	**Carátula para entrega de prácticas**
Facultad de Ingeniería	Ingeniería en Computación

Profesor(a): _____

Asignatura: Programación Orientada a Objetos

Grupo: 1

No de Práctica(s): 4

Integrante(s): 11700202-9

_____

_____

_____

No. de Equipo de cómputo empleado: _____

Semestre: 2024-1

Fecha de entrega: 29 de septiembre de 2023

Observaciones: _____

_____

CALIFICACIÓN: _____

Index.

1.- Introduction ---------- 3

1.1.- Hypothesis ---------- 3

2.- Development ---------- 3

3.- Conclusions ---------- 5

4.- References ---------- 6

Practice 4 - Objects and classes.

TicTacToe.

1.- Introduction.

This report has the objective of explaining the fourth practice of the Object oriented programming subject, in which we must explore the classes and objects implementation, working directly with the basics of the paradigm.

We must build an automatic algorithm for simulating a TicTacToe match on a 5x5 board.

1.1.- Hypothesis

We have to create two classes (Player and Grid) and make their object instances interact in the main class. As the practice doesn't indicate that both players must play optimally, we will choose a random move in each player turn (maybe in the future we could implement a minimax/maximin algorithm for optimal play).

2.- Development

The steps to follow were:

- Import the math class for generating random numbers.
- Create a class pair for better coordinate handling.
- Create the class Grid.
- Define the attributes and methods for Grid.
- Create the class Player.
- Define the attributes and methods for Player.
- Define the interactions between the instances.

intPair class.

Is a simple class, we have two attributes, first and second, both public.

Grid class abstraction.

Grid should be a n x n character matrix.

- “ ” (SPACE) is a blank space.
- O is for player A.
- X is for player B.

So we need only two attributes and using encapsulation, both are private:

- Size (Integer)
- Grid (Size x Size character matrix)

The getters for these attributes will be.

- Get size
- Get status (Status of the grid coordinates).

And the only “setter” is the makeMove method.

Public methods:

makeMove (boolean): Before checking if the move is valid (is a blank space), set the coordinate (x,y) to a player move.

Returns true if the move is done and false if the move is invalid.

isValid (boolean): Check if the space is empty for making a move.

isGameOver (boolean): Checks if any of the players won or if the game is tied.

Returns true if the game is over and false if the conditions for the game continuation are true.

winningState (boolean): Checks if the game is on a win condition by checking the rows, the columns and diagonals.

Returns true if some player won, and false if none of the conditions for winning are satisfied.

winningState and isGameOver pseudocode:

```
function isGameOver(movesA, movesB, nameA, nameB)
    a = 'O'
    b = 'X'
```

```
    if movesA < size
        return false

    if winningState(a)
        Print "Player " + nameA + " is the winner (O)"
        return true

    if winningState(b)
        Print "Player " + nameB + " is the winner (X)"
        return true

    if movesA + movesB > size * size
        Print "Tie!"
        return true

    return false

function winningState(p)
    isWon = true

    for i from 0 to size - 1
        for j from 0 to size - 1
```

```
        if grid[i][j] != p
            isWon = false
            break
        end if
        if j = size - 1
            isWon = true
            break
        end if
    end for
    if isWon = true
        break
    end if
end for

if isWon = true
    return true

for i from 0 to size - 1
    for j from 0 to size - 1
        if grid[j][i] != p
            isWon = false
            break
        end if
```

```
        if j = size - 1
            isWon = true
            break
        end if
    end for
    if isWon = true
        break
    end if
end for

if isWon = true
    return true

isWon = true

for i from 0 to size - 1
    if grid[i][i] != p
        isWon = false
        break
    end if
end for

if isWon = true
```

```
        return true


    isWon = true


    for i from 0 to size - 1
        if grid[i][size - i - 1] != p
            isWon = false
            break
        end if
    end for
    return isWon
```

printGrid (void): Displays the board on screen.

clearGrid (void): Set all the grid to blank spaces.

Player class abstraction.

Player needs of three attributes:

- Name (String).
- Side (Character).
- Number of moves (Integer).

Initially we thought that we needed a turn attribute but we could model the program in a simpler way without this attribute.

We need to encapsulate those attributes so we've made them private.

The setters for the attributes will be getName, getSide and getMoves.

We only need a setter, which is setMoves.

The builder will set the name and the side of the player.

Methods:

playerTurn (intPair): Generates a random pair of coordinates using the Math package and returns that pair.

Main class:

We set the grid size as 5 and create an instance of the Grid class. Then we create two instances of the Player class.

Then we do a while loop for various matches if the user wants.

While the game is not finished we run all the next instructions.

- Generate a valid move for player one and increase their moves.
- Print the board and check if the game is over.
- Generate a valid move for player two and increase their moves.
- Print the board and check if the game is over.

The end of the game is given by n^2 (n being the size of the board).

Pseudocode of the game:

```
play = true


while play:
    board.clearGrid()
    isValidMove = false
    movesA = 0
    movesB = 1
```

```
    for i from 1 to n * n:
        Print "Turno " + i + " del jugador " + player1.getName()
        move = player1.playerTurn(n)

        do:
                        isValidMove = board.makeMove(move.first, move.second, player1.getSide())
        while not isValidMove

        movesA += 1
        player1.setMoves(movesA)
        board.printGrid()

                    if board.isGameOver(player1.getMoves(), player2.getMoves(), player1.getName(), player2.getName()):
            break

        pressEnterToContinue()  # Pause

        Print "Turno " + i + " del jugador " + player2.getName()
        move = player2.playerTurn(n)

        do:
```

```
                    isValidMove = board.makeMove(move.first, move.second, player2.getSide())
        while not isValidMove


        movesB += 1
        player2.setMoves(movesB)
        board.printGrid()


                    if board.isGameOver(player1.getMoves(), player2.getMoves(), player1.getName(), player2.getName()):
            break


        pressEnterToContinue()


    Print "Deseas jugar otra vez?"
    Print "1: Si ---- 2: No"
    again = cout.nextInt()
    cout.nextLine()


    if again == 1:
        play = true
    else:
        play = false
```

Conclusion: The hypothesis was correct, but a fun fact is when the possible moves are few, hitting a blank space depends on the probability, being a permutation with repetition, so our complexity is exponential. Must be a way to reduce this complexity but that will be a worry for our future selves.

I've enjoyed this practice a lot because it showed me that OOP is very intuitive and programmer friendly (sometimes).

In the future I will enhance this code for implementing a minimax/maximin algorithm and maybe a visual interface without using the console.

References:

- [1] Oracle. (2023, July). Java API, Math Class [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/lang/Math.html
- [2] Ashkay L Aradya. (2023, February). Geeks for Geeks. Finding optimal move in Tic-Tac-Toe using Minimax Algorithm in Game Theory [Online]. Available: https://www.geeksforgeeks.org/finding-optimal-move-in-tic-tac-toe-using-minimax-algorithm-in-game-theory/