Fall
2022

Assignment 4 Report

HACETTEPE UNIVERSITY
DEPARTMENT OF COMPUTER ENGINEERING

LEEN SAID - 2220356194

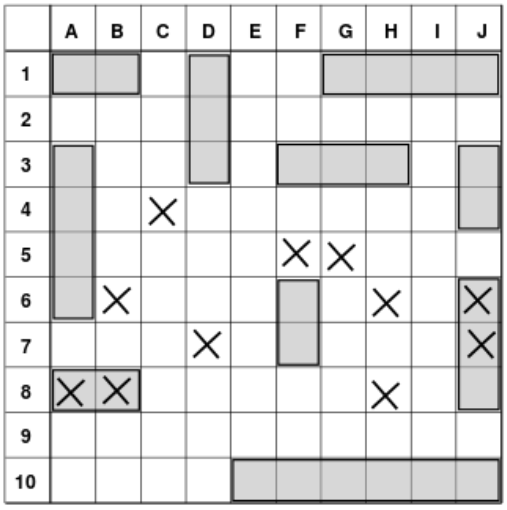
Introduction

Battleship (also known as Battleships or Sea Battle) is a strategy-type guessing game for two players. It is played on ruled grids on which each player's fleet of warships is marked. The locations of the fleets are concealed from the other player.

Players alternate turns calling "shots" at the other player's ships, and the game's objective is to destroy the opposing player's fleet. The game is played in four grids of squares 10x10, two for each player. The individual squares in the grids are identified by letter and number.

On one grid, the player arranges ships and records the shots by the opponent. On the other grid, the player records their shots. Before play begins, each player secretly arranges the ships on their hidden grid. Each ship occupies several consecutive squares on the grid, arranged either horizontally or vertically.

The type of ship determines the number of squares for each ship. The types and numbers of ships allowed are the same for each player. The ships should be hidden from the players' sight, and it is not allowed to see each other's pieces. The game is a discovery game in which players must discover their opponents' positions.



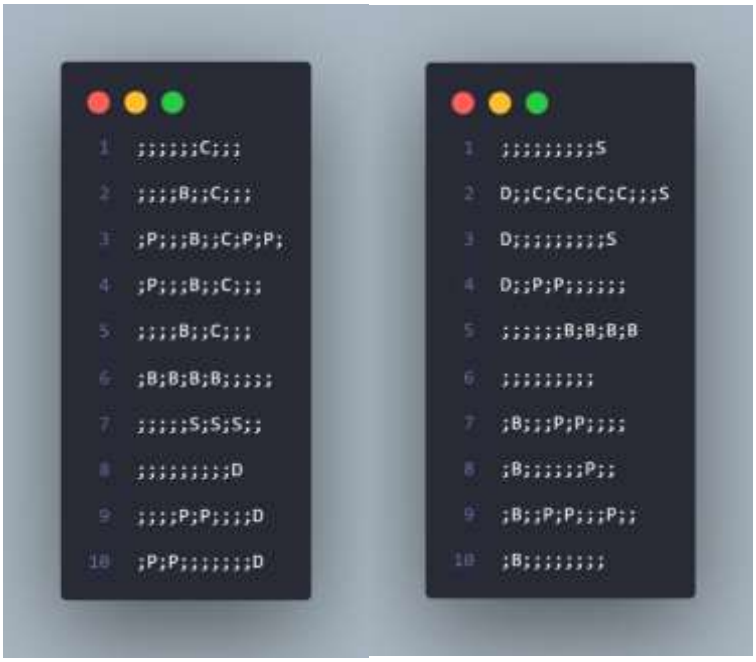
When all of the squares of a ship have been hit, the computer announces the sinking of the ship. If all a player's ships have been sunk, the game is over, and their opponent wins. If all ships of both players are sunk by the end of the round, the game is a draw.

Data

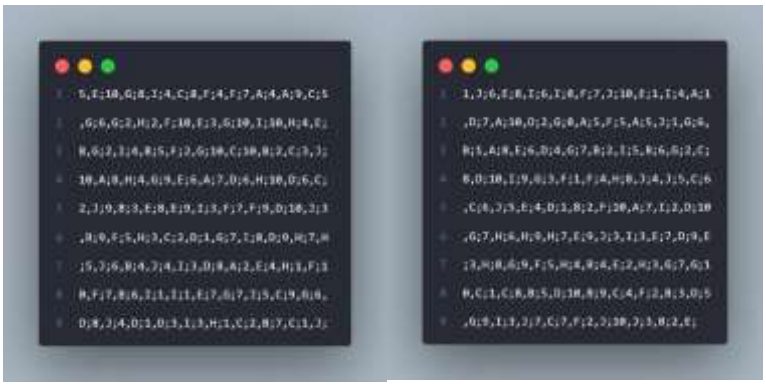
In Assignment 4, we are provided with 4 different text files from where the input for the program must be read.

They are Player1.text, Player2.txt, Player.in.txt and Player2.in.txt.

Player1.text and Player2.txt contain input from the users (players) which will be used to create the hidden grids. These hidden grids are the secret placements of the ships by the users.



Player1.in.text and Player2.in.txt contain input for coordinates of the cells to be “shot” by the users (players). These coordinates are separated by semicolons “;”.



Analysis

Apart from the main function of this program, which is to carry out the Battleship game between two players, this program should be able to read input from a text file, throw exceptions in case an error was found in the input, and then continue to read and execute other coordinates alternatively from both text files so the game can continue.

In order to solve the problem for this assignment, it's important to find a method to relate the two types of grids in the program. This method will be discussed in the Design section.

Design

- Create 4 empty lists for each grid in the game:

  two hidden grids with ship placements,

  and two 10x10 shown grids with their contents initially being –

- Read the 4 text files and assign them to 4 different variables with an IOError exception
- Create a function to identify ships of the same type available in the grid in a number greater than 1.
- Create a function to read individual coordinates from the text file and return the coordinates as a tuple. This function should throw different exception errors in case any error is found in the input format.
- Create a function that takes the coordinates generated by the previous function and locates them in the hidden grid. The function checks if this coordinate is empty or not. If empty, it changes the value in the shown grid into “O”, however, it changes the value in BOTH hidden and shown grids into “X”.
- Create a function to count the number of ships of each type in the grid
- Create 2 dictionaries with identical items, the keys of the dictionary are unique names of different ships, and the keys will be “-“ multiplied by the count of each type of ship.
- Create a function to update the dictionary of a particular player by checking for the availability of the ship type in the grid using the counter function. This dictionary will be displayed on the screen of the players.
- Create a function to determine the end of the game. This function should check EITHER or BOTH the sum of the return of the function “counter”. And will return true.
- Initiate the for loop by setting variable round =1 and using “While True:”. The round variable will be incremented after every iteration of the loop
- In the loop, if the round number is odd, it’s player1’s turn and vice versa.
- The loop will terminate if the “determine_game_end” function returns True.

Data Structures used:

As can be seen below, there's always a pair of data structures that serve the same purpose. The data structures used and their purposes are listed below:

- Player1_hidden & Player2_hidden dictionaries:
  These dictionaries contain data about the placements of ships by each player. It contains 10 keys ranging from 1-10. The values of these keys are Values1_l & Values2_l lists (see below) processed by function identify_ships(lis).

- Player1_shown & Player2_shown lists:
  These are multidimensional lists with 10 lists containing 10 elements each. The values inside these lists are initially "-". These values, however, going to change throughout the game. The players are going to be able to see these lists throughout the game.

- Values1_l & Values2_l lists:
  They are multidimensional lists, each list has the data given in each line in the text file separated by ";".

- List_of_coordinates1 & List_of_coordinates2 lists:
  These lists consist of the coordinates given by the users separated by ";".

- Dict1 and Dict2 dictionaries:
  These dictionaries are useful to update the players with the status of the ships in their grids.

- Lines list
  This list served the role of assisting in writing multiple lines of functions to a text file.

Functions used:

- **Identify_ships**

```python
# function to number Battleships and Patrol boats since there are more than 1
def identify_ships(lis):
    p = 1
    b = 1
    hidden = {}
    for i in lis:
        for k in i:
            if k == "P":
                indouter = lis.index(i)
                indinner = i.index(k)
                if lis[indouter][indinner + 1] == "P":
                    lis[indouter][indinner] = "P" + str(p)
                    lis[indouter][indinner + 1] = "P" + str(p)
                else:
                    lis[indouter][indinner] = "P" + str(p)
                    lis[indouter + 1][indinner] = "P" + str(p)
                p += 1

            if k == "B":
                indouter = lis.index(i)
                indinner = i.index(k)
                if lis[indouter][indinner + 2] == "B":
                    lis[indouter][indinner] = "B" + str(b)
                    lis[indouter][indinner + 1] = "B" + str(b)
                    lis[indouter][indinner + 2] = "B" + str(b)
                    lis[indouter][indinner + 3] = "B" + str(b)
                else:
                    lis[indouter][indinner] = "B" + str(b)
                    lis[indouter + 1][indinner] = "B" + str(b)
                    lis[indouter + 2][indinner] = "B" + str(b)
                    lis[indouter + 3][indinner] = "B" + str(b)

                b += 1

    for i in range(len(lis)):
        hidden[i] = lis[i]
    for i in hidden.values():
        if len(i) == 9:
            i.append("")
        elif len(i) == 8:
            i.append("", "")
    return hidden
```

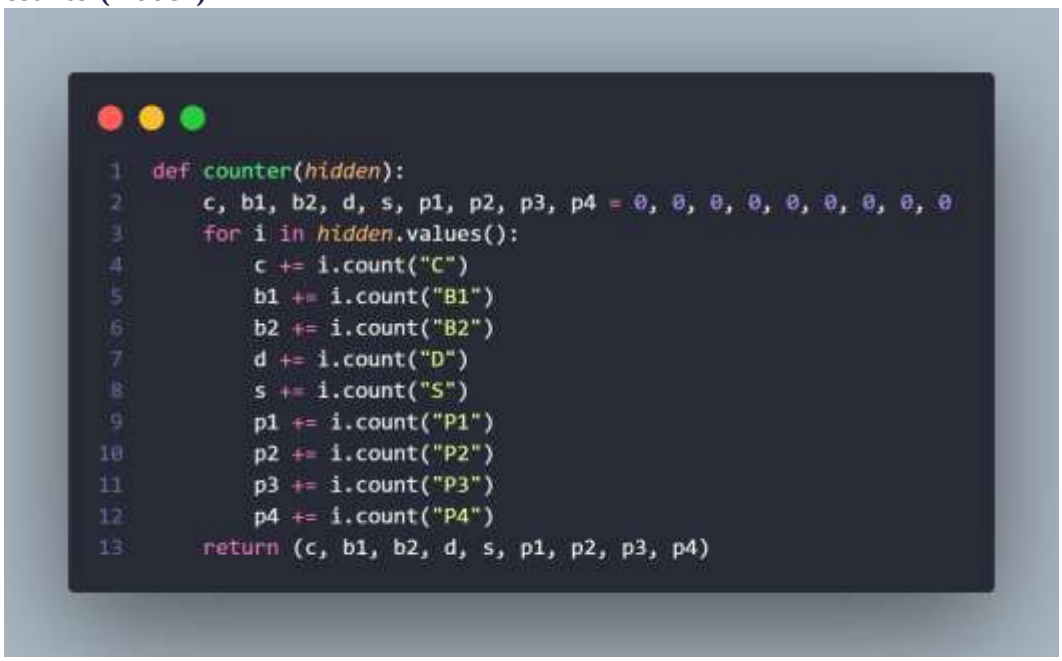
- **read_input(list_Cord,indexx)**

```
# function to return a set of row and column to work with
def read_input(list_Cord, indexx):
    val = list_Cord[indexx]
    try:
        # find the index of "," separator
        ind = val.index(",")
        # try and except
        try:
            # locate the row, here -1 is added since the indexing of the grids in our program
            # starts from 0, however it starts from 1 in the input given by users
            row = int(val[0:ind])
            if int(val[0:ind]) == "":
                raise IndexError
            try:
                column_in_letter = val[ind + 1 :]
                try:
                    if val[ind + 1 :] == "":
                        raise IndexError
                    if len(column_in_letter) > 1:
                        raise Exception
                    # convert the letter into a number according to its alphabetical order
                    column_in_int = alpha.index(column_in_letter)
                    cords = (row, column_in_int, column_in_letter)
                    return cords
                except IndexError:
                    print("IndexError: column was not given.")
                    writeto.writelines("\nIndexError: column was not given.\n")
                    return False
                except Exception:
                    print("No semi-colon separator was given!")
                    writeto.writelines("\nNo semi-colon separator was given!\n")
                    return False
            except ValueError:
                print("ValueError: inappropriate value given for column.")
                writeto.writelines(
                    "\nValueError: inappropriate value given for column.\n"
                )
                return False
        except IndexError:
            print("IndexError: row was not given.")
            writeto.writelines("\nIndexError: row was not given.\n")
            return False
        except ValueError:
            print("ValueError: inappropriate value given for row.")
            writeto.writelines("\nValueError: inappropriate value given for row.\n")
            return False
    except IndexError:
        print("IndexError: No input was given.")
        writeto.writelines("\nIndexError: No input was given.\n")
        return False
    except ValueError:
        print("No comma separator was given!")
        writeto.writelines("\nNo comma separator was given!\n")
        return False
    except:
        print("kaBOOM: run for your life")
        writeto.writelines("\nkaBOOM: run for your life\n")
        return False

```

- **update_grid(cordinates, hidden, shown)**

```
# function to update grids using the set of row and column given by user
def update_grid(cordinates, hidden, shown):
    try:
        row, column, col_letter = cordinates
        # if the cell in the hidden grid is empty turn the cell
        # with same coordinates in the hidden grid into O
        row = int(row) - 1
        if hidden[row][column] == "":
            shown[row][column] = "O"

        # if the cell in the hidden grid is NOT empty turn the cell with same
        # coordinates in the hidden grid into X as well as change the value in
        #  the hidden grid into "-" to indicate it's been emptied
        else:
            shown[row][column] = "X"
            hidden[row][column] = ""
    except:
        print("kaBOOM: run for your life")
        writeto.writelines("\nkaBOOM: run for your life\n")
```

- **counter(hidden)**

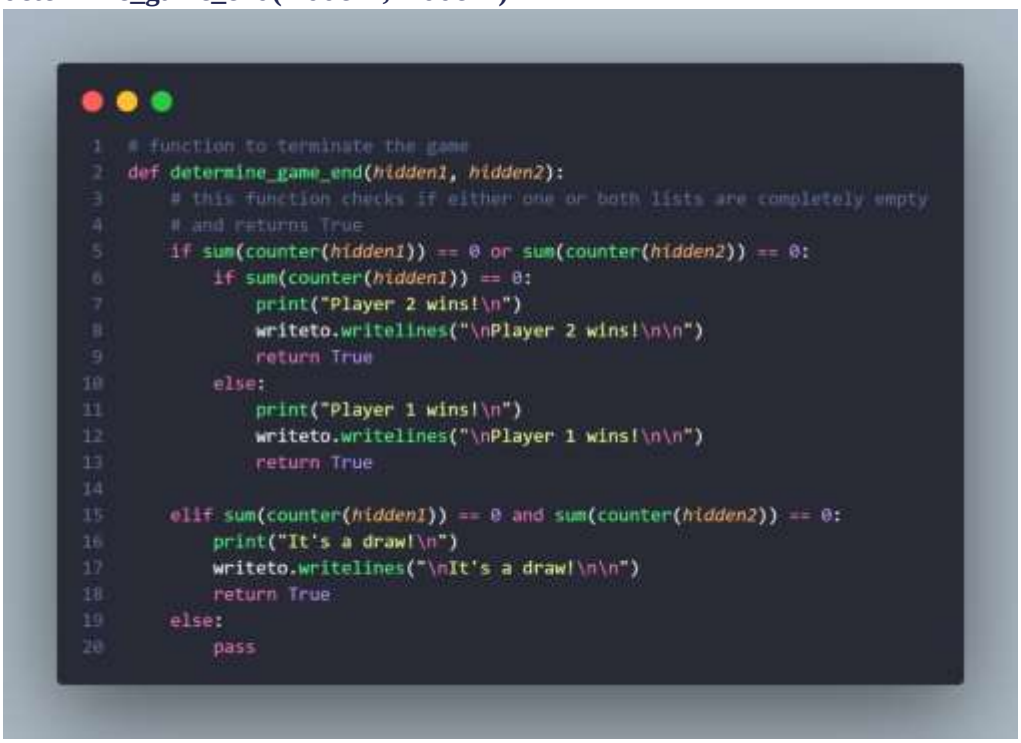


```
def counter(hidden):
    c, b1, b2, d, s, p1, p2, p3, p4 = 0, 0, 0, 0, 0, 0, 0, 0, 0
    for i in hidden.values():
        c += i.count("C")
        b1 += i.count("B1")
        b2 += i.count("B2")
        d += i.count("D")
        s += i.count("S")
        p1 += i.count("P1")
        p2 += i.count("P2")
        p3 += i.count("P3")
        p4 += i.count("P4")
    return (c, b1, b2, d, s, p1, p2, p3, p4)
```

- **update_dict(ship,dict)**

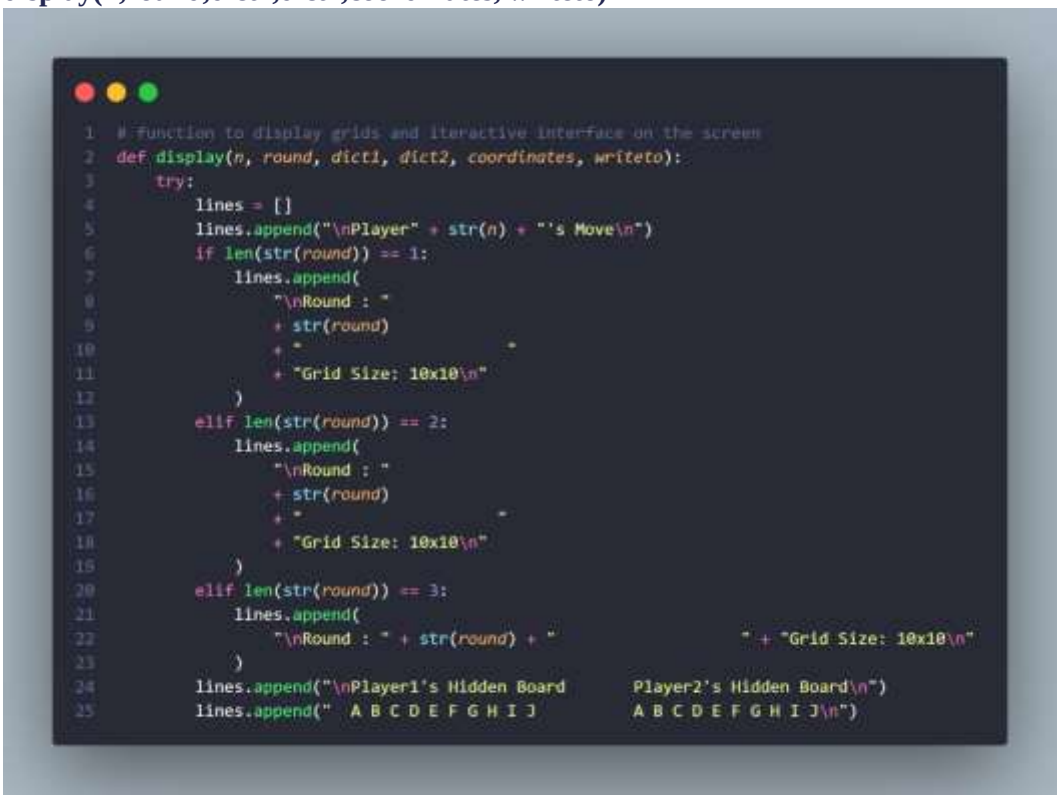
```
def update_dict(dict, hidden):
    c, b1, b2, d, s, p1, p2, p3, p4 = counter(hidden)
    if c == 0:
        dict["Carrier"] = ["X"]

    if b1 == 0:
        lb[1] = "X"
    if b2 == 0:
        lb[0] = "X"
    dict["Battleship"] = lb
    if d == 0:
        dict["Destroyer"] = ["X"]

    if s == 0:
        dict["Submarine"] = ["X"]

    if p1 == 0:
        lp[0] = "X"
    elif p2 == 0:
        lp[1] = "X"
    elif p3 == 0:
        lp[2] = "X"
    elif p4 == 0:
        lp[3] = "X"
    dict["Patrol Boat"] = lp


# function to terminate the game
def determine_game_end(hidden1, hidden2):
    # this function checks if either one or both lists are completely empty
    # and returns True
    if sum(counter(hidden1)) == 0 or sum(counter(hidden2)) == 0:
        if sum(counter(hidden1)) == 0:
            print("Player 2 wins!\n")
            writeto.writelines("\nPlayer 2 wins!\n\n")
            return True
        else:
            print("Player 1 wins!\n")
            writeto.writelines("\nPlayer 1 wins!\n\n")
            return True

    elif sum(counter(hidden1)) == 0 and sum(counter(hidden2)) == 0:
        print("It's a draw!\n")
        writeto.writelines("\nIt's a draw!\n\n")
        return True
    else:
        pass
```

- **determine_game_end(hidden1, hidden2)**



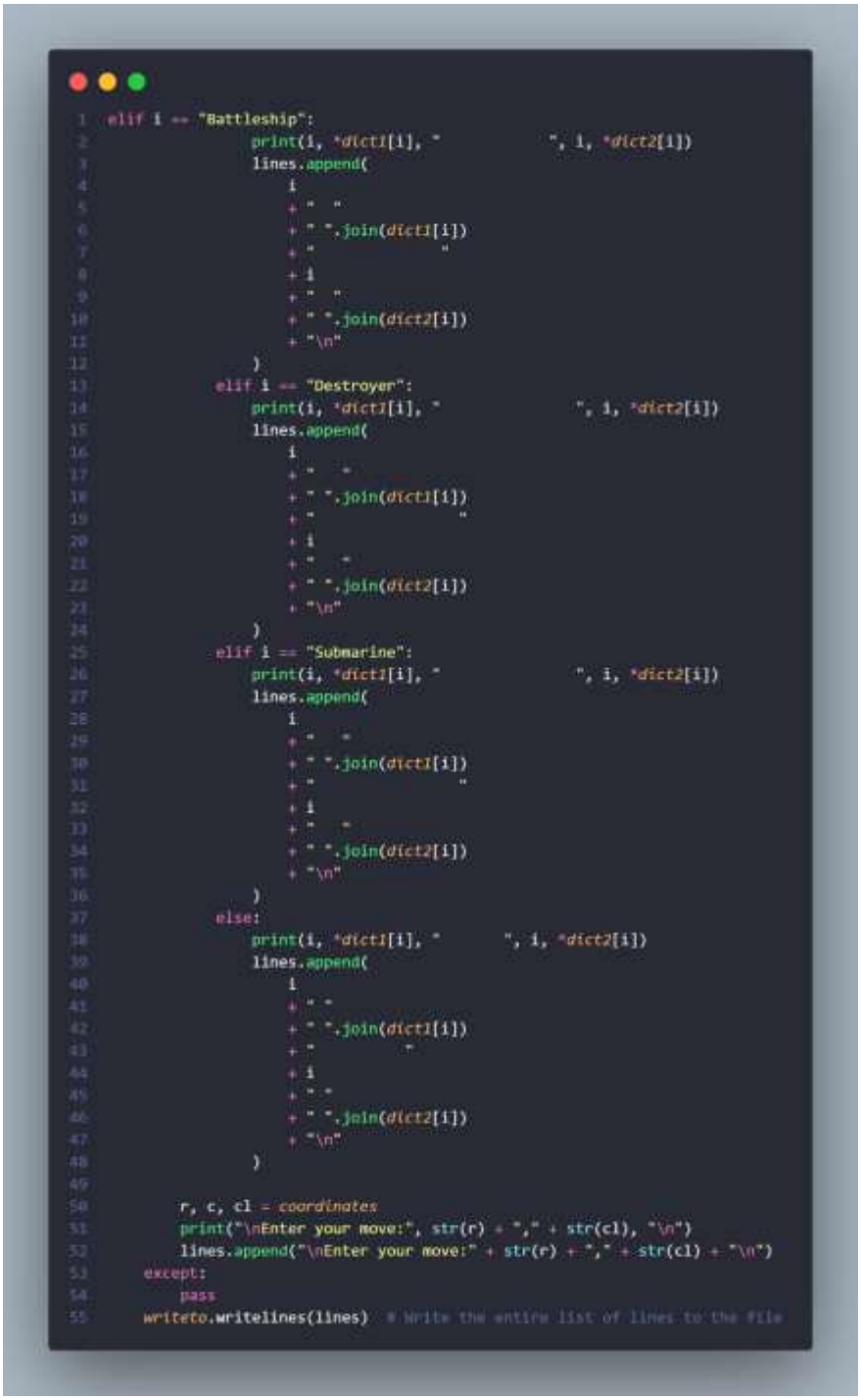
```python
# function to terminate the game
def determine_game_end(hidden1, hidden2):
    # this function checks if either one or both lists are completely empty
    # and returns True
    if sum(counter(hidden1)) == 0 or sum(counter(hidden2)) == 0:
        if sum(counter(hidden1)) == 0:
            print("Player 2 wins!\n")
            writeto.writelines("\nPlayer 2 wins!\n\n")
            return True
        else:
            print("Player 1 wins!\n")
            writeto.writelines("\nPlayer 1 wins!\n\n")
            return True

    elif sum(counter(hidden1)) == 0 and sum(counter(hidden2)) == 0:
        print("It's a draw!\n")
        writeto.writelines("\nIt's a draw!\n\n")
        return True
    else:
        pass
```

- **display(n,round,dict1,dict2,coordinates, writeto)**



```python
# function to display grids and iterative interface on the screen
def display(n, round, dict1, dict2, coordinates, writeto):
    try:
        lines = []
        lines.append("\nPlayer" + str(n) + "'s Move\n")
        if len(str(round)) == 1:
            lines.append(
                "\nRound : "
                + str(round)
                + "                    "
                + "Grid Size: 10x10\n"
            )
        elif len(str(round)) == 2:
            lines.append(
                "\nRound : "
                + str(round)
                + "                   "
                + "Grid Size: 10x10\n"
            )
        elif len(str(round)) == 3:
            lines.append(
                "\nRound : " + str(round) + "                  " + "Grid Size: 10x10\n"
            )
        lines.append("\nPlayer1's Hidden Board      Player2's Hidden Board\n")
        lines.append("  A B C D E F G H I J        A B C D E F G H I J\n")
```

```
print("\nPlayer" + str(n) + "'s Move")
        print("\nRound :", round, end="")
        print("                  ", "Grid Size: 10x10")
        print("\nPlayer1's Hidden Board       Player2's Hidden Board")
        print("   A B C D E F G H I J        A B C D E F G H I J")
        for q in range(1, 10):
            print(q, "", *player1_shown[q - 1], "    ", q, "", *player2_shown[q - 1])
            lines.append(
                str(q)
                + " "
                + " ".join(player1_shown[q - 1])
                + "        "
                + str(q)
                + " "
                + " ".join(player2_shown[q - 1])
                + "\n"
            )
        print("10", *player1_shown[9], "    ", "10", *player2_shown[9], "\n")
        lines.append(
            "10"
            + " ".join(player1_shown[9])
            + "       "
            + "10"
            + " ".join(player2_shown[9])
            + " \n"
        )
        lines.append("\n")
        for i in dict1.keys():
            if i == "Carrier":
                print(i, *dict1[i], "                ", i, *dict2[i])
                lines.append(
                    i
                    + "     "
                    + " ".join(dict1[i])
                    + "                "
                    + i
                    + "     "
                    + " ".join(dict2[i])
                    + "\n"
                )
```



```
elif i == "Battleship":
                print(i, *dict1[i], "            ", i, *dict2[i])
                lines.append(
                    i
                    + "  "
                    + " ".join(dict1[i])
                    + "              "
                    + i
                    + "  "
                    + " ".join(dict2[i])
                    + "\n"
                )
            elif i == "Destroyer":
                print(i, *dict1[i], "               ", i, *dict2[i])
                lines.append(
                    i
                    + "   "
                    + " ".join(dict1[i])
                    + "                "
                    + i
                    + "   "
                    + " ".join(dict2[i])
                    + "\n"
                )
            elif i == "Submarine":
                print(i, *dict1[i], "               ", i, *dict2[i])
                lines.append(
                    i
                    + "   "
                    + " ".join(dict1[i])
                    + "                "
                    + i
                    + "   "
                    + " ".join(dict2[i])
                    + "\n"
                )
            else:
                print(i, *dict1[i], "       ", i, *dict2[i])
                lines.append(
                    i
                    + " "
                    + " ".join(dict1[i])
                    + "            "
                    + i
                    + " "
                    + " ".join(dict2[i])
                    + "\n"
                )

        r, c, cl = coordinates
        print("\nEnter your move:", str(r) + "," + str(cl), "\n")
        lines.append("\nEnter your move:" + str(r) + "," + str(cl) + "\n")
    except:
        pass
    writeto.writelines(lines)  # Write the entire list of lines to the file
```

Programmer's catalog

Time spent:

Analyzing	Reading the assignment file, and the other input text files, as well as understanding the output the program is expected to generate took me between 2-4 hours.
Designing	Designing the method to be approached while solving the problem for this assignment took me the longest time. I wrote down my ideas and I had to find a way to connect all the data available to me together to work with the problem of the assignment to generate the required output. My work on writing the code for this problem was sparsed over the 3 weeks. It took me a minimum of 250 hours overall
Testing & reporting	Testing and reporting weren't very time-consuming. It took me between 3-5 hours to fix the small bugs faced by the program to execute the required output.

Code:

```
# Leen Said 2220356194

import sys

# create empty lists
player1_shown = []
player2_shown = []

# read the textfiles
try:
    f0 = open(sys.argv[1], "r")
    read0 = f0.readlines()
    f1 = open(sys.argv[2], "r")
    read1 = f1.readlines()
    f2 = open(sys.argv[3], "r")
    readinput1 = f2.readlines()
    f3 = open(sys.argv[4], "r")
    readinput2 = f3.readlines()
    writeto = open(sys.argv[5], "w")
except IOError as e:
    print("IOError: input file(s)", e, "is/are not reachable.")


# create list of lists using the placements of the ships by players
values1_l = []
for s in read0:
    values1 = str(s).replace("\n", "").split(";")
    values1_l.append(values1)

values2_l = []
for k in read1:
    values2 = str(k).replace("\n", "").split(";")
    values2_l.append(values2)

# create 2 multidimensional lists as the "shown grids" in the game
for i in range(10):
    player1_shown.append(["-", "-", "-", "-", "-", "-", "-", "-", "-", "-"])
    player2_shown.append(["-", "-", "-", "-", "-", "-", "-", "-", "-", "-"])


# turn the lines in the Player1/2.in.txt into one string
strr1 = ""
for cordinate1 in readinput1:
```

```
    cordinate1 = cordinate1.replace("\n", "")
    strr1 += cordinate1
    # separate the string by ";"
    list_of_coordinates1 = strr1.split(";")
# pop the last element of the list since it will be empty because the string was
split by ; at the end
list_of_coordinates1.pop()

strr2 = ""
for cordinate2 in readinput2:
    cordinate2 = cordinate2.replace("\n", "")
    strr2 += cordinate2
    list_of_coordinates2 = strr2.split(";")
list_of_coordinates2.pop()

alpha = ["A", "B", "C", "D", "E", "F", "G", "H", "I", "J"]

# function to number Battleships and Patrol boats since there are more than 1
def identify_ships(lis):
    p = 1
    b = 1
    hidden = {}
    for i in lis:
        for k in i:
            if k == "P":
                indouter = lis.index(i)
                indinner = i.index(k)
                if lis[indouter][indinner + 1] == "P":
                    lis[indouter][indinner] = "P" + str(p)
                    lis[indouter][indinner + 1] = "P" + str(p)
                else:
                    lis[indouter][indinner] = "P" + str(p)
                    lis[indouter + 1][indinner] = "P" + str(p)
                p += 1

            if k == "B":
                indouter = lis.index(i)
                indinner = i.index(k)
                if lis[indouter][indinner + 2] == "B":
                    lis[indouter][indinner] = "B" + str(b)
                    lis[indouter][indinner + 1] = "B" + str(b)
                    lis[indouter][indinner + 2] = "B" + str(b)
                    lis[indouter][indinner + 3] = "B" + str(b)
                else:
                    lis[indouter][indinner] = "B" + str(b)
```

```
                    lis[indouter + 1][indinner] = "B" + str(b)
                    lis[indouter + 2][indinner] = "B" + str(b)
                    lis[indouter + 3][indinner] = "B" + str(b)

                b += 1

    for i in range(len(lis)):
        hidden[i] = lis[i]
    for i in hidden.values():
        if len(i) == 9:
            i.append("")
        elif len(i) == 8:
            i.append("", "")
    return hidden


# function to return a set of row and column to work with
def read_input(list_Cord, indexx):
    val = list_Cord[indexx]
    try:
        # find the index of "," separator
        ind = val.index(",")
        # try and except
        try:
            # locate the row, here -1 is added since the indexing of the grids in
our program
            # starts from 0, however it starts from 1 in the input given by users
            row = int(val[0:ind])
            if int(val[0:ind]) == "":
                raise IndexError
            try:
                column_in_letter = val[ind + 1 :]
                try:
                    if val[ind + 1 :] == "":
                        raise IndexError
                    if len(column_in_letter) > 1:
                        raise Exception
                    # convert the letter into a number according to its
alphabetical order
                    column_in_int = alpha.index(column_in_letter)
                    cords = (row, column_in_int, column_in_letter)
                    return cords
                except IndexError:
                    print("IndexError: column was not given.")
                    writeto.writelines("\nIndexError: column was not given.\n")
```

```
                    return False
                except Exception:
                    print("No semi-colon separator was given!")
                    writeto.writelines("\nNo semi-colon separator was given!\n")
                    return False
            except ValueError:
                print("ValueError: inappropriate value given for column.")
                writeto.writelines(
                    "\nValueError: inappropriate value given for column.\n"
                )
                return False
        except IndexError:
            print("IndexError: row was not given.")
            writeto.writelines("\nIndexError: row was not given.\n")
            return False
        except ValueError:
            print("ValueError: inappropriate value given for row.")
            writeto.writelines("\nValueError: inappropriate value given for
row.\n")
            return False
    except IndexError:
        print("IndexError: No input was given.")
        writeto.writelines("\nIndexError: No input was given.\n")
        return False
    except ValueError:
        print("No comma separator was given!")
        writeto.writelines("\nNo comma separator was given!\n")
        return False
    except:
        print("kaBOOM: run for your life")
        writeto.writelines("\nkaBOOM: run for your life\n")
        return False


# function to update grids using the set of row and column given by user
def update_grid(cordinates, hidden, shown):
    try:
        row, column, col_letter = cordinates
        # if the cell in the hidden grid is empty turn the cell
        # with same coordinates in the hidden grid into O
        row = int(row) - 1
        if hidden[row][column] == "":
            shown[row][column] = "O"

        # if the cell in the hidden grid is NOT empty turn the cell with same
```

```
        # coordinates in the hidden grid into X as well as change the value in
        #  the hidden grid into "-" to indicate it's been emptied
        else:
            shown[row][column] = "X"
            hidden[row][column] = ""
    except:
        print("kaBOOM: run for your life")
        writeto.writelines("\nkaBOOM: run for your life\n")


# Dictionaries dict1 and dict2 are going to be printed under thr grid
lp = ["-", "-", "-", "-"]
lb = ["-", "-"]
dict1 = {
    "Carrier": ["-"],
    "Battleship": lb,
    "Destroyer": ["-"],
    "Submarine": ["-"],
    "Patrol Boat": lp,
}
dict2 = {
    "Carrier": ["-"],
    "Battleship": lb,
    "Destroyer": ["-"],
    "Submarine": ["-"],
    "Patrol Boat": lp,
}

# function counter() counts the number of each type of ship in the grid
def counter(hidden):
    c, b1, b2, d, s, p1, p2, p3, p4 = 0, 0, 0, 0, 0, 0, 0, 0, 0
    for i in hidden.values():
        c += i.count("C")
        b1 += i.count("B1")
        b2 += i.count("B2")
        d += i.count("D")
        s += i.count("S")
        p1 += i.count("P1")
        p2 += i.count("P2")
        p3 += i.count("P3")
        p4 += i.count("P4")
    return (c, b1, b2, d, s, p1, p2, p3, p4)


# function to update the dictionary using data from the list of ships
```

```
def update_dict(dict, hidden):
    c, b1, b2, d, s, p1, p2, p3, p4 = counter(hidden)
    if c == 0:
        dict["Carrier"] = ["X"]

    if b1 == 0:
        lb[1] = "X"
    if b2 == 0:
        lb[0] = "X"
    dict["Battleship"] = lb
    if d == 0:
        dict["Destroyer"] = ["X"]

    if s == 0:
        dict["Submarine"] = ["X"]

    if p1 == 0:
        lp[0] = "X"
    elif p2 == 0:
        lp[1] = "X"
    elif p3 == 0:
        lp[2] = "X"
    elif p4 == 0:
        lp[3] = "X"
    dict["Patrol Boat"] = lp


# function to terminate the game
def determine_game_end(hidden1, hidden2):
    # this function checks if either one or both lists are completely empty
    # and returns True
    if sum(counter(hidden1)) == 0 or sum(counter(hidden2)) == 0:
        if sum(counter(hidden1)) == 0:
            print("Player 2 wins!\n")
            writeto.writelines("\nPlayer 2 wins!\n\n")
            return True
        else:
            print("Player 1 wins!\n")
            writeto.writelines("\nPlayer 1 wins!\n\n")
            return True

    elif sum(counter(hidden1)) == 0 and sum(counter(hidden2)) == 0:
        print("It's a draw!\n")
        writeto.writelines("\nIt's a draw!\n\n")
        return True
```

```
    else:
        pass


# function to display grids and iteractive interface on the screen
def display(n, round, dict1, dict2, coordinates, writeto):
    try:
        lines = []
        lines.append("\nPlayer" + str(n) + "'s Move\n")
        if len(str(round)) == 1:
            lines.append(
                "\nRound : "
                + str(round)
                + "                      "
                + "Grid Size: 10x10\n"
            )
        elif len(str(round)) == 2:
            lines.append(
                "\nRound : "
                + str(round)
                + "                     "
                + "Grid Size: 10x10\n"
            )
        elif len(str(round)) == 3:
            lines.append(
                "\nRound : " + str(round) + "                    " + "Grid Size:
10x10\n"
            )
        lines.append("\nPlayer1's Hidden Board       Player2's Hidden Board\n")
        lines.append("  A B C D E F G H I J          A B C D E F G H I J\n")

        print("\nPlayer" + str(n) + "'s Move")
        print("\nRound :", round, end="")
        print("                 ", "Grid Size: 10x10")
        print("\nPlayer1's Hidden Board       Player2's Hidden Board")
        print("   A B C D E F G H I J         A B C D E F G H I J")
        for q in range(1, 10):
            print(q, "", *player1_shown[q - 1], "    ", q, "", *player2_shown[q -
1])
            lines.append(
                str(q)
                + " "
                + " ".join(player1_shown[q - 1])
                + "         "
                + str(q)
```

```
                + " "
                + " ".join(player2_shown[q - 1])
                + "\n"
            )
        print("10", *player1_shown[9], "    ", "10", *player2_shown[9], "\n")
        lines.append(
            "10"
            + " ".join(player1_shown[9])
            + "        "
            + "10"
            + " ".join(player2_shown[9])
            + " \n"
        )
        lines.append("\n")
        for i in dict1.keys():
            if i == "Carrier":
                print(i, *dict1[i], "                ", i, *dict2[i])
                lines.append(
                    i
                    + "     "
                    + " ".join(dict1[i])
                    + "                "
                    + i
                    + "     "
                    + " ".join(dict2[i])
                    + "\n"
                )
            elif i == "Battleship":
                print(i, *dict1[i], "            ", i, *dict2[i])
                lines.append(
                    i
                    + "  "
                    + " ".join(dict1[i])
                    + "              "
                    + i
                    + "  "
                    + " ".join(dict2[i])
                    + "\n"
                )
            elif i == "Destroyer":
                print(i, *dict1[i], "              ", i, *dict2[i])
                lines.append(
                    i
                    + "   "
                    + " ".join(dict1[i])
```

```
                    + "                "
                    + i
                    + "   "
                    + " ".join(dict2[i])
                    + "\n"
                )
            elif i == "Submarine":
                print(i, *dict1[i], "              ", i, *dict2[i])
                lines.append(
                    i
                    + "   "
                    + " ".join(dict1[i])
                    + "               "
                    + i
                    + "   "
                    + " ".join(dict2[i])
                    + "\n"
                )
            else:
                print(i, *dict1[i], "       ", i, *dict2[i])
                lines.append(
                    i
                    + " "
                    + " ".join(dict1[i])
                    + "          "
                    + i
                    + " "
                    + " ".join(dict2[i])
                    + "\n"
                )

        r, c, cl = coordinates
        print("\nEnter your move:", str(r) + "," + str(cl), "\n")
        lines.append("\nEnter your move:" + str(r) + "," + str(cl) + "\n")
    except:
        pass
    writeto.writelines(lines)  # Write the entire list of lines to the file


print("Battle of Ships Game")
writeto.writelines("Battle of Ships Game\n")
# set round = 1
round_of_players = 1
p1 = 0
p2 = 0
```

```
# initiate the loop
while True:
    # utilize the functions for both player1 and player2 data
    player1_hidden = identify_ships(values1_l)
    player2_hidden = identify_ships(values2_l)

    # Player1:
    if read_input(list_of_coordinates1, p1):
        coordinates1 = read_input(list_of_coordinates1, p1)
    else:
        # in case an error was found in the input given by player,
        # the program will read another of the player's input
        p1 += 1
        coordinates1 = read_input(list_of_coordinates1, p1)
    display(1, round_of_players, dict1, dict2, coordinates1, writeto)
    update_grid(coordinates1, player2_hidden, player2_shown)
    update_dict(dict1, player1_hidden)
    # increment by 1
    p1 += 1
    # check if the game terminates
    if determine_game_end(player1_hidden, player2_hidden):
        update_dict(dict1, player1_hidden)
        break
    else:
        pass

    # repeat the same for Player 2
    if read_input(list_of_coordinates2, p2):
        coordinates2 = read_input(list_of_coordinates2, p2)
    else:
        p2 += 1
        coordinates2 = read_input(list_of_coordinates2, p2)
    display(2, round_of_players, dict1, dict2, coordinates2, writeto)
    update_grid(coordinates2, player1_hidden, player1_shown)
    update_dict(dict2, player2_hidden)
    p2 += 1
    if determine_game_end(player1_hidden, player2_hidden):
        update_dict(dict2, player2_hidden)
        break
    else:
        pass

    # increment round by 1
    round_of_players += 1
```

```
# update the dictionaries once the game terminates before printing the final
information
update_dict(dict1, player1_hidden)
update_dict(dict2, player2_hidden)

# when the game terminates and either of players win
print("Final Information\n")
writeto.writelines("Final Information\n\n")
print("Player1's Hidden Board       Player2's Hidden Board\n")
writeto.writelines("Player1's Hidden Board       Player2's Hidden Board\n")
print("   A B C D E F G H I J         A B C D E F G H I J")
writeto.writelines("  A B C D E F G H I J         A B C D E F G H I J\n")
lines = []
for q in range(1, 10):
    print(q, "", *player1_shown[q - 1], "    ", q, "", *player2_shown[q - 1])
    lines.append(
        str(q)
        + " "
        + " ".join(player1_shown[q - 1])
        + "        "
        + str(q)
        + " "
        + " ".join(player2_shown[q - 1])
        + "\n"
    )
print("10", *player1_shown[9], "    ", "10", *player2_shown[9], "\n")
lines.append(
    "10"
    + " ".join(player1_shown[9])
    + "        "
    + "10"
    + " ".join(player2_shown[9])
    + " \n"
)
lines.append("\n")
for i in dict1.keys():
    if i == "Carrier":
        print(i, *dict1[i], "                 ", i, *dict2[i])
        lines.append(
            i
            + "     "
            + " ".join(dict1[i])
            + "                "
            + i
            + "     "
```

```
            + " ".join(dict2[i])
            + "\n"
        )
    elif i == "Battleship":
        print(i, *dict1[i], "            ", i, *dict2[i])
        lines.append(
            i
            + "  "
            + " ".join(dict1[i])
            + "              "
            + i
            + "  "
            + " ".join(dict2[i])
            + "\n"
        )
    elif i == "Destroyer":
        print(i, *dict1[i], "               ", i, *dict2[i])
        lines.append(
            i
            + "   "
            + " ".join(dict1[i])
            + "                "
            + i
            + "   "
            + " ".join(dict2[i])
            + "\n"
        )
    elif i == "Submarine":
        print(i, *dict1[i], "               ", i, *dict2[i])
        lines.append(
            i
            + "   "
            + " ".join(dict1[i])
            + "                "
            + i
            + "   "
            + " ".join(dict2[i])
            + "\n"
        )
    else:
        print(i, *dict1[i], "       ", i, *dict2[i])
        lines.append(
            i
            + " "
            + " ".join(dict1[i])
```

```
            + "           "
            + i
            + " "
            + " ".join(dict2[i])
            + "\n"
        )

writeto.writelines(lines)
```

User’s catalog

Battleship (also known as Battleships or Sea Battle) is a strategy-type guessing game for two players. It is played on ruled grids on which each player’s fleet of warships is marked. The locations of the fleets are concealed from the other player.

Players alternate turns calling “shots” at the other player’s ships, and the game’s objective is to destroy the opposing player’s fleet. The game is played in four grids of squares 10x10, two for each player. The individual squares in the grids are identified by letter and number.

On one grid, the player arranges ships and records the shots by the opponent. On the other grid, the player records their shots. Before play begins, each player secretly arranges the ships on their hidden grid. Each ship occupies several consecutive squares on the grid, arranged either horizontally or vertically.

The type of ship determines the number of squares for each ship. The types and numbers of ships allowed are the same for each player. The ships should be hidden from the players’ sight, and it is not allowed to see each other’s pieces. The game is a discovery game in which players must discover their opponents’ positions.

Battleship User's instructions

- Battleship game requires 2 players.
- To initiate the game, both players must specify their ship placements in a text file having 10 rows and 10 columns each containing either ";" indicating an empty cell, or a letter from the letters: "C", "B", "D", "S", "P", that correspond with the respective ships: "Carrier", "Battleship", "Destroyer", "Submarine", "Patrol Boat".
- The first round of the game starts with Player 1 where the player is required to input his coordinates in the format: number, letter.
  The number must be in the range of 1 to 10, including 1 and 10. The letter must be in the range A to J, including A and J.
  The next round will be followed by the second player's move.
- The players will be able to see the status of their ships as well as the status of their opponent's ships in the lines printed below the grids. "X" indicates that a ship has been sunk. "-" indicates that a ship is still floating.
- The players will be able to see the effect of their hits on the grids. "X" indicates that a player has successfully bombed a cell of a ship on his opponent's grid. "O" indicates that his hit had missed.
- The game will terminate once either or both of the players' ships have been completely sunk.
- In case either of the players' ships had been sunk, the opponent is to be announced the winner. However, in the case where both players' ships have been sunk, the game is announced to be a draw.