BBM 103 Introduction to Programming Laboratory 1

Assignment 4: Battle of Ships

2210356032

İbrahim Emek

01.01.2023

Contents:

ANALYSIS:

We need to create a battle of ships game. In this game, we should take the players' input to locating, and moves. They are given in files. We should read them, and process them. Firstly, we need to read "Player1.txt", and "Player2.txt" to reach the placement of ships. After reading files, we need to save them to variables in order to use them later, but we should also check if there is any error, if placements are true, and if player entered enough argument in terminal. Secondly, we need to read "Player1.in", and "Player2.in" files to reach each player's moves. We should read moves one by one, and look if the move is valid. If it is valid, we need to look if there is a ship's part in the specified location. If there is one, we need to show there is one, if there is not, we also need to show there is not one. We need to show output in every round, and look if the game is ended. If the game is over we should show the final output, and we should finish the game.

DESİGN:

Thoughts:

In the beginning, we should check if the user entered enough argument. If not, we should print the error message. Then, we should look if we can open the player's arguments (If there is file with these names). If we can open each file, we can continue, else we should print the error message and stop the process. We finished to look if the files are true, now we should look if the information in these files are true, and process them.

First files to read, and process is "Player1.txt", and "Player2.txt". These files are placements of ships. We should read each line until a blank line, and save them to a list in order to use them later. Also, we should check if these files are correct to locating. After finishing these steps, we need four board. Two of them are players' hidden board, the other two of them are empty boards which shows current situation in each round.

Now, we should code the game. We should read moves, and process them in order. First, we need to use while loop to continue until one player's moves are ended. There is a possibility which is the move is not valid. In this case, we should pass to next move. So, we should save player's number of move. If the move is valid, we can process it, increase the number and pass to next player. If not, we can only increase the number, so it passes to the next move, but not next player. Also, we need to save the round number. We should define these variables outside of the while loop, and use them whenever they are needed in the while loop. By the way, we should define a variable to save which player's turn. In the beginning of game, the turn belongs to first player, so I defined a variable which indicates it is first player's turn.

Inside while loop, first we need to look whose turn it is. After that, we should print the current board. Then, we should read the next move of this player, and look if the move is valid. If valid, we should process it and pass to next player. Else, we should print the error message, and read the next move of this player. Inside second player move, we should check if the game is over. If the game is over, we should print final board which reveals hidden ship parts, and winner information, then break the while loop.

```
with open("Battleship.out.txt", "w", encoding="utf-8") as output_file:
    try:
        player_one_location = sys.argv[1]
        player_two_location = sys.argv[2]
        player_one_input = sys.argv[3]
        player_two_input = sys.argv[4]
    except IndexError:
        print(f"IndexError: You entered less argument than expected.")
        output_file.write(f"IndexError: You entered less argument than
expected.\n")

    else:
        unopened_files = []
        try:
            first_player_locating = open(player_one_location, "r",
encoding="utf-8")
        except:
            unopened_files.append(f"{player_one_location}")
        try:
            second_player_locating = open(player_two_location, "r",
encoding="utf-8")
        except:
            unopened_files.append(f"{player_two_location}")
        try:
            first_player_index = open(player_one_input, "r", encoding="utf-
8")
        except:
            unopened_files.append(f"{player_one_input}")
        try:
            second_player_index = open(player_two_input, "r", encoding="utf-
8")
        except:
            unopened_files.append(f"{player_two_input}")

        try:
            if len(unopened_files) == 1:
                raise IOError
        except IOError:
            print(f"IOError: input file {unopened_files[0]} is not
reachable.")
            output_file.write(f"IOError: input file {unopened_files[0]}
is not reachable.\n")

        output = ""
        try:
            if len(unopened_files) > 1:
                output = ", ".join(unopened_files)
                raise IOError
        except IOError:
            print(f"IOError: input files {output} are not reachable.")
            output_file.write(f"IOError: input files {output} are not
reachable.\n")

        if len(unopened_files) == 0:  # It means every file opened properly.
            try:
                reading_locations(first_player_locating)
                reading_locations(second_player_locating)
            except IndexError:
                print("You entered less index than expected.")
                output_file.write("You entered less index than expected.\n")
            except ValueError:
                print("You didn't enter valid input")
                output_file.write("You didn't enter valid input\n")

            else:
                first_player_locating.close()
                second_player_locating.close()
                first_players_hidden_board =
create_board(player_one_location, "Player1")
                second_players_hidden_board =
create_board(player_two_location, "Player2")
                first_empty_board = create_empty_board()
                second_empty_board = create_empty_board()
                read_moves()
```

Functions:

1- reading_locations function, delete_to_check function, save_ship_locations: reading_locations function reads the file, and saves the info into all_items variable as a nested list. Then uses nested for loop to reach each item in this nested list. Then goes to delete_to_check function with this item in the nested list. If the item is not an empty string, this means there should be a ship there. We should check the letter, and look its length. If there is a ship in a row, there should be same letter next to that letter in length of the ship's length. If there is a ship in a column there should be same letter under that letter in length of the ship's length. If the both case is valid, we should check if there is same letter at the same row after the ship’s length. If there is same letter, this means first letter belongs to vertical ship, and rest letters at the row is another same ship. If there is not, this means the ship is horizontal, and there is another ship under this letter. If the function finds a ship, it deletes the ship, and in order to use it later, it saves to a dictionary by using save_ship_locations function. Nested for loop continues until the items ended. Then, checks if there is any letter. If not, it means locating the ships is successful. Else, the locating is not valid, and raise an error.

```
def reading_locations(file_name):
    """This function looks if ship locating done successfully."""
    all_items = []
    while True:
        each_line = file_name.readline()
        if each_line == "":
            break
        each_line = each_line.strip()
        each_line_items = each_line.split(";")
        all_items.append(each_line_items)  # all_items contain player.text
information.

    is_true_index = True
    at_row = -1
    for each_list in all_items:
        at_row += 1
        at_column = -1
        for each_thing in each_list:
            at_column += 1

            if each_thing == "":
                pass
            else:
                if file_name == first_player_locating:
                    player_turn = "first_player"
                else:
                    player_turn = "second_player"
                delete_to_check(player_turn, each_thing, at_row, at_column,
all_items)

        if at_column != 9:
            is_true_index = False
    if at_row != 9:
        is_true_index = False

    if not is_true_index:
        raise IndexError
    else:  # Looks if there is anything left. If there is something left, it
means the way of ship locating is false.
```

```python
        for each_list in all_items:
            for each_thing in each_list:
                if each_thing != "":
                    raise ValueError
```

```python
def save_ship_locations(player_turn, letter, at_row, at_column):
    """This function saves the ship locations to use them later."""
    ships_length = {
        "C": 5,
        "B": 4,
        "D": 3,
        "S": 3,
        "P": 2,
    }
    x = "0"
    did_fail = True
    try:
        a = ship_locations[player_turn][letter + x]
        did_fail = False
    except:
        pass
    if did_fail:  # It means there is no specified key in the dictionary.
        ship_locations[player_turn][letter + x] = []
        ship_locations[player_turn][letter + x].append([at_row, at_column])
    else:
        while not len(ship_locations[player_turn][letter + x]) <
int(ships_length[letter]):  # If there is already a
            # ship which all of its locations if found, it passes to next
key(if P0 is full, it passes to P1)
            x = str(int(x) + 1)
            did_fail = True
            try:
                b = ship_locations[player_turn][letter + x]
                did_fail = False
            except:
                pass
            if did_fail:  # It means there is no specified key in the
dictionary.
                ship_locations[player_turn][letter + x] = []
        ship_locations[player_turn][letter + x].append([at_row, at_column])
```

2- create_board function: As I said upwards, we need 4 board. We can create hidden boards using this function according to the file. It reads each line, splits and create lists, appending each item of that list to hidden board. At the end we have a hidden board.

```python
def create_board(file_name, player_num):
    """Creates a board according to the file."""
    with open(file_name, "r", encoding="utf-8") as player_input:
        hidden_board_one = []
        hidden_board_two = []
        while True:
            each_line = player_input.readline()
            each_line = each_line.strip()
            if each_line == "":
                break
            each_item = each_line.split(";")
            if player_num == "Player1":
                hidden_board_one.append(each_item)
            elif player_num == "Player2":
                hidden_board_two.append(each_item)
        if player_num == "Player1":
            return hidden_board_one
        elif player_num == "Player2":
            return hidden_board_two
```

3- create_empty_board function: It creates an 10x10 empty board which is nested list. This function provides the rest 2 board.

```python
def create_empty_board():
    """Creates an empty 10x10 board."""
    board_name = []
    for x in range(10):
        a_list = []
        for y in range(10):
            a_list.append("-")
        board_name.append(a_list)
    return board_name
```

4- read_moves function: First, we need to save the all the moves. We can reach them in .in files. After saving these info to variables, we should use while loop until game ends, or moves end. We are going to process according to player's turn, so first, we should check whose turn it is, if first player's turn, we need to do first player's process. After looking whose turn it is, we should look if the next move this player is valid with check_if_valid function.

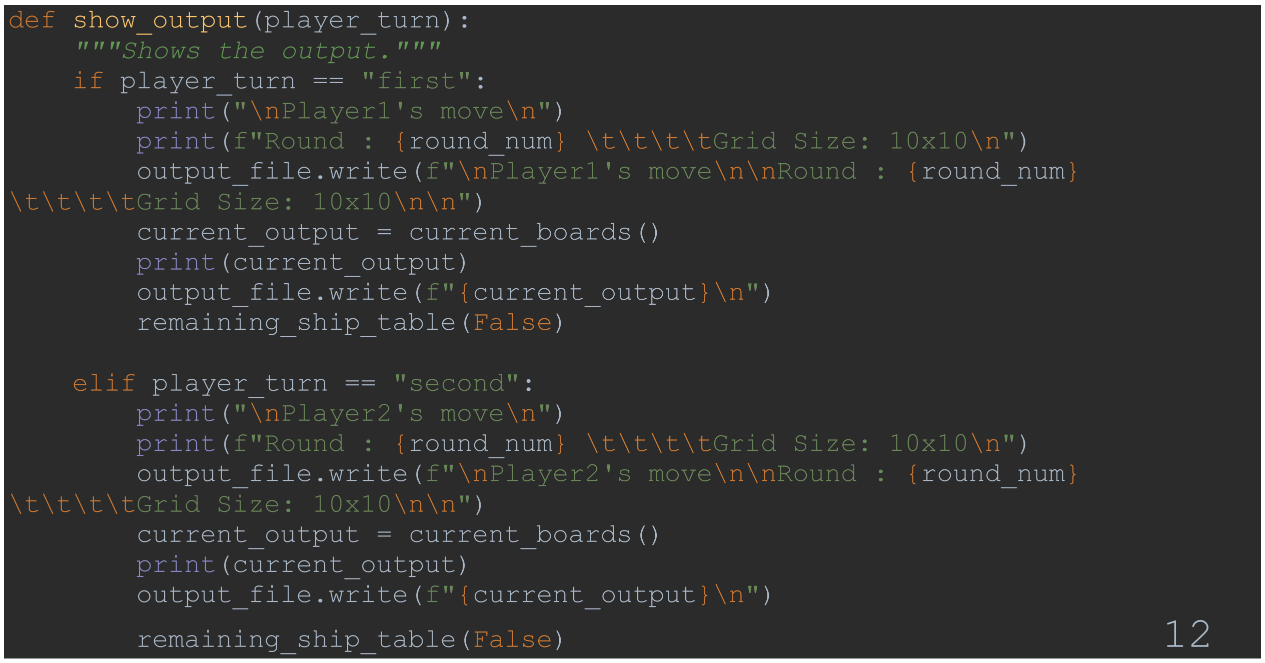
```python
def read_moves():
    """Reads every move, and process it."""
    first_file = open(player_one_input, "r", encoding="utf-8")
    second_file = open(player_two_input, "r", encoding="utf-8")
    first_move_inputs = first_file.readline()
    second_move_inputs = second_file.readline()
    first_player_moves = first_move_inputs[:-1].split(";")
    second_player_moves = second_move_inputs[:-1].split(";")

    first_at_num = 0  # indicates first player's index in first_player_moves.
    second_at_num = 0  # indicates second player's index in
second_player_moves.
    global round_num
    round_num = 0
    first_player_turn = True
    print("Battle of Ships Game")
    output_file.write("Battle of Ships Game\n")

    while first_at_num < len(first_player_moves) or second_at_num <
len(second_player_moves):
        if first_player_turn:
            if check_if_valid(first_player_moves, first_at_num):
                round_num += 1
                show_output("first")
                check_location("first", int(each_move[0]), each_move[1])
                first_at_num += 1
                ship_counts("first_player")
                first_player_turn = False
            else:  # It means the move is not valid.
                first_at_num += 1
        else:  # It is second player's turn.
            if check_if_valid(second_player_moves, second_at_num):
                show_output("second")
                check_location("second", int(each_move[0]), each_move[1])
                second_at_num += 1
                ship_counts("second_player")
                first_player_turn = True

                if len(ship_locations["first_player"]) == 0 or
len(ship_locations["second_player"]) == 0:  # It means the game is over.

                    if len(ship_locations["first_player"]) == 0 and
len(ship_locations["second_player"]) == 0:
                        print("\nIt is a Draw")
                        output_file.write("\nIt is a Draw\n")
                        show_output("final")
                        break
                    elif len(ship_locations["first_player"]) == 0:
                        convert_to_final_board("player_two_board")
                        print("\nPlayer2 wins!")
                        output_file.write("\nPlayer2 wins!\n")
                        show_output("final")
                        break
                    elif len(ship_locations["second_player"]) == 0:
                        convert_to_final_board("player_one_board")
                        print("\nPlayer1 wins!")
                        output_file.write("\nPlayer1 wins!\n")
                        show_output("final")
                        break

            else:  # It means the move is not valid.
                second_at_num += 1
```

4-A) check_if_valid function: We should check if next move is valid. If we assign a variable, and change it if everything is okay, we can understand whether everything gone correctly or not.

```python
def check_if_valid(player_moves, at_num):
    """Checks if move is valid."""
    did_success = False
    try:
        if "," not in player_moves[at_num]:
            raise IndexError
    except IndexError:
        print("IndexError: Your move is not valid because of lack of ','.")
        output_file.write("IndexError: Your move is not valid because of lack
of ','.\n")

    else:
        global each_move
        each_move = player_moves[at_num].split(",")

        try:
            a = int(each_move[0]) + 1
        except TypeError:
            print("ValueError: Your first input should be an integer.")
            output_file.write("ValueError: Your first input should be an
integer.\n")

        else:
            try:
                if each_move[1] not in alphabet:
                    raise ValueError
                else:
                    try:
                        if int(each_move[0]) > 10:
                            raise AssertionError
                        if alphabet.index(each_move[1]) > 9:
                            raise AssertionError
                        did_success = True
                    except AssertionError:
                        print(f"AssertionError: Invalid Operation")
                        output_file.write(f"AssertionError: Invalid
Operation\n")
                    else:
                        did_success = True
            except ValueError:
                print("ValueError: Your second input should be a letter.")
                output_file.write("ValueError: Your second input should be a
letter.\n")

    return did_success
```

If check_if_valid function returns “False”, it means something went wrong, and we can’t process the move, so we should skip to the next move. If we only increase the move number, it returns to while loop and continues with the next move, the turn doesn’t skip to next move. If function returns “True” it means the move is valid. Before process the move, we should print the current output with show_output function.

4-B) show_output, current_boards, remaining_ship_table functions: Firstly, show_output function looks whose turn it is, and shows the corresponding output. It prints first outputs. To show the rest outputs, it uses 2 other function. First of them is current_boards function. current_boards function prints the hidden board, and the other relative outputs. Second of them is remaining_ship_table function. We should print remained ships’ outputs. In order to print it, we should use the number of remained ships, and their name. We will change them in the latter functions. With these info, we should print the output, and other relative outputs. With these functions, showing output problem is solved.



```
def show_output(player_turn):
    """Shows the output."""
    if player_turn == "first":
        print("\nPlayer1's move\n")
        print(f"Round : {round_num} \t\t\t\tGrid Size: 10x10\n")
        output_file.write(f"\nPlayer1's move\n\nRound : {round_num}
\t\t\t\tGrid Size: 10x10\n\n")
        current_output = current_boards()
        print(current_output)
        output_file.write(f"{current_output}\n")
        remaining_ship_table(False)

    elif player_turn == "second":
        print("\nPlayer2's move\n")
        print(f"Round : {round_num} \t\t\t\tGrid Size: 10x10\n")
        output_file.write(f"\nPlayer2's move\n\nRound : {round_num}
\t\t\t\tGrid Size: 10x10\n\n")
        current_output = current_boards()
        print(current_output)
        output_file.write(f"{current_output}\n")
        remaining_ship_table(False)
    elif player_turn == "final":
        print("\nFinal information\n")
        output_file.write("\nFinal information\n")
        current_output = current_boards()
        print(current_output)
        output_file.write(f"\n{current_output}\n")
        remaining_ship_table(True)
```

```
def current_boards():
    """Shows the current boards."""
    current_output = ""
    current_output += "Player1's Hidden Board\t\tPlayer2's Hidden Board\n"
    current_output += "  "
    for x in range(10):
        current_output += " " + alphabet[x]
    current_output += "\t  "
    for x in range(10):
        current_output += " " + alphabet[x]
    current_output += "\n"
    for y in range(10):
        first_per_line = " ".join(first_empty_board[y])
        second_per_line = " ".join(second_empty_board[y])
        current_output += f"{y + 1:<2}{first_per_line}\t {y +
1:<2}{second_per_line}\n"
    return current_output
```

```
def remaining_ship_table(is_final):
    """Shows the remained ship table."""
    all_ships = {
        "Carrier": 1,
        "Battleship": 2,
        "Destroyer": 1,
        "Submarine": 1,
        "Patrol Boat": 4,
    }
    for each_ship in all_ships:
        first_board_minus = first_board_remains[each_ship[0]]
        second_board_minus = second_board_remains[each_ship[0]]
        first_board_x = all_ships[each_ship] -
first_board_remains[each_ship[0]]
        second_board_x = all_ships[each_ship] -
second_board_remains[each_ship[0]]

        if each_ship == "Carrier":
            print(f"{each_ship}\t\t{first_board_x * 'X '}{first_board_minus *
'- '}\t\t\t\t"
                  f"{each_ship}\t{second_board_x * 'X '}{second_board_minus *
'- '}")
            output_file.write(f"{each_ship}\t\t{first_board_x * 'X
'}{first_board_minus * '-' }\t\t\t\t{each_ship}\t"
                              f"{second_board_x * 'X'}{second_board_minus *
'-'}\n")
```

```
        elif each_ship == "Patrol Boat":
            print(f"{each_ship}\t{first_board_x * 'X '}{first_board_minus *
'- '}\t\t\t"
                  f"{each_ship}\t{second_board_x * 'X '}{second_board_minus *
'- '}")
            output_file.write(f"{each_ship}\t{first_board_x * 'X
'}{first_board_minus * '- '}\t\t\t"
                              f"{each_ship}\t{second_board_x * 'X
'}{second_board_minus * '- '}\n")
        else:
            print(f"{each_ship}\t{first_board_x * 'X '}{first_board_minus *
'- '}\t\t\t\t"
                  f"{each_ship}\t{second_board_x * 'X '}{second_board_minus *
'- '}")
            output_file.write(f"{each_ship}\t{first_board_x * 'X
'}{first_board_minus * '- '}\t\t\t\t"
                              f"{each_ship}\t{second_board_x * 'X
'}{second_board_minus * '- '}\n")
    if not is_final:  # we don't add this output if this output belongs to
final output.
        print(f"Enter your move: {each_move[0]},{each_move[1]}")
        output_file.write(f"\nEnter your move:
{each_move[0]},{each_move[1]}\n")
```

After showing output, we should process the move. We use the check_location function to process.

4-C) check_location function: We have four boards. Two of them are hidden boards which have info of the hidden ships, and two of them are empty boards which shows the players' moves outputs. We check if there is a ship part in the place indicated by the other player's move. If there is one or not, we change empty board info in that location.

```
def check_location(player_turn, row, column_alphabet):
    """Shows if there is a ship part in the specified location."""
    if player_turn == "first":
        if second_players_hidden_board[row -
1][alphabet.index(column_alphabet)] != "":
            second_empty_board[row - 1][alphabet.index(column_alphabet)] =
"X"
        else:
            second_empty_board[row - 1][alphabet.index(column_alphabet)] =
"O"
    elif player_turn == "second":
        if first_players_hidden_board[row -
1][alphabet.index(column_alphabet)] != "":
            first_empty_board[row - 1][alphabet.index(column_alphabet)] = "X"
        else:
            first_empty_board[row - 1][alphabet.index(column_alphabet)] = "O"
```

After process the move, we should look if a ship is sunk, and record remaining ship number. I defined ship_counts function to do these things.

4-D) ship_counts function: We used save_ship_locations to save the hidden ship parts upwards. We can use it now. We should check every ship's each location. If every location is hit, it means the ship is sunk. Then, we should decrease the number of that ship in the corresponding value in the remaining ship counts dictionary. Also, it should delete it from the dictionary which we saved the locations info in the save_ship_locations into it, which is ship_locations dictionary because if we don't delete it, it will decrease the remaining ship number again, and again.

```
def ship_counts(player_turn):
    """Looks if a ship sunk, and keeps the number of remained ship."""
    will_deleted = []
    if player_turn == "first_player":
        for each_key in ship_locations["second_player"]:  # each key 
indicates C0, B0, B1 etc.
            is_found = True
            for each_location in ship_locations["second_player"][each_key]:  
# each_location indicates each_key's location.
                if second_empty_board[each_location[0]][each_location[1]] != 
"X":
                    is_found = False

            if is_found:  # It means all parts of a ship is sunk.
                second_board_remains[each_key[0]] -= 1
                will_deleted.append(each_key)

        if len(will_deleted) != 0:  # If a ship is sunk, I am deleting it 
from the dictionary which saves all ship locations.
            for x in range(len(will_deleted)):
                ship_locations["second_player"].pop(will_deleted[x])

    elif player_turn == "second_player":
        for each_key in ship_locations["first_player"]:  # each_key indicates 
C0, B0, B1 etc.
            is_found = True
            for each_location in ship_locations["first_player"][each_key]:  # 
each_location indicates each_key's location.
```

```
                if first_empty_board[each_location[0]][each_location[1]] !=
"X":
                    is_found = False

            if is_found:  # It means all parts of a ship is sunk.
                first_board_remains[each_key[0]] -= 1
                will_deleted.append(each_key)

        if len(will_deleted) != 0:  # If a ship is sunk, I am deleting it
from the dictionary which saves all ship locations.
            ship_locations["first_player"].pop(will_deleted[0])
```

Now, we should check if the game is ended. We should check it in second player’s turn, because the round ends at the second player’s turn. We can use the remaining ship numbers. If every ship is sunk, then there is not a ship remained. If the game is ended, we should convert the boards to final board which shows if there is remained hidden parts. To do this, I defined the convert_to_final_board function.

4-E) convert_to_final_board function: If there is still a hidden part, it means not all the ships is sunk. We can use ship_locations dictionary to reach remained ship(s). If there is a ship in that dictionary, it reveals the letter in that location.

```
def convert_to_final_board(board_name):
    """This functions reveals hidden ship parts."""
    if board_name == "player_two_board":
        for each_ship_name in ship_locations["second_player"]:
            for each_location in
ship_locations["second_player"][each_ship_name]:
                if second_empty_board[each_location[0]][each_location[1]] ==
"-":
                    second_empty_board[each_location[0]][each_location[1]] =
each_ship_name[0]

    elif board_name == "player_one_board":
        for each_ship_name in ship_locations["first_player"]:
            for each_location in
ship_locations["first_player"][each_ship_name]:
                if first_empty_board[each_location[0]][each_location[1]] ==
"-":
                    first_empty_board[each_location[0]][each_location[1]] =
each_ship_name[0]
```

After converting to the final board, we should print the winner info, and final board. Finally, the game is ended.

Programmer's Catalogue:

The time I spent:

Analyzing: 1 hour

Designing: 3 hour

Implementing: 4 hour

Testing, and debugging: 6 hour

Reporting: 3 hour

I didn't do anything manually, so if you change anything in the program the program still can be used.

The function descriptions:

reading_files function: This function looks if ship locating done successfully.

delete_to_check function: This function deletes each ship until the end, and looks if there is any ship left. If there is no ship left, ship locating done successfully.

save_ship_locations function: This function saves the ship locations to use them later.

create_board function: Creates a board according to the file.

create_empty_board function: Creates an empty 10x10 board.

read_moves function: Reads every move, and process it.

check_if_valid function: Checks if move is valid.

show_output function: Shows the output.

current_boards function: Shows the current boards.

remaining_ship_table: Shows the remained ship table.

check_location function: Shows if there is a ship part in the specified location.

ship_counts function: Looks if a ship is sunk, and keeps the number of remained ship.

convert_to_final function: This function reveals hidden ship parts.

User's catalogue:

This program is battle of ships game. Firstly, you should locate your ships in the .text files, but be aware, you should locate correctly to go to next step. The ships board is like this:

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

You should locate according to that table. You should use first letter of ships to show you are locating that location, and in order to separate each location you should use ";" sign.

If you located the ships successfully, now you should enter your moves to .in files. You should be careful about your each move. Move should be like this: first, an integer to indicate the row, "," sign, a letter to indicate the column, between each move “;” sign. If the move is not valid, the game will pass to your next move. It keeps going until the game is over. Then is prints final board, and the game ends.