INFORMATICA ll

ANALISIS CLASES Y DIAGRAMAS

JUEGO OTHELLO

LUIS DAVID MUÑOZ

RAUL DAZA LIÑAN

DEPARTAMENTO DE INGENIERIA ELECTRONICA Y TELECOMUNICACIONES

FACULTAD DE INGENIERIA

UNIVERSIDAD DE ANTIOQUIA

MEDELLIN 27/10/2023

Contextualización del problema

Othello es un juego de estrategia que se lleva acabo entre 2 jugadores en un tablero de 64 posiciones con 64 fichas disponibles para cada jugador, estas se distinguen por el color diferente para cada jugador, al final gana el jugador que tenga la mayor cantidad de fichas sobre el tablero, ya sea que el tablero este completamente lleno o parcialmente lleno, pero sin movimientos disponibles para los jugadores.

Análisis

Se define una serie de clases con sus atributos y métodos que son necesarias para modelar el juego en consola.

Se desea iniciar el juego mostrando el historial de partidas jugadas que están guardadas en un archivo.txt. Se inicia la partida creando los objetos de jugadores, fichas, y tablero, seguido se llena el tablero con las posiciones iniciales de las fichas.

Inicia con el primer turno para el jugador1 que realiza la jugada inicial, se muestra en el tablero los movimientos disponibles para realizar marcados con una ‘O’ en las casillas, el jugador por consola debe ingresar el numero de fila y de columna en el cual se encuentra la casilla a la cual va a realizar jugada.

Se valida si la casilla ingresada por consola corresponde a un movimiento valido, si el movimiento no es válido, se le pide al jugador que ingrese nuevamente las coordenadas hasta que el movimiento sea válido. Si es un momento valido, se procede a implementar la lógica para cambiar las fichas al identificador del jugador en turno que acabo de realizar el movimiento en todos los lugares donde ocurra un encierro.

Se verifica si hay movimientos disponibles o si el tablero esta lleno, en caso de que el tablero aun no este lleno y haya movimientos disponibles, se cambia de turno y se repite el proceso. Si se encuentra que no hay movimientos disponibles o que el tablero está lleno, en tal caso la partida finaliza.

Una vez la partida finalice se cuentan las fichas de cada jugador y se muestran los resultados en pantalla, luego estos resultados se guardan en el archivo.txt de historial de resultados para tener registro de las partidas.

Luego le pregunta al usuario si desea jugar una nueva partida, si es afirmativo, se vuelve a iniciar otra partida. En caso negativo se finaliza el juego.

Diseño

En el main se definen dos funciones

- Menu(): despliega el menú de opciones del programa
- leerHistorial(): lee el historial de partidas del archivo que las posee

Lo siguiente es una breve descripción de las clases.

Clase ficha: Esta clase se define para modelar el color de las fichas de los jugadores, en este caso modelara con caracteres que simularan los colores. Color blanco (‘*’), color negro (‘-‘). también se añaden otros símbolos para proveer más información posición válida (‘O’), casilla vacía (‘ ’).

Clase ficha

Atributos:

- char idd

Métodos:

- ficha (char idd)
- getter y setter ficha ()

Clase jugador: Esta clase permite modelar los participantes de la partida con el nombre y el identificador de las fichas. Color blanco (*), color negro (-).

Clase jugador

Atributos:

- char idd
- string nombre

Métodos:

- jugador ()
- getters y setters ()

Clase tablero: Con esta clase se pretender realizar todas las acciones que involucren al tablero y el juego en general, se crea el tablero de juego que será una matriz de 8X8 (esto está definido por una variable global para la cantidad de filas y la cantidad de columnas), se inicializa el tablero con las 4 fichas iniciales ubicadas en su respectiva posición inicial, se realizan jugadas, se verifica si los movimientos son válidos, obtiene posiciones de las celdas donde se desean poner las fichas de una jugada a realizar, invierte el carácter de las fichas involucradas en un movimiento, se imprime el tablero de juego cada que se realicen jugadas, verifica si el tablero esta lleno o si no quedan movimientos disponibles para finalizar el juego.

Clase tablero

Atributos:

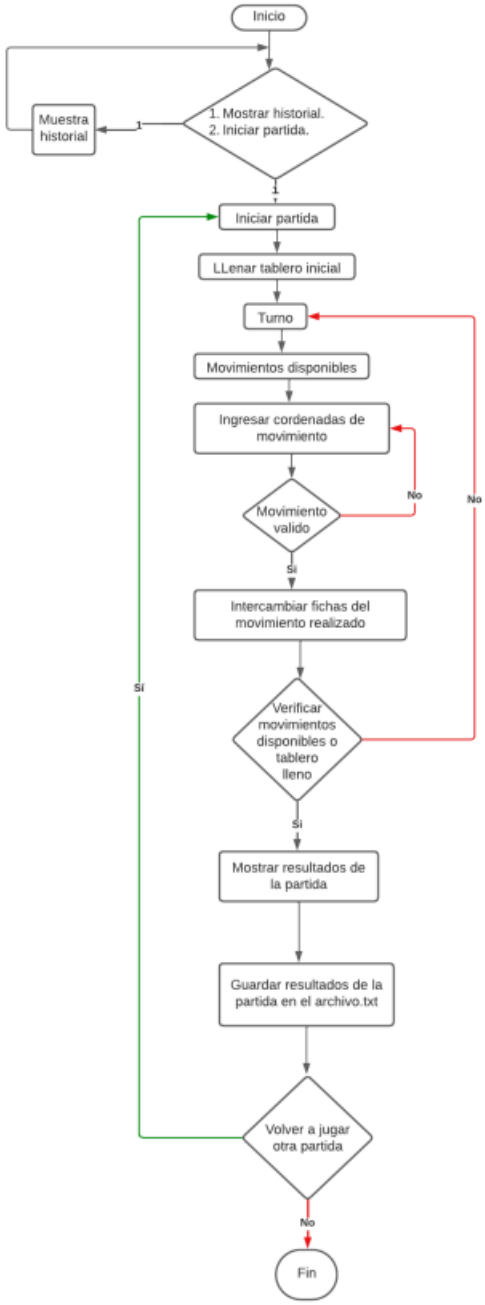
- Fichas* casillas[filas][columnas]
- Jugador jugador1
- Jugador jugador2

Métodos:

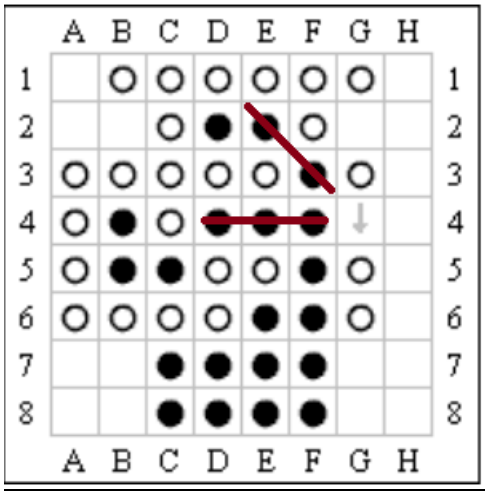
- tablero (): constructor que inicializa los atributos
- ~tablero(): elimina la memoria dinámica (las fichas de las casillas)
- iniTablero (): inicializa todas las casillas en blanco excepto las casillas del centro
- impritablero(): imprime el tablero por consola
- volteacasillas(): tiene dos funciones. verifica si hay encierros en la posición ingresada; busca las posiciones donde hay encierros y voltea las fichas en medio del encierro

- casillasjugables(): ingresa en ‘casillas’ una ‘O’ en las casillas jugables por el jugador de turno
- limpiarcasillas(): elimina las ‘O’ del método anterior
- EscArchivo(): guarda la información de la partida en el archivo del historial
- Casillavalida(): verifica si la casilla a la que se quiere hacer el movimiento es una casilla jugable
- Setter y getter de Casillas
- Totalfichas(): cuenta el total de fichas
- Win(): imprime la informacion de la partida (ganador, con cuantas fichas, etc)

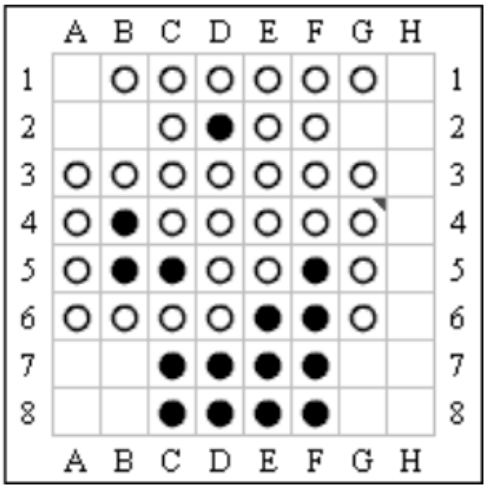
- **Diagrama de flujo del juego.**



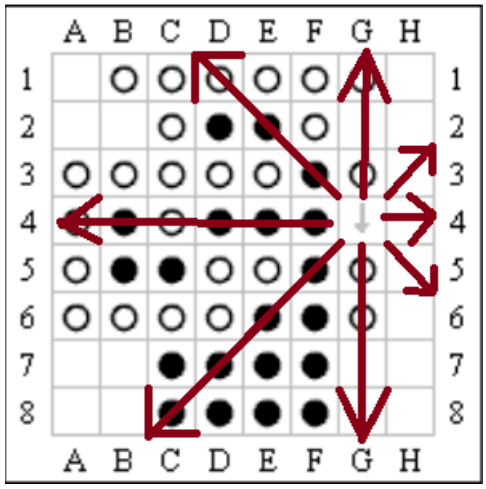
El método más complejo del programa es el método volteacasillas() perteneciente a la clase tablero. Para comprender mejor cómo funciona esta función hay que saber como ocurren los encierros. Los encierros ocurren cuando fichas del jugador que no tiene el turno quedan entre dos fichas del jugador que tiene el turno (donde una de esas fichas se acaba de poner en el turno actual). En la siguiente figura se ve cómo funciona esto.



Si se pone una ficha blanca en donde está la flecha entonces las fichas de bajo de la línea roja quedan en un encierro y pasan a ser fichas del rival como se ve en la siguiente figura.



El algoritmo creado mira en todas las direcciones buscando encierros, visualmente lo de la siguiente figura.



En total son 4 direcciones, pero se separaron en 8 sentidos diferentes, los nombres dados son: fila izquierda, fila derecha, columna arriba, columna abajo, diagonal izquierda arriba,

diagonal izquierda abajo, diagonal derecha arriba, diagonal derecha abajo. Donde la primera palabra es la dirección y la segunda es hacia el lado donde se verificará si hay un encierro, para el caso de las diagonales las 2 palabras siguientes indican en que cuadrante esta la diagonal donde se verificará si hay un encierro.
El método completo esta definido de esta manera:
bool volteacasillas(short posfila, short poscolumna, unsigned short numerojugador, bool mode)
donde posfila y poscolumna es una posición en el tablero, numerojugador es el numero del jugador con el turno actual y mode define si la función va a verificar que en esa casilla existan encierros (true) o si va a buscar todos los lugares donde hay encierros y voltear las fichas del rival en el encierro.
Lo primero es la inicialización de variables a usar internamente

```
unsigned short pos[8][2]; // matriz que guarda las posiciones de los encierros
for (int i = 0; i < 8; i++) for (int j = 0; j < 2; j++) pos[i][j] = filas*columnas;
char idenemigo;
char idpropio;
if (numerojugador == 1)
{
    idpropio = jugador1.getidd();
    idenemigo = jugador2.getidd();
}else
{
    idpropio = jugador2.getidd();
    idenemigo = jugador1.getidd();
}
```

La matriz pos guarda las posiciones donde hay encierros, se inicializa sus valores con el tamaño del tablero, además, se guarda el id del enemigo y el id propio.
Entrando a la lógica usada, la primera parte busca por cada sentido diferente buscando los encierros. En el caso de la siguiente figura, es para el encierro de fila izquierda.

```
// encierro de fila izquierda
for (int i = poscolumna-1; i >= 1; i--)
{
    if ((casillas[posfila][i]->getidd() == idenemigo) && (casillas[posfila][i-1]->getidd() == idenemigo || casillas[posfila][i-1]->getidd() == idpropio))
    {
        if((casillas[posfila][i]->getidd() == idenemigo) && (casillas[posfila][i-1]->getidd() == idpropio))
        {
            if (mode) return true;
            pos[0][0] = posfila;
            pos[0][1] = i-1;
            break;
        }
    }else
    {
        break;
    }
}
```

el bucle for itera desde la siguiente casilla a la izquierda de la casilla [posfila][poscolumna] y termina antes de la última casilla a la izquierda. Para que haya encierro las casillas siguientes a la casilla [posfila][poscolumna] deben ser fichas enemigas o ficha enemiga y ficha propia, eso es lo que verifica el primer condicional, si no es así es porque no hay encierro en esa dirección, por lo tanto, rompe el bucle.
El siguiente condicional verifica si la casilla en donde esta el bucle es enemiga y la siguiente es propia, si es así es porque allí termina el encierro, por lo tanto guarda el encierro en la matriz pos. Hay que notar que si mode es true entonces la función acaba allí la que retorna true debido a que se verificó que existen encierros para esa casilla.
Para los demás sentidos la lógica es parecida.

Lo siguiente es la lógica que voltea la fichas en los encierros. Se ve en la siguiente figura

```
if (mode) return false;
// voltea las fichas donde hay encierros
// fila izquierda
for (int i = pos[0][1]+1; i <= poscolumna; i++)
{
    if (pos[0][1] == filas*columnas) break;
    delete casillas[posfila][i];
    casillas[posfila][i] = new ficha(idpropio);
}
```

Para este punto si mode es true se devuelve false ya que no se encontró encierros.
Si mode es falso entonces se procede a voltear las fichas donde hay encierros. En la figura está el ejemplo para fila izquierda. Se inicia desde la siguiente posición guardada en la matriz pos (ya que la posición guardada ya tiene una ficha propia) y se itera hasta llegar a la posición ingresada a la función (casilla [posfila][poscolumna]), dentro del bucle se eliminan las fichas anteriores y se agregan las fichas propias. Dentro del bucle hay un condicional que verifica si pos[][] == filas*columnas ya que si esto es cierto es porque no existen encierros en ese sentido.

Experiencia de Aprendizaje

Si bien el juego propuesto se puede desarrollar sin la necesidad de usar clases, la experiencia obtenida al usar clases es gratificante ya que permite organizar el código de tal manera que es más sencilla su implementación y legibilidad.
El uso de punteros, referencias y memoria dinámica es fundamental en las implementaciones de clases para garantizar el uso adecuado de la memoria ya que evita que grandes segmentos de memoria sean copiados y además se pueda modificar los atributos de la instancia original.