RIVERSIDE CITY COLLEGE

RIVERSIDE COMMUNITY COLLEGE DISTRICT

PROJECT 2

FALL 2025

“A GOOD NAVY IS NOT A PROVOCATION OF WAR. IT IS THE SUREST GUARANTEE OF PEACE.”

A RECREATION OF THE BOARD GAME

BATTLESHIP

JOHN HUYNH

INSTRUCTOR: DR. MARK LEHR

CSC_CIS_17A

SECTION 47466

KEY FEATURES

Normal and Short Matches

Fleet Placement System

Game History

Statistics

Table of Contents

1. Introduction

For Project 2, I wanted to take the basic Battleship game I made in Project 1 and turn it into a full-fledged application using C++ classes. My main goal was to move away from the simple coding style I used before and rebuild the game using Object-Oriented Programming. This approach made it much easier to add complex features like a smarter computer opponent and a visual ship placement system.

In this version, you are not just guessing coordinates blindly. I built a visual interface that lets you actually see your ships on the board, move them around with the keyboard, and rotate them before placing them down. I also upgraded the computer player. Instead of firing randomly the whole time, it uses a custom strategy: once it hits one of your ships, it remembers that spot and starts targeting the surrounding area to finish you off.

To make all of this work, I used inheritance to create different types of players (Human and Computer) that share the same base code but behave differently. I also used templates to handle the game boards, which saved me from writing duplicate code for the ship grid and the shot grid. The result is a game that feels smoother to play and runs on a much cleaner, more organized codebase.

2. Game Play and Rules

When you run the program, you start at the **Main Menu**. You can choose to Play Battleship, view your lifetime Statistics, check the History of past games, or Quit.

Game Modes

If you choose to play, you get two options:

1. **Standard Mode:** This is the classic experience. You play on a big 10x10 grid with a full fleet of 5 ships (Carrier, Battleship, Cruiser, Submarine, Destroyer).
2. **Rapid Mode:** This is for quick testing. The board is shrunk down to 5x5, and you only have 3 ships (Cruiser, Submarine, Destroyer).

Ship Placement

Before the shooting starts, you have to place your ships. I built a visual system for this so you don't have to guess coordinates.

- **Move:** Use **W/A/S/D** to move your ship around the board.
- **Rotate:** Press **C** to flip the ship between horizontal and vertical.
- **Place:** Press **P** to lock the ship in.
- **Quit:** Press **Q** if you want to cancel setup.

The game won't let you cheat. You can't place ships on top of each other or off the edge of the board.

Taking Turns

Once the game starts, you and the computer take turns firing.

- **Your Turn:** The game prompts you to type in a coordinate like "B3". If you hit a ship, you get a "HIT!" message. If you miss, it marks it on your tracking board.
- **Computer's Turn:** The computer fires back. Unlike the random firing in my old version, this computer is smarter. It hunts randomly at first, but if it hits your ship, it starts firing at the spots right next to the hit to try and sink you.

Winning the Game

The game continues until one side loses all their ships.

- **Victory:** If you sink the entire enemy fleet, the game announces you win and saves your victory to the stats file.
- **Defeat:** If the computer sinks all your ships first, you lose.

History & Stats

After the game, you can check the **Game History** to see a replay of the boards from previous matches, or check **Statistics** to see your total wins, win percentage and accuracy, which are saved permanently between times you run the program.

3. Development Summary

This project consists of 8 separate C++ files that handle the game logic, players, ships, and grids. The final program contains:

Lines of Code	747
Comment Lines	346*
Empty	210
Total Lines	**1338**

* Formatting my comments using Doxygen doubled my comment count. Files like *Game.h* have more comments than lines of code as comments are used to explain each function call.

The program uses:

- 2 persistent binary files (history.dat and stats.dat) to maintain game records and player statistics between sessions.
- 12 Custom Classes to organize the game logic:
- Core Logic: Game, Player, HumanPlayer, ComputerPlayer.
- Data Structures: Grid (Template), Point.
- Game Objects: Ship (Abstract) and 5 derived ship types (Carrier, Battleship, etc.).
- 2 Structs (Stats, GameRecord) specifically for organizing data structures before writing them to binary files.
- Templates in the Grid class, allowing the same code to manage integer-based ship boards and character-based firing boards.
- Vectors to manage fleet sizes dynamically instead of using fixed arrays.
- Polymorphism to handle different player types (Human vs. AI) using a single code interface.

3.1 Development Overview

First Version

This game was inspired by a personal Battleship program written in Javascript back in 2019 that had sound and graphics, but was a single player game where only the player was firing. This was a very simple project and will be available near the end of the document.

Rather than Javascript, this program uses C++ programming concepts introduced throughout the semester. The code grew more complex as ship placement, turn management, shot processing, history recording, and statistics were implemented and tested.

One of the first challenges of the project was finding a practical way for the user to place their ships in real time. Early versions of the program printed a new board every time the cursor moved, causing dozens of boards to stack on top of each other in the console. The solution was to add enough blank lines to give the appearance of clearing the screen before redrawing the updated state. This allowed the cursor to move cleanly and made ship placement feel responsive.

Ensuring that ship placement stayed within the bounds of the board was another important part of development. The program needed to prevent the player from placing ships that extended off the grid, and it also needed to block any overlapping placements. Similar checks were required during gameplay so that the player couldn't fire at invalid or out-of-range coordinates. Writing these validation functions helped create a smoother experience and prevented several potential crashes.

During development, it became clear that testing the full standard game mode was time-consuming. Reaching an end-game state required manually playing through an entire match, locating the computer's fleet piece by piece, and repeatedly firing until every ship was sunk. This made it difficult to quickly generate sample data for both the game history and the statistics file during early testing.

The first workaround involved printing out the computer's board after placing the fleet as a method of knowing exactly where the ships were. Testing was faster, but it also highlighted a broader issue: the standard mode itself was simply too long for debugging. This led to the idea of adding a second game mode designed specifically for shorter sessions.

After the core gameplay was developed, the next major challenge was implementing binary file history and persistent statistics.

- history.bin stores every completed game as a GameRecord, including board states, winner, timestamps, and shot grids.
- stats.dat maintains the player's cumulative statistics across program sessions.

Memory management is also implemented to ensure the shot grids operate on dynamically allocated structures. These are allocated at runtime and deleted at the end of gameplay to avoid memory leaks.

Second Version

Moving into the second phase of development, the goal was to take this functional procedural code and completely rebuild it using an Object-Oriented architecture. I had to take my code apart and split it into separate Header (.h) and C++ (.cpp) files to keep things organized.

Here are the main architectural changes I made:

1. I got rid of the global variables. Now, the Game class handles the settings privately, and the Player class keeps its own board data safe so nothing else can mess it up.

2. Inheritance: I made a main Player class that acts as a parent. Then, I made HumanPlayer and ComputerPlayer inherit from it. This is really useful because the game can treat both of them the same way when asking for a move, even though the human types in a coordinate and the computer picks one randomly.

3. Templates: I made a custom Grid class using Templates. This lets me use the exact same grid logic for the ship numbers (integers) and the hit/miss markers (characters) without rewriting code.

4. Memory: I used dynamic memory to set up the grids. I also added a "Copy Constructor" to the Grid class to make sure that if I copy a board, it copies the data correctly and doesn't cause memory leaks.

3.2 Time Spent

The project required about **145** hours in total of development, debugging of, and preparing documentation.

80 hours were mostly spent ensuring that the project covered most of the concepts learned so far this semester.

- Structures
- Enumerations
- Pointer
- 1D and 2D Arrays
- Binary File I/O
- Functions
- Pass-by-Reference
- Menu Navigation
- Input Validation
- Dynamic memory management

The refactoring process and adding the new features took about **65** hours of development, testing, and documentation. These hours were mostly spent ensuring that the project covered our new concepts:

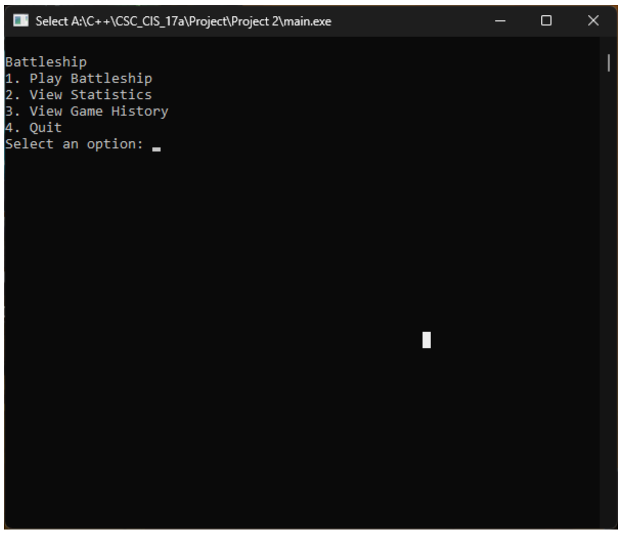
- Classes
- Objects
- Inheritance
- Polymorphism
- Templates
- Operator Overloading
- Exception Handling

4. Specifications

4.1 Sample Input/Output

4.1.1 Main Menu

When you first run the program, it shows the title and the main options.

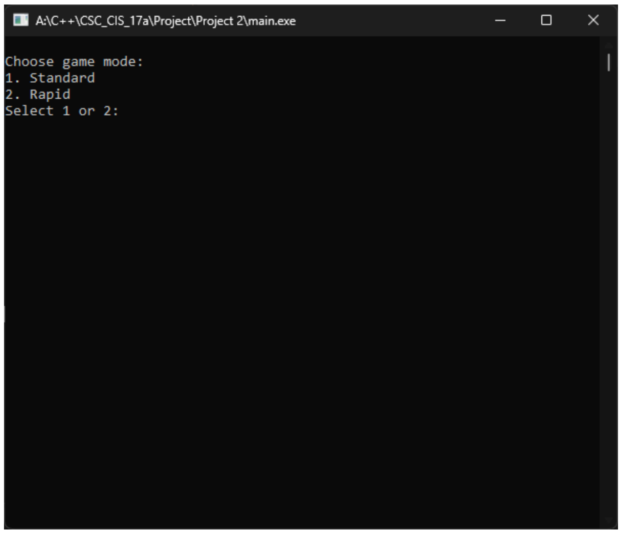


Select A:\C++\CSC_CIS_17a\Project\Project 2\main.exe

```
Battleship
1. Play Battleship
2. View Statistics
3. View Game History
4. Quit
Select an option: _
```

4.1.2 Game Mode

If you choose "Play Battleship", it asks you which mode you want. Standard uses the full 10x10 board, while Rapid is a smaller 5x5 game for quick testing. In this scenario, let's play a rapid game.

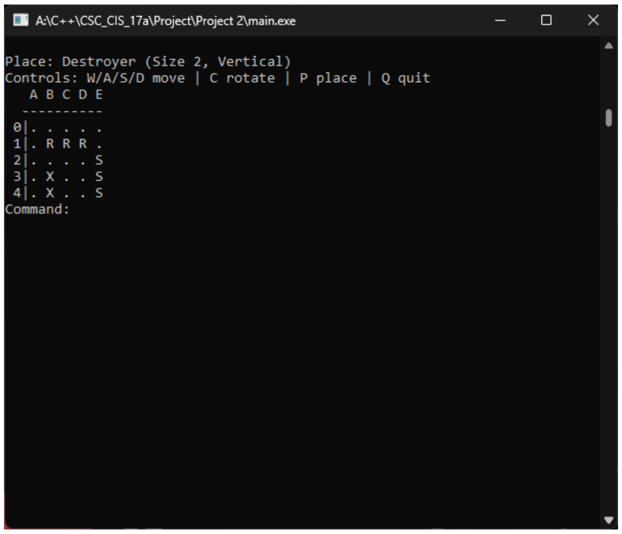


A:\C++\CSC_CIS_17a\Project\Project 2\main.exe

```
Choose game mode:
1. Standard
2. Rapid
Select 1 or 2:
```

4.1.3 Ship Placement

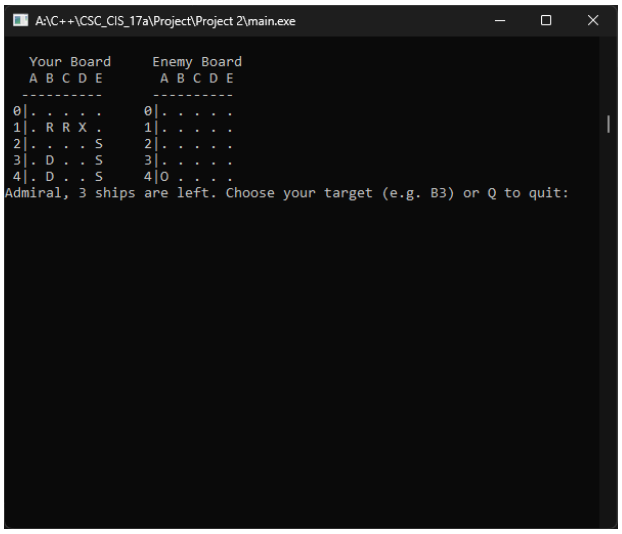
The player now sees their board, the controls and a prompt to enter the controls. You use WASD keys to move it around and C to rotate it.



```
A:\C++\CSC_CIS_17a\Project\Project 2\main.exe

Place: Destroyer (Size 2, Vertical)
Controls: W/A/S/D move | C rotate | P place | Q quit
   A B C D E
  ----------
 0|. . . . .
 1|. R R R .
 2|. . . . S
 3|. X . . S
 4|. X . . S
Command:
```

4.1.4 Gameplay

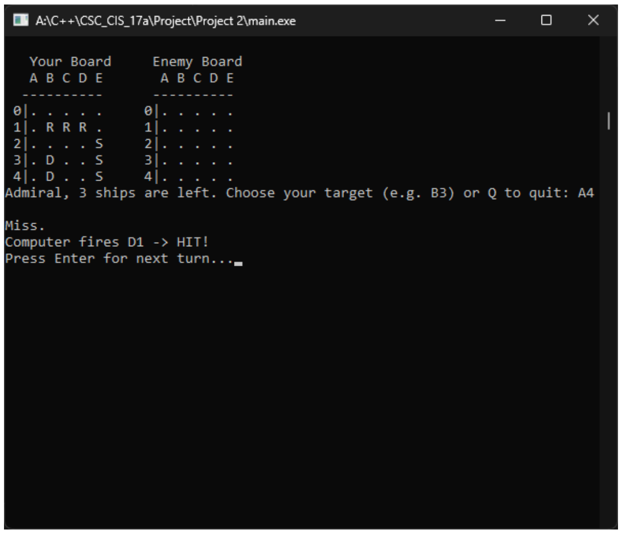
During the game, you see both boards. The left one is your board (showing where you have been hit), and the right one is the enemy board (showing your hits and misses). The new prompt uses the ship count to let you know how many enemy ships are left.



```
A:\C++\CSC_CIS_17a\Project\Project 2\main.exe

   Your Board      Enemy Board
   A B C D E        A B C D E
  ----------       ----------
 0|. . . . .      0|. . . . .
 1|. R R X .      1|. . . . .
 2|. . . . S      2|. . . . .
 3|. D . . S      3|. . . . .
 4|. D . . S      4|O . . . .
Admiral, 3 ships are left. Choose your target (e.g. B3) or Q to quit:
```

4.1.5 Computer's Turn

After you fire, the computer shoots back. If it hits you, it attacks the spots next to the hit. The game tells you exactly where it fired and what the result was.

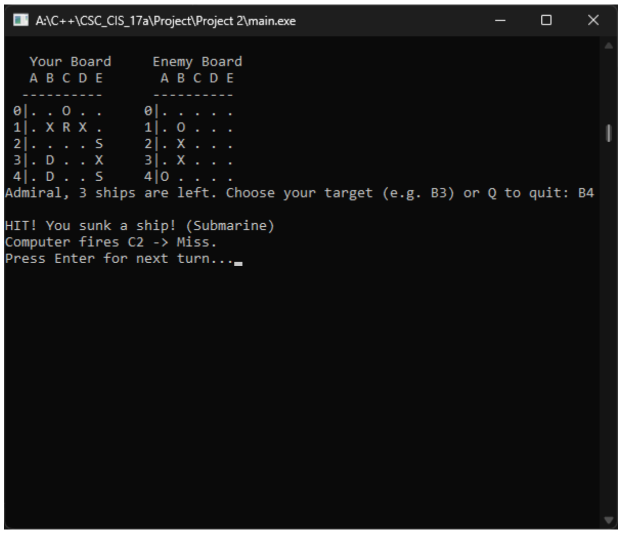


A:\C++\CSC_CIS_17a\Project\Project 2\main.exe

```
  Your Board     Enemy Board
  A B C D E       A B C D E
 ----------      ----------
0|. . . . .     0|. . . . .
1|. R R R .     1|. . . . .
2|. . . . S     2|. . . . .
3|. D . . S     3|. . . . .
4|. D . . S     4|. . . . .
Admiral, 3 ships are left. Choose your target (e.g. B3) or Q to quit: A4

Miss.
Computer fires D1 -> HIT!
Press Enter for next turn...
```

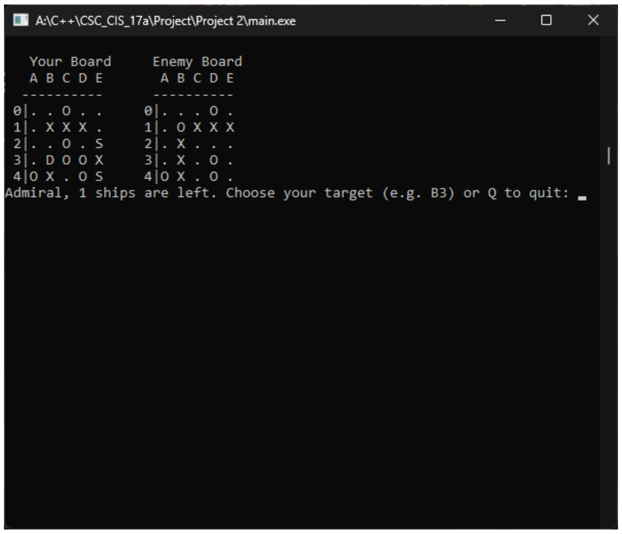
4.1.6 Ship is Sunk

When a ship has been found and sunk, the game will display the action. It will also update the count of how many ships are left to find.



A:\C++\CSC_CIS_17a\Project\Project 2\main.exe

```
  Your Board     Enemy Board
  A B C D E       A B C D E
 ----------      ----------
0|. . O . .     0|. . . . .
1|. X R X .     1|. O . . .
2|. . . . S     2|. X . . .
3|. D . . X     3|. X . . .
4|. D . . S     4|O . . . .
Admiral, 3 ships are left. Choose your target (e.g. B3) or Q to quit: B4

HIT! You sunk a ship! (Submarine)
Computer fires C2 -> Miss.
Press Enter for next turn...
```

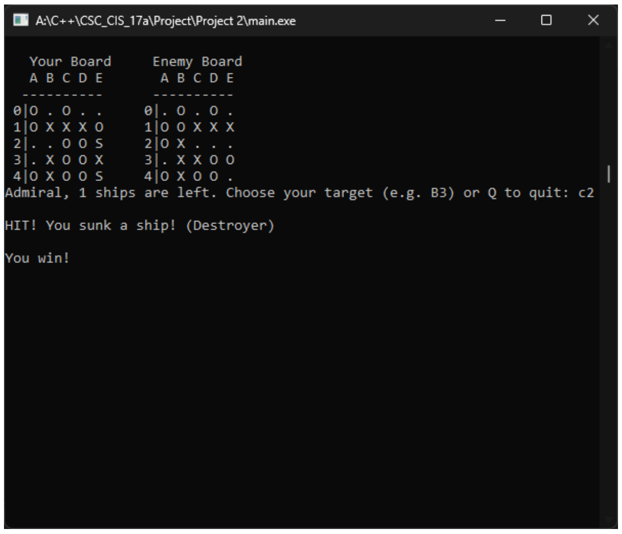


A:\C++\CSC_CIS_17a\Project\Project 2\main.exe

```
  Your Board     Enemy Board
  A B C D E       A B C D E
 ----------      ----------
0|. . O . .     0|. . . O .
1|. X X X .     1|. O X X X
2|. . O . S     2|. X . . .
3|. D O O X     3|. X . O .
4|O X . O S     4|O X . O .
Admiral, 1 ships are left. Choose your target (e.g. B3) or Q to quit:
```

4.1.7 Winner Decided

If the number of ships of the computer or player reaches zero, a message will display congratulating the winning side.

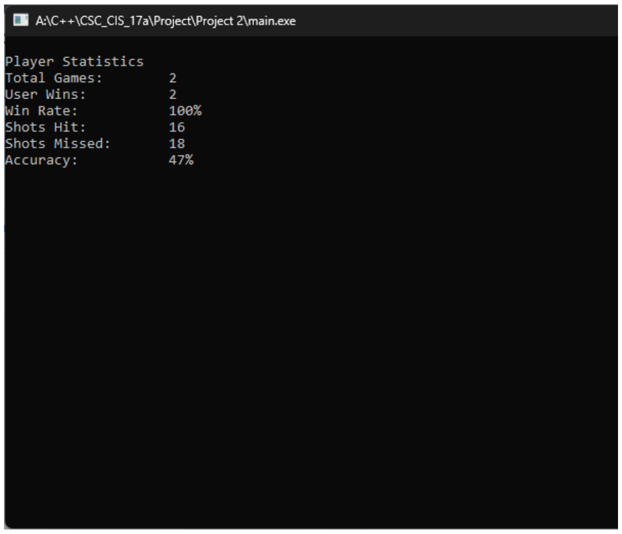


A:\C++\CSC_CIS_17a\Project\Project 2\main.exe

```
   Your Board     Enemy Board
   A B C D E       A B C D E
  ----------      ----------
 0|O . O . .     0|. O . O .
 1|O X X X O     1|O O X X X
 2|. . O O S     2|O X . . .
 3|. X O O X     3|. X X O O
 4|O X O O S     4|O X O O .
Admiral, 1 ships are left. Choose your target (e.g. B3) or Q to quit: c2

HIT! You sunk a ship! (Destroyer)

You win!
```

4.1.8 Statistics

Option 2 shows your lifetime stats. It reads from the stats.dat file to calculate your win percentage and shooting accuracy based on every shot you've ever taken.

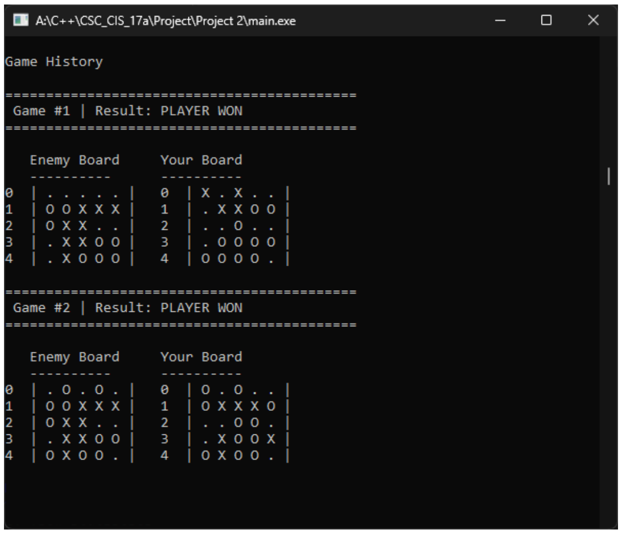


A:\C++\CSC_CIS_17a\Project\Project 2\main.exe

```
Player Statistics
Total Games:        2
User Wins:          2
Win Rate:           100%
Shots Hit:          16
Shots Missed:       18
Accuracy:           47%
```

4.1.9 Game History

If you select Option 3 from the menu, it reads the binary history file and replays the final board state of every game you've played. This helps you see how you won or lost.



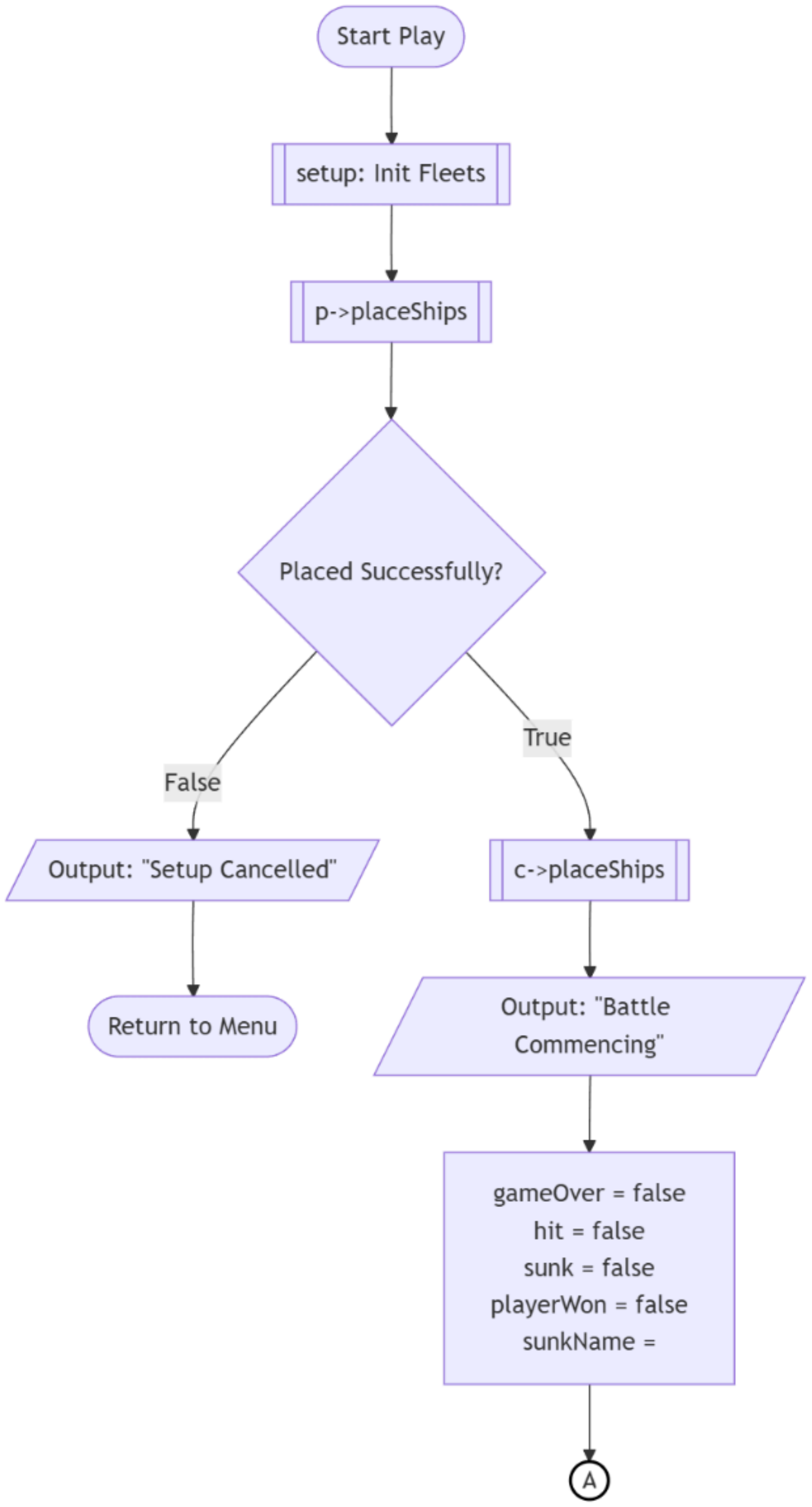
A:\C++\CSC_CIS_17a\Project\Project 2\main.exe

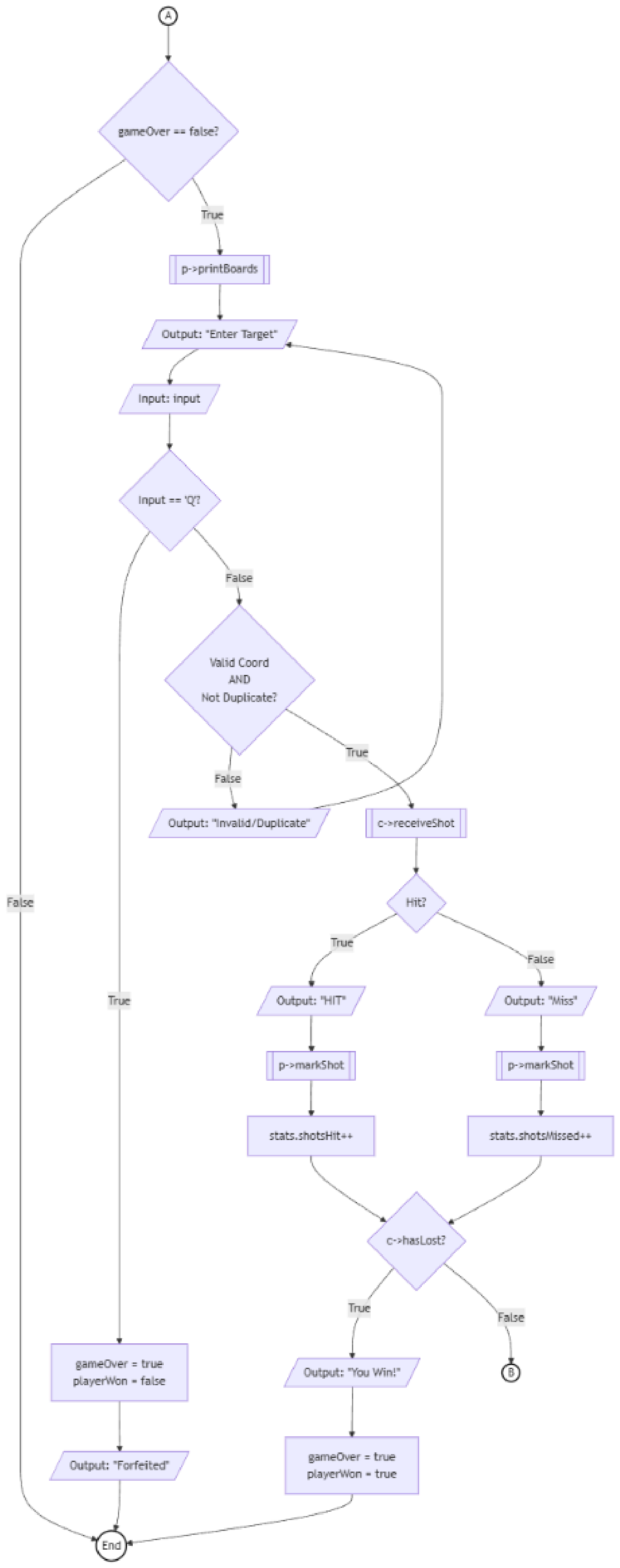
```
Game History

===========================================
 Game #1 | Result: PLAYER WON
===========================================

   Enemy Board     Your Board
   ----------      ----------
0  | . . . . . |   0  | X . X . . |
1  | O O X X X |   1  | . X X O O |
2  | O X X . . |   2  | . . O . . |
3  | . X X O O |   3  | . O O O O |
4  | . X O O O |   4  | O O O O . |

===========================================
 Game #2 | Result: PLAYER WON
===========================================

   Enemy Board     Your Board
   ----------      ----------
0  | . O . O . |   0  | O . O . . |
1  | O O X X X |   1  | O X X X O |
2  | O X X . . |   2  | . . O O . |
3  | . X X O O |   3  | . X O O X |
4  | O X O O . |   4  | O X O O . |
```

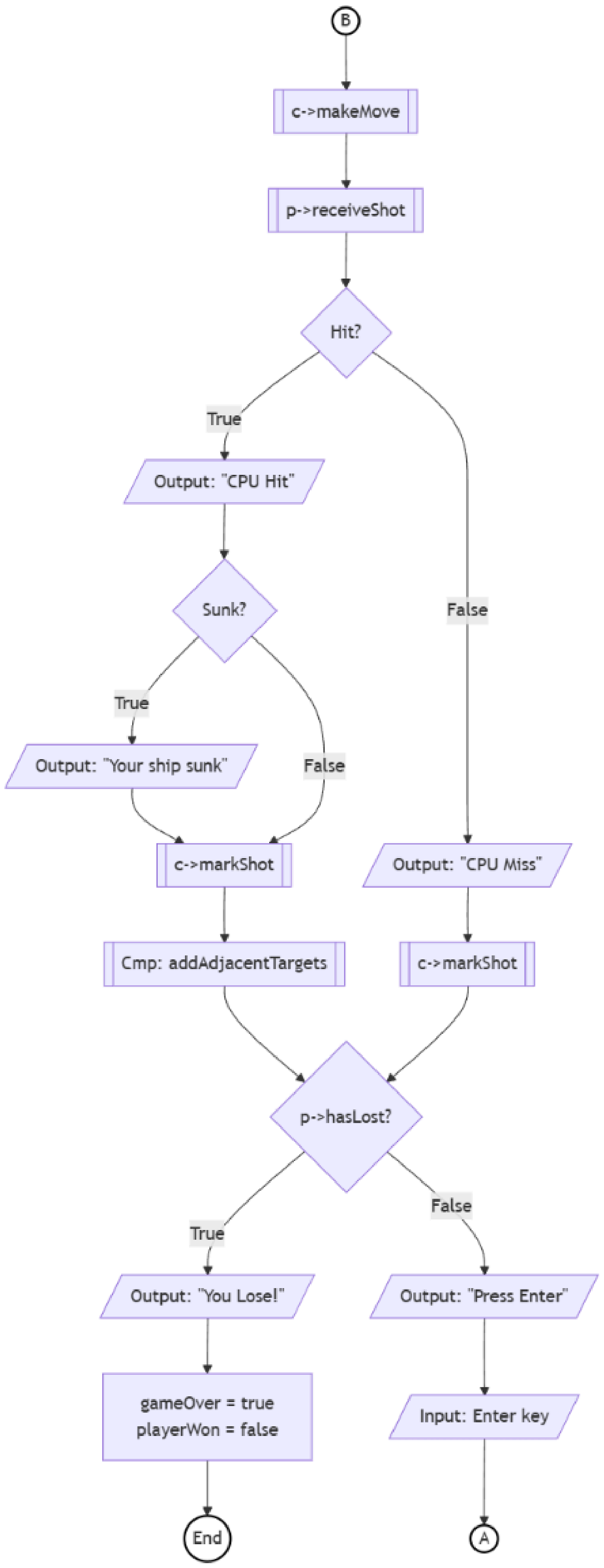
4.1.10 Quit

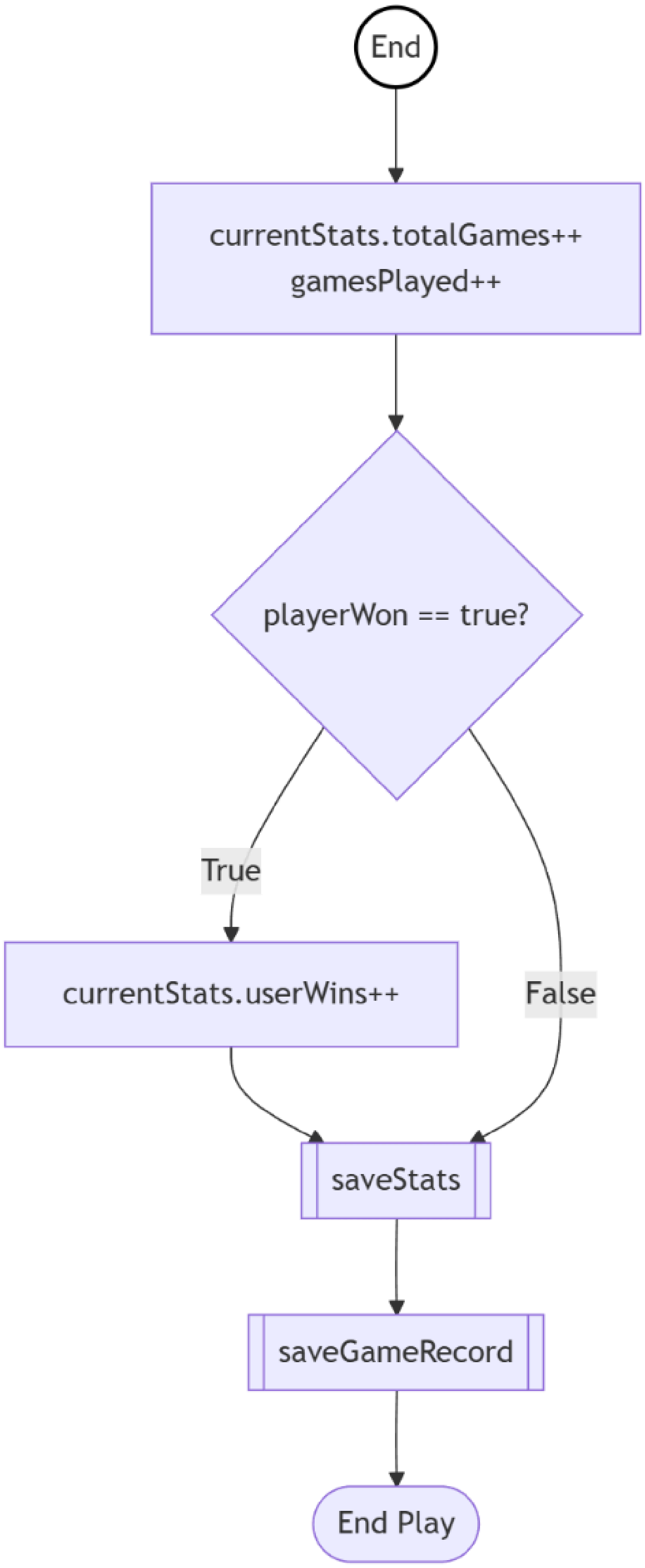
Choosing 4 closes out the program.

4.2 Flowcharts

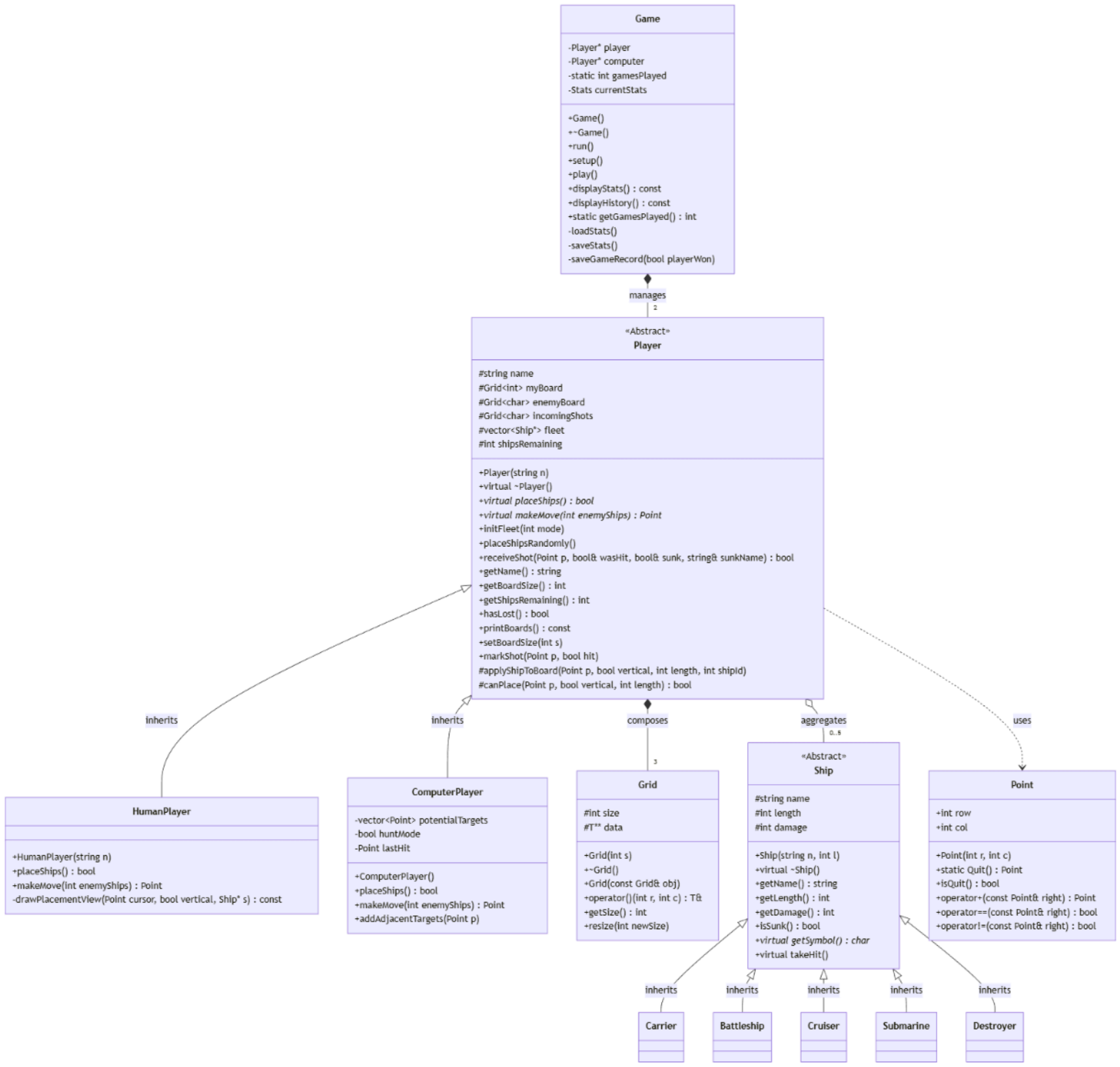



A
gameOver == false?
True
False
p->printBoards
Output: "Enter Target"
Input: input
Input == 'Q'?
True
False
Valid Coord AND Not Duplicate?
True
False
Output: "Invalid/Duplicate"
c->receiveShot
Hit?
True
False
Output: "HIT"
Output: "Miss"
p->markShot
p->markShot
stats.shotsHit++
stats.shotsMissed++
c->hasLost?
True
False
B
Output: "You Win!"
gameOver = true
playerWon = true
gameOver = true
playerWon = false
Output: "Forfeited"
End


B
c->makeMove
p->receiveShot
Hit?
True
Output: "CPU Hit"
Sunk?
True
Output: "Your ship sunk"
False
c->markShot
Cmp: addAdjacentTargets
False
Output: "CPU Miss"
c->markShot
p->hasLost?
True
Output: "You Lose!"
gameOver = true
playerWon = false
End
False
Output: "Press Enter"
Input: Enter key
A


End
currentStats.totalGames++
gamesPlayed++
playerWon == true?
True
False
currentStats.userWins++
saveStats
saveGameRecord
End Play

4.3 UML


Game
-Player* player
-Player* computer
-static int gamesPlayed
-Stats currentStats
+Game()
+~Game()
+run()
+setup()
+play()
+displayStats() : const
+displayHistory() : const
+static getGamesPlayed() : int
-loadStats()
-saveStats()
-saveGameRecord(bool playerWon)
manages
2
«Abstract»
Player
#string name
#Grid<int> myBoard
#Grid<char> enemyBoard
#Grid<char> incomingShots
#vector<Ship*> fleet
#int shipsRemaining
+Player(string n)
+virtual ~Player()
+virtual placeShips() : bool
+virtual makeMove(int enemyShips) : Point
+initFleet(int mode)
+placeShipsRandomly()
+receiveShot(Point p, bool& wasHit, bool& sunk, string& sunkName) : bool
+getName() : string
+getBoardSize() : int
+getShipsRemaining() : int
+hasLost() : bool
+printBoards() : const
+setBoardSize(int s)
+markShot(Point p, bool hit)
#applyShipToBoard(Point p, bool vertical, int length, int shipId)
#canPlace(Point p, bool vertical, int length) : bool
inherits
inherits
composes
aggregates
uses
3
0..5
HumanPlayer
+HumanPlayer(string n)
+placeShips() : bool
+makeMove(int enemyShips) : Point
-drawPlacementView(Point cursor, bool vertical, Ship* s) : const
ComputerPlayer
-vector<Point> potentialTargets
-bool huntMode
-Point lastHit
+ComputerPlayer()
+placeShips() : bool
+makeMove(int enemyShips) : Point
+addAdjacentTargets(Point p)
Grid
#int size
#T** data
+Grid(int s)
+~Grid()
+Grid(const Grid& obj)
+operator()(int r, int c) : T&
+getSize() : int
+resize(int newSize)
«Abstract»
Ship
#string name
#int length
#int damage
+Ship(string n, int l)
+virtual ~Ship()
+getName() : string
+getLength() : int
+getDamage() : int
+isSunk() : bool
+virtual getSymbol() : char
+virtual takeHit()
Point
+int row
+int col
+Point(int r, int c)
+static Quit() : Point
+isQuit() : bool
+operator+(const Point& right) : Point
+operator==(const Point& right) : bool
+operator!=(const Point& right) : bool
inherits
inherits
inherits
inherits
inherits
Carrier
Battleship
Cruiser
Submarine
Destroyer

4.4 Show of Concept

Chapter	Topic	Line #s	Notes
13	**Classes**		
	Instance of a Class	main.cpp:19	Create instance of Game class to start the program
	Private Data Members	Game.h:18	Variables like player and computer are private
	Specification vs. Implementation		Code is separated into header and source files
	Inline	Player.h:78	getName() is defined in the header
	Constructors	Game.cpp:21	Initialize game pointer
	Destructors	Game.cpp:30	Free up memory used by game pointer
	Array of Objects	Player.h:31	Vector to hold fleet of ships
	UML	Section 4.3	My Class diagram shows how Game, Player, and Grid fit together
14	**More About Classes**		
	Static	Game.h:22	static int gamesPlayed keeps a count shared across all game instances
	Friends	Player.h:22	friend class Game lets the Game controller access the Player's private board
	Copy Constructors	Grid.h:55	Copying a board copies the actual data, not just the pointer
	Operator Overloading	Point.h:61	I overloaded << so I can print coordinates easily
	Aggregation	Player.h:28	The Player class has a Grid object and has a fleet of Ships

15	**Inheritance**		
	Protected Members	Player.h:26	Child classes can access name and myBoard because they are protected, not private
	Base Class to Derived	Player.h:104	HumanPlayer inherits everything from the base Player class
	Polymorphic Associations	Game.cpp:104	The Game uses Player* pointers to handle both Human and Computer players
	Abstract C lasses	Player.h:21	Player is abstract because it has pure virtual functions like placeShips()
16	**Advanced Classes**		
	Exceptions	Grid.h:78	The grid throws an out_of_range error if you try to access a bad index
	Templates	Grid.h:19	template <class T> lets the Grid work for both int (ships) and char (shots)
	STL	Player.h:46	I used the Standard Template Library for std::vector and std::string

5. References

5.1 Javascript Program

As mentioned, the inspiration for the project came from a Javascript version of the game. This was a lab assignment for CSC-14A taught at Riverside City College in the Fall semester of 2019. It's quite telling that it was developed on an older version of Javascript, as common practice for creating variables is using *let* instead of *var.*

Unlike this program, the Javascript version was more focused on only the player guessing where the computer's ships are, not competing against it.

From the script, parseGuess led to the creation of parseTarget in this program. It also was a reminder to prevent the player from guessing a position off the board, as well as not placing a ship within the game board boundaries.

It also served as a basis for what messages to show, creating a 2D array to hold the game boards, and generating the layout of the computer's fleet.

Below is the script:

```
var view = {
        displayMessage: function(msg) {
                var messageArea = document.getElementById("messageArea");
                messageArea.innerHTML = msg;
        },
        displayHit: function(location) {
                var cell = document.getElementById(location);
                cell.setAttribute("class", "hit");
        },
        displayMiss: function(location) {
                var cell = document.getElementById(location);
                cell.setAttribute("class", "miss");
        }
};

var model = {
    boardSize: 7,
    numShips: 3,
    shipLength: 3,
    shipsSunk: 0,

        ships: [
                { locations: [0, 0, 0], hits: ["", "", ""] },
                { locations: [0, 0, 0], hits: ["", "", ""] },
                { locations: [0, 0, 0], hits: ["", "", ""] }
        ],

        fire: function(guess) {

                for(var i = 0; i < this.numShips; i++) {
                        var ship = this.ships[i];
                        var index = ship.locations.indexOf(guess);

                        // check if a ship location has already been hit
                        if ( ship.hits[index] === "hit" ) {
                                view.displayMessage("Oops, you already hit that location");
                                return true;
                        }
                        else if ( index >= 0 ) {
                                ship.hits[index] = "hit";
                                view.displayHit(guess);
                                view.displayMessage("HIT!");

                                if ( this.isSunk(ship) ) {
                                        view.displayMessage("You sank my battleship!");
                                        this.shipsSunk++;
                                }
                                return true;
                        }
                        $('#guessInput').focus();
                }
                view.displayMiss(guess);
                view.displayMessage("You Missed");
                return false;
        },

        isSunk: function(ship) {
                for (var i = 0; i < this.shipLength; i++) {
                        if (ship.hits[i] !== "hit") {
                                return false;
                        }
                }
                $('#guessInput').focus();
                return true;
        },

        generateShipLocations: function() {
                var locations;
                for (var i = 0; i < this.numShips; i++) {
                do {
                        locations = this.generateShip();
                }
                while (this.collision(locations));
                        this.ships[i].locations = locations;
                }
        },

        generateShip: function() {
                var direction = Math.floor(Math.random() * 2);
                var row, col;

                if (direction === 1) { // horizontal
                        row = Math.floor(Math.random() * this.boardSize);
                        col = Math.floor(Math.random() * (this.boardSize - this.shipLength + 1));
                } else { // vertical
                        row = Math.floor(Math.random() * (this.boardSize - this.shipLength + 1));
                        col = Math.floor(Math.random() * this.boardSize);
                }

                var newShipLocations = [];

                for (var i = 0; i < this.shipLength; i++) {
                        if (direction === 1) {
                                newShipLocations.push(row + "" + (col + i));
                        } else {
                                newShipLocations.push((row + i) + "" + col);
                        }
                }
                return newShipLocations;
        },

        collision: function(locations) {
                for (var i = 0; i < this.numShips; i++) {
                        var ship = this.ships[i];
                        for (var j = 0; j < locations.length; j++) {
                                if (ship.locations.indexOf(locations[j]) >= 0) {
                                        return true;
                                }
                        }
                }
                return false;
        }
};

var controller = {
        guesses: 0,

        processGuess: function(guess) {
                var location = parseGuess(guess);

                if (location) {
                        this.guesses++;
                        var hit = model.fire(location);
                        if (hit && model.shipsSunk === model.numShips) {
                                view.displayMessage("You sank all my battleships, in " + this.guesses +
" guesses");
                        }
                }
        }
};

// helper function to parse a guess from the user
function parseGuess(guess) {
        var alphabet = ["A", "B", "C", "D", "E", "F", "G"];

        if (guess === null || guess.length !== 2) {
                alert("Oops, please enter a letter and a number on the board.");
        }
        else {
                var firstChar = guess.charAt(0);
                var row = alphabet.indexOf(firstChar);
                var column = guess.charAt(1);
                if (isNaN(row) || isNaN(column)) {
                        alert("Oops, that isn't on the board.");
                }
                else if (row < 0 || row >= model.boardSize || column < 0 || column >= model.boardSize)
{
                        alert("Oops, that's off the board!");
                }
                else {
                        return row + column;
                }
        }
        return null;
}

// event handlers
function handleFireButton() {
        var guessInput = document.getElementById("guessInput");
        var guess = guessInput.value.toUpperCase();
        controller.processGuess(guess);
        guessInput.value = "";
}

function handleKeyPress(e) {
        var fireButton = document.getElementById("fireButton");

        e = e || window.event;
        if (e.keyCode === 13) {
                fireButton.click();
                return false;
        }
}

window.onload = init;

function init() {
        var fireButton = document.getElementById("fireButton");
        fireButton.onclick = handleFireButton;

        var guessInput = document.getElementById("guessInput");
        guessInput.onkeypress = handleKeyPress;

        model.generateShipLocations();
}
```

5.2 Online References

- Virtual Function in C++
- Exceptions, Templates, Vectors

5.3 Gaddis

Throughout the programming stage, it was essential to lean on the material in Gaddis.

Rather than save data to a normal text file, the game history needed to be saved to a binary file. Program 12-17 and 12-19 were useful references for understanding how to put data into binary files, get data out of binary files, and get the file's read and write position.

Program 12-17

```
// This program demonstrates the seekg function.
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
char ch; // To hold a character
// Open the file for input.
fstream file("letters.txt", ios::in);
// Move to byte 5 from the beginning of the file
// (the 6th byte) and read the character there.
file.seekg(5L, ios::beg);
file.get(ch);
cout << "Byte 5 from beginning: " << ch << endl;
// Move to the 10th byte from the end of the file
// and read the character there.
file.seekg(-10L, ios::end);
file.get(ch);
cout << "10th byte from end: " << ch << endl;
// Move to byte 3 from the current position
// (the 4th byte) and read the character there.
file.seekg(3L, ios::cur);
file.get(ch);
cout << "Byte 3 from current: " << ch << endl;
file.close();
return 0;
}
```

Program 12-19

```
// This program demonstrates the tellg function.
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
long offset; // To hold an offset amount
long numBytes; // To hold the file size
char ch; // To hold a character
char again; // To hold Y or N
// Open the file for input.
fstream file("letters.txt", ios::in);
// Determine the number of bytes in the file.
file.seekg(0L, ios::end);
numBytes = file.tellg();
cout << "The file has " << numBytes << " bytes.\n";
// Go back to the beginning of the file.
file.seekg(0L, ios::beg);
// Let the user move around within the file.
do
{
// Display the current read position.
cout << "Currently at position " << file.tellg() << endl;
// Get a byte number from the user.
cout << "Enter an offset from the beginning of the file: ";
cin >> offset;
// Move the read position to that byte, read the
// character there, and display it.
if (offset >= numBytes) // Past the end of the file?
cout << "Cannot read past the end of the file.\n";
else
{
file.seekg(offset, ios::beg);
file.get(ch);
cout << "Character read: " << ch << endl;
}
// Does the user want to try this again?
cout << "Do it again? ";
cin >> again;
} while (again == 'Y' || again == 'y');
// Close the file.
file.close();
return 0;
}
```

6. Program

Game.cpp

```
#include "Game.h"
#include <iostream>
#include <iomanip>
#include <string>
using namespace std;
// Initialize Static Member
int Game::gamesPlayed = 0;
/**
 * @brief Constructor. Initializes pointers and loads persistent stats.
 */
Game::Game() {
    player = NULL;
    computer = NULL;
    loadStats();
}
/**
 * @brief Destructor. Frees allocated memory for players.
 */
Game::~Game() {
    delete player;
    delete computer;
}
/**
 * @brief Main application loop. Handles menu navigation.
 */
void Game::run() {
    int choice = 0;
    do {
        cout << "\nBattleship\n";
        cout << "1. Play Battleship\n";
        cout << "2. View Statistics\n";
        cout << "3. View Game History\n";
        cout << "4. Quit\n";
        cout << "Select an option: ";
        cin >> choice;
        // Input Validation
        if (cin.fail()) {
            cin.clear(); cin.ignore(10000, '\n');
            choice = 0;
        }
        switch (choice) {
            case 1:
                setup();
                play();
                break;
            case 2:
                displayStats();
                break;
            case 3:
                displayHistory();
                break;
            case 4:
                cout << "Thanks for playing!\n";
                break;
            default:
                cout << "Invalid choice. Please try again.\n";
        }
    } while (choice != 4);
}
/**
 * @brief Configures game mode and initializes players.
 * Handles selection between Standard (10x10) and Rapid (5x5) modes.
 */
void Game::setup() {
    // Select Mode
    int choice;
    cout << "\nChoose game mode:\n";
    cout << "1. Standard\n";
    cout << "2. Rapid\n";
    cout << "Select 1 or 2: ";
    cin >> choice;
    // Input Validation
    while (choice != 1 && choice != 2) {
        cout << "Invalid input. Please choose a game mode: ";
        cin.clear();
        cin.ignore(10000, '\n');
        cin >> choice;
    }
    // Create Players
    player = new HumanPlayer("Player");
    computer = new ComputerPlayer();
    // Initialize Fleets
    Player &p = *player;
    Player &c = *computer;
    if (choice == 2) {
        p.setBoardSize(5);
        c.setBoardSize(5);
    }
    p.initFleet(choice);
    c.initFleet(choice);
}
/**
 * @brief Core gameplay loop. Manages turns, shots, and win conditions.
 */
void Game::play() {
    Player &p = *player;
    Player &c = *computer;
    if (!p.placeShips()) {
        cout << "\nGame setup cancelled. Returning to menu.\n";
        return;
    }
    c.placeShips();
```

```
cout << "\nAll ships placed! Battle commencing...\n";
// Main Game Loop
bool gameOver = false;
bool hit = false;
bool sunk = false;
bool playerWon = false;
string sunkName = "";
while (!gameOver) {
    // Player Turn
    cout << string(50, '\n');
    p.printBoards();
    // Get Valid Move from Player
    Point target = p.makeMove(c.getShipsRemaining());
    // Handle game quit
    if (target.isQuit()) {
        cout << "\nYou forfeited the match!\n";
        gameOver = true;
        playerWon = false;
        break;
    }
    // Fire at Computer
    if (c.receiveShot(target, hit, sunk, sunkName)) {
        cout << "HIT!";
        if (sunk) cout << " You sunk a ship! (" << sunkName << ")";
        p.markShot(target, true);
        currentStats.shotsHit++;
    } else {
        cout << "Miss.";
        p.markShot(target, false);
        currentStats.shotsMissed++;
    }
    cout << "\n";
    // Check Win
    if (c.hasLost()) {
        cout << "\nYou win!\n";
        gameOver = true;
        playerWon = true;
        cin.get();
        break;
    }
    // Computer's Turn
    Point cpuTarget = c.makeMove(p.getShipsRemaining());
    // Fire at Player
    if (p.receiveShot(cpuTarget, hit, sunk, sunkName)) {
        cout << "Computer fires " << cpuTarget << " -> HIT!\n";
        if (sunk) cout << "Your ship was sunk!\n";
        // Update computer target range
        c.markShot(cpuTarget, true);
        ComputerPlayer* cpuPtr = dynamic_cast<ComputerPlayer*>(computer);
        if (cpuPtr) {
            ComputerPlayer &cpu = *cpuPtr;
            cpu.addAdjacentTargets(cpuTarget);
        }
    } else {
        cout << "Computer fires " << cpuTarget << " -> Miss.\n";
```

```
            c.markShot(cpuTarget, false);
        }
        // Check Loss
        if (p.hasLost()) {
            cout << "\nYou Lose!\n";
            gameOver = true;
            playerWon = false;
        }
        // Pause between turns
        if (!gameOver) {
            cout << "Press Enter for next turn...";
            cin.ignore();
            cin.get();
        }
    }
    // Update Stats
    currentStats.totalGames++;
    if (playerWon) {
        currentStats.userWins++;
    }
    saveStats();
    saveGameRecord(playerWon);
    gamesPlayed++;
}
/**
 * @brief Loads stats from binary file safely.
 * Resets to 0 if file is missing or corrupt.
 */
void Game::loadStats() {
    currentStats.totalGames = 0;
    currentStats.userWins = 0;
    currentStats.shotsHit = 0;
    currentStats.shotsMissed = 0;
    std::ifstream in("stats.dat", std::ios::binary);
    // If file exists
    if (in) {
        Stats temp;
        in.read(reinterpret_cast<char*>(&temp), sizeof(Stats));
        if (in) {
            currentStats = temp;
        }
    }
}
/**
 * @brief Saves current stats to binary file.
 */
void Game::saveStats() {
    std::ofstream out("stats.dat", std::ios::binary);
    if (out) {
        out.write(reinterpret_cast<const char*>(&currentStats), sizeof(Stats));
    }
}
/**
 * @brief Prints player statistics and calculated accuracy to console.
 */
```

```
void Game::displayStats() const {
    cout << "\nPlayer Statistics\n";
    cout << left << setw(20) << "Total Games:" << currentStats.totalGames << "\n";
    cout << left << setw(20) << "User Wins:" << currentStats.userWins << "\n";
    if (currentStats.totalGames > 0) {
        // Calculate percentage
        int pct = (currentStats.userWins * 100) / currentStats.totalGames;
        cout << left << setw(20) << "Win Rate:" << pct << "%\n";
        cout << left << setw(20) << "Shots Hit:" << currentStats.shotsHit << "\n";
        cout << left << setw(20) << "Shots Missed:" << currentStats.shotsMissed << "\n";
        int totalShots = currentStats.shotsHit + currentStats.shotsMissed;
        if (totalShots > 0) {
            // Calculate accuracy
            int acc = (currentStats.shotsHit * 100) / totalShots;
            cout << left << setw(20) << "Accuracy:" << acc << "%\n";
        } else {
            cout << left << setw(20) << "Accuracy:" << "0%\n";
        }
    } else {
        cout << left << setw(20) << "Win Rate:" << "0%\n";
    }
}
/**
 * @brief Appends the result of the last game to the history file.
 * Handles dynamic board sizing for correct storage.
 */
void Game::saveGameRecord(bool playerWon) {
    GameRecord rec;
    rec.gameId = currentStats.totalGames;
    rec.playerWon = playerWon;
    rec.boardSize = player->getBoardSize();
    for(int r=0; r<10; r++) {
        for(int c=0; c<10; c++) {
            rec.playerShots[r][c] = '.';
            rec.cpuShots[r][c] = '.';
        }
    }
    int size = player->myBoard.getSize();
    for (int r = 0; r < size; ++r) {
        for (int c = 0; c < size; ++c) {
            char pShot = player->enemyBoard(r, c);
            if (pShot != 0) rec.playerShots[r][c] = pShot;
            char cShot = player->incomingShots(r, c);
            if (cShot != 0) rec.cpuShots[r][c] = cShot;
        }
    }
    // Append to binary file
    ofstream out("history.dat", ios::binary | ios::app);
    if (out) {
        out.write(reinterpret_cast<const char*>(&rec), sizeof(GameRecord));
    }
}
/**
 * @brief Reads and displays past game records from binary file.
 * Formats output based on the board size of each record.
 */
void Game::displayHistory() const {
    ifstream in("history.dat", ios::binary);
    if (!in) {
        cout << "\nNo game history found.\n";
        return;
    }
    GameRecord rec;
    cout << "\nGame History\n";
    while (in.read(reinterpret_cast<char*>(&rec), sizeof(GameRecord))) {
        cout << "\n" << string(43, '=') << "\n";
        cout << " Game #" << rec.gameId << " | Result: "
             << (rec.playerWon ? "PLAYER WON" : "COMPUTER WON") << "\n";
        cout << string(43, '=') << "\n\n";
        int boardWidth = 3 + (rec.boardSize * 2);
        // Print Headers aligned to board width
        cout << string(3, ' ') << "Enemy Board" << string(boardWidth - 11, ' ') << string(3, ' ') << "Your
Board\n";
        cout << string(3, ' ') << string(rec.boardSize * 2, '-') << string(6, ' ') << string(rec.boardSize * 2, '-') <<
"\n";
        for (int r = 0; r < rec.boardSize; ++r) {
            // Enemy Board
            cout << setw(2) << r << " |";
            for (int c = 0; c < rec.boardSize; ++c) {
                cout << " " << rec.playerShots[r][c];
            }
            cout << " |   ";
            // Player Board
            cout << setw(2) << r << " |";
            for (int c = 0; c < rec.boardSize; ++c) {
                cout << " " << rec.cpuShots[r][c];
            }
            cout << " |\n";
        }
    }
}
```

Game.h

```
#ifndef GAME_H
#define GAME_H
#include "Player.h"
#include <fstream>
/**
 * @class Game
 * @brief Manages the game loop, player creation, statistics, and file I/O.
 */
class Game {
    private:
        Player* player;      ///< Pointer to the human player
        Player* computer;   ///< Pointer to the computer player
        static int gamesPlayed;
        /**
         * @struct Stats
         * @brief Structure for persistent player statistics.
         */
        struct Stats {
            unsigned int totalGames;
            unsigned int userWins;
            unsigned int shotsHit;
            unsigned int shotsMissed;
        };
        Stats currentStats; ///< Holds currently loaded statistics
        /**
         * @struct GameRecord
         * @brief Structure for saving game history snapshots.
         */
        struct GameRecord {
            unsigned int gameId;
            bool playerWon;
            int boardSize;
            char playerShots[10][10];
            char cpuShots[10][10];
        };
        /**
         * @brief Loads statistics from the binary file.
         */
        void loadStats();
        /**
         * @brief Saves current statistics to the binary file.
         */
        void saveStats();
        /**
         * @brief Appends the result of the last game to the history file.
         * @param playerWon True if the human won.
         */
        void saveGameRecord(bool playerWon);
    public:
        /**
         * @brief Constructor. Initializes pointers and loads stats.
         */
    Game();
    /**
     * @brief Destructor. Cleans up player memory.
     */
    ~Game();
    /**
     * @brief Runs the main application menu loop.
     * Handles options for Play, Stats, History, and Quit.
     */
    void run();
    /**
     * @brief Handles game configuration (Mode selection, Fleet init).
     */
    void setup();
    /**
     * @brief Executes the actual gameplay loop (Turns, Shots, Win Check).
     */
    void play();
    /**
     * @brief Displays persistent statistics to the console.
     */
    void displayStats() const;
    /**
     * @brief Reads and displays past game history from the binary file.
     */
    void displayHistory() const;
    /**
     * @brief Static getter for session game count.
     */
    static int getGamesPlayed() { return gamesPlayed; }
};
#endif
```

Grid.h

```
#ifndef GRID_H
#define GRID_H
#include <iostream>
#include <vector>
#include <stdexcept>
/**
 * @class Grid
 * @brief A 2D array wrapper that manages memory safely.
 * Uses templates to store any data type (int for IDs, char for markers).
 */
template <class T>
class Grid {
    protected:
        int size; ///< Width/Height of the square grid
        T** data; ///< Pointer to the dynamic 2D array
    public:
        /**
         * @brief Constructor. Allocates memory and initializes to zero/null.
         * @param s Size of the grid (default 10).
         */
        Grid(int s = 10) : size(s) {
            data = new T*[size];
            for (int i = 0; i < size; i++) {
                data[i] = new T[size]{};
            }
        }
        /**
         * @brief Destructor. Frees dynamically allocated memory.
         */
        ~Grid() {
            if (data) {
                for (int i = 0; i < size; i++) {
                    delete[] data[i];
                }
                delete[] data;
            }
        }
        /**
         * @brief Copy Constructor. Performs a deep copy of the grid.
         */
        Grid(const Grid &obj) {
            size = obj.size;
            data = new T*[size];
            for (int i = 0; i < size; i++) {
                data[i] = new T[size];
                for (int j = 0; j < size; j++) {
                    data[i][j] = obj.data[i][j];
                }
            }
        }
        /**
         * @brief Access operator for modifying data.
         * @param r Row index.
         * @param c Column index.
         * @return Reference to the element.
         * @throws std::out_of_range if index is invalid.
         */
        T& operator()(int r, int c) {
            if (r < 0 || r >= size || c < 0 || c >= size) {
                throw std::out_of_range("Grid index out of bounds");
            }
            return data[r][c];
        }
        /**
         * @brief Access operator for reading data (const).
         */
        const T& operator()(int r, int c) const {
            if (r < 0 || r >= size || c < 0 || c >= size) {
                throw std::out_of_range("Grid index out of bounds");
            }
            return data[r][c];
        }
        int getSize() const { return size; }
        /**
         * @brief Resizes the grid. WARNING: Clears all existing data.
         * @param newSize The new dimension for the square grid.
         */
        void resize(int newSize) {
            if (size == newSize) return;
            // Delete old memory
            if (data) {
                for (int i = 0; i < size; i++) {
                    delete[] data[i];
                }
                delete[] data;
            }
            // Allocate new memory
            size = newSize;
            data = new T*[size];
            for (int i = 0; i < size; i++) {
                data[i] = new T[size]{};
            }
        }
};
#endif
```

main.cpp

```
#include "Game.h"
#include <iostream>
using namespace std;
/**
 * @brief Main execution function.
 * Initializes the Game controller and starts the main loop.
 * @return 0 on successful execution.
 */
int main() {
    // Create the game controller instance
    Game battleship;
    // Start the game application loop
    battleship.run();
    return 0;
}
```

Player.cpp

```
// Libraries
#include "Player.h"
#include <iostream>
#include <iomanip>
#include <ctime>
#include <cstdlib>
#include <cstring>
#include <limits>
#include <string>
using namespace std;
/**
 * @brief Constructor. Initializes player name and ship count.
 * @param n Name of the player.
 */
Player::Player(string n) : name(n) {
    shipsRemaining = 0;
}
/**
 * @brief Destructor. Cleans up dynamically allocated ships.
 */
Player::~Player() {
    for (int i = 0; i < fleet.size(); i++) {
        delete fleet[i];
    }
    fleet.clear();
}
/**
 * @brief Checks if a ship can be placed at a specific location.
 * Verifies boundaries and overlap with existing ships.
 * @param p Starting coordinate.
 * @param vertical Orientation flag.
 * @param length Size of the ship.
 * @return true if valid, false otherwise.
 */
bool Player::canPlace(Point p, bool vertical, int length) const {
    int size = myBoard.getSize();
    for (int k = 0; k < length; ++k) {
        int r = p.row + (vertical ? k : 0);
        int c = p.col + (vertical ? 0 : k);
        if (r < 0 || r >= size || c < 0 || c >= size) return false;
        if (myBoard(r, c) != 0) return false;
    }
    return true;
}
/**
 * @brief Writes the ship's ID to the board grid.
 * @param p Starting coordinate.
 * @param vertical Orientation flag.
 * @param length Size of the ship.
 * @param shipId Unique ID of the ship.
 */
```

```
void Player::applyShipToBoard(Point p, bool vertical, int length, int shipId) {
    for (int k = 0; k < length; ++k) {
        int r = p.row + (vertical ? k : 0);
        int c = p.col + (vertical ? 0 : k);
        myBoard(r, c) = shipId;
    }
}
/**
 * @brief Places all ships in the fleet randomly.
 * Used by ComputerPlayer and potentially for auto-setup.
 */
void Player::placeShipsRandomly() {
    int id = 1;
    int size = myBoard.getSize();
    for (int i = 0; i < fleet.size(); i++) {
        Ship &s = *fleet[i];
        bool placed = false;
        while (!placed) {
            int r = rand() % size;
            int c = rand() % size;
            bool vert = rand() % 2;
            if (canPlace(Point(r, c), vert, s.getLength())) {
                applyShipToBoard(Point(r, c), vert, s.getLength(), id);
                placed = true;
            }
        }
        id++;
    }
}
/**
 * @brief Processes a shot fired at this player.
 * Updates the board, ship health, and win condition.
 * @param p Coordinate of the shot.
 * @param wasHit [out] True if a ship was hit.
 * @param sunk [out] True if a ship sank.
 * @param sunkName [out] Name of the sunk ship.
 * @return true if shot was valid, false if invalid.
 */
bool Player::receiveShot(Point p, bool &wasHit, bool &sunk, string &sunkName) {
    wasHit = false;
    sunk = false;
    sunkName = "";
    // Check Bounds
    if (p.row < 0 || p.row >= myBoard.getSize() || p.col < 0 || p.col >=
myBoard.getSize()) {
        return false;
    }
    // Check Ship ID
    int shipId = myBoard(p.row, p.col);
    if (shipId > 0) {
        wasHit = true;
        // Mark Hit
        incomingShots(p.row, p.col) = 'X';
        Ship &s = *fleet[shipId - 1];
        s.takeHit();
        if (s.isSunk()) {
            sunk = true;
            sunkName = s.getName();
            shipsRemaining--;
        }
        return true;
    }
    // Mark Miss
    incomingShots(p.row, p.col) = 'O';
    return false;
}
/**
 * @brief Checks if the player has lost (0 ships remaining).
 * @return true if lost.
 */
bool Player::hasLost() const {
    return shipsRemaining == 0;
}
/**
 * @brief Initializes the fleet based on game mode.
 * @param mode 1 = Standard (5 ships), 2 = Rapid (3 ships).
 */
void Player::initFleet(int mode) {
    // Clear out any ships
    for (int i = 0; i < fleet.size(); i++) {
        delete fleet[i];
    }
    fleet.clear();
    // Standard Mode
    if (mode == 1) {
        fleet.push_back(new Carrier());
        fleet.push_back(new Battleship());
        fleet.push_back(new Cruiser());
        fleet.push_back(new Submarine());
        fleet.push_back(new Destroyer());
    }
    // Rapid Mode
    else {
        fleet.push_back(new Cruiser());
        fleet.push_back(new Submarine());
        fleet.push_back(new Destroyer());
    }
    shipsRemaining = fleet.size();
}
/**
 * @brief Displays both Player and Enemy boards side-by-side.
 */
void Player::printBoards() const {
    int size = myBoard.getSize();
    // Header
    cout << string(3, ' ') << "Your Board" << string(size * 2 - 8, ' ') << string(3,
' ') << "Enemy Board\n";
    cout << string(3, ' ');
    for (int c = 0; c < size; ++c) cout << (char)('A' + c) << " ";
    cout << string(6, ' ');
    for (int c = 0; c < size; ++c) cout << (char)('A' + c) << " ";
    cout << "\n";
    cout << string(2, ' ') << string(size * 2, '-') << string(6, ' ') << string(size
* 2, '-') << "\n";
    for (int r = 0; r < size; ++r) {
        // Left Side: Player Board (Shows ships + incoming shots)
        cout << setw(2) << r << "|";
        for (int c = 0; c < size; ++c) {
            // Check if we have been shot here first!
            char shot = incomingShots(r, c);
            if (shot != 0) {
                cout << shot << " ";
            } else {
                int id = myBoard(r, c);
                char sym = '.';
                if (id > 0) {
                    Ship &s = *fleet[id - 1];
                    sym = s.getSymbol();
                }
                cout << sym << " ";
            }
        }
        cout << "   "; // Spacer
        // Right Side: Enemy Board (Shows my shots)
        cout << setw(2) << r << "|";
        for (int c = 0; c < size; ++c) {
            char mark = enemyBoard(r, c);
            if (mark == 0) mark = '.';
            cout << mark << " ";
        }
        cout << "\n";
    }
}
/**
 * @brief Helper to draw the placement interface with "ghost" ship.
 */
void HumanPlayer::drawPlacementView(Point cursor, bool vertical, Ship* s) const {
    // Clear screen
    cout << string(50, '\n');
    cout << "Place: " << s->getName() << " (Size " << s->getLength() << ", " <<
(vertical ? "Vertical" : "Horizontal") << ")\n";
    cout << "Controls: W/A/S/D move | C rotate | P place | Q quit\n";
    int size = getBoardSize();
    // Header
    cout << string(3, ' ');
    for (int c = 0; c < size; ++c) cout << (char)('A' + c) << " ";
    cout << "\n" << string(2, ' ') << string(size * 2, '-') << "\n";
    for (int r = 0; r < size; ++r) {
        cout << setw(2) << r << "|";
        for (int c = 0; c < size; ++c) {
            bool isGhost = false;
            // Render Ghost Ship at cursor
            for (int k = 0; k < s->getLength(); ++k) {
                int ghostR = cursor.row + (vertical ? k : 0);
                int ghostC = cursor.col + (vertical ? 0 : k);
                if (r == ghostR && c == ghostC) {
                    isGhost = true;
                    break;
                }
            }
            char sym = '.';
            int shipId = myBoard(r, c);
            if (isGhost) {
                sym = 'X';
            } else if (shipId > 0) {
                sym = fleet[shipId-1]->getSymbol();
            }
            cout << sym << " ";
        }
        cout << "\n";
    }
}
/**
 * @brief Interactive loop for user to place ships.
 * @return true if successful, false if user quits.
 */
bool HumanPlayer::placeShips() {
    Point cursor(0, 0);
    bool vertical = false;
    int shipId = 1;
    for (int i = 0; i < fleet.size(); i++) {
        Ship &s = *fleet[i];
        bool placed = false;
        while (!placed) {
            drawPlacementView(cursor, vertical, &s);
            cout << "Command: ";
            char cmd;
            cin >> cmd;
            cmd = toupper(cmd);
            int size = myBoard.getSize();
            switch (cmd) {
                case 'W': if (cursor.row > 0) cursor.row--; break;
                case 'S': if (cursor.row < size - 1) cursor.row++; break;
                case 'A': if (cursor.col > 0) cursor.col--; break;
                case 'D': if (cursor.col < size - 1) cursor.col++; break;
                case 'C': vertical = !vertical; break;
                case 'Q':
                    return false;
                case 'P':
                    if (canPlace(cursor, vertical, s.getLength())) {
                        applyShipToBoard(cursor, vertical, s.getLength(), shipId);
                        placed = true;
                        shipId++;
                    } else {
                        cout << "Invalid Position!\n";
                        cin.ignore(); cin.get();
                    }
                    break;
                }
            }
        }
    return true;
}
/**
 * @brief Prompts user for a target.
 * @param enemyShips Count of enemy ships for display.
 * @return Target point or Quit signal.
 */
Point HumanPlayer::makeMove(int enemyShips) {
    string input;
    int r, c, rowNum;
    char colChar;
    while (true) {
        cout << "Admiral, " << enemyShips << " ships are left. Choose your target 
(e.g. B3) or Q to quit: ";
        cin >> input;
        if (toupper(input[0]) == 'Q') {
            return Point::Quit();
        }
        if (sscanf(input.c_str(), "%c%d", &colChar, &rowNum) == 2) {
            colChar = toupper(colChar);
            // Check if column letter is valid
            if (colChar >= 'A' && colChar < 'A' + getBoardSize()) {
                c = colChar - 'A';
                r = rowNum;
                // Check if row number is valid
                if (r >= 0 && r < getBoardSize()) {
                    // Check if we already shot there
                    if (enemyBoard(r, c) == 0) {
                        return Point(r, c);
                    } else {
                        cout << "Already shot there.\n";
                        continue;
                    }
                }
            }
        }
        cout << "Invalid coordinate. Try again.\n";
    }
}
/**
 * @brief Constructor. Starts in Hunt mode.
 */
ComputerPlayer::ComputerPlayer() : Player("Computer") {
    huntMode = true;
}
/**
 * @brief Places ships randomly.
 * @return Always true.
 */
```

```
bool ComputerPlayer::placeShips() {
    placeShipsRandomly();
    return true;
}
/**
 * @brief Adds adjacent cells to the target stack after a hit.
 */
void ComputerPlayer::addAdjacentTargets(Point p) {
    potentialTargets.push_back(Point(p.row - 1, p.col));
    potentialTargets.push_back(Point(p.row + 1, p.col));
    potentialTargets.push_back(Point(p.row, p.col - 1));
    potentialTargets.push_back(Point(p.row, p.col + 1));
}
/**
 * @brief AI Logic: Switches between random Hunting and smart Targeting.
 */
Point ComputerPlayer::makeMove(int enemyShips) {
    Point target;
    bool valid = false;
    int size = getBoardSize();
    while (!valid) {
        if (potentialTargets.empty()) huntMode = true;
        if (huntMode) {
            // Fire randomly at the board
            target.row = rand() % size;
            target.col = rand() % size;
        } else {
            // Fire at adjacent spots found
            target = potentialTargets.back();
            potentialTargets.pop_back();
        }
        // Validate target
        if (target.row >= 0 && target.row < size && target.col >= 0 && target.col <
size) {
            if (enemyBoard(target.row, target.col) == 0) valid = true;
        }
    }
    return target;
}
```

Player.h

```
#ifndef PLAYER_H
#define PLAYER_H

#include "Grid.h"
#include "Ships.h"
#include "Point.h"
#include <vector>
#include <string>

/**
 * @class Player
 * @brief Base class for all players.
 * Handles the game boards, fleet management, and shot logic.
 */
class Player {

    // Allows the Game class to access protected members like the boards
    friend class Game;

    protected:
        std::string name;           ///< Player's name
        Grid<int> myBoard;          ///< Where player's ships are (0=empty, ID=ship)
        Grid<char> enemyBoard;      ///< Where player has shot ('X'=hit, 'O'=miss)
        Grid<char> incomingShots;   ///< Where player has been shot
        std::vector<Ship*> fleet;   ///< List of player ships
        int shipsRemaining;         ///< How many ships are still alive

        /**
         * @brief Writes ship ID numbers onto the board grid.
         */
        void applyShipToBoard(Point p, bool vertical, int length, int shipId);

        /**
         * @brief Checks if a ship fits on the board without overlapping.
         */
        bool canPlace(Point p, bool vertical, int length) const;

    public:

        Player(std::string n);
        virtual ~Player();

        /**
         * @brief Handles ship setup. Returns false if user quits.
         */
        virtual bool placeShips() = 0;

        /**
         * @brief Determines the next shot target.
         * @param enemyShips Used for the prompt text.
         */
        virtual Point makeMove(int enemyShips) = 0;

        /**
         * @brief Creates the fleet based on game mode (Standard vs Rapid).
         */
        void initFleet(int mode);

        /**
         * @brief Randomly places all ships (used by Computer).
         */
        void placeShipsRandomly();

        /**
         * @brief Processes a shot fired at this player.
         * @param sunkName If a ship sinks, this string gets updated with its name.
         * @return true if valid shot, false if out of bounds.
         */
        bool receiveShot(Point p, bool &wasHit, bool &sunk, std::string &sunkName);

        // Getters
        std::string getName() const { return name; }
        int getBoardSize() const { return myBoard.getSize(); }
        int getShipsRemaining() const { return shipsRemaining; }
        bool hasLost() const;

        /**
         * @brief displays both boards (My Board vs Enemy Board) side-by-side.
         */
        void printBoards() const;

        void setBoardSize(int s) {
            myBoard.resize(s);
            enemyBoard.resize(s);
            incomingShots.resize(s);
        }

        // Marks the result of player's shot on the tracking board
        void markShot(Point p, bool hit) {
            enemyBoard(p.row, p.col) = (hit ? 'X' : 'O');
        }
};
```

```
/**
 * @class HumanPlayer
 * @brief Handles user input for ship placement and targeting.
 */
class HumanPlayer : public Player {
   private:
      // Visual interface for the "WASD" placement mode
      void drawPlacementView(Point cursor, bool vertical, Ship* s) const;

   public:
      HumanPlayer(std::string n) : Player(n) {}
      virtual ~HumanPlayer() {}

      bool placeShips() override;
      Point makeMove(int enemyShips) override;
};

/**
 * @class ComputerPlayer
 * @brief AI opponent with a simple Hunt/Target strategy.
 */
class ComputerPlayer : public Player {
   private:
      std::vector<Point> potentialTargets;    ///< Stack of adjacent cells to try after a hit
      bool huntMode;                          ///< True = Searching randomly, False = Targeting specific area
      Point lastHit;

   public:
      ComputerPlayer();
      virtual ~ComputerPlayer() {}

      bool placeShips() override;
      Point makeMove(int enemyShips) override;

      // Pushes Up/Down/Left/Right coordinates to the target stack
      void addAdjacentTargets(Point p);
};

#endif
```

Point.h

```
#ifndef POINT_H
#define POINT_H
#include <iostream>
/**
 * @class Point
 * @brief Represents a (row, column) coordinate on the game board.
 */
class Point {
    public:
        int row; ///< Row index (0-9)
        int col; ///< Column index (0-9)
        /**
         * @brief Constructor. Defaults to (0,0).
         */
        Point(int r = 0, int c = 0) : row(r), col(c) {}
        /**
         * @brief Returns a special coordinate (-1, -1) signaling a Quit action.
         */
        static Point Quit() { return Point(-1, -1); }
        /**
         * @brief Checks if this point represents a Quit signal.
         */
        bool isQuit() const { return row == -1 && col == -1; }
        /**
         * @brief Adds two points together (used for relative movement).
         */
        Point operator+(const Point& right) const {
            return Point(row + right.row, col + right.col);
        }
        /**
         * @brief Equality check.
         */
        bool operator==(const Point& right) const {
            return (row == right.row && col == right.col);
        }
        /**
         * @brief Inequality check.
         */
        bool operator!=(const Point& right) const {
            return !(*this == right);
        }
        /**
         * @brief Stream insertion operator.
         * Prints coordinate in "A1" format (Column Letter + Row Number).
         */
        friend std::ostream& operator<<(std::ostream& out, const Point& p) {
            out << (char)('A' + p.col) << p.row;
            return out;
        }
};
#endif
```

Ships.h

```
#ifndef SHIPS_H
#define SHIPS_H
#include <string>
#include <iostream>
/**
 * @class Ship
 * @brief Abstract base class for all ship types.
 * Tracks name, length, and damage.
 */
class Ship {
    protected:
        std::string name;
        int length;
        int damage;
    public:
        /**
         * @brief Constructor.
         * @param n Name of ship.
         * @param l Length of ship.
         */
        Ship(std::string n, int l) : name(n), length(l), damage(0) {};
        // Destructor
        virtual ~Ship() {}
        // Getters
        std::string getName() const { return name; }
        int getLength() const { return length; }
        int getDamage() const { return damage; }
        /**
         * @brief Returns true if damage equals length.
         */
        bool isSunk() const { return damage >= length; }
        /**
         * @brief Returns the char symbol for the board.
         */
        virtual char getSymbol() const = 0;
        /**
         * @brief Increments damage when hit.
         */
        virtual void takeHit() {
            damage++;
        }
};
/**
 * @class Carrier
 * @brief Size 5 Ship. Symbol: 'A'
 */
```

```
class Carrier : public Ship {
    public:
        Carrier() : Ship("Aircraft Carrier", 5) {}
        char getSymbol() const override { return 'A'; }
};
/**
 * @class Battleship
 * @brief Size 4 Ship. Symbol: 'B'
 */
class Battleship : public Ship {
    public:
        Battleship() : Ship("Battleship", 4) {}
        char getSymbol() const override { return 'B'; }
};
/**
 * @class Cruiser
 * @brief Size 3 Ship. Symbol: 'R'
 */
class Cruiser : public Ship {
    public:
        Cruiser() : Ship("Cruiser", 3) {}
        char getSymbol() const override { return 'R'; }
};
/**
 * @class Submarine
 * @brief Size 3 Ship. Symbol: 'S'
 */
class Submarine : public Ship {
    public:
        Submarine() : Ship("Submarine", 3) {}
        char getSymbol() const override { return 'S'; }
};
/**
 * @class Destroyer
 * @brief Size 2 Ship. Symbol: 'D'
 */
class Destroyer : public Ship {
    public:
        Destroyer() : Ship("Destroyer", 2) {}
        char getSymbol() const override { return 'D'; }
};
#endif
```