Hacettepe University
Department of Computer Engineering

BBM103 Assignment 4 : Battle of Ships Report

Ezgi Ekin - 2210356029

03.01.2023

Table of Contents

1. Analysis

We are asked to write a program which is a guessing game named “Battleship”. The game involves strategy to make the right moves.

The game is played on 10x10 four grids, two of which contain ship positions that are arranged before the game begins and players cannot see each other's ships. The other two grid is used for making the moves and announcing if the moves are a “hit” or “miss” by printing “O” for “miss” and “X” for “hit”. The game aims to hit all ships of other players.

Grids are identified by letters for columns and numbers for rows. Ships can be placed horizontally or vertically on the grids. The type of ship determines its length of squares on the grid. Players have to use the same type of ships in the game.

There are 5 types of ships in the game
A ship named “Carrier” has a length of 5 squares and a count of 1.
A ship named “Battleship” has a length of 4 squares and a count of 2.
A ship named “Destroyer” has a length of 3 squares and a count of 1.
A ship named “Submarine” has a length of 3 squares and a count of 1.
A ship named “Patrol Boat” has a length of 2 squares and a count of 4.

When a ship’s all squares have been hit, the game announces this information.

The game is played in a series of rounds. In each round player bombs a target square at the other player's board. In every round, both of the players have to make a move. This means the game cannot announce a winner when player 1 made a move and bombs the last ship of player 2, instead it has to ask for one more move from player 2 too. If this move bombs the last ship of player 1 then the game is a draw. Otherwise, Player 1 wins the game.

Initial ship positions and players’ moves should be read from txt files. These files must be checked for correctness and if they are not in the wanted form or files are unreachable, a suitable error must be displayed to the user.

For every round a hidden form of both players' boards must be displayed, “-“ for floating “X” for sunk, and “O” for miss. At the end of the game if there are remaining ships their initial letters must be shown on the board in the final round.

After running the program on the terminal with arguments, the output of the program must be displayed to both “Battleship.out” file and the terminal.

2. Design

Reading from files and placing ships

When the game starts program needs to check if the given argument files(Player1 ships, Player 2 ships, Player1 moves, Player2 moves) exist or if their name is misspelled or not. If they are incorrect, file names must be collected in a list to prompt an error message to the user. If this list is not empty program must be terminated. Otherwise, ships must be positioned at the grid. "Battleship" and "Patrol boat" ships positions must be taken from optional txt files. It is not needed to check these optional files for correctness. If there are less arguments than expected in the command line program must be terminated.

In ship txt files, the space between semicolons represents empty squares and other ones represent ships with their initial letters so the program must split from semicolons. A grid must be created with nested list comprehension that contains initial letters for ships and "- " for spaces.
Battleship and patrol boat indexes must be taken from optional files and kept in another list

If the player enters an unwanted type of ship or makes a mistake with the length of a ship, the program must prompt an error to the user and exit the code.

There will be a function for this purpose.

Defining players and ships

The "Ship" and "Player" classes will be used for this purpose. This will make the job easier for other functions. Such as, executing a move by a player or counting sunk ships.

Each ship will have count, symbol, and length information.

Each player will have ship grids, ship positions from optional files, hidden boards, and a list that contains their moves.

Making the moves and handling exceptions

There will be a function that takes three parameters player, enemy, and move index.

Before making the moves, their correctness must be controlled by try ,except blocks.
After splitting moves from the file, if a move is missing an operand or operands, “IndexError” must be prompted. İf the operands exist but the program cannot interpret them “ValueError” must be prompted. If the operands exceed the boundaries of the grid “AssertionError” must be prompted.
After these errors, the program must keep reading the same player's input file for a valid move and then move on to other player. If an unpredictable error occurs ” kaBOOM: run for your life!” must be printed and the program must be terminated.

If there is a ship at move index then that index in the hidden board must be changed to “X”, if the index is empty hidden board must be changed to “O”.

Hidden boards are multidimensional python lists since they have rows and columns in them.

Counting sunk and remaning ships

For this purpose, the function must take two parameters, the type of the ship and the player.

The function will get all hit positions at the hidden board in a list and check if the list contains all indexes of the wanted type of ship. İf there is a ship that is been hit completely, the counter goes up by one. It will return a symbolic message for sunk ships and the number of remaining ships.

This step uses a multi-dimensional python list since there is more than one ship for one type of ship most of the time.

Writing moves to screen

At every round, hidden boards and remaining ships must be displayed to screen then other player's moves must be prompted too.
To do that, an output function must be written that can both print to the terminal and to out file. Row numbers and column letters must be shown on the hidden board.
At the final round, if there are remaining ships their initials must be shown on the hidden board.

Declaring the winner

In each round program must check if the game is over. This can be handled by counting the remaining ship's function. If this function returns 0 for both players then the game is a draw. Otherwise one of the players is the winner of the game.

3. Programmer’s Catalogue

Description Of Program And Functions

- Main part of the program

```
alphabet = ["A", "B", "C", "D", "E", "F", "G", "H", "I", "J"]
hidden_board1 = [["-" for i in range(10)] for j in range(10)]
hidden_board2 = [["-" for k in range(10)] for m in range(10)]

outfile = open("Battleship.out", "w")

# check if the files exist
file = []
for ele in sys.argv[1:5]:
    try:
        open(ele, "r")
    except IOError:
        file.append(ele)


if file:
    output_function(f"IOError: input file(s) {', '.join(file)} is/are not
reachable.\n")
    exit()
else:
    try:
        player1_txt, player2_txt = open(sys.argv[1], "r"), open(sys.argv[2],
"r")
        player1_in, player2_in = open(sys.argv[3], "r"), open(sys.argv[4],
"r")
    except IndexError:
        output_function("IndexError: Missing Argument")
        exit()

    try:
        player1_moves = player1_in.read().rstrip(";").split(";")
        player2_moves = player2_in.read().rstrip(";").split(";")

        act_ships1, opt_ships1 = place_ships(player1_txt,
"OptionalPlayer1.txt")
        act_ships2, opt_ships2 = place_ships(player2_txt,
"OptionalPlayer2.txt")

        player1 = Player(act_ships1, opt_ships1, hidden_board1,
player1_moves)
        player2 = Player(act_ships2, opt_ships2, hidden_board2,
player2_moves)

        output_function("Battle of Ships Game\n\n")

        ship_str = f"Carrier\t\t{' '.join(count_ships(carrier,
player1)[1])}\t\t\t\tCarrier\t\t{' '.join(count_ships(carrier,
player2)[1])}\n" \
                   f"Battleship\t{' '.join(count_ships(battle_ship,
player1)[1])}\t\t\t\tBattleship\t{' '.join(count_ships(battle_ship,
player2)[1])}\n" \
                   f"Destroyer\t{' '.join(count_ships(destroyer,
player1)[1])}\t\t\t\tDestroyer\t{' '.join(count_ships(destroyer,
player2)[1])}\n" \
                   f"Submarine\t{' '.join(count_ships(submarine,
player1)[1])}\t\t\t\tSubmarine\t{' '.join(count_ships(submarine,
player2)[1])}\n" \
                   f"Patrol Boat\t{' '.join(count_ships(patrol_boat,
player1)[1])}\t\t\tPatrol Boat\t{' '.join(count_ships(patrol_boat,
player2)[1])}\n"

        output_function(f"Player1's Move\n\n")
        output_function(f"Round : {1}\t\t\t\t\tGrid Size: 10x10\n\n")
        output_function("Player1's Hidden Board\t\tPlayer2's Hidden Board\n")
        output_function(f"  {' '.join(alphabet)}\t\t  {'
'.join(alphabet)}\n")
        for j in range(10):
            output_function(
                f"{str(j + 1) : <2}{' '.join(player1.board[j])}\t\t{str(j +
1) : <2}{' '.join(player2.board[j])}\n")
        output_function("\n")
        output_function(ship_str)
        output_function(f"\nEnter your move: {player1.moves[0]}\n")

        i, j = 0, 0
        round = 1
        while i < len(player1.moves):
            try:
                move_player(player1, player2, i)
            except (IndexError, ValueError, AssertionError) as e:
                output_function(f"{e}\n")
                output_function(f"Enter your move: {player1.moves[i+1]}\n")
                i += 1
                continue
            else:
                i += 1

            write_moves(player2, j, round)
            while j < len(player2.moves):
                try:
                    move_player(player2, player1, j)
                except (IndexError, ValueError, AssertionError) as e:
                    output_function(f"{e}\n")
                    output_function(f"Enter your move:
{player2.moves[j+1]}\n")
                    j += 1
                    continue
                else:
                    round += 1
                    j += 1
                    break

            if game_over()[1]:
                final_write()
                break
            else:
                write_moves(player1, i, round)

    except Exception:
        output_function("kaBOOM: run for your life!\n")
        exit()
```

The game is being played in this part.
There are two while loops to move on to the next move of the current player if there is an error in the input file.

- Class Ship

```
class Ship:
    def __init__(self, count, symbol, length):
        self.count = count
        self.symbol = symbol
        self.length = length

carrier = Ship(1, "C", 5)
battle_ship = Ship(2, "B", 4)
destroyer = Ship(1, "D", 3)
submarine = Ship(1, "S", 3)
patrol_boat = Ship(4, "P", 2)
```

This class has 3 attributes
Count, for number of the ships
Symbol, for the initial of the ship’s name
Length, for the number of squares of the ship
These attributes are used for mostly counting the remaining ships.

- Class Player

```
class Player:
    def __init__(self, ships, opt_ships, board, moves):
        self.ships = ships
        self.opt_ships = opt_ships
        self.board = board
        self.moves = moves

player1 = Player(act_ships1, opt_ships1, hidden_board1, player1_moves)
player2 = Player(act_ships2, opt_ships2, hidden_board2, player2_moves)
```

This class has 4 attributes
This class has 4 attributes
Ships, for general ship layout of the player
Opt_ships, for “battleship” and “patrol boat” positions
Board, for hidden boards of players
Moves, for the list containing moves of the players
This class makes the code easier to read and shorter since the program can use the same function for both players.

- output_function(out)

```
def output_function(out):
    print(out, end="")
    outfile.write(out)
```

Writes to both terminal and “Battleship.out” file.

- place_ships(file_name, opt_file)

```
def place_ships(file_name, opt_file):
    player_lines = file_name.read().splitlines()
    player = [item.split(";") for item in player_lines]


    for list1 in player:
        for element in list1:
            if element.isalpha():
                if element not in ["C", "B", "D", "S", "P"]:
                    raise Exception


    def count_letter(ship):
        count = sum(1 if element == ship.symbol else 0 for lis1 in player for
element in lis1)
        return count == ship.length * ship.count
    for element in [carrier, battle_ship, destroyer, submarine, patrol_boat]:
        if count_letter(element):
            pass
        else:
            raise Exception

    player_ships = [["-" if ele == "" else ele.upper() for ele in j] for
count, j in enumerate(player)]

    b_list = []
    p_list = []
    with open(opt_file) as opt:
        optlines = opt.read().splitlines()
    for line in optlines:
        pure = line.rstrip(";")
        ship_info = pure[3:].split(";")
        index_x = int(ship_info[0].split(",")[0])
        index_y = ship_info[0].split(",")[1]
        alp_index = alphabet.index(index_y)

        if pure[0] == "B":
            if ship_info[1] == "right":
                b_list.append([(index_x, alphabet[alp_index]), (index_x,
alphabet[alp_index+1]), (index_x, alphabet[alp_index+2]), (index_x,
alphabet[alp_index+3])])
            elif ship_info[1] == "down":
                b_list.append([(index_x, alphabet[alp_index]), (index_x+1,
alphabet[alp_index]), (index_x+2, alphabet[alp_index]), (index_x+3,
alphabet[alp_index])])

        elif pure[0] == "P":
            if ship_info[1] == "right":
                p_list.append([(index_x, alphabet[alp_index]), (index_x,
alphabet[alp_index+1])])
            elif ship_info[1] == "down":
                p_list.append([(index_x, alphabet[alp_index]), (index_x+1,
alphabet[alp_index])])
    hard_ones = [b_list, p_list]

    return player_ships, hard_ones
```

This function takes the ship txt file and the optional player txt file as a parameter.

first letters and spaces are splitted from txt file and checked if the letters are matching with initials of ship types. If not, Exception is raised. Then there is a function to count every letter in the txt file. If the count of a letter exceeds the ship's length*ship's count "Exception" is raised. If there is no exception raised ship layout will be created in a list comprehension.

Then "battleship" and "patrol boat" positions are appended to a list to later use them in other functions.

The function returns the ship layout and positions of "b" and "p"s.

- move_player(player, enemy, index)

```python
def move_player(player, enemy, index):

    if "," not in player.moves[index] or len(player.moves[index]) == 0 or
player.moves[index].split(",")[0] == "" or player.moves[index].split(",")[1]
== "" :
        raise IndexError(f"IndexError: Missing index at move
'{player.moves[index]}'.")

    if player.moves[index].split(",")[1].isalpha() is False or
player.moves[index].split(",")[0].isdigit() is False or
len(player.moves[index]) > 4:
        raise ValueError(f"ValueError: '{player.moves[index]}' is not a valid
move.")

    player_x, player_y = int(player.moves[index].split(",")[0]),
player.moves[index].split(",")[1]

    assert 1 <= int(player.moves[index].split(",")[0]) <= 10 and
player.moves[index].split(",")[1] in alphabet, "AssertionError: Invalid
Operation."

    column = alphabet.index(player.moves[index].split(",")[1])

    assert not enemy.board[player_x - 1][column].isalpha(), "AssertionError:
Invalid Operation."

    if enemy.ships[player_x-1][column].isalpha():
        enemy.board[player_x-1][column] = "X"
    else:
        enemy.board[player_x-1][column] = "O"
```

This function takes 3 parameters, player, enemy, and player move index counter.

First, exceptions will be raised by checking missing indexes or inappropriate moves.
İf the move format is correct and correctly splitted then its boundaries will be checked by using assertion. Also if the target square is already been hit before, AssertionError must be raised.

If no exception is raised target square must be changed with the correct symbol in the enemy hidden board depending on if that square contains a ship or not.

- count_ships(ship_type, player)

```python
def count_ships(ship_type, player):

    board = []
    for count, row in enumerate(player.board):
        for i in range(len(row)):
            if row[i] == "X":
                board.append((count+1, alphabet[i]))
    if ship_type.symbol == "B":
        liste = []
        for items in player.opt_ships:
            for item in items:
                if len(item) == 4:
                    liste.append(item)


    elif ship_type.symbol == "P":
        liste = []
        for items in player.opt_ships:
            for item in items:
                if len(item) == 2:
                    liste.append(item)

    else:
        liste = []
        for count, row in enumerate(player.ships):
            for i in range(10):
                if row[i] == ship_type.symbol:
                    liste.append((count+1, alphabet[i]))
        liste = [liste]


    c = 0
    for items in liste:
        if set(items).issubset(set(board)):
            c += 1
    return ship_type.count-c, "X"*c+"-"*(ship_type.count-c)
```

This function counts the remaining ship of wanted type in the player's board and returns it with a symbolic expression to prompt the user.

- write_moves(player, index, round)

```
def write_moves(player, index, round):
    ship_str = f"Carrier\t\t{' '.join(count_ships(carrier,
player1)[1])}\t\t\t\tCarrier\t\t{' '.join(count_ships(carrier,
player2)[1])}\n" \
        f"Battleship\t{' '.join(count_ships(battle_ship,
player1)[1])}\t\t\t\tBattleship\t{' '.join(count_ships(battle_ship,
player2)[1])}\n" \
        f"Destroyer\t{' '.join(count_ships(destroyer,
player1)[1])}\t\t\t\tDestroyer\t{' '.join(count_ships(destroyer,
player2)[1])}\n" \
        f"Submarine\t{' '.join(count_ships(submarine,
player1)[1])}\t\t\t\tSubmarine\t{' '.join(count_ships(submarine,
player2)[1])}\n" \
        f"Patrol Boat\t{' '.join(count_ships(patrol_boat,
player1)[1])}\t\t\tPatrol Boat\t{' '.join(count_ships(patrol_boat,
player2)[1])}\n"

    output_function(f"\n{'Player1' if player == player1 else 'Player2'}'s
Move\n\n")
    output_function(f"Round : {round}\t\t\t\t\tGrid Size: 10x10\n\n")
    output_function("Player1's Hidden Board\t\tPlayer2's Hidden Board\n")
    output_function(f"  {' '.join(alphabet)}\t\t  {' '.join(alphabet)}\n")
    for j in range(10):
        output_function(f"{str(j + 1) : <2}{'
'.join(player1.board[j])}\t\t{str(j + 1) : <2}{'
'.join(player2.board[j])}\n")
    output_function("\n")
    output_function(ship_str)
    output_function(f"\nEnter your move: {player.moves[index]}\n")
```

This function prints rounds to the user. Takes three parameters to write the round and next move of the other player.

- final_write()

```
def final_write():
    ship_str = f"Carrier\t\t{' '.join(count_ships(carrier,
player1)[1])}\t\t\t\tCarrier\t\t{' '.join(count_ships(carrier,
player2)[1])}\n" \
               f"Battleship\t{' '.join(count_ships(battle_ship,
player1)[1])}\t\t\t\tBattleship\t{' '.join(count_ships(battle_ship,
player2)[1])}\n" \
               f"Destroyer\t{' '.join(count_ships(destroyer,
player1)[1])}\t\t\t\tDestroyer\t{' '.join(count_ships(destroyer,
player2)[1])}\n" \
               f"Submarine\t{' '.join(count_ships(submarine,
player1)[1])}\t\t\t\tSubmarine\t{' '.join(count_ships(submarine,
player2)[1])}\n" \
               f"Patrol Boat\t{' '.join(count_ships(patrol_boat,
player1)[1])}\t\t\tPatrol Boat\t{' '.join(count_ships(patrol_boat,
player2)[1])}\n"

    output_function(f"{game_over()[0]}")
    output_function("Final Information\n\n")
    output_function("Player1's Board\t\t\t\tPlayer2's Board\n")
    output_function(f"  {' '.join(alphabet)}\t\t  {' '.join(alphabet)}\n")
    for j in range(10):
        output_function(
            f"{str(j + 1) : <2}{' '.join(rem_ships(player1)[j])}\t\t{str(j +
1) : <2}{' '.join(rem_ships(player2)[j])}\n")
    output_function("\n")
    output_function(ship_str)
```

This function prints the final information of the board.

- game_over()

```
def game_over():
    a = count_ships(carrier, player1)[0] + count_ships(battle_ship,
player1)[0] + count_ships(destroyer, player1)[0] + count_ships(submarine,
player1)[0] + count_ships(patrol_boat, player1)[0]
    b = count_ships(carrier, player2)[0] + count_ships(battle_ship,
player2)[0] + count_ships(destroyer, player2)[0] + count_ships(submarine,
player2)[0] + count_ships(patrol_boat, player2)[0]
    if a == 0 and b == 0:
        return "It is a Draw!\n\n", True
    elif a == 0:
        return "Player2 Wins!\n\n", True
    elif b == 0:
        return "Player1 Wins!\n\n", True
    else:
        return None, False
```

In every round, this function calculates the remaining ships of both players and if one of them or both of them is equal to zero it announces to game's final information.

- rem_ships_board(player)

```
def rem_ships(player):
    liste = []
    for count, row in enumerate(player.ships):
        for i in range(len(row)):
            if row[i].isalpha():
                liste.append([count, i])

    for item in liste:
        if player.board[item[0]][item[1]].isalpha():
            pass
        else:
            player.board[item[0]][item[1]] = player.ships[item[0]][item[1]]

    return player.board
```

This function is used for printing the final information of the board. İt returns the hidden board but replaces the “-“s with the initial letters of the ships.

Time Spent On This Assignment

Time spent on analysis	Since this is a well-known game, understanding its rules took a little time, and understanding which exceptions I should handle and how I should handle them took a little bit more time than understanding the rules. I can say this part took 2 hours.
Time spent on design and implementation	This part involves deciding on which data structure to use and thinking about how to make things easier and more smooth for the code. Exception handling took a long time too. Overall this part took 20 hours.
Time spent on testing and reporting	There are many scenarios to test with this game, so I think the most important part is testing. And to clearly express myself, I need to write my report carefully. This part took 6 hours for me.

Reusability Of The Code

Since this is a game program and many other game rules are like each other, this program is likely to be used again in other studies. I think any programmer that wants to write another game or other version of battleship can understand my code and manipulate it for their needs.

4. User Catalogue

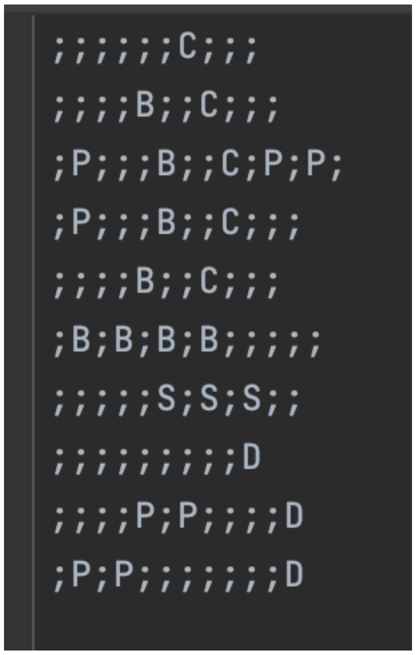
User Manual

The program must be executed by writing

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

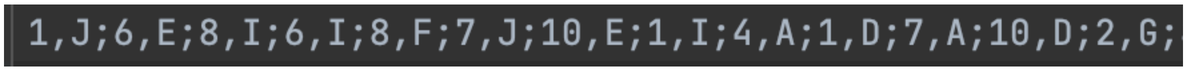
to terminal.

File names can be different, they just have to have the same content and same input format as the example ones.



```
;;;;;;C;;;
;;;;B;;C;;;
;P;;;B;;C;P;P;
;P;;;B;;C;;;
;;;;B;;C;;;
;B;B;B;B;;;;;
;;;;;S;S;S;;
;;;;;;;;;D
;;;;P;P;;;;D
;P;P;;;;;;;D
```

Player.txt format



```
1,J;6,E;8,I;6,I;8,F;7,J;10,E;1,I;4,A;1,D;7,A;10,D;2,G;
```

Player.in format

Txt and input Files must be in the same directory as the python file.

After running the program “Battleship.out” will be in the directory or moves can be seen in the terminal too.