BBM103

Introduction to Programming Laboratory I

Assignment 4

Battle of Ships

B2210356060

Ömer Kayra Çetin

03.01.2023 23:00

Analysis

We are expected to make a battleship game. The game is a strategy game played between two players. Players take turns to shoot the opponent's ships. The game is played on four 10x10 grid of squares, two for each player. Each player places their ships on one grid and keeps the record of the shots on the other grid. Players can't see each other's ships and try to find and sunk them.

In each turn, each player says a coordinate to shoot. The other player says it's a miss or a shot. Then player's keep record of this shots until a player's all ships are sunk. If the player's that started second ships are all sunk, he has a one more shot to make for it to be more equal. The game continues until a player's all ships are sunk, in this situation the opposing player wins. If their all ships are sunk at the end of the round, the game is a draw.

Each player has the same type and number of ships. Each player has a 5 square long Carrier, two 4 square long Battleship, a 3 square long Destroyer, a 3 square long Submarine and four 2 square long Patrol Boat. Ships can be placed vertically or horizontally. Once a game is all shot, the player announces that ship has sunk.

In our game, keeping record of the shots and announcing the sunk ships, checking if there is a winner is all done by the computer. The players give their ship placements in a .txt format and their shots in a .in format. Then the computer uses them to produce an output file which shows the round's played by opponent and the winner at the end of the file. In each round, each player makes a shot and computer marks the hidden board depending on if it's a miss or a shot. The computer also keeps track of the sunk ships for each player. At the end, each player's actual board is shown with the miss or shot signs.

Design

Taking Inputs

Player1.txt and Player2.txt has player ship's coordinate information. They are split by ";". I can read them line by line for further use.

Player1.in and Player2.in has player shot's coordinates in a single line, split by ";" character. I can read them and split from ";" character to use them individually.

Making Player Boards

I can use the Player1.txt and Player2.txt and make each player a hidden and actual board. In the actual board, player's will have their ships placed. In the hidden board, player's shots will be recorded.

I make two 10x10 empty boards for each player. It is done by making each element equal to '-'. I use list comprehension to do that. They are two-dimensional python lists. One board which will be the actual one will be used to be placed ships on them. I read player text files and using a nested loop, I place them.

Keeping Track of Ships

First, I make two player a ship tracker using dictionaries. The indexes are the ships' first letters. The values are '-' for each type of ship multiplied by its number initially. The values will turn into 'X' for each ship sunk.

I make a ship flag dictionary for each player; its indexes are the ships' first letters as well. Their values are each 'floating' initially. Once a ship is sunk, its value will be changed to 'sunk'.

Then I use a function to locate ships coordinates for each player. The located coordinates are in a dictionary as values where each key is its ship related to it.

Then I use a function to check and update the sunk ships. This function checks a player's ships using the coordinates to see if they are all shot. If they are, it updates the corresponding dictionaries.

Shots

I use the previously read Player1.in and Player2.in contents to make each player their shots. In a for loop which will be the main one, player's take rounds shooting each other's boards. The shots will be in a list format and using separate indexes for each player they will shoot the next shot. By looking at their respective actual boards, their hidden board will be updated if the shot hits a ship. This will happen by making that coordinate 'X' in the hidden board. If it is a miss the same process happens but 'O' will be used.

Declare Winner

To declare the winner, I use a check loss function to take a look at each player's ship flag dictionaries to see if they are all sunk. This function is called at the end of the round for each player. If their ships are all sunk at the same time, it's a draw. If only one of theirs all ships are sunk, the other player wins. The game will continue until a player wins or a player's shots are finished.

Produce Output

Using the previously created data structures, I make functions to return strings in a formatted way.

I make a show board function to return each player's hidden board side by side.

I make a show ship counter function to return each player's ship types and '-' for floating ships and 'X' using one of the data structures created previously.

I use a output text string to collect all the necessary strings returned.

In the main loop, I add additional texts to output text string to cleanly and correctly produce the output as well as using the other show functions.

When the main loop is ended, final information will be printed and to correctly format the final information I use a make final board function. This function will make each player's final board showing shots, misses and unshot ships all together. I add this information to output text.

At the end the output text will be added to a Battleship.out file to be looked at later. Also, the contents of the Battleship.out will be printed to console as well.

Programmer's Catalogue

The Overall Time Spent for Implementing

I spent nearly two hours to completely understand the assignment. Designing, implementing, testing, and reporting happened simultaneously and repeatedly, and it required approximately 20 hours.

Functions

read_txt function

```
def read_txt(path):
    """Takes player txt file path as input and returns a list where each item is a line."""
    with open(path) as f:
        return f.read().split("\n")
```

This function takes player txt file path as input and returns a list where each item is a line.

read_in function

```
def read_in(path):
    """Takes player in file path as input and returns a list where each item is a shot coordinat."""
    with open(path) as f:
        return f.read().split(";")[:-1]  # Last item is empty so I don't include it
```

This function takes player in file path as input and returns a list where each item is a shot coordinate.

make_empty_board function

```
def make_empty_board():
    """Returns a two-dimensional 10x10 list where each item is '-'."""
    return [["-" for _ in range(10)] for _ in range(10)]
```

This function takes no arguments and returns a two-dimensional python list using list comprehension. The list's items are lists that filled with ten '-' characters.

make_ship_board function

```
def make_ship_board(ship_pos_list):
    """Takes player txt file list as input and returns a list where each ship is placed."""
    b_ship = make_empty_board()
    for i in range(len(ship_pos_list)):
        line = ship_pos_list[i]
        j = 0  # j is a counter for ; char to correctly place ships
        for char in line:
            if char == ";":
                j += 1
            else:
                b_ship[i][j] = char
    return b_ship
```

This function takes a parameter called ship_pos_list that is essentially player text file's read value. It makes an empty board and by the help of a nested loop, it places the ships onto the empty board. Each line has ';' as separators and initials of ship types. In the nested loop, these lines are checked one char at a time and by the help of a counter named j, the ships are correctly placed. It returns the board.

make_ship_counter function

```
def make_ship_counter():
    """Returns a dictionary where keys are ship types, and their values are '-' times ship numbers."""
    # This dictionary will be used to correctly show which ships are sunk.
    ship_counter = dict()
    ship_counter["C"] = "-"
    ship_counter["D"] = "-"
    ship_counter["S"] = "-"
    ship_counter["B"] = "- -"
    ship_counter["P"] = "- - - -"
    return ship_counter
```

This function takes no arguments and returns a dictionary where keys are ship types, and their values are '-' times ship numbers. This dictionary will be used to correctly show which ships are sunk.

make_ship_flags function

```
def make_ship_flags():
    """Returns a dictionary where each key is a ship name and their values are all 'floating'."""
    # This will be used to check if a ship is floating or sunk.
    ship_names = ["C", "D", "S", "B1", "B2", "P1", "P2", "P3", "P4"]
    return {name: "floating" for name in ship_names}
```

This function takes no arguments and returns a dictionary where each key is a ship name, and their values are all 'floating'. This will be used to check if a ship is floating or sunk.

find_ship_pos function

```python
def find_ship_pos(board, opt_path):
    """Takes board and optional player txt file path as inputs and returns a dict where each key is a ship
    and their values are lists containing the ship's coordinates as tuples."""
    # Make a ship_pos dictionary, it's keys are ship names and values are its positions on the board
    ship_pos = dict()
    # Generate position lists for single ships
    c_pos, d_pos, s_pos = list(), list(), list()
    # Search single ship positions to add their values to its lists
    for i in range(10):
        for j in range(10):
            if board[i][j] == "C":
                c_pos.append((i, j))
            elif board[i][j] == "D":
                d_pos.append((i, j))
            elif board[i][j] == "S":
                s_pos.append((i, j))
    # Assign the single ships to their positions inside the dictionary
    ship_pos["C"], ship_pos["D"],  ship_pos["S"] = c_pos, d_pos, s_pos
```

```python
    # Assign the single ships to their positions inside the dictionary
    ship_pos["C"], ship_pos["D"],  ship_pos["S"] = c_pos, d_pos, s_pos
    # Use the optinional txt files to correctly place ship categories that have more than one ship
    with open(opt_path) as f:
        lines = f.read().split("\n")  # Seperate lines
        for line in lines:
            name, content = line.split(":")
            pos, direction = content.split(";")[:-1]  # Don't include last item as it is empty
            # Get the x coordinate - 1 to correctly use in board lists which start from 0 as indexes
            x_cor = int(pos.split(",")[0]) - 1
            # Get the y coordinate as an integer to correctly use in board lists which start from 0 as indexes
            # Ord function gets the decimal ascii value of a given char
            # Since it is an upper char subtracting 65 would give the correct y coordinate
            y_cor = ord(pos.split(",")[1]) - 65
            if name[0] == "B":  # Check if it's a battleship or patrol boat
                # Check the direction to correctly get the coordinates
                if direction == "right":
                    ship_pos[name] = [(x_cor, y_cor), (x_cor, y_cor+1), (x_cor, y_cor+2), (x_cor, y_cor+3)]
                else:
                    ship_pos[name] = [(x_cor, y_cor), (x_cor+1, y_cor), (x_cor+2, y_cor), (x_cor+3, y_cor)]
            else:
                if direction == "right":
                    ship_pos[name] = [(x_cor, y_cor), (x_cor, y_cor+1)]
                else:
                    ship_pos[name] = [(x_cor, y_cor), (x_cor+1, y_cor)]
    return ship_pos
```

This function takes board and optional player txt file path as inputs and returns a dictionary where each key is a ship, and their values are lists containing the ship's coordinates as tuples.

The way the function works is explained in the comments.

show_ship_counter function

```
def show_ship_counter(s1_counter, s2_counter):
    """Takes p1's and p2's ship counter dictionaries as inputs and returns a txt string consisting of
    their ship counter infos."""
    txt = ""
    txt += f"Carrier\t\t{s1_counter['C']}\t\t\t\tCarrier\t\t{s2_counter['C']}\n"
    txt += f"Battleship\t{s1_counter['B']}\t\t\t\tBattleship\t{s2_counter['B']}\n"
    txt += f"Destroyer\t{s1_counter['D']}\t\t\t\tDestroyer\t{s2_counter['D']}\n"
    txt += f"Submarine\t{s1_counter['S']}\t\t\t\tSubmarine\t{s2_counter['S']}\n"
    txt += f"Patrol Boat\t{s1_counter['P']}\t\t\tPatrol Boat\t{s2_counter['P']}\n\n"
    return txt
```

This function takes p1's and p2's ship counter dictionaries as inputs and returns a txt string consisting of their ship counter information. This return txt will be added to a general output txt string later.

show_boards function

```
def show_boards(b1, b2):
    """Takes p1's and p2's boards as inputs and returns a txt string consisting of their board infos."""
    txt = "Player1's Hidden Board\t\tPlayer2's Hidden Board\n"
    txt += "   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):
        txt += "%-2d" % (i+1)
        for j in range(10):
            if j != 9:
                txt += "%-2s" % b1[i][j]
            else:
                txt += "%s" % b1[i][j]
        txt += "\t\t"
        txt += "%-2d" % (i+1)
        for j in range(10):
            if j != 9:
                txt += "%-2s" % b2[i][j]
            else:
                txt += "%s" % b2[i][j]
        txt += "\n"
    txt += "\n"
    return txt
```

This function takes p1's and p2's boards as inputs and returns a txt string consisting of their board information. By looping two of the lists at the same time and correctly formatting them, the returned txt string will be constructed.

check_update_sunk function

```
def check_update_sunk(board, ship_count, ship_pos, ship_flag):
    """Takes a player's board, ship counter, ship positions and ship flags as input and
    checks for sunk ships and updates them. Returns none."""
    # Keys in ship_pos dictionary is each ship individually
    for key in ship_pos:
        all_shot = True
        for pos in ship_pos[key]:
            x_cor, y_cor = pos
            if board[x_cor][y_cor] != "X":
                all_shot = False
                break
        if all_shot and ship_flag[key] == "floating":
            ship_flag[key] = "sunk"
            # For updating ship count I use the first character of the key to correctly match the key for ship count
            ship_count[key[0]] = ship_count[key[0]].replace("-", "X", 1)
```

This function takes a player's board, ship counter, ship positions and ship flags as input and checks for sunk ships and updates them. Returns none. It checks if a ship is all shot, and it has not already been shot. If that's the case, it updates the ship as sunk and places a 'X' to ship counter. Any complexity is explained in the comments.

check_loss function

```
def check_loss(flags):
    """Takes a player's ship flags and returns True if they are all sunk, otherwise False."""
    for value in flags.values():
        if value == "floating":
            return False
    return True
```

This function takes a player's ship flags and returns True if they are all sunk, otherwise False.

make_final_board function

```
def make_final_board(board, hidden_board):
    """Takes a player's actual board and hidden board as inputs and returns a board where each unshot ship
    is shown."""
    for i in range(10):
        for j in range(10):
            if board[i][j] != "-" and hidden_board[i][j] == "-":
                hidden_board[i][j] = board[i][j]
    return hidden_board
```

This function takes a player's actual board and hidden board as inputs and returns a board where each unshot ship is shown. It compares two lists and if there is a ship in a coordinate which hasn't been shot, it shows that ship there.

try_paths function

```
def try_paths(path_list):
    """Takes a list of paths as input and tries to open them. Returns none.
    Raises IOError if it can't open at least a file with a message that has unopenable file name(s)."""
    error_names_list = []
    for path in path_list:
        try:
            with open(path) as f:
                pass
        except IOError as err:
            error_names_list.append(os.path.basename(err.filename))
    if len(error_names_list) != 0:
        msg = ", ".join(error_names_list)
        raise IOError(msg)
```

This function takes a list of paths as input and tries to open them. Returns none. Raises IOError if it can't open at least a file with a message that has unopenable file name(s).

shot_ship function

```
def shot_ship(all_shots, shot_index, actual_board, hidden_board):
    """Takes a player's all shots, shot index and opponent's actual board, hidden board as inputs.
    Makes the shot and updates the opponent's hidden board accordingly. Returns None."""
    global output_txt
    shot = all_shots[shot_index]
    output_txt += f"Enter your move: {shot}\n\n"
    # Check for missing arguments
    if len(shot.split(",")) < 2 or "" in shot.split(","):
        raise IndexError(f"IndexError: Missing argument. You entered '{shot}' ,"
                         f"the correct format would be such as '3,A'.\n\n")
    # Check for more than 2 arguments
    if len(shot.split(",")) > 2:
        raise ValueError(f"ValueError: Too many arguments. You entered '{shot}' ,"
                         f"the correct format would be such as '3,A'.\n\n")
    # Check the arguments' validation
    if not shot.split(",")[0].isnumeric() or not shot.split(",")[1].isalpha():
        raise ValueError(f"ValueError: Bad arguments. You entered '{shot}' ,"
                         f"the correct format would be such as '3,A'.\n\n")

    pshot_x = int(shot.split(",")[0]) - 1  # Minus 1 to correctly use in board lists
    pshot_y = ord(shot.split(",")[1]) - 65  # Minus 65 to correctly use in board lists
```

```
    # Check if the shots are valid
    assert pshot_x in valid_cor
    assert pshot_y in valid_cor

    # Change the board accordingly if it's a miss or shot
    if actual_board[pshot_x][pshot_y] != "-":
        hidden_board[pshot_x][pshot_y] = "X"
    else:
        hidden_board[pshot_x][pshot_y] = "O"
```

This function takes a player’s all shots, shot index and opponent's actual board, hidden board as inputs. Makes the shot and updates the opponent's hidden board accordingly. Returns None. It adds necessary strings to the output txt string which is the main output string. It also checks for the validity of the shots and according to the situation an error is raised.

Preparation for the Main Loop

```
# The output string which will be written into the actual output file
output_txt = ""

# Valid coordinates
valid_cor = [i for i in range(10)]
```

From now on, everything will be in a try block to handle any kind of exception.

```
try:
    # Get the necessary file paths
    cwd = os.getcwd()
    player1txt_path = os.path.join(cwd, sys.argv[1])
    player2txt_path = os.path.join(cwd, sys.argv[2])
    player1in_path = os.path.join(cwd, sys.argv[3])
    player2in_path = os.path.join(cwd, sys.argv[4])
    opt_p1_path = os.path.join(cwd, "OptionalPlayer1.txt")
    opt_p2_path = os.path.join(cwd, "OptionalPlayer2.txt")
    # Test if files are openable using try_paths function
    p_list = [player1txt_path, player2txt_path, player1in_path, player2in_path]
    try_paths(p_list)

    # Get ship location lists
    p1_ships = read_txt(player1txt_path)
    p2_ships = read_txt(player2txt_path)

    # Get shot lists
    p1_shots = read_in(player1in_path)
    p2_shots = read_in(player2in_path)
```

Get the file paths of the arguments and test if they are openable using try_paths function. If they are all openable continue to get necessary data structures for the main loop.

```
    # Use ship location lists to generate p1's and p2's actual board
    p1_board = make_ship_board(p1_ships)
    p2_board = make_ship_board(p2_ships)

    # Generate p1's and p2's hidden boards
    p1_hidden_board = make_empty_board()
    p2_hidden_board = make_empty_board()

    # Get ship counters
    p1_ship_count = make_ship_counter()
    p2_ship_count = make_ship_counter()

    # Get p1's and p2's ship positions
    p1_ship_pos = find_ship_pos(p1_board, opt_p1_path)
    p2_ship_pos = find_ship_pos(p2_board, opt_p2_path)

    # Get p1's and p2's ship flags
    p1_ship_flags = make_ship_flags()
    p2_ship_flags = make_ship_flags()

    output_txt += "Battle of Ships Game\n\n"
```

Continue to get the necessary data structures and add the title to the output_txt string.

```
    p1scount = 0  # Player1's shot count
    p2scount = 0  # Player2's shot count
    round_c = 1  # Round counter
```

Declare counters for the main loop.

Main Loop

```
# Continue until one of the player's shots finish
while p1scount < len(p1_shots) and p2scount < len(p2_shots):
    # Player 1's shot
    output_txt += "Player1's Move\n\n"
    output_txt += f"Round : {round_c}\t\t\t\t\tGrid Size: 10x10\n\n"
    output_txt += show_boards(p1_hidden_board, p2_hidden_board)
    output_txt += show_ship_counter(p1_ship_count, p2_ship_count)
```

If a player's shots finish before there is a winner, loop stops. Start the loop by player 1's shot. First add the output_txt necessary strings.

```
    # Try to make a shot, if it is not valid, add the error message to the output txt and
    # try to make the next shot of the same player
    while True:
        try:
            shot_ship(p1_shots, p1scount, p2_board, p2_hidden_board)
        except IndexError as e:
            output_txt += str(e)
            if p1scount < len(p1_shots):
                p1scount += 1
            else:
                break
        except ValueError as e:
            output_txt += str(e)
            if p1scount < len(p1_shots):
                p1scount += 1
            else:
                break
        except AssertionError:
            output_txt += "AssertionError: Invalid Operation.\n\n"
            if p1scount < len(p1_shots):
                p1scount += 1
            else:
                break
        else:
            break

    check_update_sunk(p2_hidden_board, p2_ship_count, p2_ship_pos, p2_ship_flags)
```

This code segment is explained in the comment. This basically checks the validity of the shot. It takes another shot from the player until it finds a valid one or the player's shots finished. At the end, it calls the check_update_sunk function. Check the functions segment to see what it does.

```
    # Player 2's shot
    output_txt += "Player2's Move\n\n"
    output_txt += f"Round : {round_c}\t\t\t\t\tGrid Size: 10x10\n\n"
    output_txt += show_boards(p1_hidden_board, p2_hidden_board)
    output_txt += show_ship_counter(p1_ship_count, p2_ship_count)
```

```
    while True:
        try:
            shot_ship(p2_shots, p2scount, p1_board, p1_hidden_board)
        except IndexError as e:
            output_txt += str(e)
            if p2scount < len(p2_shots):
                p2scount += 1
            else:
                break
        except ValueError as e:
            output_txt += str(e)
            if p2scount < len(p2_shots):
                p2scount += 1
            else:
                break
        except AssertionError:
            output_txt += "AssertionError: Invalid Operation.\n\n"
            if p2scount < len(p2_shots):
                p2scount += 1
            else:
                break
        else:
            break

    check_update_sunk(p1_hidden_board, p1_ship_count, p1_ship_pos, p1_ship_flags)
```

Basically, do the same steps for the second player.

```
    round_c += 1
    p1scount += 1
    p2scount += 1

    # Check if the game has ended

    if check_loss(p1_ship_flags) and check_loss(p2_ship_flags):
        output_txt += "It is a Draw!\n\n"
        break
    elif check_loss(p2_ship_flags):
        output_txt += "Player1 Wins!\n\n"
        break
    elif check_loss(p1_ship_flags):
        output_txt += "Player2 Wins!\n\n"
        break
```

When the round ends, increment the count variables by one. Then check if the game has ended using check loss function.

```
    # Print final information
    output_txt += "Final Information\n\n"
    p1_final_board = make_final_board(p1_board, p1_hidden_board)
    p2_final_board = make_final_board(p2_board, p2_hidden_board)
    output_txt += "Player1's Board\t\t\t\tPlayer2's Board"
    # Correctly format the boards to get only the board part not the headings
    output_txt += "\n" + show_boards(p1_final_board, p2_final_board).split("\n", 1)[1]
    output_txt += show_ship_counter(p1_ship_count, p2_ship_count)
```

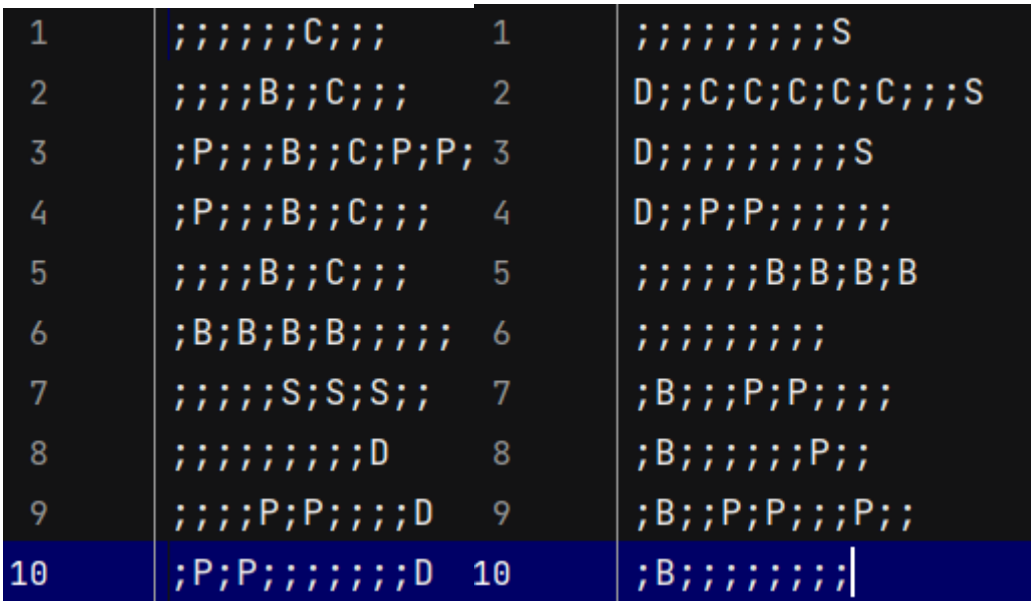
When the loop ends, print the final information.

```
except IOError as e:
    output_txt += "IOError: input file(s) " + str(e) + " is/are not reachable."
except Exception:
    output_txt += "kaBOOM: run for your life!"
finally:
    # Write the output_txt to an output file
    # Print the output_txt to command line
    print(output_txt)
    with open("Battleship.out", "w") as file:
        file.write(output_txt)
```

This catches the exceptions in the biggest try block that would cause the game to end immediately. Finally, it prints the contents of the output_txt to the command line and writes the contents of the output_txt to a file called ‘Battleship.out’.

User’s Catalogue

User Manual



1. Each player should create a .txt file containing 10 lines. In each line their ships will be present. The format should be up above, separated by ‘;’.

```
5,E;10,G;8,I;4,C;8,F;4,F;7,A;4,A;9,C;5,G;6,G;2,H;2,F;10,E;3,G;10,I;10,H;4,E;8,G;2,I;4,B;5,F;2,G;10,C;10,B;2,C;3,J;10
```

2. Each player should create .in file containing one line. On that line there is player’s shot coordinates separated by ‘;’. Any wrong formatted shot won’t be effective.
3. Then all the files will be gathered in one file with the main program.

```
python Assignment4.py "Player1.txt" "Player2.txt" "Player1.in" "Player2.in"
```

4. The program will be run by typing this in the command line.
5. In the same folder, ‘Battleship.out’ file will be created.
6. Battleship.out file can be used to look at the rounds played and see the winner.

Restrictions of the Program

Most of the cases where an error happens is handled. However, if a player exceeds their ship number, or places them horizontally there is nothing the program can do even if it is not allowed in the game. Any other unexpected error would cause the program to stop immediately without knowing why it occurred.