Hacettepe University

Department of Computer Engineering

Battle of Ships Assignment Report

Course Name:

BBM103 – Introduction to Programming Lab

Assignment – 4

Mert ERGÜN - B2220356062

24.12.2022

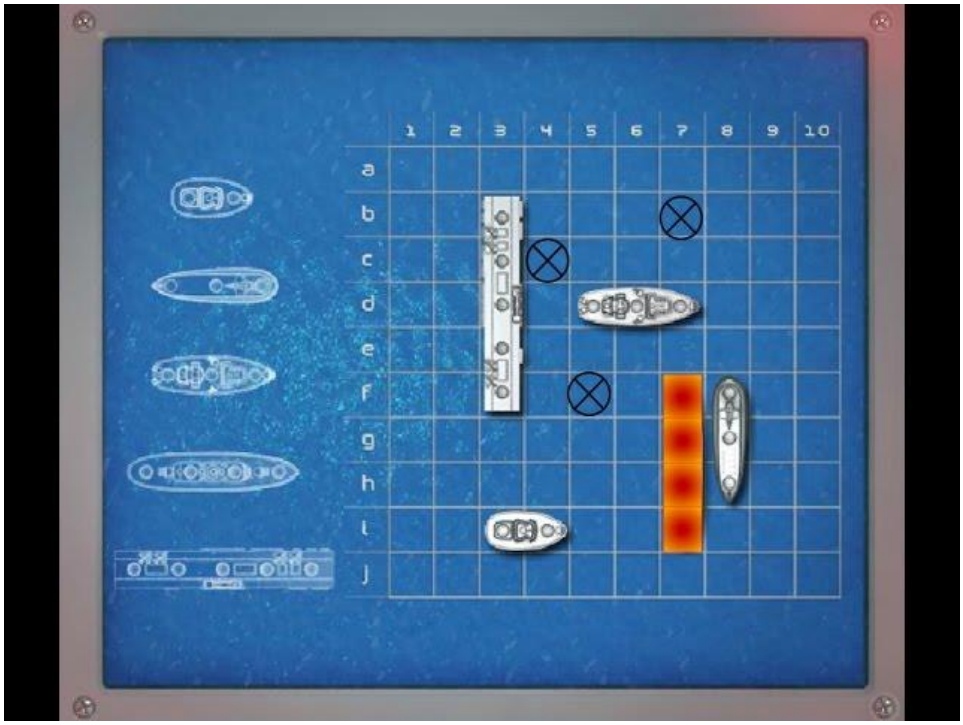
Table of Contents

1- Introduction

In this assignment, we were asked to design a simple Battleship game.

Sink the Admiral (also known as Battle of Ships or Sea Battle) is basically a two-player guessing-based strategy game. It is played on a 10 by 10 grid. Both players take turns shooting and trying to hit ships in each other's grid. Ships that are sunk are marked as sunk. At the end of the game, the first player whose ships are all sunk loses the game.

In normal gameplay, four 10x10 grids are used. Each square in these grids can be identified by a letter and a number. In one grid the player records his ships and the shots fired at his ships, in the other one he records the shots fired at the other player. Each ship can be placed horizontally and vertically. They cannot overlap each other.



2- Analysis

The problem to be solved in this assignment was to write a program that takes the moves from two different ".in" files and the initial ship positions from two different ".txt" files, plays each round in turn, makes the moves and prints the winner (and all other rounds) on the screen and in an ".out" file.

In addition to working fine under normal conditions, it was an assignment that had to correct or skip all erroneous input, never print a basic python error to the user, consider all situations that would generate an error, and hence the main difficulty.

The program works by the user giving 4 additional arguments to the console in addition to the program file:

The first two additional arguments are "Player1.txt" and "Player2.txt" which contain the initial ship positions.

The last two additional arguments are "Player1.in" and "Player2.in" which contain the user's moves throughout the game.

The program checks the entered arguments, looks at the files, removes missing or incorrect data and starts the game.

The output of the program at the end of the process can be as follows:

- Every move played and its effects on the grids.
- Error messages about incorrect files.
- Who is the winning player, how many rounds the game lasted, public grids at the end of the game.
- In case of a tie, a warning message that the game ends in a draw and the grids are publicly available.

3- Design

The code I wrote for the game consists of 11 different functions that work in conjunction with each other. Even though each function's task is independent of each other, they connect to each other to work. It analyzes the input data and writes on the output file (and to the console of course) according to the data.

3.0 – Clearing the Output File and Writing to It

"Append" mode is used to write data to the output file. This causes the data in the file not to be deleted, but to overlap each time the program is restarted. This function was created to prevent this and to delete the data in the output file every time the program is restarted and start the process from scratch.

Basically, what it does is to open the output file in "Write" mode for one time and completely erase its contents. It also creates the output file from scratch.

The "Write output file" function, which I used afterwards, is called in various places in the code to write the desired data into the output file. It can be made easier to use by adding a feature like `"+\n"` at the end, but this is not the case for this code.

3.1 – Checking Arguments and Input Files

It checks the arguments that the program is executed with on the console, flags files that are not in the directory where the program is located, and stops the program after printing an error message about not being able to read these files. This prevents the program from running with a missing or problematic file. It also makes sure that only four arguments were entered with the program in the first place. If there are extra or missing arguments, the program will give the same error.

The data taken as arguments form the file names. Therefore, if an incorrect filename is entered, the program terminates by stating that this file cannot be accessed.

The correct way to call the program is;

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

3.2 – Reading Ship Positions (.txt Files)

Once the designed program has verified the required files, it now proceeds to read the contents of the files. First it reads the "initial ship positions". To do this it needs to look at the files named "Player1.txt" and "Player2.txt". It opens these files with the argument names given in the console. It makes the data in the format separated by ";" open for reading and terminates the program with an error if there is incorrect input. The reason for terminating the program here is that if an incorrect input is encountered in the initial ship positions, it is not possible to correct it.

The program then adds a "-" to the ship lists created for the players, assuming that if there is nothing between the “;” elements, there is no ship there. If there is an element there, it recognizes that there is a ship and adds it.

3.3 – Reading Player Moves (.in Files)

After reading the starting ship positions, it is time to read the players' moves. During the game, the players' moves stored in the "Player1.in" and "Player2.in" files are read out and played in order in each round.

The program reads the moves for both players separated by a ";" element. It adds each move separately to the list of moves for that player. It examines the elements added to the list, removes the incorrect ones and prints the error message on the screen at the beginning of the game. In this way, the user is aware that an incorrect game will be played. In addition, after deleting an erroneous move, it replaces it with the next move (unless, of course, it is also erroneous).

3.4 – Defining Ships

After receiving the starting position of the ships and the players' moves, the program starts to identify the ships on the players' lists. This step is especially crucial when there is more than one of the same ship (Battleship and Patrol Boat). Thanks to this step, when two different patrol boats are hit, players don't get a nonsense message like "one patrol boat sunk". Each patrol boat is identified separately. In addition, the size of the ships is also kept in a dictionary in this function, ready to be used later. Finally,

for ease of use and comprehension, in order to avoid using keywords such as ord, char, etc., the numeric equivalents of the letters that will be found in the players' grids throughout the game are also kept in a dictionary.

3.5 – Playing Turns

Once all the preparations are complete, it is time for the players to play their rounds. The round play function takes the player as an argument, and determines the other player separately. In each round it prints what needs to be printed on the screen and in the output file. The number of turns, the moves, whose turn it is, the "secret" boards of the players. Also printed under the board are that player's sunk or unsinkable ships.

From the list of the player's moves, the move of that turn is taken, played, and if the move hits a ship, a length is deleted from the list of ship lengths of the player hit, so that when all the lengths are deleted, it is certain that the ship is sunk. If not, the hit coordinate is replaced with the letter "O".

3.6 – Win Condition

When all ships of any player sink, the game is over and that player loses the game. In this case, the "winCondition" function is activated and gives information such as who the winning player is, the final state of the players' grids, and how many rounds the game lasted.

It is first called for the winning player. It prints that player's win on the screen and in the file, and prints the final information in the same way.

3.7 – Draw Condition

If one player has sunk all the other player's ships but has one ship left, the game is not over, the player who has sunk all his ships gets one last shot. If that player makes good use of this shot and manages to hit his opponent's last remaining ship, the game ends in a draw. In this case the "drawCondition" function is activated. It is not called for

any player, it prints the final information to the screen and file as well as the fact that the game is a draw.

3.8 – Playing Game in Main Loop

Once the endgame conditions and round play functions have been fully defined, it is time for the main game loop function to call these functions on a case-by-case basis. This function is recursive and constantly calls itself again from within itself. It takes the number of rounds as argument. It is called at the beginning of the game with argument "1". If it is indeed called with argument "1" (if the game has just started), it prints the name of the game on the screen.

Calls the round play function for both players in turn. It creates a counter of integers (initially zero) to count the unsinkable ships of both players. At the end of the round, it adds ships to these counters according to the players' data in the ship length dictionary.

It compares the unsinkable ships counters, if one of the two players has no unsinkable ships left (the counter is zero) then that player has lost the game, in which case it calls the Win Condition function.

If one player has all ships sunk and the other player has only one unsinkable ship, it plays the last round.

If both players have no unsinkable ships left, it calls the Draw Condition function.

If both players have unsinkable ships, it calls itself again, increasing the number of turns by one.

4 – Programmer’s Catalogue

Python code for “Battleships” program without any blank or comment line is:

```
import sys


def clear_output_file():
    """
    Clears the output file.
    """

    with open("Battleship.out", 'w') as f:
        pass


def write_output_file(message):
    """
    Writes the given message to the output file.
    """

    with open("Battleship.out", 'a') as f:
        f.write(message)


def check_arguments():
    """
    Checks if the number of arguments is correct.
    """

    try:
        if len(sys.argv) != 5:
            write_output_file("Invalid number of arguments. Expected 4, got " + str(len(sys.argv) - 1)
+ ".\n")
            raise ValueError("Invalid number of arguments. Expected 4, got " + str(len(sys.argv) - 1) +
".")
    except ValueError as e:
        print("ValueError:", e)
        sys.exit()


def check_input_files():
    """
    Checks if the input files are reachable.
    """
```

```
    problemMakers = []


    try:
        open(sys.argv[1])
    except FileNotFoundError:
        problemMakers.append(sys.argv[1])
    try:
        open(sys.argv[2])
    except FileNotFoundError:
        problemMakers.append(sys.argv[2])
    try:
        open(sys.argv[3])
    except FileNotFoundError:
        problemMakers.append(sys.argv[3])
    try:
        open(sys.argv[4])
    except FileNotFoundError:
        problemMakers.append(sys.argv[4])


    if len(problemMakers) > 0:
        print("IOError: input file(s) " + ", ".join(problemMakers) +" is/are not reachable.")
        write_output_file("IOError:  input  file(s)  "  +  ",  ".join(problemMakers)  +"  is/are  not
reachable.\n")
        sys.exit()


def read_ship_pos():
    """
    Reads the ship positions from the input files.
    """


    with open(sys.argv[1], 'r') as f:
        player1 = f.readlines()


    with open(sys.argv[2], 'r') as f:
        player2 = f.readlines()
    global playerShips1, playerShips2
    playerShips1 = []
    playerShips2 = []
    for row in player1:
        row = row.strip('\n')
```

```
        values = row.split(";")
        try:
            if len(values) != 10:
                raise IndexError("Player1.txt file should contain exactly 10 positions.")
        except IndexError as e:
            print(e)
            write_output_file(e)
            sys.exit()

        for value in values:
            if value != "":
                try:
                    assert len(value) == 1
                    assert value in "BCPSD"
                except AssertionError:
                    print("IndexError: Player1.txt file contains an invalid ship position.")
                    write_output_file("IndexError: Player1.txt file contains an invalid ship
position.\n")
                    sys.exit()

        values = [value if value else "-" for value in values]

        playerShips1.append(values)
    for row in player2:
        row = row.strip('\n')
        values = row.split(";")
        try:
            if len(values) != 10:
                raise IndexError("Player1.txt file should contain exactly 10 positions.")
        except IndexError as e:
            print(e)
            write_output_file(e)
            sys.exit()

        for value in values:
            if value != "":
                try:
                    assert len(value) == 1
                    assert value in "BCPSD"
                except AssertionError:
                    print("IndexError: Player1.txt file contains an invalid ship position.")
```

```
                    write_output_file("IndexError:   Player1.txt   file   contains   an   invalid   ship
position.\n")
                    sys.exit()

        values = [value if value else "-" for value in values]

        playerShips2.append(values)

def read_player_moves():
    """
    Reads the player moves from the input files.
    """

    global playerMoves1, playerMoves2

    with open(sys.argv[3], 'r') as f:
        playerMoves1 = f.readline().strip(";").split(";")
    for element in playerMoves1:
        try:
            if element[0] == "1" and element[1] == "0":
                assert len(element) == 4
            else:
                assert len(element) == 3
        except AssertionError:
            print("IndexError: Player1.in file should contain exactly 3 characters in a move.")
            write_output_file("IndexError: Player1.in file should contain exactly 3 characters in a
move.\n")
            playerMoves1.remove(element)
            continue
        except IndexError:
            print("IndexError: Player1.in file should contain exactly 3 characters in a move.")
            write_output_file("IndexError: Player1.in file should contain exactly 3 characters in a
move.\n")
            playerMoves1.remove(element)
            continue

        try:
            assert int(element[0]) in range(1, 11) or int(element[0:2]) in range(10,11)
            if element[0] == "1" and element[1] == "0":
                assert element[3] in "ABCDEFGHIJ"
            else:
                assert element[2] in "ABCDEFGHIJ"
```

```
        except AssertionError:
            print("AssertionError: Invalid operation in Player1.in file.")
            write_output_file("AssertionError: Invalid operation in player1.in file.\n")
            playerMoves1.remove(element)
            continue
        except ValueError:
            print("ValueError: Player1.in file contains an invalid move.")
            write_output_file("ValueError: Player1.in file contains an invalid move.\n")
            playerMoves1.remove(element)
            continue
        except IndexError:
            print("IndexError: Player1.in file contains an invalid move.")
            write_output_file("IndexError: Player1.in file contains an invalid move.\n")
            playerMoves1.remove(element)
            continue

    with open(sys.argv[4], 'r') as f:
        playerMoves2 = f.readline().strip(";").split(";")
    for element in playerMoves2:
        try:
            if element[0] == "1" and element[1] == "0":
                assert len(element) == 4
            else:
                assert len(element) == 3
        except AssertionError:
            print("IndexError: Player2.in file should contain exactly 3 characters in a move.")
            write_output_file("IndexError: Player2.in file should contain exactly 3 characters in a
move.\n")
            playerMoves2.remove(element)
            continue
        except IndexError:
            print("IndexError: Player2.in file should contain exactly 3 characters in a move.")
            write_output_file("IndexError: Player2.in file should contain exactly 3 characters in a
move.\n")
            playerMoves2.remove(element)
            continue

        try:
            assert int(element[0]) in range(1, 10) or int(element[0:2]) in range(10, 11)
            if element[0] == "1" and element[1] == "0":
                assert element[3] in "ABCDEFGHIJ"
```

```
            else:
                assert element[2] in "ABCDEFGHIJ"
        except AssertionError:
            print("AssertionError: Invalid operation in Player2.in file.")
            write_output_file("AssertionError: Invalid operation in Player2.in file.\n")
            playerMoves2.remove(element)
            continue
        except ValueError:
            print("ValueError: Player2.in file contains an invalid move.")
            write_output_file("ValueError: Player2.in file contains an invalid move.\n")
            playerMoves1.remove(element)
            continue
        except IndexError:
            print("IndexError: Player2.in file contains an invalid move.")
            write_output_file("IndexError: Player2.in file contains an invalid move.\n")
            playerMoves2.remove(element)
            continue

def define_ships(playerShip):
    """
    Defines the ships and their sizes in given player's ship positions lists.
    """

    global shipsLen, letters_to_numbers



    shipsLen = {
        "B1_1": 4, "B2_1": 4, "B2_2": 4, "B1_2": 4,
        "P3_1": 2, "P4_1": 2, "P3_2": 2, "P4_2": 2,
        "P1_1": 2, "P2_1": 2, "P1_2": 2, "P2_2": 2,
        "C_1": 5, "C_2": 5, "D_1": 3, "D_2": 3, "S_1": 3, "S_2": 3
    }



    letters_to_numbers = {
        "A": 0, "B": 1, "C": 2, "D": 3, "E": 4, "F": 5,
        "G": 6, "H": 7, "I": 8, "J": 9
    }

    problemMakers = ["B14", "B24", "P42", "P32","P12","P22"]
```

```
    for ship in problemMakers:
        for i in range(10):
            for j in range(10):
                match = False
                if playerShip[i][j]==ship[0]:
                    for n,w in [(1,0),(0,1),(-1,0),(0,-1)]:
                        try:
                            if playerShip[i+n][j+w]==ship[0] and (playerShip[i+n*2][j+w*2]==ship[0] or
ship[0]=="P"):
                                match = True
                                direction = (n,w)
                        except:
                            pass
                    if match:
                        playerShip[i][j]=ship[:2]
                        for t in range(1,int(ship[-1])):
                            playerShip[i+t * direction[0]][j+t * direction[1]] = ship[:2]
                        break
            if match:
                break


def playTurn(player):
    """
    Plays the turn of the given player. Takes the player as a parameter.
    """

    global playerMoves1, playerMoves2, playerShips1, playerShips2


    if player == 1:
        opposer = "_2"
        players_move = playerMoves1
        opposer_ship = playerShips2
    else:
        opposer = "_1"
        players_move = playerMoves2
        opposer_ship = playerShips1


    print("Player{}'s Move\n".format(player))
    write_output_file("Player{}'s Move\n\n".format(player))
```

```
print("Round: {}\t\t\t\t\tGrid Size: 10x10\n".format(gameturn))
write_output_file("Round: {}\t\t\t\t\tGrid Size: 10x10\n\n".format(gameturn))

print("Player1's Hidden Board:\t\tPlayer2's Hidden Board:\n")
write_output_file("Player1's Hidden Board:\t\tPlayer2's Hidden Board:\n\n")

print("  A B C D E F G H I J\t\t  A B C D E F G H I J")
write_output_file("  A B C D E F G H I J\t\t  A B C D E F G H I J\n")

for i in range(10):
    print(f"{i+1}", end="")
    write_output_file(f"{i+1}")
    for j in range(10):
        if playerShips1[i][j][0] == "O":
            to_print = "O"
        elif playerShips1[i][j][0] == "X":
            to_print = "X"
        else:
            to_print = "-"
        print(" "*(i!=9 or j!=0) + to_print, end="")
        write_output_file(" "*(i!=9 or j!=0) + f"{to_print}")
    print("\t\t", end="")
    write_output_file("\t\t")
    print(f"{i+1}", end="")
    write_output_file(f"{i+1}")
    for j in range(10):
        if playerShips2[i][j][0] == "O":
            to_print = "O"
        elif playerShips2[i][j][0] == "X":
            to_print = "X"
        else:
            to_print = "-"
        print(" "*(i!=9 or j!=0) + to_print, end="")
        write_output_file(" "*(i!=9 or j!=0) + f"{to_print}")
    print("")
    write_output_file("\n")
print("")
write_output_file("\n")
for j in [["C"], ["B1", "B2"], ["D"], ["S"], ["P1", "P2", "P3", "P4"]]:
    ship_1 = ""
```

```
    ship_2 = ""
    for k in j:
        if shipsLen[k + "_1"] == 0:
            ship_1 = "X " + ship_1
        else:
            ship_1 = ship_1 + "- "
        if shipsLen[k + "_2"] == 0:
            ship_2 = "X " + ship_2
        else:
            ship_2 = ship_2 + "- "
        if k[0] == "C":
            ship = "Carrier"
            tab_number = "\t\t"
        elif k[0] == "B":
            ship = "Battleship"
            tab_number = "\t"
        elif k[0] == "D":
            ship = "Destroyer"
            tab_number = "\t"
        elif k[0] == "S":
            ship = "Submarine"
            tab_number = "\t"
        elif k[0] == "P":
            ship = "Patrol Boat"
            tab_number = "\t"
        if k[0] == "P":
            tab_number2 = "\t" * 3
        else:
            tab_number2 = "\t" * 4
    print(f"{ship}{tab_number}{ship_1[:-1]}{tab_number2}{ship}{tab_number}{ship_2}")
    write_output_file(f"{ship}{tab_number}{ship_1[:-1]}{tab_number2}{ship}{tab_number}{ship_2}\n")
print("")
write_output_file("\n")
move = players_move[gameturn-1]
print("Enter your move: {}\n".format(move))
write_output_file("Enter your move: {}\n\n".format(move))
move = move.split(",")

try:
    shipsLen[opposer_ship[int(move[0]) - 1][letters_to_numbers[move[1]]]+opposer] -= 1
```

```
    except:
        pass


    opposer_ship[int(move[0])-1][letters_to_numbers[move[1]]] = ("O" if opposer_ship[int(move[0])-1][
        letters_to_numbers[move[1]]]    ==    "-"    else    "X")    +    opposer_ship[int(move[0])-
1][letters_to_numbers[move[1]]]


def winCondition(player):
    """
    Win condition function. It takes the player as a parameter. Prints the winner, last non-hidden tables
and writes it to the output file.
    """


    print("Player{} Wins!\n".format(player))
    write_output_file("Player{} Wins!\n\n".format(player))


    print("Final Information:\n")
    write_output_file("Final Information:\n\n")


    print("Player1's Board\t\t\t\tPlayer2's Board")
    write_output_file("Player1's Board\t\t\t\tPlayer2's Board\n")


    print("  A B C D E F G H I J\t\t  A B C D E F G H I J")
    write_output_file("  A B C D E F G H I J\t\t  A B C D E F G H I J\n")


    for i in range(10):
        print(f"{i+1}", end="")
        write_output_file(f"{i+1}")
        for j in range(10):
            to_print = playerShips1[i][j][0]
            print(" "*(i!=9 or j!=0) + to_print, end="")
            write_output_file(" "*(i!=9 or j!=0) + f"{to_print}")
        print("\t\t", end="")
        write_output_file("\t\t")
        print(f"{i+1}", end="")
        write_output_file(f"{i+1}")
        for j in range(10):
            to_print = playerShips2[i][j][0]
            print(" "*(i!=9 or j!=0) + to_print, end="")
            write_output_file(" "*(i!=9 or j!=0) + f"{to_print}")
        print("")
```

```
            write_output_file("\n")
        print("")
        write_output_file("\n")
        for j in [["C"], ["B1", "B2"], ["D"], ["S"], ["P1", "P1", "P3", "P4"]]:
            ship_1 = ""
            ship_2 = ""
            for k in j:
                if shipsLen[k + "_1"] == 0:
                    ship_1 = "X " + ship_1
                else:
                    ship_1 = ship_1 + "- "
                if shipsLen[k + "_2"] == 0:
                    ship_2 = "X " + ship_2
                else:
                    ship_2 = ship_2 + "- "
                if k[0] == "C":
                    ship = "Carrier"
                    tab_number = "\t\t"
                elif k[0] == "B":
                    ship = "Battleship"
                    tab_number = "\t"
                elif k[0] == "D":
                    ship = "Destroyer"
                    tab_number = "\t"
                elif k[0] == "S":
                    ship = "Submarine"
                    tab_number = "\t"
                elif k[0] == "P":
                    ship = "Patrol Boat"
                    tab_number = "\t"
                if k[0] == "P":
                    tab_number2 = "\t" * 3
                else:
                    tab_number2 = "\t" * 4
            print(f"{ship}{tab_number}{ship_1[:-1]}{tab_number2}{ship}{tab_number}{ship_2}")
            write_output_file(f"{ship}{tab_number}{ship_1[:-1]}{tab_number2}{ship}{tab_number}{ship_2}\n")

def drawCondition():
    """
    Draw condition function. Prints the draw message and writes it to the output file.
```

```
"""

print("It's a Draw!\n")
write_output_file("It's a Draw!\n\n")

print("Final Information:\n")
write_output_file("Final Information:\n\n")

print("Player1's Board\t\t\t\tPlayer2's Board")
write_output_file("Player1's Board\t\t\t\tPlayer2's Board\n")

print("  A B C D E F G H I J\t\t  A B C D E F G H I J")
write_output_file("  A B C D E F G H I J\t\t  A B C D E F G H I J\n")

for i in range(10):
    print(f"{i+1}", end="")
    write_output_file(f"{i+1}")
    for j in range(10):
        to_print = playerShips1[i][j][0]
        print(" "*(i!=9 or j!=0) + to_print, end="")
        write_output_file(" "*(i!=9 or j!=0) + f"{to_print}")
    print("\t\t", end="")
    write_output_file("\t\t")
    print(f"{i+1}", end="")
    write_output_file(f"{i+1}")
    for j in range(10):
        to_print = playerShips2[i][j][0]
        print(" "*(i!=9 or j!=0) + to_print, end="")
        write_output_file(" "*(i!=9 or j!=0) + f"{to_print}")
    print("")
    write_output_file("\n")
print("")
write_output_file("\n")
for j in [["C"], ["B1", "B2"], ["D"], ["S"], ["P1", "P1", "P3", "P4"]]:
    ship_1 = ""
    ship_2 = ""
    for k in j:
        if shipsLen[k + "_1"] == 0:
            ship_1 = "X " + ship_1
        else:
```

```
                ship_1 = ship_1 + "- "
            if shipsLen[k + "_2"] == 0:
                ship_2 = "X " + ship_2
            else:
                ship_2 = ship_2 + "- "
            if k[0] == "C":
                ship = "Carrier"
                tab_number = "\t\t"
            elif k[0] == "B":
                ship = "Battleship"
                tab_number = "\t"
            elif k[0] == "D":
                ship = "Destroyer"
                tab_number = "\t"
            elif k[0] == "S":
                ship = "Submarine"
                tab_number = "\t"
            elif k[0] == "P":
                ship = "Patrol Boat"
                tab_number = "\t"
            if k[0] == "P":
                tab_number2 = "\t" * 3
            else:
                tab_number2 = "\t" * 4
        print(f"{ship}{tab_number}{ship_1[:-1]}{tab_number2}{ship}{tab_number}{ship_2}")
        write_output_file(f"{ship}{tab_number}{ship_1[:-1]}{tab_number2}{ship}{tab_number}{ship_2}\n")

def playGame(turn):
    """
    Playing the game function. It takes the turn as a parameter. Calls other functions to play the game.
    """

    global gameturn, last_turn
    gameturn = turn

    if turn == 1:
        print("Battle of Ships Game\n")
        write_output_file("Battle of Ships Game\n\n")
    playTurn(1)
    playTurn(2)
```

```
    nonsunk_ships1 = 0
    nonsunk_ships2 = 0

    for ship in shipsLen:
        if ship[-1] == "1":
            nonsunk_ships1 += shipsLen[ship]
        else:
            nonsunk_ships2 += shipsLen[ship]

    if nonsunk_ships1 == 0 and nonsunk_ships2 == 0:
        drawCondition()

    elif nonsunk_ships1 == 0:
        if nonsunk_ships2 == 1 and not last_turn:
            playGame(turn+1)
            last_turn = True
        else:
            winCondition(2)

    elif nonsunk_ships2 == 0:
        if nonsunk_ships1 == 1 and not last_turn:
            playGame(turn+1)
            last_turn = True
        else:
            winCondition(1)

    else:
        playGame(turn+1)

if __name__ == "__main__":
    try:
        clear_output_file()
        check_arguments()
        check_input_files()

        read_ship_pos()
        read_player_moves()

        define_ships(playerShips1)
```

```
        define_ships(playerShips2)

        last_turn = False
        playGame(1)

    except Exception as e:
        print("kaBOOM: run for your life!")
        sys.exit()
```

Time Spent for Analysis	The analysis part was not particularly intense for this program. There was relatively little to research, everything was as clear and fixed as it could be. Still, I had to do a lot of research on how to implement what I had in my head before I started writing the code. It was especially challenging to think about how to distinguish between multiple ships of the same type. I must say that I didn't spend any extra time researching Battleships, a game I've been playing quite intensively since I was a kid. Considering this, I can say that the Analysis part took about just 2 hours for me.
Time Spent for Design	The design part was by far the most difficult part of this assignment. The code was very complex, and every time problems were solved, bigger problems appeared. Especially the part about handling all the errors and sending custom error messages to the user was challenging as it was necessary to check all the inputs one by one. Also differentiating between different ships of the same type was one of the most mind-numbing parts of the project. Trying to make sure that the code works all the time, entering many incorrect inputs to see different error messages, etc. made the design time even longer. In the end, when I consider all these things, I can say that the design part took me at least 15 hours for this project.
Time Spent for Reporting	Since I had already established a template for reporting from previous assignments and I was more experienced, the reporting part didn't scare me too much this time. After the design part was completely finished, I sat down for reporting and it took about 5 hours.

4.1 – Reusability of Program

Since the designed program is basically designed to work on its own, it may not be stable to be embedded in an external application. Apart from that, all of the comment lines in the program code are written in detail in order to explain to the user and the developer what each line does. With these comment lines and documentation in mind, the program can be made to run in any external application. However, it would be completely problematic if only a part of the code was used, as the entire code is made up of small snippets of code that are interconnected and use common and global variables. Missing one of them can cause the whole program to crash and give incorrect results.

5 – User's Catalogue

5.1 – User Manual

To play Battleships, you will need a .txt and .in file for each player before the game, in a format to be specified, containing the starting ship positions and the moves to be made each turn during the game.

Once the files are prepared, the file containing the first player's ship positions is named "Player1.txt". For the second player this file should be called "Player2.txt". The files containing the moves should be named "Player1.in" and "Player2.in" respectively.

After the files are prepared and named accordingly, run the program from the console as follows:

```
python3 Assignment4.py “Player1.txt” “Player2.txt” “Player1.in” “Player2.in”
```

If there are no errors in the files, the program runs, plays each round separately, and prints the output to the screen and to the file named "Battleship.out" to be created in the folder where the “.py” file of the program is located. Note that the output file is reset every time the program is restarted!

5.1.1 – Format of Ship Positions File

Files containing ship positions must be in ".txt" format. Each row in it corresponds to a row on your table in the game.

You start by placing the first element in each row. If you do not want to put any ships at the beginning of the line, just leave it blank. You can separate your ships, or spaces, with the ";" element. Between two ";" elements you can put either a letter (the initial letter of the ship you want to place) or a space.

Any other input will cause an error and the program will give you an error telling you which file has the problem.

Example:

```
;;;;;;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
```

5.1.2 – Format of Move File

The files containing the moves must be in ".in" format. The entire contents of the file must be written in one line, using new lines will cause the program to error and not work.

Each move in the file is separated from each other by the ";" element. At the beginning of the file, the move to be played in the first round of the game is placed, followed by ";" to move to the next move. One ";" is left at the end of the file.

Moves must always be in the form "NUMBER,LETTER", indicating the row and then the column of the point to be hit.

If the game is expected to last for how many rounds, that move should be written. Too many moves will not cause an error, but if a move is entered, the program will close with an error about insufficient moves.

Incorrectly entered moves do not cause the program to close, the program continues to search for moves afterwards.

Example:

5,E;10,G;8,I;4,C;8,F;4,F;7,A;4,A;

6 – Grading Table

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	3
Function Usage	15	15
Correctness, File I/O	30	30
Exceptions	20	20
Report	20	20
There are several negative evaluations	…	…