**Battleship Game**

**Bruce Tieu**

**SID: 107841636**

**10/21/20**

**CSCI 2312 - Design Document**

1. Problem Description  
   This program will be a battleship game. Battleship is a guessing game for two players. The objective of the game is to destroy the opposing player’s entire fleet. Players are given two grids each: one to arrange their ships and record the torpedoes fired by the opponent; and the other to let the player record their own shots and whether they hit or missed. Players will alternate turns by ‘firing torpedoes’ at the other player’s ships. Here, ‘firing a torpedo’ means allowing the player to take a guess at where on the grid the opponent may have placed a ship.
2. Input Requirements  
   A detailed **list of all external inputs** (from files or keyboard) including a description of the **data type** and **range of valid values** for each input. For input file format and interactive user input, you need to write what data type is used for every field and valid value and length.

File Input:

One external input to consider is reading in an external file called “ship\_placement.csv”. This file contains (1) the type of ship, (2) the grid square for the ship placement, and (3) whether the ship is placed vertically or horizontally (V, H) on the field. To represent this data, we can use some type of map with key and value pairs. For example,

Ship1 = {‘ship’: ”battleship”, ‘location’: “B2”, ’orientation’: “V”}

Ship2 = {‘ship’: ”cruiser”, ‘location’: “J2”, ’orientation’: “H”}

Ship3 = {‘ship’: ”carrier”, ‘location’: “G4”, ’orientation’: “V”}

Ship4 = {‘ship’: ”submarine”, ‘location’: “C10”, ’orientation’: “H”}

Ship5 = {‘ship’: “destroyer”, ‘location’: “J6”, ‘orientation’: “V”}

Then, to make these ships accessible, we can perhaps enumerate all maps into a vector:

shipVector = < {‘ship’: ”battleship”, ‘location’: “B2”, ’orientation’: “V”}, {‘ship’: ”cruiser”, ‘location’: “J2”, ’orientation’: “H”}, {‘ship’: ”carrier”, ‘location’: “G4”, ’orientation’: “V”}, {‘ship’: ”submarine”, ‘location’: “C10”, ’orientation’: “H”}, {‘ship’: “destroyer”, ‘location’: “J6”, ‘orientation’: “V”} >

To access the third ship’s location for example, one can write:

shipVector[2][‘location’]

In terms of data types, we have a map of strings, ie map<string, string>, and a string of vector mappings, ie vector<map<string, string>>. It is important to consider that there are limits to the values the map can take for each key. Namely,

ship = {“carrier”, “cruiser”, “battleship”, “submarine”, “destroyer”}

orientation = {‘V’, ‘H’}

The ‘location’ is tricky because it’s dependent on the ‘orientation’, and also the size of the ship. The main thing to avoid is the possibility of overlapping ships. Assuming the ships do not overlap, ‘location’ can take on string values between “A1” and “J10”, which represent individual squares on the 10 x 10 grid. For example, A1 -> 00, A2 -> 01, etc. Finally, the size of the ship depends on the type of ship. Carriers span 5 blocks, Battleships span 4 blocks, Cruiser and Submarines span 3 blocks, and Destroyers span 2 blocks. So, these are integer values.

Keyboard Input:

The player will have a chance to play against the computer via the keyboard. The player runs the program, and they are prompted with a message like “Enter a grid location (A1 – J10) or ‘q’ to quit: “. The player will need to enter a string in the valid range. They will be prompted to keep entering grid coordinates until it is correct. The player can also quit out of the program by typing ‘q’.

1. Output Requirements  
   A detailed **list or description of all outputs (**to files or to the screen).

Console output:

After the player takes a guess at where on the grid their opponent may have placed the ship, there will be a message informing the player if a ship has been hit or not. Then, the computer will take a random guess, which will also be displayed to the console. For example:

Enter a grid location (A1 – J10) or ‘q’ to quit: A1

That’s a miss! Computer has 5 ships that are afloat.

Computer guessing …

Computer guesses A1!

Computer missed. You have 5 ships that are afloat.

Enter a grid location (A1 – J10) or ‘q’ to quit: B1

That’s a hit! Computer has 4 ships that are afloat.

Computer guessing…

Computer guesses B1!

Computer missed. You have 5 ships that are afloat.

…

It would be ideal if 10 x 10 grids can appear next to each other on the console which shows the locations that have already been guessed by both the computer and the player, and updates the grid as ships are fired. This would greatly enhance the user experience but will be implemented as time permits.

There will also be messages that lets the player know if he/she has won the game, or if any errors occur in the input.

1. Problem Solution Discussion  
   A summary description of the solution steps with algorithms analysis (1 paragraph, approximately 100 words). If any unusual techniques or algorithms are used that need further explanation, and additional paragraph may be used.

There are at least two main algorithms to consider:

1. Determining placement of ships.
2. Keeping track of which ships have been hit and the game score.

Addressing 1 – Pseudocode:

1. Read in the ship\_placement.csv file
2. Populate a dictionary like data structure with the keys being ‘shipType’, ‘shipLocation’, ‘shipOrientation’ and the values corresponding to the data
   1. Also add another key with the ‘size’ of the ship. For example, Cruisers are 5 blocks large
3. Using the data structure in b, fill the grid with ships
   1. Parse location of ship into rows and columns of a matrix. For example, A1 -> (0, 0) or row 0, column 0 of the matrix
   2. Check the size and orientation of the ship
   3. Loop through the grid and place the ship in the correct location. For example, If the size is 5 and orientation is horizontal, we have a loop like:

for (int i = column, i < size; i++) {

GRID[row][i] = battleship.substr(1,1);

}

So if the orientation is horizontal, fix the row and index the column by 1. If the orientation is vertical, fix the column and index the row by 1. Here, battleship.substr(1,1) is the first letter of the ship that is being placed.

Checking for invalid placements – idea 1:

One way to check for overlaps / invalid placements is to compare the current placement of the ship to the placement of all previous ships. For example, suppose all previous placement locations so far are stored in a vector:

prevPlacements = {‘B2’,’B3’,’B4’,’B5’,’B5’, … }

and the location of the current ship is:

currPlacement = {‘B2’,’C2’,’D2,’E2’}

Then we know that this is an invalid placement since ‘B2’ appears in both vectors / arrays. The problem with this approach is that it is very slow. It requires the vectors / arrays to be populated, and then a double for loop is needed to check if there are elements in both containers. There could be a maximum of 100 elements in prevPlacements, and a maximum of 5 elements in currPlacement at a time. So in general, this is an O(n^2) operation.

Addressing 2:

We can consider adding a Score class which keeps track of the locations that have been fired. Using the vector of maps which was created when the ships were read in, we can compare what was fired to value of the location key. For example, if “B2” was the location, we can check if “B2” is a value under the ‘location key’ of each map. If there is a match, we can return the key value corresponding to the ship type. In the worst case this is probably an O(n) operation since we are looping through the vector of maps described in the **Input Requirements** section. The logic follows similarly for the computer.

1. (UML diagrams) A UML Class Diagram and Sequence Diagram  
   UML class diagram and Interaction (Sequence) diagram with your choice of classes and inheritance. Show the relationships among the classes and how your classes interact each other. Out of so many references available for these two diagrams, following two references have good starting points to draw the diagrams for this final project.

(Updated as of 12/3/20) UML Class Diagram**:** https://app.diagrams.net/?libs=general;uml#G1J2AZ-1A2k6-oFZq35VbexxZxEXQ1XZ0E

UML Sequence Diagram https://app.diagrams.net/#G1vT9pQopGkR1taxofLPMbZVWIWONw7gbo

1. Overall Software Architecture  
   A brief description of ***major*** functions and ***their main roles*** in the program. You need to explain how the entire program is constructed and how the functions are related each other. You don't have to explain every little function. Mention only the most important functions.

(Updated as of 12/3/20) Functions in the Grid class:

1. createGrid(): Creates the grid for the player and the grid for the computer.
2. printGrid(): Prints out the grid for the player and the grid for the computer.
3. clearGrid(): This method clears the grid.
4. getRow() : get the row from the Grid
5. getColum(): get the column from the Grid.

(Updated as of 12/3/20) Functions in the Game class:

1. buildGame(): Initiates the start of the game.
2. playGame(): plays the game
3. printResults() : Prints the result of the game.

(Updated as of 12/3/20) Ship Class:

This class is meant to hold Ship objects.

(Updated as of 12/3/20) Functions in the Human class (inherits from Player).

1. humanFires() : Fire a torpedo and see if there’s a hit
2. readShipsFromFile() : read the ships into a vector
3. placeShipsOnBoard() : place the ships on the board.
4. \_addSizeToShips(): add size to the ships
5. \_numComputerShipsHit() : count how many computer ships were hit
6. \_numComputerShipsMissed() : count how many computer ships were missed

(Updated as of 12/3/20) Functions in the Player class (base class for users)

1. shipsAreInBounds() : check if ships are in bounds
2. getPossibleShipLocations() : get the possible ship arrangements
3. shipDontOverlap() : check that the ships don’t overlap
4. allShipsAreIncluded() : check that all the ships are included
5. printShipVector() : print out the resulting ship vector.

(Updated as of 12/3/20) Functions in the Computer class (inherits Player class):

1. randShipPlacement() : randomize the ship placement
2. randShipLocation() : randomize the ship location
3. randShipOrientation() : randomize the ship orientation
4. generateCorrectShipPlacements() : generate the correct ship placements
5. placeShipsOnBoard() : place the ships on the board
6. computerFires() : computer fires a torpedo

(Updated as of 12/3/20) Functions in the Helpers class

1. parseLocationToIndex(): Parse a location to row and col integers
2. parseLocationToString(): Parse a location to row and column string representation
3. isAlphabet(): Check the input is alphabetical
4. isInBound() : check the ship location is in bounds
5. isRightLenght() : check the input is of the correct length
6. isSpace() : check that there are no spaces in the input
7. isEmpty() : check that the input is not empty
8. toUpper() : convert the choice to be all uppercase

(Updated as of 12/3/20) GridIndex class:

1. Create a struct to hold the row and column positions of the matrix.

(Updated 12/3/20) Main class:

1. Build the game instance and then play the game.
2. Test scenarios and cases

Write at least 10 test scenarios and 2 – 3 test cases for each test scenarios.

Test scenario 1: Verify that the ship placement is valid.

Test case 1: Ship1: ‘Carrier’, Location: ‘B2’, Orientation: ‘H’, Ship2: ‘Battleship’, Location: ‘D4’, Orientation: ‘V’, Ship3: ‘Cruiser’, Location: ‘G4’, Orientation: ‘H’, Ship4: ‘ Submarine’, Location: ‘G5’, Orientation: ‘V’, Ship5: ‘Destroyer’, Location: ‘G8’, Orientation: ‘H’

Test result 1: Success (No overlap of ships)

Test case 2: Ship1: ‘Carrier’, Location: ‘J1’, Orientation: ‘H’, Ship2: ‘Battleship’, Location: ‘D4’, Orientation: ‘V’, Ship3: ‘Cruiser’, Location: ‘A4’, Orientation: ‘H’, Ship4: ‘ Submarine’, Location: ‘A9’, Orientation: ‘H’, Ship5: ‘Destroyer’, Location: ‘A1’, Orientation: ‘H’

Test result 2: Failed (There’s overlapping of ships)

Test case 3: Ship is placed on H7 horizontally and it is 5 blocks large

Test result 3: Failed (7 + 5 = 12 > 10 and is out of bounds).

Test case 4: Ship is placed on F4 vertically and is 5 blocks tall

Test result 4: Success (4 + 5 = 9 <= 10 and is within the grid bounds vertically)

Test case 5: Ship is placed on H4 horizontally and it is 5 blocks large

Test result 5: Success (5 + 4 = 9 <= 10 and it is within the grid bounds horizontally)

Test case 6: Ship is placed on H10 vertically and is 5 blocks large

Test result 6: Failed (10 + 5 = 15 > 10 and it is outside of the grid bounds)

Test Scenario 2: Correctly handle all types of user input.

Test Case 1: A1

Test Result 1: Success

Test Case 2: A11:

Test result 2: Failed (the Grid is only 10 x 10)

Test Case 3: A 1

Test result 3: Failed (cannot have spaces in input)

Test Case 4: 1A

Test result 4: Failed (Grid is row by column (choose a letter from A-J then 1 – 10)

Test case 5: 00

Test result 5: Failed (Input needs to be in the format (A-J) followed by (1-10)

with no spaces or other characters)

Test case 6: ‘Q’ or ‘q’

Test result 6: Success (This indicates the player wants to quit and so the game terminates)

Test case 7: Anything other than test case 1 – test case 5

Test result 7: Failed

Test Scenario 3: Verify whether a ship has been hit or not (both for player and computer)

Test case 1: Ship placement at ‘B2’, player fires ‘B3’

Test result 1: Output to the console: “Missed!” (Failed)

Test case 2: Ship placement at ‘B2’, player fires ‘B2’

Test result 2: Output to the console: “Hit!”. Allow the user to keep firing or display another message saying that the ship has been sunk (Success)

Test scenario 4: Verify the input file only has valid ships

Test case 1: Carrier, Battleship, Cruiser, Submarine, Destroyer

Test result 1: Success

Test case 2: Carrier, Battleship, Cruiser, Submarine

Test result 2: Failed (Missing Destroyer)

Test case 3: Carrier, Carrier, Battleship, Cruiser, Submarine, Destroyer

Test result 3: Failed (Must not contain a duplicate ship)

Test case 4: Carrier, Battleship, Cruiser, Submarine, Destroyer, Yacht

Test result 4: Ship must only be Carrier, Battleship, Cruiser, Submarine, and Destroyer

Test Scenario 5: Verify that each ship has the correct size

Test case 1: Carrier: 5, Battleship: 4, Cruiser: 3, Submarine: 3, Destroyer: 2

Test result 1: Success

Test case 2: Carrier: 5, Battleship: 5, Crusier: 4, Submarine: 3, Destroyer: 2

Test result 2: Failed (Battleship is 4, Cruiser is 3)

Test Scenario 6: Verify that the orientation of the ship supplied in the input file is valid

Test case 1: H

Test result 1: Success

Test case 2: anything but H, h, V, v

Test result 2: Failed

Test Scenario 7: Verify the correct output is displayed when a ship is hit or missed

Test case 1: Computer ship is located on A1 and player fires ‘A1’

Test result 1: Success (A message is displayed indicating that the ship was hit)

Test case 2: Computer ship is located on A1 and player fires ‘B1’

Test result 2: Success (A message is displayed indicating that the ship was not hit)

Test case 3: Computer ship is located on ‘A1’ and player fires ‘B1’

Test result 3: Failed (A message is printed indicating the ship was actually hit)

Test scenario 8: Verify that the game ends and starts properly

Test case 1: No ships are sunk

Test result 1: Game continues onward (Success – this is expected to happen)

Test case 2: All opponents ships are sunk

Test result 2: Game continues onward (Failure – the game is supposed to end)

Test case 3: At least one of opponent’s ships are sunk

Test result 3: Game continues until all ships from one player are sunk (Success). Or the game ends (failure – the game should end when all ships of a player have been sunk)

Test Scenario 9: Update the grid accordingly when a ship has or hasn’t been hit

Test case 1: Opponent fires a torpedo on the square ‘B7’

Test result 1: The grid displays an ‘X’ on ‘B7’ to indicate the slot has been taken (Success)

Test Case 2: Opponent fires a torpedo on the square ‘B8’

Test result 2: The grid doesn’t display anything (Failed)

Test Scenario 10: Track if a ship or all ships have been sunk

Test case 1: Submarine ship is located on ‘A1’, ‘A2’, and player fires ‘A1’ on the first turn and ‘A2’ on the second turn

Test result 1: The computer’s ship is sunk (Success but allow the program to continue executing if this is not the last ship that has been sunk. Otherwise, end the game.)

Test case 2: Submarine ship is located on ‘A1’, ‘A2’, and player fires ‘A1’.

Test result 2: This ship is not sunk yet (Failed, but allow the program to continue executing

Test Scenario 11: Check if the file can be read

Test case 1: The file is of type txt

Test result 1: Failed (Output a message with invalid data type and stop execution)

Test case 2: The file is a csv, but with incorrect data

Test result 2: Failed (Output a message saying that invalid data is supplied, stop execution)

Test case 3: The file is a csv with correct data supplied

Test result 3: Success

Test Scenario 12: Verify that the grid which is printed out is the correct dimensions

Test case 1: The grid is less than 10 x 10

Test result 1: Failed (The grid must be exactly 10 x 10)

Test case 2: The grid is greater than 10 x 10

Test result 2: Failed (The grid must be exactly 10 x 10)

Test case 3: The grid is 10 x 10

Test result 3: Success (The grid is the correct dimension)

(Added as of 12/3/20) Status of the program:

The program was developed on Clion with g++ and C++11. It successfully runs on csegrid server. The main requirements have been fulfilled, and some of the extra credit was completed except for the computer guessing logic.