BShip Documentation

Michal Bodzianowski

4/12/18

Status: Complete **w/ Extra Credit** (with possible future additions)

Testing Status: Thoroughly tested and seemingly working on CSEGrid

Table of Contents

Problem Description

The task at hand was to create a program that could allow a user to play the classic game of *Battleship* with a computer. The game *Battleship* consists of 2 players having their own 10x10 grids on which they place 5 ships, ranging in size from 5 squares in a line to 2 squares in a line. Each player takes turns calling a coordinate on the other player's grid, in which case the other player must inform the other whether the coordinate being "torpedoed" results in a hit, a sink, or a miss. The first player then records this result on his secondary grid, as a way to keep track of which spaces they have called. The game ends when one player's ship are completely sunk- the other player wins. A tie is possible within some rulesets.

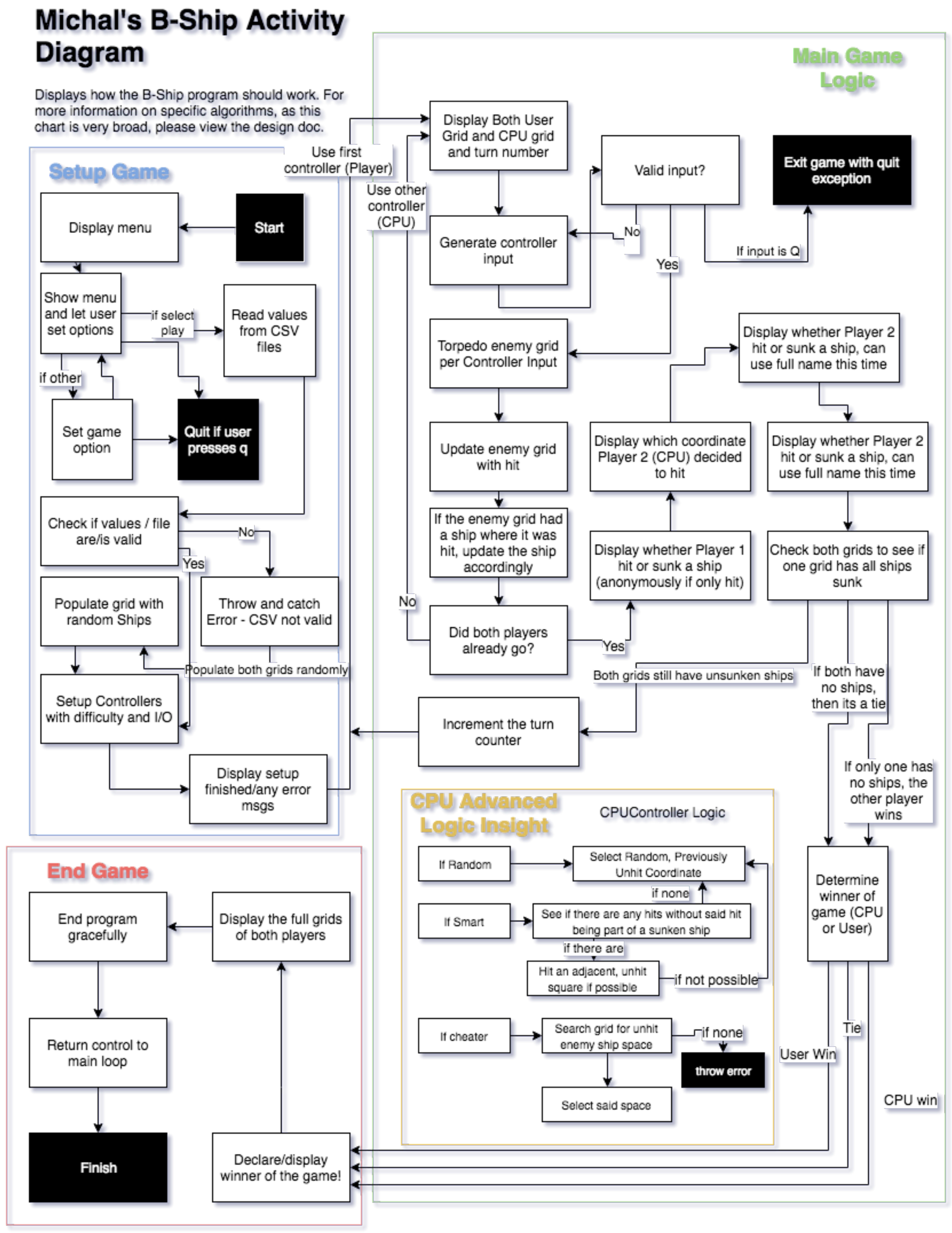
The problem called for the user to be able to also specify a .CSV file containing the values and placement of their ships, and to use a random-based algorithm to generate the ship placement for the computer. The problem also called for all other user input to be done through the terminal, and for most inputs to be checked, scrubbed, and include error throws and catching as necessary, to demonstrate knowledge of the exception system. Optional parts of the problem include advanced hit detection, and advanced CPU logic beyond randomly torpedoing as the program calls for it. Additionally, ships must have names, and the grid and information must be presented in a user friendly format. The program architecture must include use of inheritance, and must be in a multi-file format. The program, must of course, be thoroughly tested to ensure it properly works as expected.

Overall Software Architecture

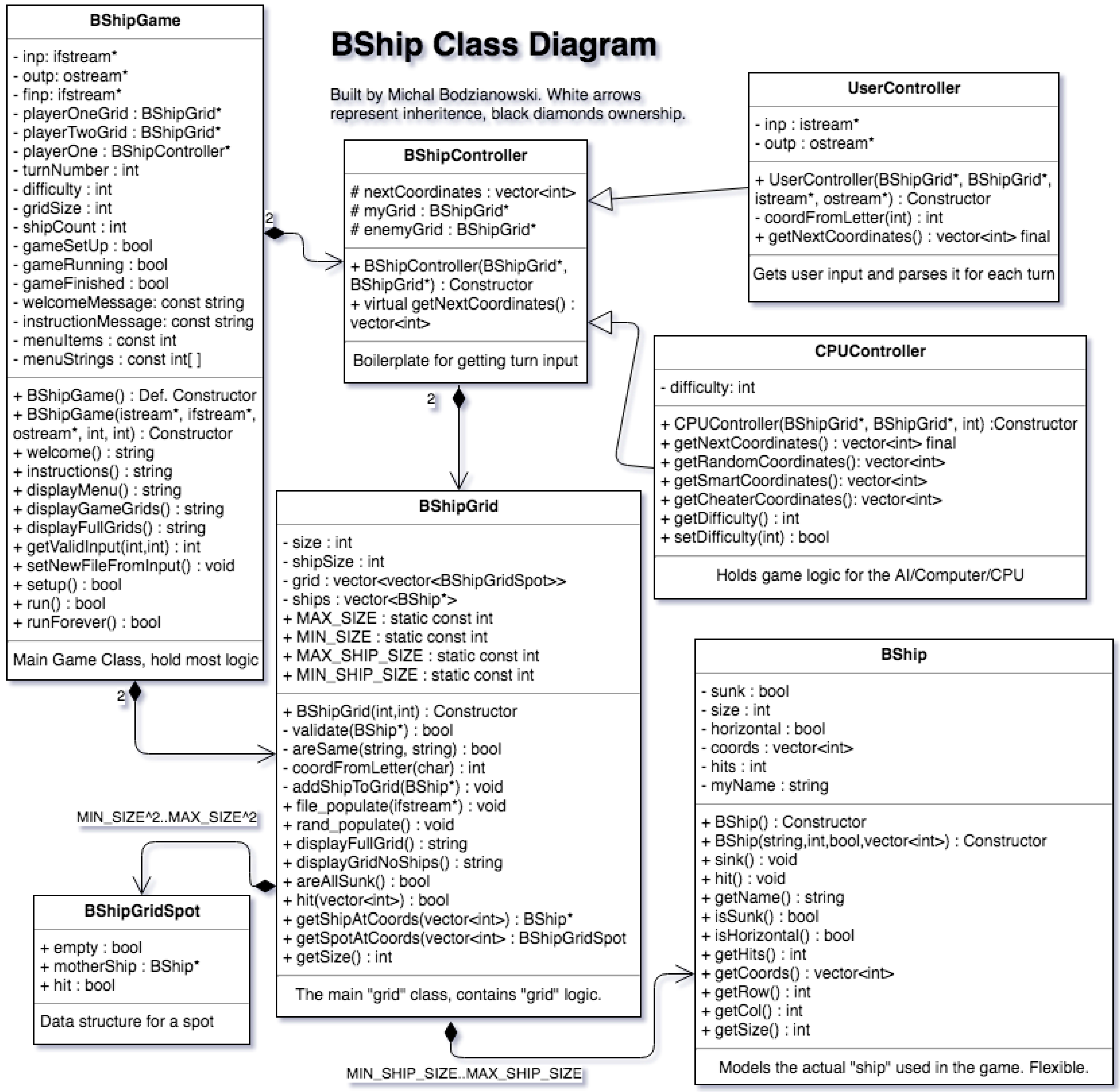
The program was designed in order to provide extremely modular use in any situation. It is plug and play at its current state, and with a few simple modifications (and more tedious testing), could be adapted for use in almost any situation. The program is also designed to be self-contained for this reason, and therefore its logic is stored in an object oriented design. The program has three core classes, which store most of the logic- BShipGame, BShipGrid, and BShipController. The Game class stores most of the high level game logic, and passes out information and responsibilities to other classes for the most part. The Grid class takes the burden of handling most of the game's data, and also models the game's board. The Controller class is a template class, which is then modeled by the UserController class and CPUController class. These classes provide an interface for the game to be played. The remaining two classes, BShipGridSpot and BShip, are auxiliary classes used primarily by BShipGrid.

The program has three main phases when launched. First, setup is called. This first routine allows a period of where the game can modified to the user's liking. A friendly and easy to use menu is displayed, and the user then proceeds to choose an option. Depending on what the user chooses, either a game setting is modified, the program proceeds to the next phase, or the program quits. The next phase is the game phase. During this phase, the game runs and can no longer be modified. The two player game proceeds with each player guessing a coordinate, and then the grid updating to reflect the hit coordinate, complete with logic for handling hit ships. The game continues until one player sinks (or hits all coordinates containing enemy ships) all of the others ships. The third phase then cleans up the game, displays results, and gracefully ends the game.

UML Diagrams (Activity)


Michal's B-Ship Activity Diagram
Displays how the B-Ship program should work. For more information on specific algorithms, as this chart is very broad, please view the design doc.
Setup Game
Start
Display menu
Show menu and let user set options
if select play
Read values from CSV files
if other
Set game option
Quit if user presses q
Check if values / file are/is valid
No
Yes
Throw and catch Error - CSV not valid
Populate grid with random Ships
Populate both grids randomly
Setup Controllers with difficulty and I/O
Display setup finished/any error msgs
Main Game Logic
Use first controller (Player)
Use other controller (CPU)
Display Both User Grid and CPU grid and turn number
Generate controller input
Valid input?
No
Yes
If input is Q
Exit game with quit exception
Torpedo enemy grid per Controller Input
Update enemy grid with hit
If the enemy grid had a ship where it was hit, update the ship accordingly
Did both players already go?
No
Yes
Display whether Player 1 hit or sunk a ship (anonymously if only hit)
Display which coordinate Player 2 (CPU) decided to hit
Display whether Player 2 hit or sunk a ship, can use full name this time
Display whether Player 2 hit or sunk a ship, can use full name this time
Check both grids to see if one grid has all ships sunk
Both grids still have unsunken ships
If both have no ships, then its a tie
If only one has no ships, the other player wins
Increment the turn counter
Determine winner of game (CPU or User)
Tie
User Win
CPU win
CPU Advanced Logic Insight
CPUController Logic
If Random
Select Random, Previously Unhit Coordinate
if none
If Smart
See if there are any hits without said hit being part of a sunken ship
if there are
Hit an adjacent, unhit square if possible
if not possible
If cheater
Search grid for unhit enemy ship space
if none
throw error
Select said space
End Game
End program gracefully
Display the full grids of both players
Return control to main loop
Finish
Declare/display winner of the game!

UML Diagrams (Class)



BShip Class Diagram

Built by Michal Bodzianowski. White arrows represent inheritence, black diamonds ownership.

BShipGame

- inp: ifstream*
- outp: ostream*
- finp: ifstream*
- playerOneGrid : BShipGrid*
- playerTwoGrid : BShipGrid*
- playerOne : BShipController*
- turnNumber : int
- difficulty : int
- gridSize : int
- shipCount : int
- gameSetUp : bool
- gameRunning : bool
- gameFinished : bool
- welcomeMessage: const string
- instructionMessage: const string
- menuItems : const int
- menuStrings : const int[ ]

+ BShipGame() : Def. Constructor
+ BShipGame(istream*, ifstream*, ostream*, int, int) : Constructor
+ welcome() : string
+ instructions() : string
+ displayMenu() : string
+ displayGameGrids() : string
+ displayFullGrids() : string
+ getValidInput(int,int) : int
+ setNewFileFromInput() : void
+ setup() : bool
+ run() : bool
+ runForever() : bool

Main Game Class, hold most logic

BShipController

nextCoordinates : vector<int>
myGrid : BShipGrid*
enemyGrid : BShipGrid*

+ BShipController(BShipGrid*, BShipGrid*) : Constructor
+ virtual getNextCoordinates() : vector<int>

Boilerplate for getting turn input

UserController

- inp : istream*
- outp : ostream*

+ UserController(BShipGrid*, BShipGrid*, istream*, ostream*) : Constructor
- coordFromLetter(int) : int
+ getNextCoordinates() : vector<int> final

Gets user input and parses it for each turn

CPUController

- difficulty: int

+ CPUController(BShipGrid*, BShipGrid*, int) :Constructor
+ getNextCoordinates() : vector<int> final
+ getRandomCoordinates(): vector<int>
+ getSmartCoordinates(): vector<int>
+ getCheaterCoordinates(): vector<int>
+ getDifficulty() : int
+ setDifficulty(int) : bool

Holds game logic for the AI/Computer/CPU

BShipGrid

- size : int
- shipSize : int
- grid : vector<vector<BShipGridSpot>>
- ships : vector<BShip*>
+ MAX_SIZE : static const int
+ MIN_SIZE : static const int
+ MAX_SHIP_SIZE : static const int
+ MIN_SHIP_SIZE : static const int

+ BShipGrid(int,int) : Constructor
- validate(BShip*) : bool
- areSame(string, string) : bool
- coordFromLetter(char) : int
- addShipToGrid(BShip*) : void
+ file_populate(ifstream*) : void
+ rand_populate() : void
+ displayFullGrid() : string
+ displayGridNoShips() : string
+ areAllSunk() : bool
+ hit(vector<int>) : bool
+ getShipAtCoords(vector<int>) : BShip*
+ getSpotAtCoords(vector<int> : BShipGridSpot
+ getSize() : int

The main "grid" class, contains "grid" logic.

BShipGridSpot

+ empty : bool
+ motherShip : BShip*
+ hit : bool

Data structure for a spot

BShip

- sunk : bool
- size : int
- horizontal : bool
- coords : vector<int>
- hits : int
- myName : string

+ BShip() : Constructor
+ BShip(string,int,bool,vector<int>) : Constructor
+ sink() : void
+ hit() : void
+ getName() : string
+ isSunk() : bool
+ isHorizontal() : bool
+ getHits() : int
+ getCoords() : vector<int>
+ getRow() : int
+ getCol() : int
+ getSize() : int

Models the actual "ship" used in the game. Flexible.

Input Requirements

User Inputs	
Main Menu	A single integer number 1- 8 inclusive must be entered. All other inputs will be denied.
Options	A single integer number, the range is determined and communicated by the option. Range limitations for certain options exist for program security and stability.
Game	A single character followed by a single integer number. Range for both is determined by grid size. This must be a valid coordinate pair
File Inputs	
CSV File	Must be formatted properly, see below for an example. Must have 1 more lines than the ship count. Has a header line, and body lines follow the format of (Ship Name),(Coordinate Pair),(**H**orizontal or **V**ertical)
Options (CSV Filepath)	String, which contains a valid filepath to a .csv file.

CSV Example (file starts next line)

```
TypeOfShip,Location,HorizOrVer
Carrier,E5,V
Battleship,G3,V
Destroyer,F2,H
Tug Boat,H3,V
Submarine,A1,V
```

CSV Explanation

TypeOfShip,Location,HorizOrVer -- This is the required static header line.
Carrier,E5,V -- Notice there are NO spaces
Battleship,G3,V --All ships are checked for validity on a grid
Destroyer,F2,H -- Name of ship, followed by coordinate, followed by H or V
Tug Boat,H3,V --The Tug Boat is a small easter egg...try it out!
Submarine,A1,V --H or V specifies whether it is horizontal or vertical

How the above ships get placed on a 10 x 10 grid:

x	A	B	C	D	E	F	G	H	I	J
1	Su									
2	Su					De	De			
3	Su						Ba	Tu		
4							Ba			
5					Ca		Ba			
6					Ca		Ba			
7					Ca					
8					Ca					
9					Ca					
10										

Output Requirements

Name	**Description**	**Example**

Menu
Or Startup
	Displays a menu (first line only on startup).
	
	Menu names are meant to be self explanatory.
	
	<u>Simulate!</u> pits 2 CPUs against each other.
		`Welcome to Michal's BShip Game!`
		`---Menu (Enter # to Select)---`
		`1: View Instructions`
		`2: Set Difficulty (BONUS!)`
		`3: Set New CSV File`
		`4: Set Grid Size (BONUS!)`
		`5: Set Ship Count (BONUS!)`
		`6: Play!`
		`7: Become A Robot / Simulate! (BONUS!)`
		`8: Quit`

Instructions	Displays the Instructions	`---Instructions---`
		`-Enter Q at any time to quit the game.`
		`-0 = Miss, X = Hit.`
		`-When it's your turn, enter coordinates in the format '(Letter)(Number)', i.e. 'A1'`
		`-Game ends when all of one or the others ships are sunk!`
		`------------------`

Difficulties	Displays the Difficulties	`-----Difficulties-----`
		` 0 - Default (Normal)`
		` 1 - Easy / 2 - Normal`
		` 3 - Hard / 4 - Semi-Pro`
		` 5 - Pro / 6 - World Champ`
		`   7 - c h e a t e r`
		`----------------------`

Instruction/Set	Gives user instruction.	`Enter a valid grid size (5-25):`

Game Display	Displays game information.
	
	First displays the turn number, with an arrow pointing out to easily be able to locate turns post game in a standard terminal width.
		`--------------------------`
		`|         Turn 8         | - - - - - - - - - - - - - - > 8`
		`--------------------------`
		
		`-CPU's Grid:-`
		`X | A | B | C | D | E | F | G | H | I | J |`
		`____________________________________________`
		`1 | 0   ~   ~   ~   ~   ~   ~   ~   ~   ~`
		`2 | X   ~   ~   ~   ~   X   ~   ~   ~   ~`
		`3 | X   ~   ~   ~   X   X   ~   ~   ~   ~`
		`4 | ~   ~   ~   ~   ~   ~   ~   ~   ~   ~`
		`5 | ~   ~   ~   ~   ~   ~   ~   ~   0   ~`
		`6 | ~   ~   ~   ~   ~   ~   ~   ~   ~   ~`
		`7 | ~   ~   ~   ~   ~   ~   ~   ~   ~   ~`
		`8 | ~   ~   ~   ~   ~   ~   ~   ~   ~   ~`
		`9 | ~   ~   ~   ~   ~   ~   ~   ~   ~   ~`

	Displays the CPU grid with hidden ships. Displays your hits and misses. Displays your entire grid, and CPU's hits and misses. Displays turn information and notifications. Asks for user input.	10| ~ 0 ~ ~ ~ ~ ~ ~ ~ ~ ——————————————————————— -Your Grid:- X | A | B | C | D | E | F | G | H | I | J | ——————————————————————— 1 | 0 ~ ~ ~ ~ ~ ~ ~ ~ ~ 2 | X 0 ~ ~ ~ X ~ ~ ~ ~ 3 | X ~ ~ ~ X X ~ ~ ~ ~ 4 | Ca ~ ~ ~ Su ~ ~ ~ ~ ~ 5 | Ca ~ ~ ~ Su ~ ~ ~ ~ ~ 6 | Ca ~ ~ ~ ~ Ba Ba Ba Ba ~ 7 | ~ Cr ~ ~ ~ ~ ~ ~ ~ ~ 8 | ~ Cr ~ ~ ~ ~ ~ ~ ~ ~ 9 | ~ Cr ~ ~ ~ 0 ~ ~ ~ ~ 10| ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ——————————————————————— You sunk a Destroyer! The CPU guessed F2. The CPU sunk your Destroyer! Please enter a coordinate pair to hit (A-J)(1-10):
Post Game Display	Displays Winner Displays the full grids of both players, so you can see what you missed on the CPU grid. Displays Good Game! message before returning.	CPU Wins! -- Full Grids -- -CPU's Grid:- X | A | B | C | D | E | F | G | H | I | J | ——————————————————————— 1 | 0 0 0 0 ~ ~ 0 0 X 0 2 | 0 0 0 ~ ~ 0 0 0 X 0 3 | ~ ~ ~ ~ ~ ~ 0 0 X 0 4 | ~ 0 ~ ~ 0 ~ 0 0 X 0 5 | ~ 0 0 0 X 0 X 0 ~ ~ 6 | X X X 0 X 0 X ~ 0 ~ 7 | ~ 0 ~ 0 X 0 0 ~ ~ ~ 8 | ~ ~ ~ ~ X 0 ~ ~ ~ ~ 9 | 0 ~ ~ 0 X 0 0 ~ 0 0 10| 0 ~ 0 ~ 0 ~ Su Su Su ~ ——————————————————————— -Your Grid:- X | A | B | C | D | E | F | G | H | I | J | ——————————————————————— 1 | ~ ~ ~ ~ ~ 0 ~ 0 X ~ 2 | ~ ~ ~ ~ ~ ~ 0 ~ X ~ 3 | 0 0 0 0 0 ~ ~ 0 X 0 4 | ~ 0 ~ ~ 0 ~ ~ ~ X ~ 5 | 0 0 0 0 X 0 X ~ 0 ~ 6 | X X X 0 X 0 X ~ 0 ~ 7 | ~ 0 ~ 0 X 0 0 ~ 0 ~ 8 | ~ ~ ~ 0 X ~ 0 0 0 ~ 9 | 0 ~ 0 0 X 0 0 0 0 0 10| 0 ~ ~ ~ ~ 0 X X X 0 ——————————————————————— Good Game!
Exception	Displays an exception	Exception - File Format Error - Not enough values in file. Consider changing Ship Count

Menu Item Walkthrough

Just a few extra details on what each menu option does.

1. **View Instructions**

Displays the instruction string and returns back to menu.

2. **Set Difficulty**

Allows the user to set the difficulty of the CPU, from Easy to Cheater. The higher the integer value entered, the higher the difficulty. The Difficulty menu is also displayed for ease of reference. The difficulty is validated.

3. **Set New CSV File**

Allows the enter to enter a new file path from which the CSV file will be loaded. If invalid, the user is informed and must try again. File path can be relative to the current directory the program is in. Returns to menu after.

4. **Set Grid Size**

Allows the user to set a grid size (n where the grid is nxn) from 5-25. A larger grid size generally means a longer game. Returns upon valid entry.

5. **Set Ship Count**

Allows the user to determine the amount of ships. Along with grid size, the amount of allowed ships is dynamically calculated, and this information is related when the user enters a number that is too high for the grid size.

6. **Play!**

Starts the game normally and exits the menu.

7. **Become A Robot / Simulate!**

First replaces Player 1 (the UserController, or user) with a CPUController, but otherwise starts the game normally and exits the menu. The game plays automatically.

8. **Quit**

Quits the game, and returns control back to main loop.

Problem Solution Discussion

The problem was solved using a modular object oriented approach. Despite the fact that designing a modular solution is generally more inefficient and takes more work, it provides significantly more versatility than simply writing to the solution. Because of this, the program can easily be modified beyond its original boundaries and work with very reasonable accuracy. This is demonstrated in the final program.

The object oriented approach was designed in mind with a hand off principle. The classes were created from a top-down approach, where the main class (BShipGame) started off creating the basic game logic with very vague functions to uncreated classes, such as areAllShipsSunk, and then those functions were implemented more specifically in a subclass (BShipGrid), which then still used vague functions (BShip.isSunk), until finally the bottom classes were able to provide enough specificity without complexity.

This results in a very readable program at all levels. The only exception is validate(BShip) in BShipGrid, which was written outside of this structure, and merely adapted for use from another program, since collision is pretty complicated. Otherwise, the readability allows for very smooth debugging, as the logic is made better traceable and it is easier to identify and locate problems.

A big part of the program that is unique to this solution is the ability to use, and even create different “controllers”. The BShipController class is a template for other classes to use to be able to interface and play the game. The program currently contains UserController, which takes user input (or even file input) and CPUController (see UML for more details). But in theory, other controllers could be created such as AIController, and used. Currently in the program there is an option for two CPUControllers to play the game without any additional logic. This is only possible through the use of inheritance.

Classes/Inheritance/Data Structures

For more class/inheritance information, please view the UML Class Diagram on page 6

- **BShipGame** is the main class which owns every other class.
- **BShipGrid** contains the grid and owns the "class" BShipGridSpot and BShip
- **BShipController** is a template (parent) class which can obtain coordinates to hit on a grid
    - **UserController** is a child of BShipController and gets coordinates via user or possibly file input
    - **CPUController** is a child of BShipController and gets coordinates via computer randomness or logic
- **BShipGridSpot** is a class that really should be a struct, but that turned out to cause some problems with memory so it's a class. It is a data structure that contains data for a spot on the grid, including which ship, if any, is on it.
- **BShip** is a class which models the actual ship which is put on the grid and has logic for getting hit and automatically flagging itself as sunk.

Vectors were chosen over arrays in nearly every case, as vectors are more flexible and dynamic than arrays which is very important in a modular solution. Arrays were mainly used in static or non-dynamic cases (such as the menu or static definitions) since vectors were not necessary. These arrays would most likely still be replaced in a fully dynamic solution, unlike a mostly dynamic solution like the program is currently in.

Custom data structures such as **BShipGridSpot** were used to increase readability over a single int with different values. It's easier to parse BShipGridSpot.motherShip than check which int matches with what BShip, or check if it's -1, or 0, or something convoluted like that.

Final Statements / Future Outlook

The program contains many additional features over the problem statement. The main 4 include difficulty settings, dynamic grid size, dynamic ship count, and simulation mode. I'm not sure if I've already mentioned this, but since ship count is dynamic, there is no limit to what type of ship you can specify. So yes, you could write in 5 Carriers into the CSV and it would work. That is the user's choice. Considering the CPU has a cheat mode perhaps this isn't such a bad idea tactically. Specifying a different file path while the program is running WAS supposed to be a bonus feature, but apparently that turned out to be required via a question on Piazza. These features are a way to hopefully make up a bit for the lateness of this assignment, and I hope you can appreciate them.

Also, as a glimpse into other possible features that weren't fully fleshed out, you can specify a "Tug Boat" in the CSV file as a valid ship name. This special 1 length ship is your glass cannon- it's hard to find, but very easy to sink. Put it somewhere you don't think the CPU will check, and victory...is probably certain? The CPU will never randomly generate a Tug Boat, so there is no need to worry about that regard. Maybe if this project wasn't already so late and overdue I'd add a file that allows the user to specify special ships (perhaps even some 2D ones...?), and along with that a config file that preloads settings which is already within grasp.

This is, I think, the biggest takeaway, is that once I get started, it is easy for me to get drawn into the challenge of the assignment, and because of that I tend to get ideas about how I can do more with it.

Thanks for playing BShip,
-Michal

Additionally, the late night comments in the code are worth a read over.