

Nombre de la clase: Node

Breve descripción de la clase: Clase usada para almacenar la información necesaria para el algoritmo de la IA.

Descripción de los atributos:

- Pair Coord: La coordenada de la jugada que se va a evaluar
- Casilla Player: Color de la jugada que se evalúa.
- Score: Puntuación asociada a la jugada mediante una heurística.

3.4. Clase Tree

Nombre de la clase: Tree<T>

Breve descripción de la clase: Clase genérica que permite almacenar objetos de clase T en su interior en formato de árbol N-ario.

Descripción de los atributos:

- root: Objeto de clase T almacenado. Representa el valor del nodo.
- leafs: Lista de Tree's. Representa los nodos hijos (adyacentes a this).
- parent: Tree que representa el nodo padre del mismo (si es la raíz, éste será null).
- locate: Mapa de hash que permite el acceso rápido a los nodos.

Descripción de funcionalidades principales:

- Collection<Tree> getSubTrees(): devuelve el atributo "leafs" para permitir navegar por la estructura.
- static <T> Collection<T> getSuccessors(T of, Collection<Tree<T>> in): Busca y devuelve los objetos de clase T en los hijos del nodo.

3.5. Comprobar legalidad de jugadas

Para comprobar que las jugada se atiene a las normas de la partida, hemos de comprobar que, al colocar una ficha, ésta "encierra" al menos una ficha del color contrario. Con encierra nos referimos a que dicha ficha quede entre dos fichas de color contrario en alguna dirección recta. Para comprobar esto recorremos el tablero desde la posición en la que se quiere realizar la jugada, y se comprueba que haya alguna adyacente del color contrario. Si la condición contraria se cumple, hay que comprobar que la siguiente posición después de salir de la serie de fichas del color contrario sea del mismo color que la jugada.

Representación:

- - - J O O O X

- - - - - - - -

J sería una jugada correcta si J es de "color X".

- - - J O O O -

- - - - - - - -

En cambio, J no sería adecuada aquí, pues aunque encuentra una ficha del color contrario, al final de esta hay una posición vacía.

Para ello implementamos el siguiente código:

```
protected boolean is_legal(Pair p, Casilla c) {
        if (matrix[p.first()][p.second()] != Casilla.VACIA) {return false;}
        for (Pair dir : directions) {
                Pair start = p.sum(dir);
                if (inBounds(start) && matrix[start.first()][start.second()] == c.contrary()) {
                        while (inBounds(start)) {
                            if (matrix[start.first()][start.second()] == c) {return true;}
                            if (matrix[start.first()][start.second()] == Casilla.VACIA) {break;}
                            start = start.sum(dir);
                        }
                }
        }
        return false;
}
```

3.6. Buscar jugadas legales

Para buscar jugadas legales para un color, el algoritmo consiste en el siguiente concepto:

La base de la legalidad de una jugada reside en que la ficha que se quiera poner sea adyacente a una del color contrario, así que sólo hay que buscar las posiciones adyacentes al color contrario a la jugada que estén vacías. Una vez se tienen las posiciones vacías hay que descartar las ilegales.

Para ello implementamos el siguiente código:

```
public ArrayList<Pair> getLegalMoves(Casilla c) {
        ArrayList<Pair> positions;
        if (c == Casilla.BLANCA) {
                positions = (ArrayList<Pair>) negras.clone();
        } else  positions = (ArrayList<Pair>) blancas.clone();
        ArrayList<Pair> Legals = new ArrayList<Pair>(0);
        for (int i = 0; i < positions.size(); ++i) {
                Get_empty_nearby(positions.get(i), Legals);
        }
        int s = Legals.size();
        if (!Legals.isEmpty()) {
                for (int j = s-1; j >= 0; --j) {
                        if (!is_legal(Legals.get(j), c)) { Legals.remove(j); }
                }
        }
        return Legals;
}
```

3.7. Realizar una jugada

Para poder realizar una jugada (asumiendo que ésta es legal), hay que colocar la ficha en la posición deseada y cambiar el color de todas las fichas enemigas que queden "encerradas". Esto consiste en una idea parecida a la comprobación de la legalidad de una jugada, pero en este caso no hay que salir cuando se encuentra una dirección que cumpla la premisa. Lo que hacemos consiste en: una vez compruebas que una dirección es válida, damos marcha atrás y cambiar los colores de las fichas enemigas en el camino. Ésto último se repite por cada dirección. Hay que tener en cuenta que los cambios de color no se pueden realizar hasta que se han comprobado todas las direcciones, pues si editas simultáneamente puede causar problemas.

Para ello implementamos el siguiente código:

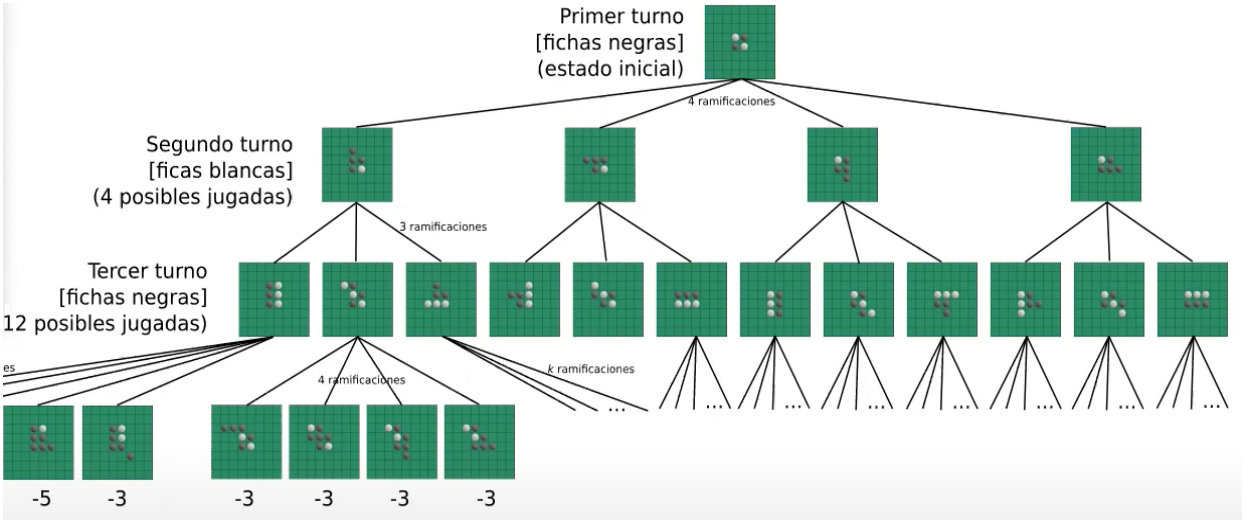
```
public ArrayList<Pair> commitPlay(Pair p, Casilla c) {
        ArrayList<Pair> swaps = new ArrayList<>(0);
        if (inBounds(p) && is_legal(p, c)) {
                matrix[p.first()][p.second()] = c;
                addtoColor(p, c);
                swaps.add(p);
                swapEnemy(p, c, swaps);
        }
        return swaps;
}
protected void swapEnemy(Pair p, Casilla c, ArrayList<Pair> swaps) {
        if (matrix[p.first()][p.second()] == c) {
           for (Pair dir : directions) { Pair start = p.sum(dir);
                if (inBounds(start) && matrix[start.first()][start.second()] == c.contrary()) {
                    while(inBounds(start)){
                        if (matrix[start.first()][start.second()] == c) {
                                start = start.sub(dir);
                                while(!start.convertString().equals(p.convertString())) {
                                    removefromColor(start   ,c.contrary());   addtoColor(start,
                                    c); swaps.add(start); start = start.sub(dir);
                                } break;
                        }
                        if (matrix[start.first()][start.second()] == Casilla.VACIA) { break; }
                        start = start.sum(dir);
                    }
                }
           }
           for(int i = 1; i < swaps.size(); ++i) { swapTile(swaps.get(i)); }
        }
}
```

```
protected boolean inBounds(Pair p) {
      if (p.first() >= 0 && p.first() < 8 && p.second() >= 0 && p.second() < 8) {
         return true;
      } else {  return false; }
}

protected void swapTile(Pair p) {
      Casilla c = matrix[p.first()][p.second()];
      matrix[p.first()][p.second()] = c.contrary();
}

protected void addtoColor(Pair p, Casilla c) {
      if(c == Casilla.BLANCA) { blancas.add(p); }
      else if(c == Casilla.NEGRA) {  negras.add(p); }
 }

protected void removefromColor(Pair p, Casilla c) {
      if(c == Casilla.BLANCA) {  blancas.remove(p); }
      else if(c == Casilla.NEGRA) { negras.remove(p); }
}
```

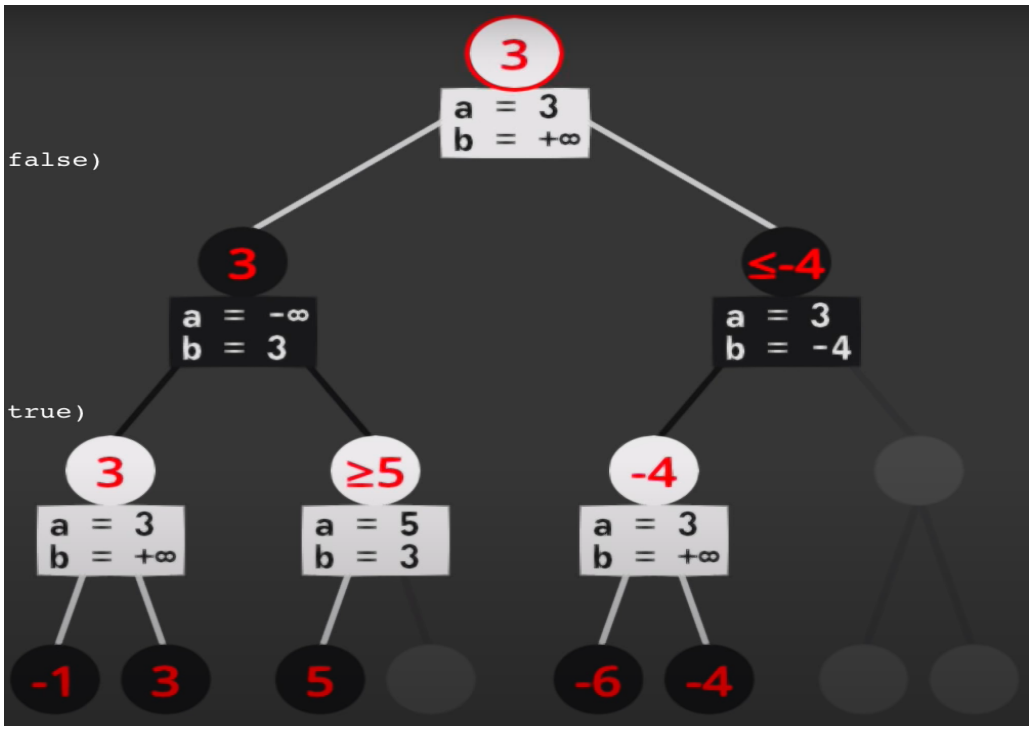
3.8. Escoger jugada IA

Para que la IA pueda escoger un movimiento necesitamos que pueda hacer una evaluación sobre todos los que tiene disponibles. Para ello, intentamos que haga una "predicción" sobre cada jugada, y con esto escoja la que más beneficio traiga a futuro. El modo en que hacemos esto consiste en, por cada jugada disponible, la simula (movimiento A) e intenta predecir el movimiento del enemigo(movimiento B). Sobre el estado del tablero donde A y B ya han ocurrido, intenta hacer una jugada más, y de estas jugadas se selecciona la jugada más óptima. Para esto usamos el algoritmo Minimax Alpha Beta para minimizar pérdidas. El algoritmo consiste en generar el peor caso posible para sí mismo, y de esta situación escoger el movimiento que más beneficio le traiga.

En esta imagen vemos cómo se van generando el árbol simulando cada jugada para que así dependiendo de la altura del árbol llegar a los nodos hijos y empezar a realizar el algoritmo ALFA - BETA.



En la imagen a continuación mostramos un breve ejemplo de cómo el algoritmo funciona y va podando a medida que va realizando la evaluación de cada nodo.



Para ello implementamos el siguiente código:

```
private boolean iaMove(IA ia) {
        Tree jugadas = createTree(t, ia);
        Pair c = ia.escogerMovimiento(jugadas);
        return !(t.commitPlay(c, ia.getColor()).isEmpty());
}
```

```
Tree createTree(Tablero tablero, IA ia) {
        Tree jugadas = new Tree(new Node());
        createTree_rec(jugadas, ia.getDepth(), tablero, ia, ia.getColor());
        return jugadas;
}
```

```
private static void createTree_rec(Tree tree, int depth, Tablero tablero, IA player, Casilla color) {
        if (depth >= 0) {
                ArrayList<Pair> legalMoves = tablero.getLegalMoves(color);
                for (Pair p : legalMoves) {
                        Tablero tDeepCopy = tablero.DeepCopy();
                        ArrayList<Pair> pa = tDeepCopy.commitPlay(p, color);
                        Tree tr;
                        int score = player.obtenerScore(pa);
                        tr = tree.addLeaf(new Node(p, color, score));
                        createTree_rec(tr, depth - 1, tDeepCopy, player, color.contrary());
                }
        }
}
```

```
public Pair escogerMovimiento(Tree<Node> t) {
        Node maxEval = new Node(NEGATIVE_INFINITY);
        for (Tree<Node> child : t.getSubTrees()) {
                Node eval = alpha_beta(child, this.depth, NEGATIVE_INFINITY,
                POSITIVE_INFINITY, true);
                if(maxEval.getScore() < eval.getScore()) {
                        maxEval.setScore(eval.getScore());
                        maxEval.setCord(child.getRoot().getCord());
                }
        }
        Node movimiento = maxEval;
        return movimiento.getCord();
}
private Node alpha_beta(Tree<Node> t, int depth, int alpha, int beta, boolean
maximizingPlayer) {
        if (depth == 0 || t.getSubTrees().isEmpty()) {
                return t.getRoot();
        }
        if (maximizingPlayer) {
            Node maxEval = new Node(NEGATIVE_INFINITY);
            for (Tree<Node> child : t.getSubTrees()) {
                Node eval = alpha_beta(child, depth - 1, alpha, beta, false);
                int scr = ((maxEval.getScore() > eval.getScore()) ? maxEval.getScore() :
                eval.getScore());
                maxEval.setScore(scr);
                alpha = ((alpha > eval.getScore()) ? alpha : eval.getScore());
                if (beta <= alpha) { break; }
            }
            return maxEval;
        }
```

```
        else {
            Node minEval = new Node(POSITIVE_INFINITY);
            for (Tree<Node> child : t.getSubTrees()) {
                Node eval = alpha_beta(child, depth - 1, alpha, beta, true);
                int scr = ((minEval.getScore() < eval.getScore()) ? minEval.getScore() :
                eval.getScore());
                minEval.setScore(scr);
                beta = ((beta < eval.getScore()) ? beta : eval.getScore());
                if (beta <= alpha) { break; }
            }
            return minEval;
        }
}
```

4. Juegos de Prueba.

4.1. Clase Tablero

- **Prueba 1**

Descripción: Comprobar si una jugada es legal.

Objetivos: Ver si la comprobación se hace correctamente.

Otros elementos integrados en la prueba:

import othello.data.Casilla;

import othello.data.Pair;

Entrada:

2 3

1 (Negra)

Salida:

correct

Resultado de la prueba: ok

- **Prueba 2**

Descripción: Comprobar si una jugada es legal.

Objetivos: Ver si la comprobación se hace correctamente.

Otros elementos integrados en la prueba:

import othello.data.Casilla;

import othello.data.Pair;

Entrada:

1 1

1 (Negra)

Salida:

Incorrect

Resultado de la prueba: ok

- **Prueba 3**

Descripción: Comprobar si una jugada es legal.

Objetivos: Ver si la comprobación se hace correctamente.

Otros elementos integrados en la prueba:

import othello.data.Casilla;

import othello.data.Pair;

Entrada:

-111111 11111111

1 (Negra)

Salida:

Incorrect

Resultado de la prueba: ok

- **Prueba 4**

Descripción: Conseguir jugadas legales.

Objetivos: Conseguir todas las jugadas legales para un color en un determinado estado de tablero.

Otros elementos integrados en la prueba:

import othello.data.Casilla;

import othello.data.Pair;

Entrada:

1 (Negra)

new Tablero();

Salida:

[2/3, 3/2, 5/4, 4/5]

correct

Resultado de la prueba: ok

4.2. Clase Ctrl Domain

- **Prueba 1**

Descripción: Crear un perfil.

Objetivos: Crear un perfil exitosamente.

Otros elementos integrados en la prueba:

import othello.domain.CtrlDomain;

Entrada: alias

Salida: alias.json

correct

Resultado de la prueba: ok

- **Prueba 2**

Descripción: Crear un perfil.

Objetivos: Crear un perfil exitosamente.

Otros elementos integrados en la prueba:

import othello.domain.CtrlDomain;

Entrada: alias

Salida: alias ya existe, prueba de nuevo

Incorrect

Resultado de la prueba: ok

- **Prueba 3**

Descripción: Borrar un perfil.

Objetivos: Borrar un perfil exitosamente.

Otros elementos integrados en la prueba:

Estadística

Entrada: alias

Salida: alias borrado correctamente

correct

Resultado de la prueba: ok

- **Prueba 4**

Descripción: Crear un perfil.

Objetivos: Borrar un perfil exitosamente.

Otros elementos integrados en la prueba:

Estadística

Entrada: alias

Salida:

alias no existe.

Incorrect

Resultado de la prueba: ok

- **Prueba 5**

Descripción: Imprimir el ranking.

Objetivos: Imprimir el ranking ordenado en orden descendiente por score.

Otros elementos integrados en la prueba:

Estadística

Entrada: alias1.json(5 score), alias2.json(3 score), alias3.json(0 score)

Salida: alias1: Puntos: 5

Victorias: 2

Derrotas: 1

Empates: 0

alias2: Puntos: 3

Victorias: 1

Derrotas: 0

Empates: 0

alias 3: Puntos: 0

Victorias: 0

Derrotas: 2

Empates: 0

correct

Resultado de la prueba: ok

4.3. Clase Escenario

- **Prueba 1**

Descripción: Undo.

Objetivos: deshacer una acción hecha previamente.

Otros elementos integrados en la prueba:

import othello.data.Casilla;

import othello.data.Pair;

import java.util.ArrayList;

Entrada: Undo

Salida:

Tablero previo

Correct

Resultado de la prueba: ok

- **Prueba 2**

Descripción: Undo.

Objetivos: deshacer una acción hecha previamente.

Otros elementos integrados en la prueba:

import othello.data.Casilla;

import othello.data.Pair;

import java.util.ArrayList;

Entrada: Undo en tablero base

Salida:

Tablero base

Incorrect

Resultado de la prueba: ok

- **Prueba 3**

Descripción: Redo.

Objetivos: Rehacer una acción deshecha previamente.

Otros elementos integrados en la prueba:

import othello.data.Casilla;

import othello.data.Pair;

import java.util.ArrayList;

Entrada: Redo

Salida:

Tablero posterior

Correct

Resultado de la prueba: ok

- **Prueba 4**

Descripción: Redo.

Objetivos: Rehacer una acción deshecha previamente.

Otros elementos integrados en la prueba:

import othello.data.Casilla;

import othello.data.Pair;

import java.util.ArrayList;

Entrada: Redo con Stack vacio

Salida:

Tablero sin cambios

Incorrect

Resultado de la prueba: ok

4.4. Clase Partida.

- **Prueba 1**

Descripción: PrintTree.

Objetivos: Mostrar el árbol de predicción en función del estado del tablero.

Otros elementos integrados en la prueba:

```
import othello.data.Casilla;
import othello.data.Pair;
import othello.domain.IA;
import othello.domain.tablero.Tablero;
import othello.domain.CtrlDomain;
```

Entrada: new Tablero()

Salida:

```
Node{cord=-1/-1, player=VACÍA, score=0}
Node{cord=2/4, player=BLANCA, score=25}
Node{cord=2/3, player=NEGRA, score=25}
Node{cord=4/2, player=BLANCA, score=35}
Node{cord=3/2, player=BLANCA, score=25}
Node{cord=5/2, player=BLANCA, score=25}
Node{cord=2/2, player=BLANCA, score=40}
Node{cord=1/2, player=BLANCA, score=1}
Node{cord=4/5, player=NEGRA, score=25}
Node{cord=5/3, player=BLANCA, score=25}
Node{cord=5/2, player=BLANCA, score=25}
Node{cord=5/4, player=BLANCA, score=25}
Node{cord=5/5, player=BLANCA, score=25}
Node{cord=5/6, player=BLANCA, score=1}
Node{cord=2/5, player=NEGRA, score=25}
Node{cord=5/3, player=BLANCA, score=25}
Node{cord=4/2, player=BLANCA, score=25}
Node{cord=3/5, player=BLANCA, score=25}
```

Node{cord=2/6, player=BLANCA, score=1}
Node{cord=3/5, player=BLANCA, score=25}
Node{cord=2/3, player=NEGRA, score=25}
Node{cord=4/2, player=BLANCA, score=25}
Node{cord=3/2, player=BLANCA, score=25}
Node{cord=5/2, player=BLANCA, score=25}
Node{cord=2/2, player=BLANCA, score=25}
Node{cord=1/2, player=BLANCA, score=1}
Node{cord=4/5, player=NEGRA, score=25}
Node{cord=5/3, player=BLANCA, score=35}
Node{cord=5/2, player=BLANCA, score=25}
Node{cord=5/4, player=BLANCA, score=25}
Node{cord=5/5, player=BLANCA, score=40}
Node{cord=5/6, player=BLANCA, score=1}
Node{cord=2/5, player=NEGRA, score=25}
Node{cord=5/3, player=BLANCA, score=25}
Node{cord=4/2, player=BLANCA, score=25}
Node{cord=1/5, player=BLANCA, score=1}
Node{cord=2/4, player=BLANCA, score=25}
Node{cord=5/3, player=BLANCA, score=25}
Node{cord=3/2, player=NEGRA, score=25}
Node{cord=2/4, player=BLANCA, score=25}
Node{cord=2/3, player=BLANCA, score=25}
Node{cord=2/5, player=BLANCA, score=25}
Node{cord=2/2, player=BLANCA, score=25}
Node{cord=2/1, player=BLANCA, score=1}
Node{cord=5/4, player=NEGRA, score=25}
Node{cord=3/5, player=BLANCA, score=35}
Node{cord=2/5, player=BLANCA, score=25}
Node{cord=4/5, player=BLANCA, score=25}
Node{cord=5/5, player=BLANCA, score=40}

Node{cord=6/5, player=BLANCA, score=1}
Node{cord=5/2, player=NEGRA, score=25}
Node{cord=2/4, player=BLANCA, score=25}
Node{cord=3/5, player=BLANCA, score=25}
Node{cord=4/2, player=BLANCA, score=25}
Node{cord=5/1, player=BLANCA, score=1}
Node{cord=4/2, player=BLANCA, score=25}
Node{cord=3/2, player=NEGRA, score=25}
Node{cord=2/4, player=BLANCA, score=35}
Node{cord=2/3, player=BLANCA, score=25}
Node{cord=2/5, player=BLANCA, score=25}
Node{cord=2/2, player=BLANCA, score=40}
Node{cord=2/1, player=BLANCA, score=1}
Node{cord=5/4, player=NEGRA, score=25}
Node{cord=3/5, player=BLANCA, score=25}
Node{cord=2/5, player=BLANCA, score=25}
Node{cord=4/5, player=BLANCA, score=25}
Node{cord=5/5, player=BLANCA, score=25}
Node{cord=6/5, player=BLANCA, score=1}
Node{cord=5/2, player=NEGRA, score=25}
Node{cord=2/4, player=BLANCA, score=25}
Node{cord=3/5, player=BLANCA, score=25}
Node{cord=6/2, player=BLANCA, score=1}
Node{cord=5/3, player=BLANCA, score=25}
Correct

Resultado de la prueba: ok

- **Prueba 2**

Descripción: PartdaIAvsIA.

Objetivos: Comprobar que una partida se desarrolla correctamente.

Otros elementos integrados en la prueba:

```
import othello.data.Casilla;
import othello.data.Pair;
import othello.domain.IA;
import othello.domain.tablero.Tablero;
import othello.domain.CtrlDomain;
```

Entrada:

```
new IA(Casilla.BLANCA)
new IA(Casilla.NEGRA)
new Tablero()
```

Salida:

la partida se desarrolla correctamente.

Resultado de la prueba: ok

4.5. Clase IA

- **Prueba 1**

Descripción: testScore.

Objetivos: Comprobar que al hacer una jugada, los swaps de color se reflejan correctamente con la heurística..

Otros elementos integrados en la prueba:

```
import java.util.ArrayList;
import othello.data.Casilla;
import othello.data.Node;
import othello.data.Pair;
import othello.data.Tree;
```

Entrada:

2 3

1 3

0 3

Salida:

9

Resultado de la prueba: ok

- **Prueba 2**

Descripción: testAlphaBeta.

Objetivos: Comprobar que la IA escoge correctamente la coordenada a partir del árbol

Otros elementos integrados en la prueba:

```
import java.util.ArrayList;
import othello.data.Casilla;
import othello.data.Node;
import othello.data.Pair;
import othello.data.Tree;
```

Entrada:

2 3 5

4 2 9

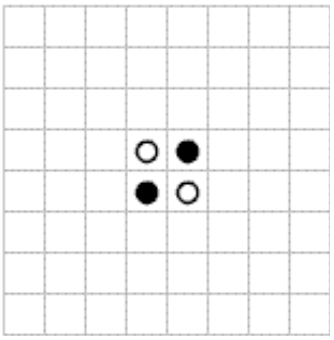
4 3 -1

Salida:

4/2

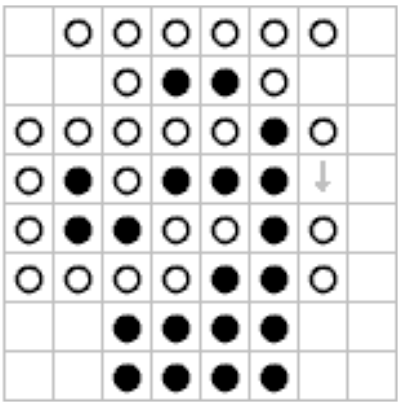
Resultado de la prueba: ok

5. Reglas del juego “Othello”



En el othello, se emplea un tablero de 8 filas por 8 columnas y 64 fichas idénticas, redondas, blancas por una cara y negras por la otra (u otros colores). A un jugador se le asigna un color y se dice que lleva las fichas de ese color, lo mismo para el adversario con el otro color.

En el othello, de inicio, se colocan cuatro fichas tal como se ve en el diagrama de la izquierda: dos fichas blancas y dos negras.

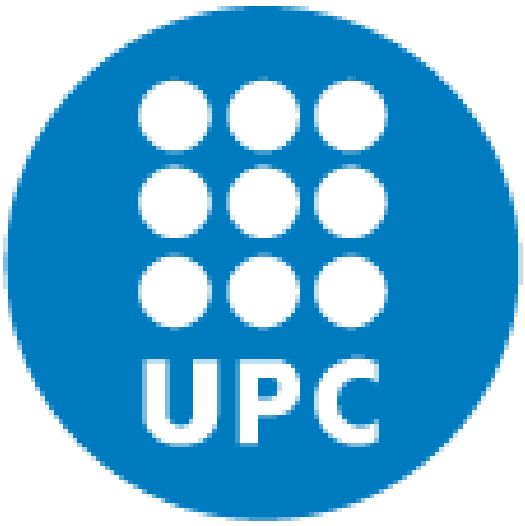


En el otelo se comparten las sesenta y cuatro fichas. Empezando por quien lleva las fichas negras los jugadores deben hacer un movimiento por turno, a menos que no puedan hacer ninguno, pasando en ese caso el turno al jugador contrario. El movimiento consiste en colocar una ficha de forma que flanquean una o varias fichas del color contrario y voltear esas fichas para que pasen a mostrar el propio color.

Se voltean todas las fichas que se han flanqueado en ese turno al colocar la ficha del color contrario. Esas fichas, para que estén flanqueadas, deben formar una línea continua recta (diagonal u ortogonal) de fichas del mismo color entre dos fichas del color contrario (una de ellas la recién colocada y la otra ya presente). En el siguiente ejemplo juegan las blancas donde indica la flecha y se puede ver qué fichas se voltean.

La partida finaliza cuando ningún jugador puede mover (normalmente cuando el tablero está lleno de fichas) y gana quien en ese momento tenga sobre el tablero más fichas mostrando su color.


FIB


UPC