Hacettepe University

Department Of Computer Engineering

BBM 103 Assignment 2 Report

Mert Can Köseoğlu – 2220356055

23.11.2022

CONTENTS

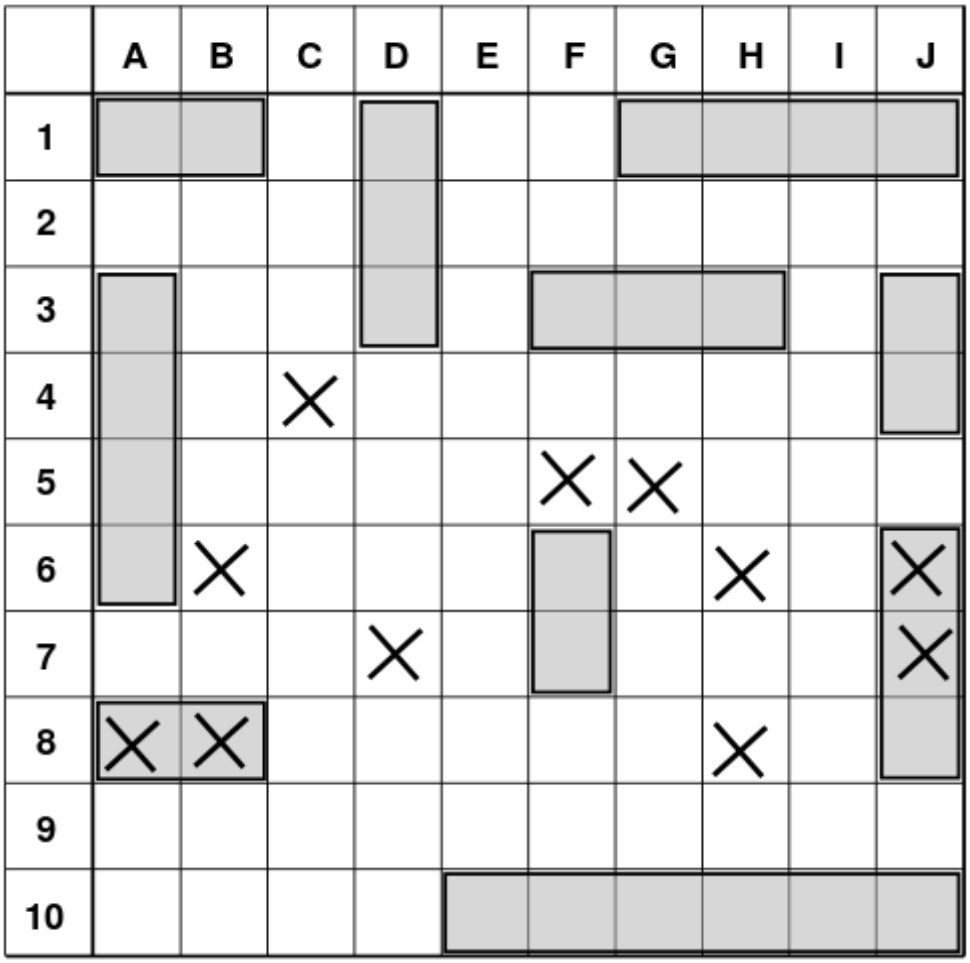
ANALYSİS

Battleship (also known as **Battleships** or **Sea Battle**) is a strategy type guessing game for two players. It is played on ruled grids (paper or board) on which each player's fleet of warships are 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 objective of the game is to destroy the opposing player's fleet.

The game is played on four grids, two for each player. The grids are typically square – usually 10×10 – and the individual squares in the grid 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 own shots. Before play begins, each player secretly arranges their ships on their primary grid. Each ship occupies a number of consecutive squares on the grid, arranged either horizontally or vertically. The number of squares for each ship is determined by the type of ship. The ships cannot overlap (only one ship can occupy any given square in the grid). The types and numbers of ships allowed are the same for each player. These may vary depending on the rules. The ships should be hidden from players sight and it's not allowed to see each other's pieces. The game is a discovery game which players need to discover their opponents ship positions.

Ships for the Battleship game

Numbers.	Class of ship	Size
1	Carrier	5 CCCCC
2	Battleship	4 BBBB
1	Destroyer	3 DDD
1	Submarine	3 SSS
4	Patrol Boat	2 PP



Sample hidden 10x10 grid with ships and other players attack.

If other players bombs part of a ship, that place in hidden board will be “X”. Else it will be “-“.

If all part of ship for every ship is bombed by other player, other player wins the game. (If player 1 hits all the ships, player 2 has one chance to attack. If player 2 hits all the ships just after player 1 then game is draw.)

Design

This code reads in two input files containing information about the initial placement of ships on a game board, and two additional input files containing a series of moves made by two players. The solution then processes this input and simulates a game of battleship between the two players, outputting the results to the console and a file called "Battleship.out". It also includes error handling to handle the case where the required input files are not reachable

Overall, the solution appears to be well-structured and easy to follow. It includes appropriate comments to explain the purpose of various sections of the code, and makes use of appropriate data structures to store and manipulate the input data. It also includes appropriate error handling to handle potential issues with the input files.

Data Structures

Here is a list of the data structures used in this code, along with a brief description of their purpose:

1. **board_p1**: This is a 2D list that represents player 1's game board. It is initialized with 10 rows and 10 columns of "-" characters, which represent empty spaces on the board.
2. **board_p2**: This is a 2D list that represents player 2's game board. It is also initialized with 10 rows and 10 columns of "-" characters.
3. **player1_hidden_board**: This is a 2D list that represents player 1's hidden board, which contains the placements of the player's ships. It is read in from the "Player1.txt" input file and processed by splitting each line on the ";" character.
4. **player2_hidden_board**: This is a 2D list that represents player 2's hidden board. It is read in from the "Player2.txt" input file and processed in a similar way to player1_hidden_board.
5. **player1_move**: This is a list of lists of strings that represents the moves made by player 1 in the game. It is read in from the "Player1.in" input file and processed by splitting on the ";" and "," characters.
6. **player2_move**: This is a list of lists of strings that represents the moves made by player 2 in the game. It is read in from the "Player2.in" input file and processed in a similar way to player1_move.

Pseudocode

1. Start the program
2. Open the output file "Battleship.out" in append mode
3. Print "Battle of Ships Game" and add a newline character to both the output file and the console
4. Set the variable "letters" equal to a string with spaces and the capital letters A through J
5. Create a dictionary called "column_letters" with keys equal to the capital letters A through J and values equal to the integers 0 through 9
6. Create a 2D list called "board_p1" with 10 sublists, each containing 10 elements all set to "-". This will represent Player 1's board
7. Create a 2D list called "board_p2" with 10 sublists, each containing 10 elements all set to "-". This will represent Player 2's board
8. Try to open the following files: "Player1.txt", "Player2.txt", "Player1.in", "Player2.in"
9. If any of these files cannot be found, raise an IOError with a message indicating which file(s) could not be found
10. If all of the files are found, create an empty list called "player1_hidden_board" and open "Player1.txt" in read mode
11. Read each line of "Player1.txt" and split it by the ";" character. Add each line to "player1_hidden_board" as a separate list
12. Close "Player1.txt"
13. Iterate through each line and item in "player1_hidden_board" and if the item is an empty string, set it to "-"
14. Create an empty list called "player2_hidden_board" and open "Player2.txt" in read mode
15. Read each line of "Player2.txt" and split it by the ";" character. Add each line to "player2_hidden_board" as a separate list
16. Close "Player2.txt"
17. Iterate through each line and item in "player2_hidden_board" and if the item is an empty string, set it to "-"
18. Open "Player1.in" in read mode and read the entire file into a string
19. Split the string by the ";" character and store the resulting list in "player1_move". Remove the last element of "player1_move"
20. Open "Player2.in" in read mode and read the entire file into a string
21. Split the string by the ";" character and store the resulting list in "player2_move". Remove the last element of "player2_move"
22. Close "Player1.in" and "Player2.in"
23. Define a function called "players_round" that takes in a parameter "rounds"
24. Inside the function, print "Round : {} \t\t\t\t\tGrid Size: 10x10 \n".format(rounds) to both the console and the output file
25. Print "Player1’s Hidden Board \tPlayer2’s Hidden Board" to both the console and the output file
26. Print "letters" to both the console and the output file
27. Define a function called "print_grid" that takes in parameters "grid_p1" and "grid_p2"
28. Create a for loop to iterate over "grid_p1" and "grid_p2"

Programmer’s Catalogue

This code is playing a game of battleship, a popular game where players place ships on a grid and take turns guessing the location of the other player's ships in an effort to sink them.

The code first opens two optional input files for players and an output file for writing results. It then initializes two grids for the two players, **board_p1** and **board_p2**, with each element initialized to **"-"**.

Next, the code reads in four input files for players 1 and 2: a **.txt** file for each player's hidden board and a **.in** file for each player's moves. It processes the input from these files, storing them in **player1_hidden_board**, **player2_hidden_board**, **player1_move**, and **player2_move** respectively.

The code then defines a function **players_round** that takes an argument **rounds** and prints out the round number and grid size. It also defines a function **print_grid** that takes two grids as arguments and prints them out with row and column labels.

Finally, the code enters a loop where the two players take turns making moves until one of them wins. The code checks the validity of each move and updates the grids accordingly. It also keeps track of the number of rounds played and ends the game when one player has no ships left.
This code appears to be trying to simulate a game of battleship. It reads in two player's hidden boards from **Player1.txt** and **Player2.txt**, as well as the moves for each player from **Player1.in** and **Player2.in**. It then simulates rounds of the game, printing out each player's hidden board and marking their opponent's board with either an **X** or an **O** depending on whether the move was a hit or a miss. Finally, it prints out the winner or if the game is a draw.

Functions

1.

```
def players_round(rounds):
    print("Round : {} \t\t\t\t\tGrid Size: 10x10 \n".format(rounds))
    print("Round : {} \t\t\t\t\tGrid Size: 10x10 \n".format(rounds),
file=output_file)
    print("Player1’s Hidden Board \tPlayer2’s Hidden Board")
    print("Player1’s Hidden Board \tPlayer2’s Hidden Board",
file=output_file)
    print(letters, "\t   ", letters)
    print(letters, "\t   ", letters, file=output_file)
```

The **players_round()** function is used to print out the round number, the grid size, and the two player's hidden boards. It takes in an integer argument **rounds** which represents the current round number. It then prints out this round number along with the grid size of 10x10. It also prints out the player's hidden boards with appropriate labels.

2.

```
def print_grid(grid_p1, grid_p2):
    num = 1
    for line_1, line_2 in zip(grid_p1, grid_p2):
        if num != 10:
            print(num, "", " ".join(line_1), " \t", end="")
            print(num, "", " ".join(line_2))
            print(num, "", " ".join(line_1), " \t", end="",
file=output_file)
            print(num, "", " ".join(line_2), file=output_file)
            num += 1
        else:
            print(num, " ".join(line_1), " \t", end="")
            print(num, " ".join(line_2), "\n")
            print(num, " ".join(line_1), " \t", end="", file=output_file)
            print(num, " ".join(line_2), "\n", file=output_file)
```

This function is used to print the game board for both players. It takes two 2D lists as input, **grid_p1** and **grid_p2**, which represent the boards for player 1 and player 2 respectively.

The function first initializes a variable **num** to 1, which will be used to print the row number of the board. It then iterates over the rows of the two boards, **line_1** and **line_2**, and prints them side by side. If the current row number is not 10, it prints the row number, followed by the row of the board, and a tab character. If the row number is 10, it omits the tab character. Finally, it increments the row number by 1

3.

```
def carrier(hidden_1, hidden_2):
    if "C" not in hidden_1:
        c_p1 = "X"
    else:
        c_p1 = "-"
    print("Carrier     \t{}\t\t\t\t".format(c_p1), end="")
    print("Carrier     \t{}\t\t\t\t".format(c_p1), end="", file=output_file)

    if "C" not in hidden_2:
        c_p2 = "X"
    else:
        c_p2 = "-"
    print("Carrier     \t{}".format(c_p2))
    print("Carrier     \t{}".format(c_p2), file=output_file)
```

```
def battleship(hidden_1, hidden_2):
    if "B" not in hidden 1:
        b_p1 = "X"
    else:
        b_p1 = "-"
    print("Battleship\t{} {}\t\t\t\t".format(b_p1, b_p1), end="")
    print("Battleship\t{} {}\t\t\t\t".format(b_p1, b_p1), end="",
file=output_file)

    if "B" not in hidden_2:
        b_p2 = "X"
    else:
        b_p2 = "-"
    print("Battleship\t{} {}".format(b_p2, b_p2))
    print("Battleship\t{} {}".format(b_p2, b_p2), file=output_file)


def destroyer(hidden_1, hidden_2):
    if "D" not in hidden_1:
        d_p1 = "X"
    else:
        d_p1 = "-"
    print("Destroyer \t{}\t\t\t\t".format(d_p1), end="")
    print("Destroyer \t{}\t\t\t\t".format(d_p1), end="", file=output_file)

    if "D" not in hidden_2:
        d_p2 = "X"
    else:
        d_p2 = "-"
    print("Destroyer \t{}".format(d_p2))
    print("Destroyer \t{}".format(d_p2), file=output_file)


def submarine(hidden_1, hidden_2):
    if "S" not in hidden_1:
        s_p1 = "X"
    else:
        s_p1 = "-"
    print("Submarine \t{}\t\t\t\t".format(s_p1), end="")
    print("Submarine \t{}\t\t\t\t".format(s_p1), end="", file=output_file)

    if "S" not in hidden_2:
        s_p2 = "X"
    else:
        s_p2 = "-"
    print("Submarine \t{}".format(s_p2))
    print("Submarine \t{}".format(s_p2), file=output_file)


def patrol_boat(hidden_1, hidden_2):
    if "P" not in hidden_1:
        p_p1 = "X"
    else:
        p_p1 = "-"
    print("Patrol Boat\t{} {} {} {}\t\t\t".format(p_p1, p_p1, p_p1, p_p1),
end="")
    print("Patrol Boat\t{} {} {} {}\t\t\t".format(p_p1, p_p1, p_p1, p_p1),
end="", file=output_file)
```

```
    if "P" not in hidden_2:
        p_p2 = "X"
    else:
        p_p2 = "-"
    print("Patrol Boat\t{} {} {} {}".format(p_p2, p_p2, p_p2, p_p2))
    print("Patrol Boat\t{} {} {} {}".format(p_p2, p_p2, p_p2, p_p2),
file=output_file)
```

The function **carrier** takes in two parameters, **hidden_1** and **hidden_2**, which represent the hidden boards of player 1 and player 2, respectively. It checks whether the string "C" is in **hidden_1** and **hidden_2** and assigns the value "X" or "-" to variables **c_p1** and **c_p2**, respectively, depending on whether "C" is present or not. It then prints the value of **c_p1** and **c_p2** with the string "Carrier" in a formatted string. The output is also written to the output file.

This function can be reused by other programmers if they want to check whether the string "C" is present in a given list and print the result with the string "Carrier". The function can be called with the desired list as the argument, like so: **carrier(hidden_board_1, hidden_board_2)**. It is same for other ships functions.

4.

```
def attack_to_bomb(hidden_board, board, move):
    print("Enter your move: {},{}".format(move[0], move[1]), "\n")
    print("Enter your move: {},{}".format(move[0], move[1]), "\n",
file=output_file)
    row = int(move[0]) - 1
    col = column_letters.get(move[1])
    if hidden_board[row][col] == "-":
        hidden_board[row][col] = "O"
        board[row][col] = "O"
    else:
        hidden_board[row][col] = "X"
        board[row][col] = "X"
```

This function is used to print the game board for both players. It takes two 2D lists as input, **grid_p1** and **grid_p2**, which represent the boards for player 1 and player 2 respectively.

The function first initializes a variable **num** to 1, which will be used to print the row number of the board. It then iterates over the rows of the two boards, **line_1** and **line_2**, and prints them side by side. If the current row number is not 10, it prints the row number, followed by the row of the board, and a tab character. If the row number is 10, it omits the tab character. Finally, it increments the row number by 1.

This function is called **attack_to_bomb** and it is used to attack a specific position on the board. It takes three arguments:

- **hidden_board**: a list of lists representing the board where the ships are placed and the attacks are stored.
- **board**: a list of lists representing the board where the attacks are shown to the player.
- **move**: a list containing the row and column of the position to attack.

The function starts by printing the move that was made and storing the row and column of the position to attack in variables **row** and **col**. Then, it checks if the position on the **hidden_board** is a dash (meaning there is no ship there). If it is, it changes the value on both the **hidden_board** and **board** to "O" (meaning a miss). Otherwise, it means there was a ship and the value on both boards is changed to "X" (meaning a hit).

5.

```
def print_error():
    print(error, "\n")
    print(error, "\n", file=output_file)
```

This function is used to print an error message. It takes no arguments and simply prints the error message stored in the variable **error** to the console, followed by a newline. It also writes the error message to the file **output_file**, also followed by a newline.

6.

```
def final_information():
    print("Final Information \n")
    print("Final Information \n", file=output_file)
    print("Player1's Board \t\t\tPlayer2's Board")
    print("Player1's Board \t\t\tPlayer2's Board", file=output_file)
    print(letters, "\t   ", letters)
    print(letters, "\t   ", letters, file=output_file)
```

The function **final_information()** is used to print out the final information about the players' boards at the end of the game. It prints a message indicating that it is the final information and then prints the players' boards with their corresponding labels. The labels are the column letters "A B C D E F G H I J" that are printed above each board to indicate the column number. The function does not take any arguments and does not return any values. It simply prints the final information to the console and to the output file "Battleship.out".

User Catalogue

To use this code, the user would need to follow these steps:

1. Create four text files named "Player1.txt", "Player2.txt", "Player1.in", and "Player2.in" in the same directory as the code file.
2. In "Player1.txt" and "Player2.txt", the user should enter the positions of the ships on the respective player's board in the following format: "A1;A2;A3;A4;A5" (without quotes). Each ship should be on a separate line and the positions should be separated by semicolons.
3. In "Player1.in" and "Player2.in", the user should enter the moves for the respective player in the following format: "A1,B2;C3,D4;E5,F6" (without quotes). Each move should be on a separate line and the positions should be separated by semicolons.
4. Run the code file from the command line by typing "python filename.py Player1.txt Player2.txt Player1.in Player2.in", replacing "filename.py" with the name of the code file and the four file names with the names of the text files created in steps 1-3.
5. The code will then execute the game and output the results to both the console and a file named "Battleship.out".

This code can be reused by other programmers as long as they follow the same input format for the text files. The code can also be modified by other programmers to meet their specific needs, such as changing the size of the board or the number of ships. To call the functions in this code, the user would simply need to use the function name followed by the required parameters. For example, to call the "players_round" function, the user could write "players_round(rounds)" where "rounds" is a variable representing the current round of the game.