Battleship Game Design Document

Github Repository link : https://github.com/Arshbir1/BattleShip_Project

Table of Contents

Team Details

Team Name: **Quantum Codex**

• Arshbir Singh Dang (IMT2023132)

• Nishant Khatri (IMT2023009)
• Siddharth Goswami (IMT2023542)
• Manav Jindal (IMT2023535)
• Sanyam Verma (IMT2023040)
• Mayank Tanwar (IMT2023012)

1. Introduction and Objectives

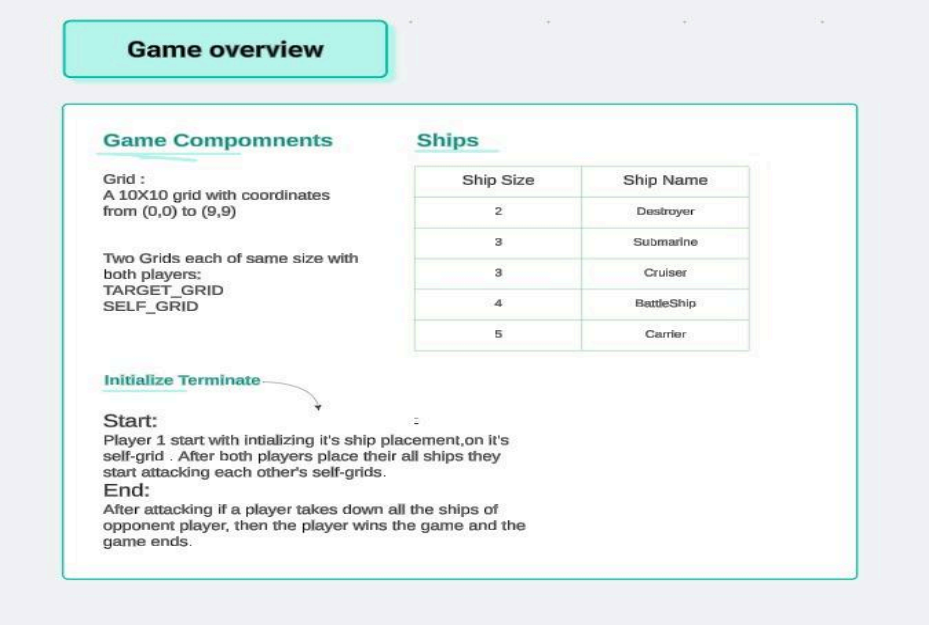
1.1 Project Overview

The **Battleship Game** is a multiplayer game designed for two players to engage in naval warfare on a 10x10 grid. The project is implemented using a **command-line interface (CLI)** for player interactions, **C++ for backend game logic**, and **Java for the middleware** that communicates between the client/server and backend via **JNI (Java Native Interface)**. The goal is to allow two players to play over a network, placing ships, attacking opponent ships, and detecting the outcome of each move.

1.2 Objectives

The primary objectives of this project are:

- To create a multiplayer version of the Battleship game that works in real-time.
- To use a **command-line interface (CLI)** to interact with the game, making it accessible through any terminal.
- To develop a **server-client architecture** where the server handles game logic and the client processes user input and outputs game results.
- To integrate the **C++ backend** with the **Java middleware** using **JNI** for seamless communication.



2. Technical Specifications

2.1 Backend (Game Logic)

The backend of the Battleship game is implemented in **C++**, which handles the core game logic. This includes:

- **Board Representation**: A 10x10 grid for each player to place ships.
- **Ship Placement**: Players can place ships on their boards by specifying **starting coordinates**, **orientation (1 for vertical, 0 for horizontal)**, and **ship type**. The types of ships are:
    - **Carrier** (size = 5)
    - **Battleship** (size = 4)
    - **Cruiser** (size = 3)
    - **Submarine** (size = 3)
    - **Destroyer** (size = 2) Example: `carrier 4 4 1` means placing a **carrier** (size = 5) at coordinates (4, 4) with (1) as a vertical orientation and (0) as horizontal orientation.
- **Attack Mechanism**: Players can attack by specifying only **starting coordinates** (e.g., `5 4` for attacking the cell at row 5, column 4). The backend determines whether the attack hits or misses and updates the game state accordingly.
- **Game Status**: The backend manages the game state, detects when a player wins, and sends the results.

2.2 Communication Protocol

The **middleware**, written in **Java**, is responsible for handling user input from the command line, forwarding actions to the backend via **JNI**, and receiving responses. The communication is done using **Java Native Interface (JNI)** to allow Java code to call C++

functions in the backend. The middleware ensures that player actions, such as placing ships and making attacks, are properly transmitted and results are displayed to the players.

Through JNI, the **Java server/client** interacts with the **C++ backend** to access the game logic, such as placing ships and attacking. The JNI interface serves as a bridge between the two languages, enabling seamless integration of C++ functionality into Java-based server/client operations.

3. Input/Output Requirements

3.1 Input (Command line to Backend)

The user interacts with the game through the command line by entering text-based commands:

- **Ship Placement**: For placing ships, the input follows the format:
  css
  Copy code
  `[ship_type] [starting_x] [starting_y] [orientation]`
- Where:
  - `ship_type` is the type of ship (carrier, battleship, cruiser, submarine, destroyer).
  - `starting_x` and `starting_y` are the coordinates on the grid (starting position).
  - `orientation` is 1 for vertical or 0 for horizontal. Example: `carrier 4 4 1` places a **carrier** (size = 5) at coordinates (4, 4) with a vertical orientation.
- **Attack**: For attacking, the input follows the format:
  css
  Copy code
  `[x] [y]`
- Where:
  - `[x]` and `[y]` are the coordinates of the cell to attack. Example: `5 4` would attack the cell at row 5, column 4.

3.2 Output (Backend to Command line)

The backend sends updates to the CLI:

- **"HIT"** or **"MISS"** for attack results.
- **"SUNK"** when a ship is completely destroyed.
- **"WIN"** or **"LOSE"** when the game ends. The results are displayed in the terminal interface for the player to see.

4. Development Setup

4.1 Game Logic(C++ Backend)

The **C++ backend** is responsible for running the game logic. To set up the server:

- Install a C++ development environment (e.g., GCC).
- Ensure the project files are properly configured for compiling the C++ code.
- The server listens for incoming client connections and processes game commands.
- **JNI Integration**: The server code includes JNI headers that allow the Java middleware to call C++ functions directly.

4.2 Server and Client Setup (Java Middleware)

The **Java middleware** acts as an intermediary between the command line interface and the C++ backend:

- Install Java Development Kit (JDK).
- The client handles input and output operations for the user interface.
- The client sends requests to the backend using JNI and receives game status updates.
- **JNI Integration**: Java calls backend functions via JNI to place ships, process attacks, and retrieve results.

4.3 Version Control and Repository Structure

The project uses **Git** for version control. The repository structure is organized as follows:

- **/bin**: Compiled executable files and libraries.
- **/src**: Source code for both C++ and Java implementations.
- **README.md:** Instructions for running the project files

Github Repository link : https://github.com/Arshbir1/BattleShip_Project

5. Architecture and Communication

5.1 Data Flow

1. The **client** sends player commands (e.g.,`"[ship_type] [starting_x] [starting_y] [orientation]"`, `“[attacking_x] [attacking_y]"`) to the **middleware**.
2. The middleware forwards the commands to the **backend** (C++ server) through **JNI**.
3. The backend processes the commands, updates the game state, and sends a response back to the middleware.

4. The middleware displays the game results (e.g., "HIT", "MISS") to the player via the CLI.

5.2 Advantages of the Architecture

- **Separation of Concerns**: The game logic is isolated from the user interface, making the code easier to maintain and extend.
- **Flexibility**: The server and client are loosely coupled, meaning they can be updated independently.
- **Scalability**: The architecture supports multiple clients interacting with a single backend server, making it scalable.
- **Efficient Communication**: JNI provides a seamless connection between the Java middleware and the C++ backend, ensuring quick data exchanges and efficient performance.

6. Workflow

6.1 Backend (Game Logic)

The backend handles the game's core logic, including:

- Processing player moves.
- Updating game status after each action.
- Detecting the winner and ending the game.

6.2 Command line (User Interface)

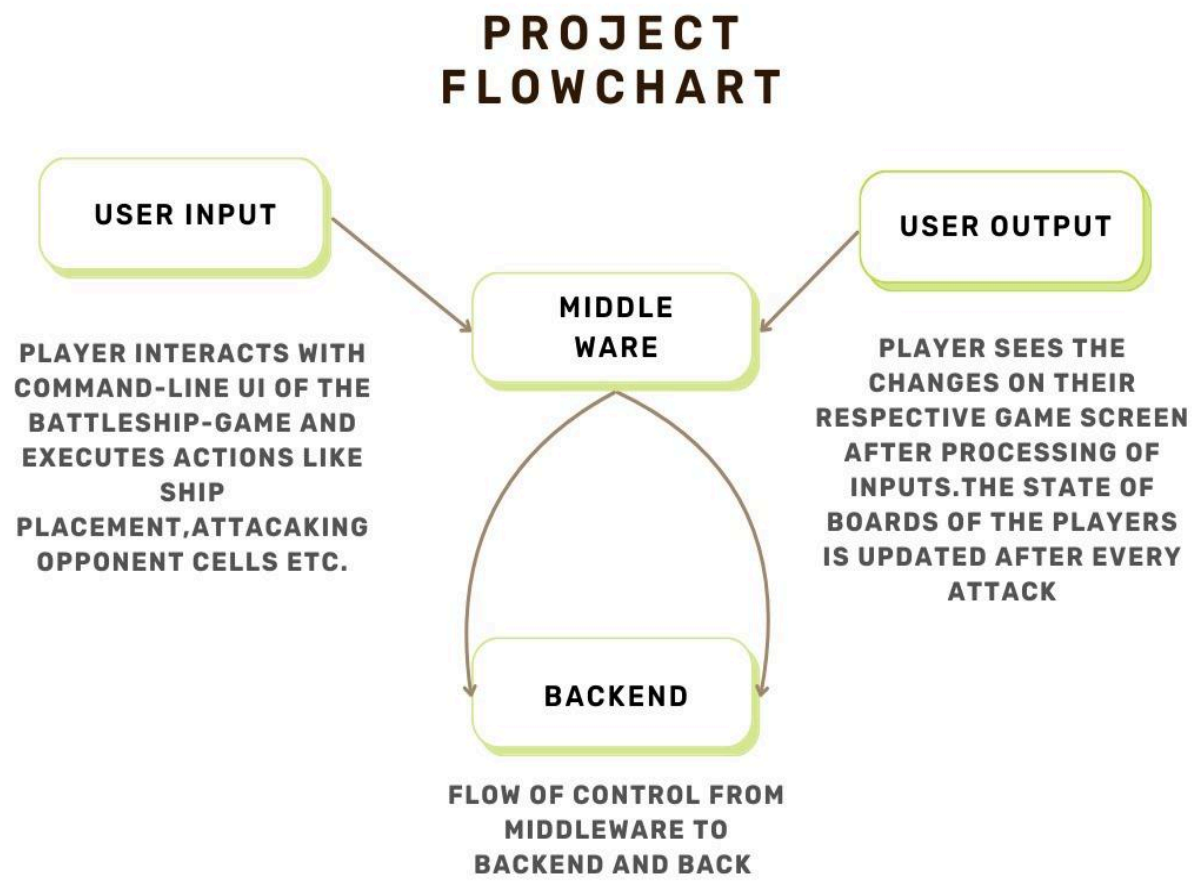
The CLI provides a simple interface for players to interact with the game:

- Accepts player input via text commands.
- Displays the game board and results to the user.
- Notifies players of game status changes (e.g., HIT, MISS, WIN).

6.3 Middleware (Communication Layer)

The middleware facilitates communication between the client (CLI) and the backend:

- Sends player commands to the backend via **JNI**.
- Receives game status updates and relays them to the player.



7. Project Structure and Important Files

7.1 Project Directory Structure

```
BATTLESHIP_PROJECT
├── .vscode
├── bin
│   ├── BattleshipClient.class
│   ├── BattleshipGame.dll
│   ├── BattleshipJNI.class
│   ├── BattleshipJNI.o
│   ├── BattleshipServer.class
│   ├── GameEngine.class
│   └── GameLogic.o
├── docs
│   └── README.txt
├── lib
│   ├── jni_md.h
│   └── jni.h
```

```
├── src
│   ├── cpp
│   │   ├── include
│   │   │   ├── BattleshipJNI.h
│   │   │   └── GameLogic.h
│   │   ├── BattleshipJNI.cpp
│   │   └── GameLogic.cpp
│   └── java
│       ├── BattleshipClient.java
│       ├── BattleshipJNI.java
│       ├── BattleshipServer.java
│       └── GameEngine.java
```

7.2 Key Files

- **BattleshipJNI.cpp**: Contains the C++ backend game logic and JNI implementations to interface with Java.
- **BattleshipJNI.java**: Java file that defines the JNI methods used to call the C++ backend functions.
- **BattleshipJNI.h**: Machine generated header file of **BattleshipJNI.java** according to which we **BattleshipJNI.cpp** was designed .
- **GameEngine.java**: Handles the client-side logic, receiving and displaying game updates.
- **BattleshipClient.java**: Main client program for handling user inputs and managing communication with the backend.
- **GameLogic.cpp**: Contains the core game logic functions such as placing ships, handling attacks, and updating the game state.
- **GameLogic.h**: Header file for **GameLogic.cpp**, defining the functions and game-related data structures (e.g., grid, ships).
- **BattleshipServer.java**: The main server program that manages the multiplayer game session, receiving player actions and invoking the backend via JNI to process them.
- **BattleshipGame.dll**: The dynamically linked library (DLL) containing the compiled C++ backend logic, used by the middleware for JNI integration.

8. Communication Protocol

8.1 Data Format

The data exchanged between the client and the server uses a **text-based protocol**, with commands like:

- **"SHIP_TYPE [starting_x] [starting_y] [orientation]"** // Placing Ships
- **"[attacking_x] [starting_y]"** // Attacking

8.2 Example Message Flow

1. **Client to Server**: `"[ship_type] [starting_x] [starting_y] [orientation]"`
2. **Server Response**: `"SHIP PLACED"`
3. **Client to Server**: `“[attacking_x] [attacking_y]"`
4. **Server Response**: `"MISS"`

9. Testing Strategy

9.1 Unit Testing

- Test each individual C++ function for placing ships, attacking, and game status.
- Test Java methods to ensure proper integration with JNI.

9.2 Integration Testing

- Test the full flow between the client (Java) and server (C++) using JNI.
- Ensure commands are correctly processed, and results are displayed properly.

9.3 User Acceptance Testing

- Run test sessions where two players can interact with the system and validate all game features (ship placement, attacking, win/loss detection).

10. Conclusion

This Battleship game project implements a multiplayer, networked version of the classic board game. By utilizing a command-line interface, C++ for backend logic, and Java for middleware communication, the project achieves efficient data flow and separation of concerns, ensuring scalability, maintainability, and an engaging user experience. With comprehensive testing and error handling, the system ensures a smooth gameplay experience.
