Hacettepe University

Department of Computer Engineering

BBM103 Assignment 4 Report

TA: Hayriye Çelikbilek

Samed Ökçeci - 2210356015

Analysis

Lets take a quick glance to the Battleship game. The game is played on the board like a chess board. We are given 5 different ships.

No.	Class of ship	Size	Count	Label
1	Carrier	5	1	CCCCC
2	Battleship	4	2	BBBB
3	Destroyer	3	1	DDD
4	Submarine	3	1	SSS
5	Patrol Boat	2	4	PP

We should place them horizontally or vertically. Cross placing is not allowed. The players cannot see each others ships. After all ships are placed, the game is ready to start. First player makes a blindly move (without seeing opponents ship), if move hits a part of any ship, we must place X on the equilavent part of the board; if it doesn't, we must place O to equilavent part of the board. The game continues until one player hits all the ships of his opponent. In this situation, the player is the winner. The game ends.

In Assignment 4, we should code a program in which we can play Battleship game. We are given 6 files; Player1.txt, Player2.txt, Player1.in, Player2.in, OptionalPlayer1.txt, OptionalPlayer2.txt.

We should get initial positions of the ships from Player1.txt and Player2.txt.

As its name states OptionalPlayer1.txt and OptionalPlayer2.txt are optional to find Battleship and Patrol Boats (B, P). (There are 2 Battleships and 4 Patrol Boats, so it can be a little bit intriguing to identify multiple boats named as the same.)

We should get all moves of the players from Player1.in and Player2.in files. All moves are given at once, but we have to make only one move round by round.

We should print the current situation of the boards every round. Therefore players can see, if they hit.

After all the ships of one player are shot, for example Player2s ships, we should print Player1 is the winner, and print last situation of the boards. Winners board should include remaining ships.

Design

I coded inputShips_to_dict_converter() function to convert ship locations in Player1.txt, Player2.txt to two independent multidimensional dictionaries, which hold coordinates of ships for each player. Example of Player1's ships: {'B1': {'B6': '-', 'C6': '-', 'D6': '-', 'E6': '-'}, 'P1': {'B3': '-', 'B4': '-'}, 'P2': {'B10': '-', 'C10': '-'}, 'C': {'G1': '-', 'G2': '-', 'G3': '-', 'G4': '-', 'G5': '-'}, …

B1, P1, P2, C.. stands for the name of the ships. For example B1 means Battleship 1. P1 means Patrol Boat 1. There is individual boat name including its number for each boat.

I used this line of code to seperate all moves depending on ';' key and create a list:

```
p1Moves =  p1MovesFile.readlines()[0].replace('\n','').split(';')
p2Moves =  p2MovesFile.readlines()[0].replace('\n','').split(';')
```

If I execute print(p1Moves), It prints:

['5,E', '10,G', '8,I', '4,C', '8,F', '4,F', '7,A', '4,A', '9,C', '5,G', '6,G', '2,H', '2,F', '10,E', '3,G', '10,I', '10,H', '4,E', '8,G', '2,I', '4,B',...]

I coded create_hiddenboard() function to create two 10X10 identical hiddenboards. This hiddenboards have all the coordinates that a board can have. print(p1hiddenboard)-->

{'A1': '-', 'B1': '-', 'C1': '-', 'D1': '-', 'E1': '-', 'F1': '-', 'G1': '-', 'H1': '-', 'I1': '-', 'J1': '-', 'A2': '-', 'B2': '-', 'C2': '-', 'D2': '-',... 'G10': '-', 'H10': '-', 'I10': '-', 'J10': '-',}

I coded inputMoves_seperater_and_checker() function which pops 0th index of p1Moves or p2Moves (that means it pops current move from list) then checks if its syntax correct. If it is correct, it returns current move, and finishes function; otherwise it prints error message and calls itself until syntax of the move is correct.

If the move is correct, I call make_move function which searchs for move coordinate in p1ShipsDict (or p2ShipsDict) ,and if move coordinate corresponds to a coordinate in dictionary, it changes its value to 'X' and its hidderboards value to 'X'. Otherwise it changes its hiddenboards value to 'O'. Then calls ship_condition_checker() function to check if all coordinates are shot. If all cordinates shot game ends, otherwise it continues.

For every output, I call save_output() function to print my output to command line and to Battleship.out file.

I coded play_round() function to print every rounds output and make_move of the each player.

To make sure every player play one move each round, I coded a while function and called twice play_round funtion. After two play_round() function executed, I check if game ends by searching both players ships.

If one of both players or both players ships sink. It prints both hiddenboards and finishes the game.

Programmer's Catalogue

Analyzing: 6 hours

Designing: 9 hours

Implementing: 10 hours

Testing: 12 hours(to fix bugs)

Reporting: 8 hours

Overall: 45 hours

open_file(fileName,method):

```
def open_file(fileName,method):
    return open(fileName,method,encoding='UTF-8') #opens a file specified as name and method
```

This functions takes two parameters, fileName and method. As its name stands, fileName is the name of the file, and method is opening method. For example:

P1InputFile = open_file('Player1.txt', 'r')

save_output(fileName, text):

```
def save_output(fileName, text):
    fileName.write(text+'\n') #Writes 'text' to the output file
    print(text) #prints 'text' to the command line
```

It takes two parameters. FileName and text. It prints text to command line, and to the Battleship.out file.

create_hiddenboard(rowsxcolumns)

```
def create_hiddenBoard(rowsxcolumns='10x10'):
    rows, columns = rowsxcolumns.split('x') #rowsxcolumns 10x10. it seperates into two 10, 10
    rows, columns = int(rows), int(columns) #converts them into integers
    uppercase_characters = string.ascii_uppercase[:rows] #takes uppercase characters until 10th index. ABCDEFGHIJ
    hiddenBoard = dict.fromkeys([str(letter)+str(number) for number in range(1,columns+1) for letter in uppercase_characters],'-')
    #creates a dictionary with A1: '-', A2: '-' ... J10: '-' .
    hiddenBoard.update({'C': 0, 'B': 0, 'S': 0, 'P': 0, 'D': 0}) #adds this key value pairs to count sunk ships
    return hiddenBoard #returns whole dictionary
```

It needs only one parameter. Syntax should be like ‘rowxcolumn’. If you don’t enter any parameter default is 10x10.

hiddenboard_visualizer(boardName, player_name)

```
def hiddenboard_visualizer(boardName, player_name): #this function simply creates an empty list and appends every hiddenboard line into list.
    row_list = []
    row_list.append(f"{player_name}'s Hidden Board")
    uppercase_characters = string.ascii_uppercase[:10]
    row_list.append('  '+' '.join(uppercase_characters))

    for number in range(1,10+1): #this for loop takes values of hiddenboard as horizontally and appends to list
        row = f"{number}"
        for letter in uppercase_characters:
            if number<=9:
                row += " " + boardName[str(letter)+str(number)]
            else:
                row += boardName[str(letter)+str(number)] + ' '
        row_list.append(row)


    row_list.append('')
    row_list.append('Carrier     {}       '.format(boardName['C']*'X '+(1-boardName['C'])*'- ')) #gets sinked ships and multiplies by X, multiplies unsinkable ships by -
    row_list.append('Battleship  {}     '.format(boardName['B']*'X '+(2-boardName['B'])*'- '))
    row_list.append('Destroyer   {}       '.format(boardName['D']*'X '+(1-boardName['D'])*'- '))
    row_list.append('Submarine   {}      '.format(boardName['S']*'X '+(1-boardName['S'])*'- '))
    row_list.append('Patrol Boat {} '.format(boardName['P']*'X '+(4-boardName['P'])*'- '))
    row_list.append('')

    return row_list
```

It takes only two parameters. boardName and player_name. For example. p1HiddenBoard, ‘Player1’. Returns list of current hiddenboard output and condition of the ships.

ship_condition_checker(hiddenboardName, dictName, changed_key)

```
def ship_condition_checker(hiddenBoardName,dictName,changed_key): #checks if a ship is sinked or not
    for value in dictName[changed_key].values():
        if value == '-':
            return False
    else:
        hiddenBoardName[changed_key[0]] += 1
```

It takes three parameters. HiddenBoardName, dictName, changed_key. It checks all of the values of the shipDict. In one of the values, if value is ‘-’, it returns False; otherwise it adds 1 to sinked ships.

make_move(move, hiddenBoardName, dictName, movesList)

```
def make_move(move, hiddenBoardName, dictName, movesList): #this function hits the ship in the coordinate which is given. if it hits, it checks if ship is sinked.
    changed_key = None
    for key in dictName.keys():
        if dictName[key].get(move)!=None:
            hiddenBoardName[move] = 'X'
            dictName[key][move] = 'X'
            changed_key = key
            break
    else:
        hiddenBoardName[move] = "O"

    if changed_key!=None: #here it checks if ship is sinked
        ship_condition_checker(hiddenBoardName,dictName,changed_key)
```

It checks if a ship exists, on the same coordinate as move. If it exists updates its value to X and its hiddenboards value to X. Otherwise it updates hiddenboards value to O.

hiddenboard_finalizer(hiddenBoard, dictName)

```
def hiddenBoard_finalizer(hiddenBoard,dictName): #this is executed when a player wins.
    dicts_list = "C,D,S,B1,B2,P1,P2,P3,P4".split(',') #it simply takes parts of the ships which haven't shot. updates this parts to current hiddenboard of the winner.
    for i in dicts_list:
        for j in (list(dictName[i].keys())):
            if dictName[i][j]=='-':
                hiddenBoard[j] = str(i[0])
```

It helps to show remaining ships of the winner player. It simply searches for the ‘-’ valued ships in the dictionary of the winner player. When it finds ‘-’ updates winner players

hiddenboard with the abbrevation of the ship. (like one of these C, P, S, B, D)

is_game_finished(dictName)

```
def is_game_finished(dictName): #checks if game end
    for names_of_inner_dicts in dictName.values():
        for dict_values in names_of_inner_dicts.values():
            if dict_values == '-':
                return False
    else:
        return True
```

It checks if game ended. It simply looks all of the keys, if it finds only one '-', it returns False.

play_round(round, playerName, movesList, p1HiddenBoard, p2HiddenBoard, hiddenboard, shipsDict)

```
def play_round(round,playerName,movesList,p1HiddenBoard,p2HiddenBoard,hiddenBoard,shipsDict): #every round it is executed.
    save_output(outputFile, f"{playerName}'s Move\n")
    save_output(outputFile,f"Round : {round}\t\t\t\t\tGrid Size: 10x10\n")

    p1HiddenBoardList = hiddenboard_visualizer(p1HiddenBoard,'Player1')    #takes current output of players hiddenboards as list
    p2HiddenBoardList = hiddenboard_visualizer(p2HiddenBoard,'Player2')



    for index in range(len(p1HiddenBoardList)):
        save_output(outputFile,f"{p1HiddenBoardList[index]}\t\t{p2HiddenBoardList[index]}") #prints their output lists line by line



    move_coordinate, move = inputMoves_seperater_and_checker(movesList,shipsDict) #this code checks current move if it has any errors.
                                                                                   #if not returns move and move_coordinate for example: B1, 1,B

    save_output(outputFile,'Enter your move: {}\n'.format(move))

    make_move(move_coordinate,hiddenBoard,shipsDict,movesList) #it makes a new move, and updates hiddenboards and dictionaries
```

It is executed every round, and it prints the hiddenboards of the both players next to each other. Then it

calls inputMoves_seperater_and_checker function. Then it calls make_move function.

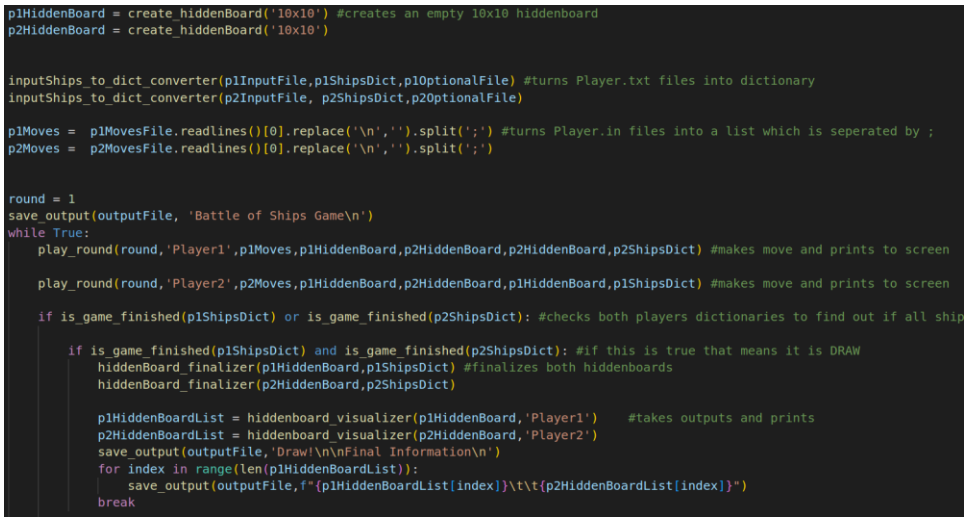
main()

```
def main():
    global p1InputFile, p2InputFile, p1ShipsDict, p2ShipsDict,p1HiddenBoard, p2HiddenBoard, outputFile
    outputFile = open_file('Battleship.out','w')

    p1ShipsDict = {}
    p2ShipsDict = {}
    error_list = []

    try: #trys to open a file, it FileNotFoundError happens it adds the file name to error name.

        try:
            if sys.argv[1]=='Player1.txt': #checks if Player1.txt exists
                p1InputFile = open_file(sys.argv[1],'r') #Opens file
            else:
                raise FileNotFoundError #if Player1.txt doesnt exist
        except FileNotFoundError:
            error_list.append('Player1.txt')

        try:
            if sys.argv[2]=='Player2.txt':
                p2InputFile = open_file(sys.argv[2],'r')
            else:
                raise FileNotFoundError
        except FileNotFoundError:
            error_list.append('Player2.txt')


        try:
            if sys.argv[3]=='Player1.in':
                p1MovesFile = open_file(sys.argv[3],'r')
            else:
```

It uses try, except to check if input files entered correctly.

```
        try:
            if sys.argv[4]=='Player2.in':
                p2MovesFile = open_file(sys.argv[4],'r')
            else:
                raise FileNotFoundError

        except FileNotFoundError:
            error_list.append('Player2.in')


        try:
            p1OptionalFile = open_file('OptionalPlayer1.txt','r')

        except FileNotFoundError:
            error_list.append('OptionalPlayer1.txt')


        try:
            p2OptionalFile = open_file('OptionalPlayer2.txt','r')

        except FileNotFoundError:
            error_list.append('OptionalPlayer2.txt')

    except:
        print('kaBOOM: run for your life!')
        exit()

    finally:
        if len(error_list)!=0:
            save_output(outputFile,f"IOError: input file(s) {','.join(error_list)} is/are not reachable.")
            exit()
```

It uses try, except to check if input files entered correctly. If some of the files are not opened, it prints error output and finishes the program.



```
p1HiddenBoard = create_hiddenBoard('10x10') #creates an empty 10x10 hiddenboard
p2HiddenBoard = create_hiddenBoard('10x10')


inputShips_to_dict_converter(p1InputFile,p1ShipsDict,p1OptionalFile) #turns Player.txt files into dictionary
inputShips_to_dict_converter(p2InputFile, p2ShipsDict,p2OptionalFile)

p1Moves =  p1MovesFile.readlines()[0].replace('\n','').split(';') #turns Player.in files into a list which is seperated by ;
p2Moves =  p2MovesFile.readlines()[0].replace('\n','').split(';')


round = 1
save_output(outputFile, 'Battle of Ships Game\n')
while True:
    play_round(round,'Player1',p1Moves,p1HiddenBoard,p2HiddenBoard,p2HiddenBoard,p2ShipsDict) #makes move and prints to screen

    play_round(round,'Player2',p2Moves,p1HiddenBoard,p2HiddenBoard,p1HiddenBoard,p1ShipsDict) #makes move and prints to screen

    if is_game_finished(p1ShipsDict) or is_game_finished(p2ShipsDict): #checks both players dictionaries to find out if all ship

        if is_game_finished(p1ShipsDict) and is_game_finished(p2ShipsDict): #if this is true that means it is DRAW
            hiddenBoard_finalizer(p1HiddenBoard,p1ShipsDict) #finalizes both hiddenboards
            hiddenBoard_finalizer(p2HiddenBoard,p2ShipsDict)

            p1HiddenBoardList = hiddenboard_visualizer(p1HiddenBoard,'Player1')    #takes outputs and prints
            p2HiddenBoardList = hiddenboard_visualizer(p2HiddenBoard,'Player2')
            save_output(outputFile,'Draw!\n\nFinal Information\n')
            for index in range(len(p1HiddenBoardList)):
                save_output(outputFile,f"{p1HiddenBoardList[index]}\t\t{p2HiddenBoardList[index]}")
            break
```

Creates hiddenboard for both players. Converts both Player.txt files into dictionaries. Then takes both moves files and converts them to list. In while loop, both players plays their turn (play_round), every round program checks if game is finished. If game finishes breaks the loop, otherwise increases the round by one.

```
    elif is_game_finished(p2ShipsDict): #if this is True that means Player1 won

        hiddenBoard_finalizer(p1HiddenBoard,p1ShipsDict) #converts Player1 output board to show remaining boots
        p1HiddenBoardList = hiddenboard_visualizer(p1HiddenBoard,'Player1')    #converts hiddenboard to a list
        p2HiddenBoardList = hiddenboard_visualizer(p2HiddenBoard,'Player2') #converts hiddenboard to a list
        save_output(outputFile,'Player1 Wins!\n\nFinal Information\n')
        for index in range(len(p1HiddenBoardList)):  #prints hiddenboards next to each other
            save_output(outputFile,f"{p1HiddenBoardList[index]}\t\t{p2HiddenBoardList[index]}")
        break


    round += 1 #round counter for output.
```

User Catalogue

To run this program, you have to put 6 files as the same location with the program. They should be named: Player1.txt, Player2.txt, Player1.in, Player2.in, OptionalPlayer1.txt, OptionalPlayer2.txt.

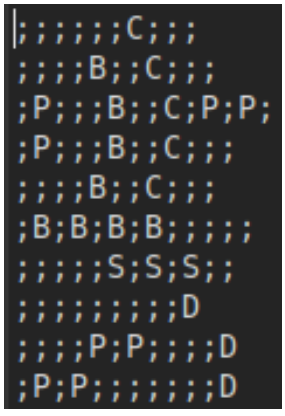
Command line command must be exactly the same(they are in the same line):

python3 Assignment4.py "Player1.txt" "Player2.txt" "Player1.in" "Player2.in"

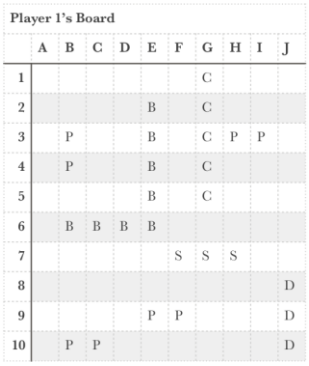
Player1.txt, Player2.txt (mandatory)

You should pay attention for uppercase and lowercase. It is important for program.

Player1.txt and Player2.txt syntax should be like:



(Player1.txt)



Player 1's Board

	A	B	C	D	E	F	G	H	I	J
1							C			
2					B		C			
3		P			B		C	P	P	
4		P			B		C			
5					B		C			
6		B	B	B	B					
7						S	S	S		
8										D
9					P	P				D
10		P	P							D

(What Player1.txt represents)

In each line, there must be nine semicolons. Every semicolon represents vertical line between coordinates. If you put C to leftmost, that means C is in the A column. There are ten rows which represents rows of the board.

Optional File (mandatory)

```
B1:6,B;right;
B2:2,E;down;
P1:3,B;down;
P2:10,B;right;
P3:9,E;right;
P4:3,H;right;
```

Syntax: ShipName:Starting Coordinate;direction(only right and down);

Player1.in and Player2.in (mandatory)

Example (it should be only one line):

5,E;10,G;8,I;4,C;8,F;4,F;7,A;4,A;9,C;5,G;6,G;2,H;2,F;10,E;3,G;10,I;10,H;4,E;8,G;2,I;4,B;5,F;2,G;10,C;10,

Syntax: row number,column name;

Row numbers should be between 1 and 10 (1 and 10 are included). Column letters should be from A to J. (A and J are included)

You should have enough moves to make sure at least one player win. (if both players win its draw)

Evaluation

Evaluation	Points	Evaluate Yourself / Guess Grading
Readable Codes and Meaningful Naming	5	5.

Evaluation	Points	Evaluate Yourself / Guess Grading
Using Explanatory Comments	5	5
Efficiency (avoiding unnecessary actions)	5	5
Function Usage	15	15
Correctness, File I/O	30	30
Exceptions	20	20
Report	20	18
There are several negative evaluations	…	…