CS112 Project Report: BattleShip

Submitted by:

Mohammad Hassaan	2024302
Hamza Ali	2024208
Raja Hamza Sikandar	2024532

Contents

1 Introduction

Our group recreated the Battleship board game in C++. It is a text-based strategy game that will be implemented with Object-Oriented Programming (OOP) concepts. It will emulate the traditional naval combat game with both Player vs Player and Player vs Computer modes on an 8x8 grid battlefield. This project will be used as an example of real-world implementation of all basic OOP concepts such as classes, inheritance, polymorphism, encapsulation, abstraction, and dynamic memory allocation (DMA).

2 Motivation

From the very start, our shared familiarity with Battleship's mechanics made it a natural choice—no time lost learning new rules meant we could dive straight into solid C++ design. When we sat down to pick a theme, the naval connection resonated on both a personal and aesthetic level: several teammates have family ties to the military, and we wanted that to shine through not just in ship names (all drawn from Pakistan Navy vessels) but in the look-and-feel of the opening gameplay screen, which we styled to evoke a command-center display. Choosing Battleship let us stay motivated by playing to our strengths—strategic grid logic and risk management—while building something truly our own.

3 Purpose and Objectives

Our goal was straightforward: recreate the classic Battleship experience as faithfully as possible while showcasing clean, maintainable C++ code. Rather than chasing arbitrary metrics, we held ourselves to a standard of "real-world fidelity"—ship placement, hit detection, and turn flow all had to mirror the tabletop version. At the same time, we sought to innovate: adding a quick "Demo Mode" that instantly plays out a full game, plus two distinct AI difficulty levels (random "Normal" and adjacency-targeting "Smart") to demonstrate flexible AI design. Throughout, we adopted test-first practices for core logic (placement rules, shot resolution) and kept our code modular so that adding features—like a future GUI or network play—would slot in without upheaval.

4 Problem Statement

Our Battleship software is a modular, console-based C++ application that solves the core challenges of input validation, state management, AI strategy, and extensibility by dividing responsibilities among five main components:

1. **Display Stability**: Early on, advancing turns would garble the console layout—fixing that required refactoring our rendering loop so each frame cleanly cleared and redrew only the necessary elements.

2. **AI Behavior**: Coding the "Smart" mode AI—so that once it scores a hit it systematically targets surrounding cells—took careful state tracking to avoid repeated shots and to handle corner and edge cases.

3. **Rule Enforcement**: We needed robust, class-level checks to prevent overlapping or off-board ship placements, and unit tests to catch any regressions.

4. **Time Management**: Completing this while juggling other quizzes and assignments—forced us to break the project into mini-sprints (grid logic first, then AI, then modes, then polish) and to communicate tightly through daily check-ins.

By solving these, we delivered a Battleship game that not only feels true to the original but also stands on a rock-solid, extensible codebase ready for future growth.

5 Proposed Solution

Our Battleship software is a modular, console-based C++ application that solves the core challenges of input validation, state management, AI strategy, and extensibility by dividing responsibilities among five main components:

5.1 Player Module

- Function: Encapsulates both human and CPU players.
- Solution: Each `Player` holds two `Board` objects—one for its own ships and one to track shots at the opponent—so all shot logic, scorekeeping, and turn actions (`attack()`) are localized and reusable.

5.2 Board Module

- Function: Manages an 8×8 grid of cells and ship placement.
- Solution: A 2D array plus a `std::vector<Ship>` lets `placeShip()` and `processShot()` enforce placement rules, update cell states (empty, ship, hit, miss), and detect when all ships are destroyed.

5.3 Game Module

- Function: Orchestrates the game loop, menu navigation, and mode selection.
- Solution: `Game::run()` drives the main menu (Play, Rules, Demo, Toggle AI), while `play()` handles turn order. Smart-mode logic (`cpuSmartTurn()`) and demo mode are cleanly integrated, ensuring CPU behavior andbeginitemize enditemize rapid walkthroughs slot into the same loop without code duplication.

5.4 UI Module

- Function: Abstracts all console I/O and rendering.
- Solution: Methods like `drawGameBoard()`, `displayMainMenu()`, and input helpers (`getPlayerTarget()`, `getShipDirection()`) centralize user interac-

tion, so core logic remains free of raw `cout`/`cin` calls and is easier to maintain or replace with a GUI later.

5.5 Main Entry Point

- Function: Bootstraps the application.
- Solution: `main()` seeds the RNG, instantiates `Game`, and calls `run()`, keeping startup trivial and decoupled from game logic.

6 OOP Concepts Used

6.1 Encapsulation

- All class data members (e.g., `Player::score`, `Board::grid`, `Ship::positions`) are private. Public getters/setters and methods (`getScore()`, `incrementScore()`, `processShot()`) control access, preventing invalid state changes.

6.2 Abstraction

- The `UI` class hides console-specific details behind methods like `clearScreen()` and `drawGameBoard()`. The `Game` class exposes a high-level API (`run()`, `play()`) that orchestrates gameplay without exposing its internal loops or data structures.

6.3 Composition

- `Player` → `Board`: Each `Player` object owns two `Board` instances.
- `Board` → `Ship`/`Position`: A `Board` holds multiple `Ship` objects, each of which contains a collection of `Position` objects. This "has-a" relationship models real-world Battleship entities directly in code.

6.4 Polymorphism (Interface Uniformity)

- Both human and CPU players share the same `Player` interface. The `Game` class invokes `attack()` on a generic `Player&`, allowing human or CPU turns to proceed via the same method calls. CPU modes (Normal vs. Smart) switch behavior internally, yet the `Game` loop remains unchanged.

6.5 Extensibility via Inheritance

- Although no subclass hierarchy was strictly required for the initial implementation, every core module (`Player`, `Board`, `UI`, `Game`) exposes virtual or public interfaces that can be extended. For example, one could derive a

`NetworkedPlayer` from `Player` or replace `UI` with a `GuiUI` subclass without altering game logic.

Together, these OOP principles ensure our Battleship implementation is robust, maintainable, and ready for future enhancements.

7 Flow Diagram/Architecture

Figure 1: Class Diagram showing the architecture and flow of the Battleship game

8 Screenshots

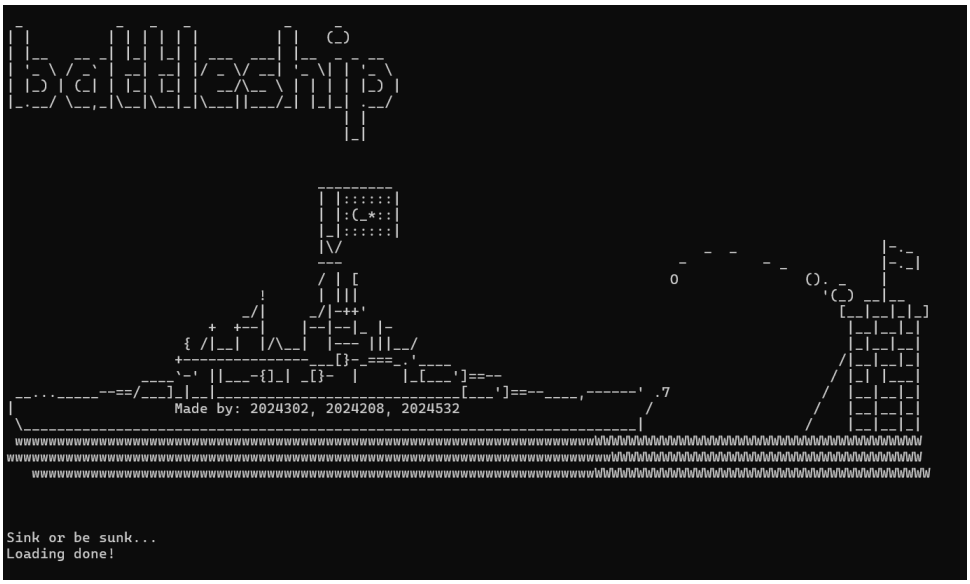

Figure 2: Welcome Splash Screen for the Battleship Game

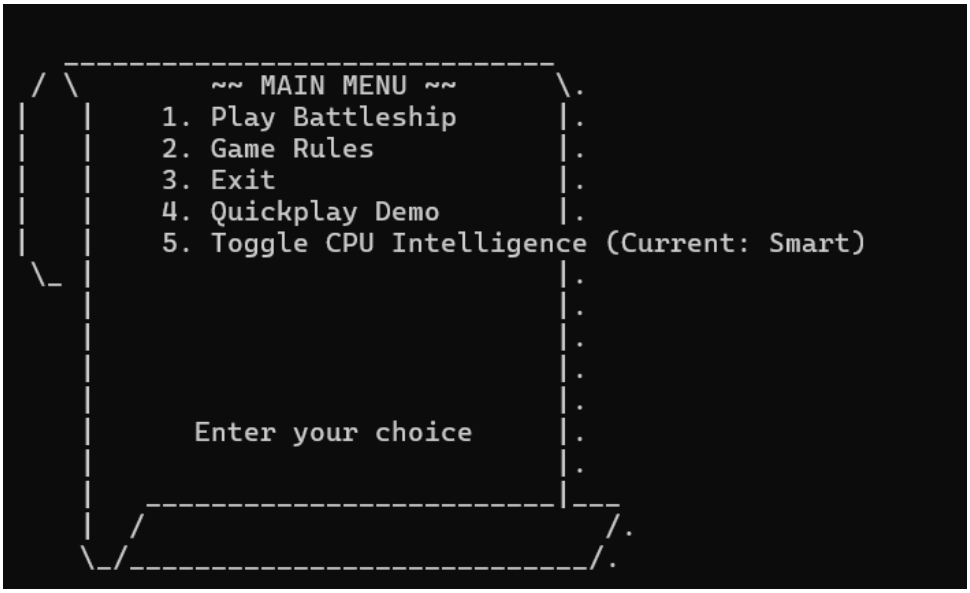

Figure 3: Main Menu of the game

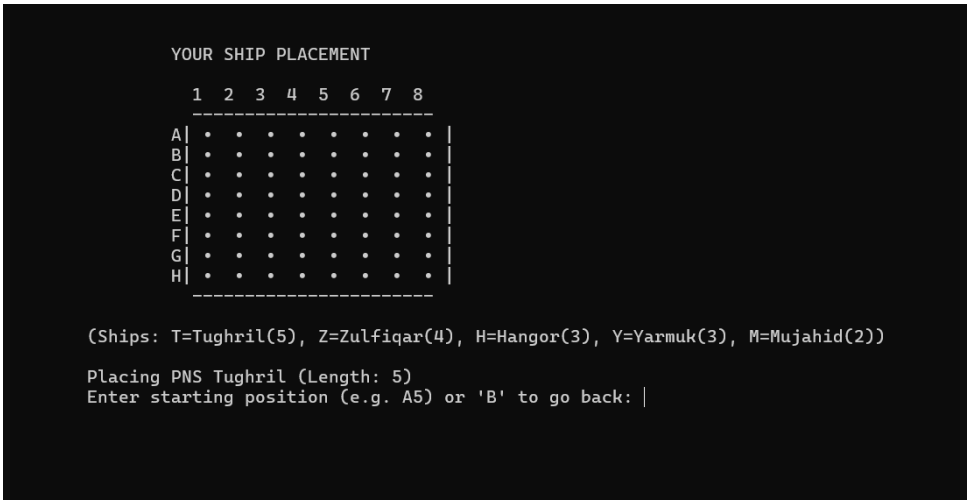

Figure 4: Manual placement of ships screen

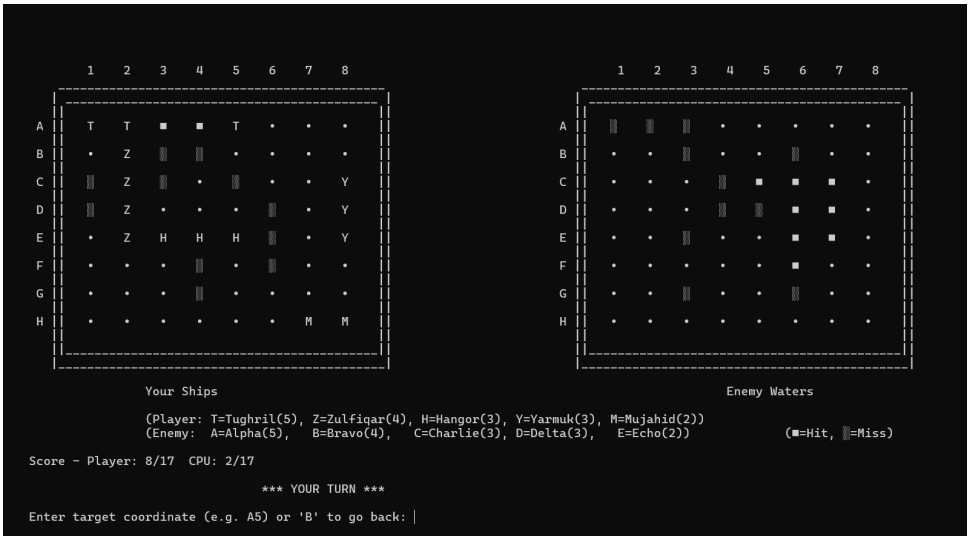

Figure 5: Gameplay Screen

9 Conclusion

Through the Battleship project, we not only delivered a faithful C++ recreation of a classic naval strategy game—with manual and random ship placement, two AI difficulty modes, and a demo walkthrough—but also honed essential software-engineering skills. By architecting a clean separation of concerns across Player, Board, Game, and UI modules, we practiced encapsulation, composition, and abstraction in a tangible context. Test-driven development guided us as we enforced placement rules and stable rendering, while implementing the smart-mode AI deepened our understanding of stateful algorithms and edge-case handling.

Completing this project in a tight window taught us effective sprint planning, clear team communication, and disciplined version control. In the end, we emerged with a robust, extensible codebase and renewed confidence in C++ object-oriented design—ready to evolve this foundation into GUI or networked versions, or to tackle even more complex systems in future courses and professional work.

10 Appendix

The source code for this analysis is available at: `https://github.com/hamziAli/Battleship`