UNIVERSIDAD DE ANTIOQUIA

Facultad de Ingeniería

Informe Final

Parcial II

Informática II

Robinson Correa Morales
CC: 1001471074

Juan José Díaz Zuluaga
CC: 1001456540

Docente
Aníbal José Guerra Soler

2023-2
UNIVERSIDAD DE ANTIOQUIA
DEPARTAMENTO DE I. ELECTRÓNICA Y TELECOMUNICACIONES
FACULTAD DE INGENIERÍA

SEDE MEDELLÍN

Noviembre 2023

Medellín, Colombia

Universidad de Antioquia
Departamento de I. Electrónica y Telecomunicaciones
Correa Morales Robinson
Díaz Zuluaga Juan José

Facultad de Ingeniería
Informática II

Segundo examen parcial – Informe Final

1 - Planteamiento del problema:

Se debe realizar una aplicación de consola que permita jugar el juego de mesa *Othello.* El juego requiere de dos jugadores y se juega sobre un tablero de 8 x 8 aunque también existen versiones de otras dimensiones.

El juego consiste en colocar fichas blancas o negras (según el jugador) e ir cambiando el color de las fichas del oponente según la siguiente regla:

> El jugador puede poner una ficha en una posición si y solo si el movimiento encierra de manera lineal (vertical, horizontal y diagonal) una o más fichas del oponente, provocando que las fichas encerradas cambien al color del jugador (se le añaden al total de ficas en el tablero).
>
> Si el jugador no puede hacer un movimiento, pierde el turno.
>
> El juego termina cuando la partida se bloquea (no existen movimientos para ningún jugador) o el tablero está completamente lleno. Gana quien tiene la mayor cantidad de fichas.

Para desarrollar el juego, no se permite el uso de contenedores de la STL, además se deben modelar las distintas clases que representen los objetos y la posibilidad de manejar archivos de texto para tener un historial de partidas.

2 - Análisis del problema:

Para realizar el modelado de una partida, se utilizará un enfoque de programación orientada a objetos, en la que se generan múltiples clases considerando una jerarquía a través de la anidación; es decir, se parte desde lo más simple hacia lo más general.

La idea es que a través de la estructura de clase se modele correctamente el juego, de manera que los objetos interactúen entre sí por medio de los atributos y métodos. En este caso el orden parte desde el juego, el cual contiene tanto a los jugadores como a las fichas.

- **Clase ficha:** Esta clase es la clase más simple de todas, permitirá modelar una ficha, conociendo su color, siendo blanca (*), negra (-) o nula (0). La ficha nula es una ficha sin color que representa un espacio vacío del tablero. Esta solo puede cambiarse por una negra o una blanca si se realiza un movimiento que modifique la casilla.

- **Clase jugador:** Esta clase permite modelar los jugadores que interactúan en el tablero, a través de ella se puede conocer la cantidad de fichas que posee, el número de movimientos que tiene y el color que lo representa.

- **Clase tablero:** Esta clase permitirá modelar el tablero de juego, con la posibilidad de generar tableros de tamaños $n * n$ y dimensiones pares y mayores a dos.
  Parte de esta clase se puede modelar como una matriz compuesta de arreglos dinámicos los cuales contienen los diferentes objetos ficha que participan en el juego. Estos objetos pueden cambiar de acuerdo con las reglas y además pueden crearse nuevos durante la partida.
  Además, el tablero tendrá como atributo, un arreglo con dos objetos jugador, el cual le permitirá saber quiénes están jugando y sus fichas.

- **Clase Juego:** La clase juego permite controlar todos los métodos y atributos asociados con la manipulación de la partida, es decir el iniciar una partida, crear un tablero de juego, verificar cuando finaliza la partida y finalmente almacenar los resultados del juego en un archivo de texto.

Es importante mencionar que entre las clases ficha y jugador no hay una interacción directa ya que ambos lo hacen por medio de la clase tablero, el cual contiene a dos jugadores y además una matriz dinámica que almacena los objetos ficha.

La partida se inicia desde la clase juego, la cual inicializa a través del método constructor los dos atributos más importantes de este nivel de jerarquía: el tablero de juego y una matriz dinámica que almacena la información de los jugadores; particularmente el número de fichas y los nombres de ambos.

En cada turno se debe verificar si el jugador tiene movimientos disponibles e indicar cuales son estos, ya que, de no haber, ocurrirá un cambio de turno. Si existen movimientos posibles el jugador debe indicar la posición en la que debe ir la ficha y si la jugada es válida, pasa al siguiente turno y continúa la partida.

En caso de que la posición indicada no sea válida, se volverán a pedir las coordenadas hasta que esta sea correcta.

El juego terminará de dos maneras:

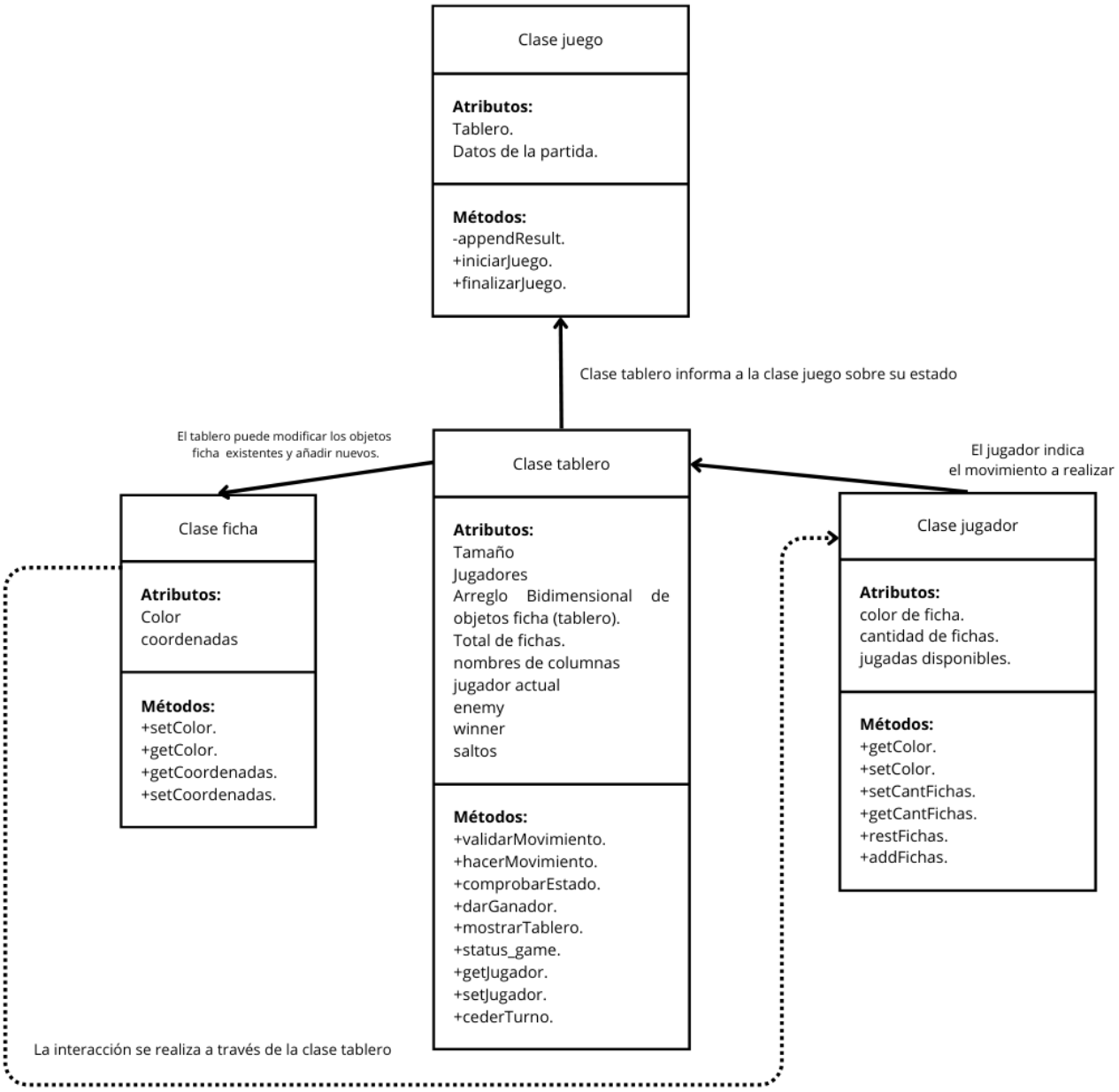
- Si no existen más movimientos posibles para ninguno de los jugadores
  (Es decir se producen **2** saltos de turno consecutivos)
- Si el tablero se ha llenado por completo.

Cualquiera de estos dos casos se verificará en la clase tablero y dará por finalizada la partida; el jugador con más fichas sobre el tablero será el ganador.

Al finalizar se deben mostrar los resultados de la partida y almacenarlos el historial de todos los juegos que se han dado. Para ello se deben guardar los nombres de los jugadores y con cuantas fichas finalizó cada uno en un archivo de texto.

Universidad de Antioquia
Departamento de I. Electrónica y Telecomunicaciones
Correa Morales Robinson
Díaz Zuluaga Juan José

3 - Diagrama de Clases y Estructuras de Datos:



Universidad de Antioquia
Departamento de I. Electrónica y Telecomunicaciones
Correa Morales Robinson
Díaz Zuluaga Juan José

Tabla de Atributos y Métodos

Clase Juego:

Identificador	Acceso	Descripción
string **Datosjuego	Privado	Matriz dinámica que almacena los datos de los jugadores
void set_jugadores()	Privado	Función interna que pasa los datos a la matriz dinámica
void set_fichas()	Privado	Función interna que pasa los datos a la matriz dinámica
void append_results()	Privado	Añade una fila al archivo de texto donde se almacenan los resultados de las partidas
void jugar()	Privado	Realiza el control de turnos y pide los movimientos nuevos
unsigned int entradaNum()	Privado	Realiza la verificación de entradas numéricas de movimientos
unsigned int entradaString()	Privado	Realiza la verificación de entradas string de movimientos
void reglas	Privado	Imprime las reglas del juego por pantalla antes de comenzar
tablero Table	Público	Objeto de la clase tablero sobre el que se juega la partida
void start_game()	Público	Introduce el juego y guarda los nombres de los jugadores
void juego_finalizado()	Público	Da por finalizada la partida y entrega el ganador y resultados

Clase Jugador:

Identificador	Acceso	Descripción
char color	Privado	Variable que define el color de las fichas del jugador
unsigned int fichas	Privado	Variable que define el número de fichas que posee
char getColor()	Público	Método getter del color
void setColor()	Público	Método setter del color
unsigned int getFichas()	Público	Método getter del número de fichas
void addFichas()	Público	Suma fichas al contador del jugador
void restFichas()	Público	Resta fichas al contador del jugador
void setFichas()	Público	Método setter del número de fichas

Universidad de Antioquia
Departamento de I. Electrónica y Telecomunicaciones
Correa Morales Robinson
Díaz Zuluaga Juan José

Clase Tablero:

Identificador	Acceso	Descripción
ficha **matriz	Privado	Matriz dinámica donde se almacenan objetos ficha
unsigned int fichas	Privado	Número de fichas que se encuentran en juego
std::string *nombres_col	Privado	Almacena las letras que representan las columnas
short jug_actual	Privado	Variable del jugador que tiene el turno
short enemy	Privado	Variable del jugador contrario al turno
short winner	Privado	Variable que almacena al jugador que gana la partida
short saltos	Privado	Variable que almacena el número de saltos de turno para finalizar la partida
bool checkFlip()	Privado	Retorna si la jugada produce un flip de fichas
void robarFichas()	Privado	Ejecuta el robo de fichas en el tablero (Flip)
void nombreCols()	Privado	Añade los nombres de las columnas a el array dinámico
void printNombresCol()	Privado	Imprime en consola las letras de las columnas
player jugadores [2]	Público	Objetos jugador que interactúan sobre la clase tablero
string *getNombres_col()	Público	Retorna un apuntador a string con los nombres de las columnas
unsigned int current_player()	Público	Retorna el jugador actual
unsigned int enemy_player()	Público	Retorna el jugador adversario al actual
bool comprobarMov()	Público	Verifica si un movimiento es válido o no
void hacerMovimiento()	Público	Ejecuta el movimiento en el tablero
void printTablero()	Público	Imprime en consola el tablero completo con las fichas
bool comprobarEstado()	Público	Retorna si la partida ya ha finalizado
short darGanador()	Público	Entrega el valor del jugador ganador 0 o 1
void status_game()	Público	Imprime el número de fichas de cada jugador que están en el tablero
void cederTurno()	Público	Pasa el turno al otro jugador cambiando el jugador actual
void addSaltos()	Público	Añade un salto al contador

Clase Ficha:

Identificador	Acceso	Descripción
char color	Privado	Variable que define el color de la ficha
unsigned int x	Privado	Variable que define la posición x de la ficha
unsigned int y	Privado	Variable que define la posición y de la ficha
void setColor()	Público	Método setter del color
char getColor()	Público	Método getter del color
void printColor()	Público	Método para imprimir el color de una ficha
unsigned int getX()	Público	Método getter de la posición en x
void setX()	Público	Método setter de la posición en x
unsigned int getY()	Público	Método getter de la posición en y
void setY()	Público	Método setter de la posición en y

4 - Algoritmos Importantes

Durante la realización de este programa se requirieron un gran número de funciones y algoritmos, los cuales fueron enumerados anteriormente en las tablas de atributos y métodos. Aunque particularmente se requiere profundizar en algunos de ellos para llegar a un mejor entendimiento.

4.1 - Impresión de Resultados:

Para implementar este algoritmo, se requiere de varias partes, siendo una de las más importantes el manejo de archivos. Para esta aplicación, se utiliza la librería 'fstream', que permite abrir, leer y modificar archivos de diferentes tipos. En este caso, se emplea la memoria dinámica para leer y almacenar la información de las partidas. Estos datos se guardan en un formato similar al de un archivo CSV, utilizando una estructura separada por comas y saltos de línea para indicar un nuevo registro. Los registros se almacenan en la memoria no volátil a través de un archivo de texto. La memoria dinámica se utiliza para que, una vez que se imprimen por pantalla todos los registros de partidas, se libere ese espacio en memoria.

4.2 - Comprobación de movimientos:

Esta función se encarga de verificar si una ficha puede ser colocada en una casilla o no según las reglas de Othello, utiliza una segunda función llamada checkFlip. Para comprobar un movimiento se hace lo siguiente:

Se verifica que los datos ingresados (fila, columna) no estén fuera del tablero, si es el caso, el movimiento es incorrecto.

Si los datos están dentro del tablero, se procede a verificar si la posición ingresada está ocupada, si es el caso, el movimiento es incorrecto.

Si no hay problemas con los pasos anteriores, se procede a determinar el color del jugador actual y a evaluar en la función auxiliar checkFlip si es posible generar un encierro en cualquiera de las direcciones en línea recta alrededor de la posición ingresada incluyendo diagonales.

4.3 – Comprobación de encierro (checkFlip):

Esta función se encarga de comprobar si existe una ficha del jugador actual en una dirección especifica, para ello, recibe la posición inicial que corresponde a la casilla donde se colocará la futura ficha, el color a evaluar y una dirección determinada por 2 constates. Un ciclo se encargará de buscar una ficha del color evaluado, en caso de llegar al borde del tablero, retornará falso.

El funcionamiento es simple, tomará la posición de la casilla seleccionada y en cada iteración sumará a la fila y la columna las constantes ingresadas como parámetro, esto provocará que la casilla a comprobar se desplace en una línea recta. Ejemplo:

Si las constantes son $dy = 0$ y $dx = 1$, al sumarlas a las variables fila y columna, generará un desplazamiento a la derecha.

4.4 – Invertir el color de las fichas (robarFichas):

Esta función maneja una lógica muy similar a checkFlip con la diferencia de que busca el color del adversario y no el propio. Si encuentra una ficha del adversario, invierte su color y se la añade al jugador. El algoritmo termina cuando no encuentra más fichas del adversario en la dirección indicada, la cual utiliza el mismo concepto de desplazamiento que checkFlip.

4.5 – Construcción del tablero:

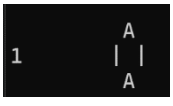
Para construir el tablero, en el constructor de la clase se asigna memoria para un arreglo de n apuntadores a objetos tipo ficha, luego a cada uno de estos apuntadores se les asigna memoria para n objetos tipo ficha, donde n es una contante global, lo que permite generar tableros de distintos tamaños sin afectar el juego.

4.6 – Impresión del tablero:

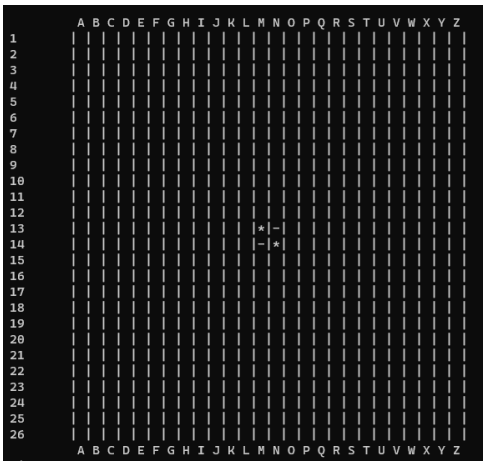
Se hace uso de un atributo de la clase tablero, el cual es un arreglo de strings, que contienen los nombres de las diferentes columnas (A, B, C, …,H), los cuales son mostrados gracias a una función. Luego se procede a imprimir desde el ciclo externo el numero de la fila y luego desde el ciclo interno el contenido de dicha fila haciendo uso de un método público de la clase ficha, cada uno seguido de una barra vertical.

Luego, se vuelven a imprimir los nombres de las columnas en la parte inferior.

Tablero de 1 posición.



Tablero de 676 posiciones



5 – Estructuras de datos

Para seleccionar la estructura de datos del programa se analizaron los diferentes requerimientos de la aplicación entre los cuales estaban el no uso de contenedores STL y el aprovechamiento de la memoria dinámica.

Como se mencionó anteriormente esta aplicación fue planteada para utilizar la programación orientada a objetos, por lo que se tiene una jerarquía de distintos niveles entre cada clase, utilizando la anidación para definir estás relaciones. Es decir, se parte de lo más básico, como la clase ficha y se van diseñando las clases más generales, las cuales anidan las clases anteriores como atributos. Este es el caso del tablero, que tiene como atributos fichas y jugadores, pero a su vez esta misma clase está dentro del ámbito de la clase juego.

En este caso el tablero está formado por una matriz dinámica, la cual básicamente se compone de filas de arreglos dinámicos y un arreglo de apuntadores a las filas mencionadas anteriormente (Concepto de apuntador doble). Esta matriz dinámica contiene en su interior a los objetos ficha que interactúan en el tablero, los cuales guardan información acerca de su posición, y color. Y Además la clase tablero guarda información de los turnos de los jugadores y como cambian las fichas a lo largo de la partida con los movimientos realizados.

Finalmente, la clase juego utilizará una matriz dinámica para manejar el histórico de partidas. Y esta se borrará tan pronto como la información se pase al archivo de texto, ya que, al tener esta información almacenada en una sección de memoria no volátil, no hace falta conservarla directamente en el programa.

6 – Experiencia de Aprendizaje

Durante el desarrollo de este programa se encontraron múltiples dificultades de diferentes niveles que obligaron a pensar las cosas de una manera nueva utilizando realmente la creatividad y las herramientas al alcance que se tenían con los temas vistos en el curso.

Particularmente existieron tres momentos que llevaron retos más importantes que lo demás, inicialmente el entender cómo se realiza un modelo con programación orientada a objetos lleva a hacerse muchas preguntas acerca de la interacción entre objetos, la anidación y finalmente como se integran todas las partes que componen el sistema. Por ello el lograr llegar a una estructura adecuada que fuera fiel a cómo funcionan las cosas en la vida fue la primera dificultad abordada. Y fue superada utilizando pensamiento estructurado y apoyándose de herramientas como diagramas de clases.

El segundo reto conceptual encontrado fue con el uso de la memoria dinámica, ya que al intentar hacer una aplicación realmente eficiente hay que prestar especial cuidado a los elementos que utilizan memoria; por ello elementos como las matrices de información fueron diseñadas en memoria dinámica, con el objetivo de que al acabar cada juego esa

memoria no se quedara inutilizada, sino que se liberase al terminar una partida, o al leer la información para la que fue diseñada como contenedor.

Finalmente, el reto considerablemente más complicado fue el diseño de las comprobaciones en cada movimiento, esto principalmente se debía a que “othello” es un juego lleno de casos diversos y que pueden hacer cambiar el rumbo de la partida de manera muy abrupta. Por eso mismo el lograr crear funciones que pudieran manipulas todos los movimientos posibles que pueden ocurrir en una casilla tan grande como 8x8 hace que la complejidad aumente de manera notable. Para solucionar este problema se utilizó la metodología conocida como: “Divide y vencerás”, ya que el usar funciones más pequeñas para tratar los diferentes movimientos y opciones hacia la tarea mucho más sencilla.

Verdaderamente esta tarea estuvo llena de retos personales a la hora de integrar todas las partes del código, pero fue posible lograr un resultado satisfactorio y funcional avanzando con las ideas y utilizando un pensamiento racional para solucionar los problemas en vez de utilizar la programación con “fuerza bruta” y hacer que las cosas funcionasen como pudieran.