Chess Game Project Proposal Multiplayer using Client Server Client Architecture

Team Details

Team Name: The Start-up Squad

- **Archit Jaju** IMT2023128
- **Sarthak Raj** IMT2023569
- **Kunal Narang** IMT2023577
- **Pushkar Jitendra Kulkarni** IMT2023087
- **Harjot Singh** IMT2023064
- **Yash Khetan** IMT2023578

Github Repository - https://github.com/ayhm23/Chess_project.git

Table of Contents

1. Introduction and Objectives

1.1 Project Overview

This project implements a real-time, cross-platform multiplayer chess game using string-based communication. The frontend (JavaFX) is used for the user interface, while the backend (C++) acts as a server managing game logic and synchronizing the state. Communication between the frontend and backend is established through TCP sockets in a client-server-client architecture.

1.2 Objectives

1. **Cross-device functionality (Windows):** Use Java for frontend and C++ for backend to ensure a seamless experience across devices.
2. **Real-time communication:** Enable players to exchange moves efficiently using string-based protocols over TCP sockets.
3. **User-friendly interface:** Build a responsive graphical interface in JavaFX.

4. **Centralized architecture:** Manage game state and logic in a single backend server, ensuring consistency and reliability.

2. Technical Specifications

2.1 Frontend (User Interface)

- **Language/Framework:** Java with JavaFX for graphical components.
- **Files:** Five Java frontend files implementing the chess launcher, board rendering, and player interaction.
- **Socket Communication:** Java's `java.net.Socket` class facilitates TCP communication with the backend.

2.2 Backend (Game Logic)

- **Language:** C++
- **Single Backend File:** The C++ file manages server operations, game logic, and state synchronization.
- **Socket Communication:** C++ uses `std::string` and socket APIs for receiving and sending data to clients.

2.3 Communication Protocol

- **String-based Communication:** Strings are exchanged between backend and frontend for clarity and compatibility.
- **TCP Sockets:** Reliable communication is ensured using the TCP protocol for both data integrity and synchronization.

3. Input/Output Requirements

3.1 Input (Frontend to Backend)

- **Move Commands:** Serialized data representing the player's move.
  - **Example:** `{ "player" "e2" "e4" }`
- **Game Requests:** Actions like starting a game or requesting synchronization.
  - **Example:** `{ "action": "start_game", "player": "black" }`

3.2 Output (Backend to Frontend)

- **Game State Updates:** Serialized data about the updated board state.
  - **Example:** `{ "board": [...], "turn": "black", "status": "in_progress" }`
- **Error Handling:** Notifications for invalid moves or connection issues.
  - **Example:** `{ "error": "illegal_move", "details": "Invalid knight move" }`

4. Development Setup

4.1 Server Setup (C++ Backend)

- **Libraries:**
  - Standard C++ socket APIs.
  - Game logic implemented in a single backend file.

4.2 Client Setup (Java Frontend)

- **Libraries:** JavaFX for UI, and `java.net.Socket` for networking.
- **Frontend Files:** Five Java files include the chess launcher, main board, event handlers, and communication modules.

4.3 Version Control and Repository Structure

- **Repository Structure:**

```
plaintext
Copy code
chess-game-client-server/
├── server/                 # Backend (C++) code
├── java-client/            # Java frontend code
├── docs/                   # Documentation
└── README.md               # Project overview
```

5. Architecture and Communication

5.1 Data Flow

- **Frontend (Client):** Sends player actions (e.g., moves) to the server.
- **Backend (Server):** Validates moves, updates the game state, and broadcasts updates to both clients.

5.2 Connection Phase

- The server starts, awaiting connections.
- Two Java clients connect, establishing the game session.

5.3 Game Phase

- Clients send moves to the server.
- Server processes moves, updates game state, and notifies both clients.

5.4 Synchronization

- The server provides updated game state on request for disconnected clients.

5.5 Game Completion

- The server detects and notifies players of game-ending conditions (e.g., checkmate).

6. Workflow

6.1 Backend (Game Logic)

- Validates moves, updates game state, and detects game-ending conditions.

6.2 Frontend (User Interface)

- Renders the board, captures user moves, and communicates with the server.

6.3 Middleware (Communication Layer)

- Manages string message exchange using string-based protocols over TCP.

7. Project Structure and Important Files

7.1 Project Directory Structure

Refer to the updated structure under Section 4.3.

7.2 Key Files

- **backend.cpp:** C++ backend file implementing server logic and game rules.
- **Frontend Files:**
  - `ChessLauncher.java`: Client launcher for connecting to the server.
  - `ChessBoard.java`: Renders the chessboard and manages game interactions.
  - `PieceManager.java`: Handles user move input.
  - `ChessTile.java`: Manages tile placement.

8. Communication Protocol

8.1 Data Format

- Strings formatted as JSON-like messages for compatibility and readability.

8.2 Example Message Flow

- **Frontend to Backend:** `{ "player" "e2" "e4" }`
- **Backend to Frontend:** `{ "valid move", “invalid move” }`

9. Testing & Logging

9.1 Testing Strategy

- Unit testing for game logic and UI updates.
- Integration testing for socket-based communication.

9.2 Error Handling

- Graceful handling of invalid inputs and network issues.

9.3 Logging Mechanisms

- Backend logs key events (moves, disconnections).
- Frontend logs user actions and communication errors.

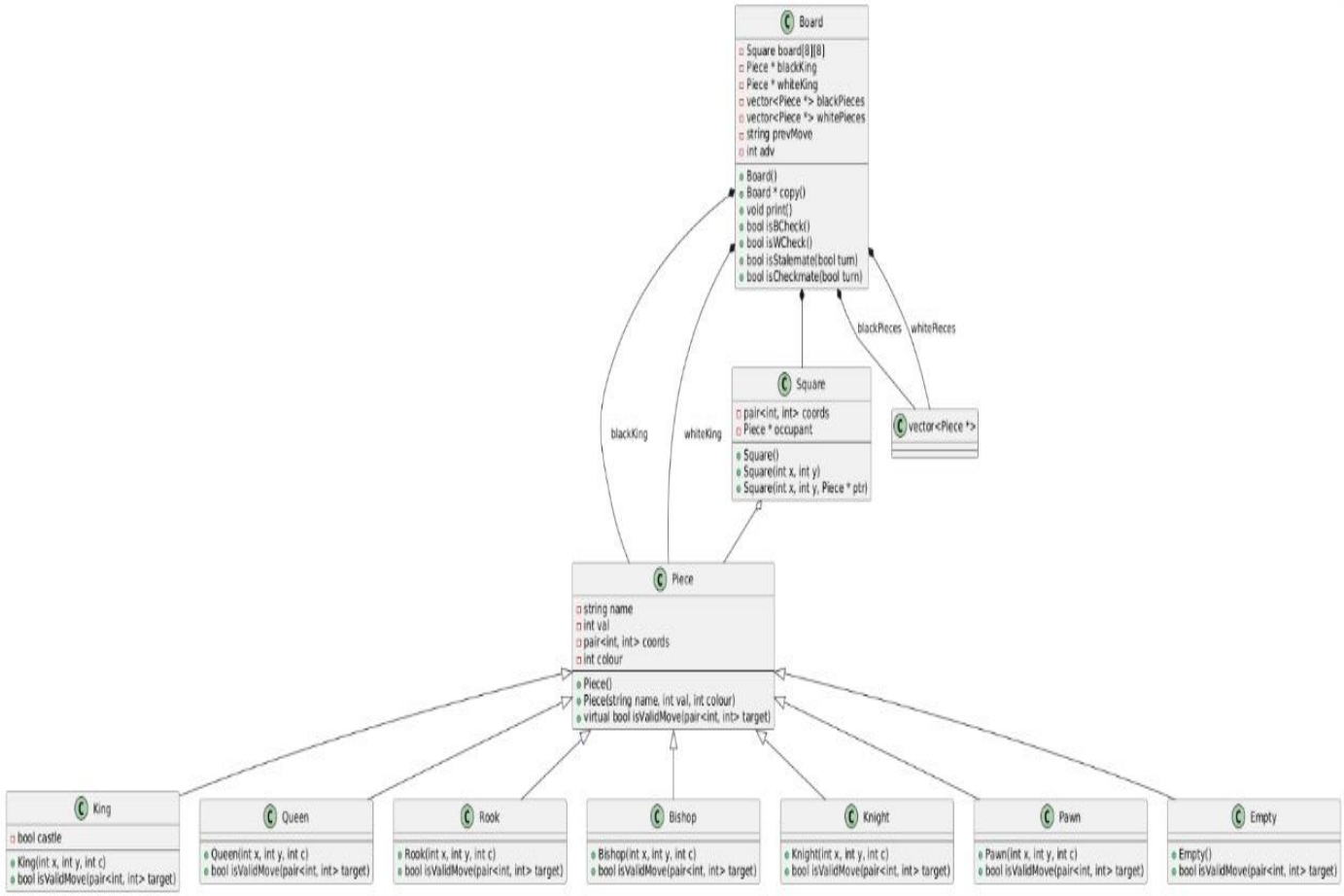
10. UML Diagrams

This section outlines the UML diagrams for various components of the chess game project. Each diagram will provide a detailed visualization of the structure and interactions within the system.

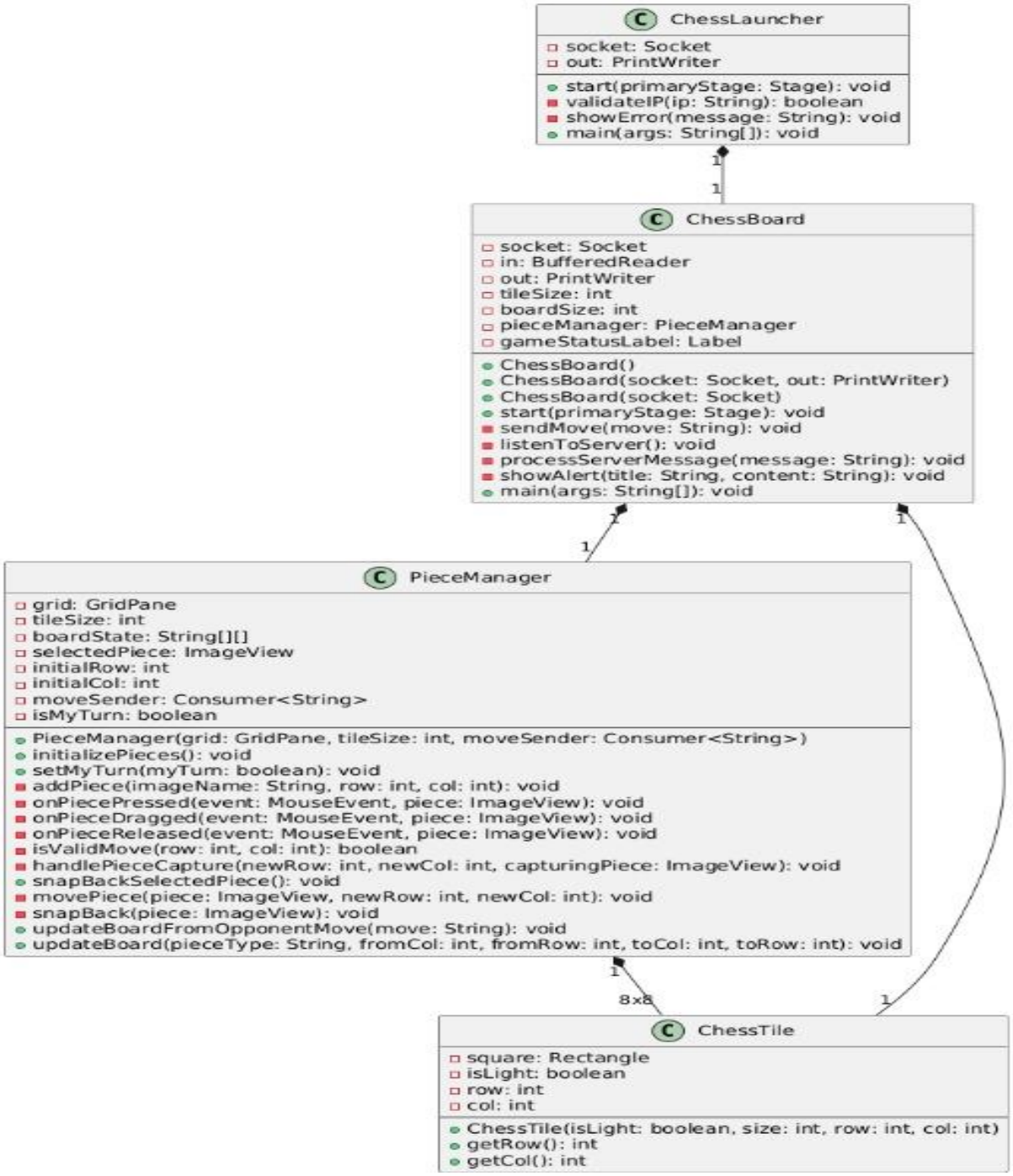
10.1 Class Diagram

The **class diagram** will illustrate the structure of the system, including classes, attributes, methods, and the relationships between the Java frontend classes (`ChessBoard.java`, `PieceManager.java`, `ChessTile.java`, etc.) and the backend (`backend.cpp`).

1. Backend


Board
- Square board[8][8]
- Piece * blackKing
- Piece * whiteKing
- vector<Piece *> blackPieces
- vector<Piece *> whitePieces
- string prevMove
- int adv
+ Board()
+ Board * copy()
+ void print()
+ bool isBCheck()
+ bool isWCheck()
+ bool isStalemate(bool turn)
+ bool isCheckmate(bool turn)
blackPieces
whitePieces
Square
- pair<int, int> coords
- Piece * occupant
+ Square()
+ Square(int x, int y)
+ Square(int x, int y, Piece * ptr)
vector<Piece *>
blackKing
whiteKing
Piece
- string name
- int val
- pair<int, int> coords
- int colour
+ Piece()
+ Piece(string name, int val, int colour)
+ virtual bool isValidMove(pair<int, int> target)
King
- bool castle
+ King(int x, int y, int c)
+ bool isValidMove(pair<int, int> target)
Queen
+ Queen(int x, int y, int c)
+ bool isValidMove(pair<int, int> target)
Rook
+ Rook(int x, int y, int c)
+ bool isValidMove(pair<int, int> target)
Bishop
+ Bishop(int x, int y, int c)
+ bool isValidMove(pair<int, int> target)
Knight
+ Knight(int x, int y, int c)
+ bool isValidMove(pair<int, int> target)
Pawn
+ Pawn(int x, int y, int c)
+ bool isValidMove(pair<int, int> target)
Empty
+ Empty()
+ bool isValidMove(pair<int, int> target)

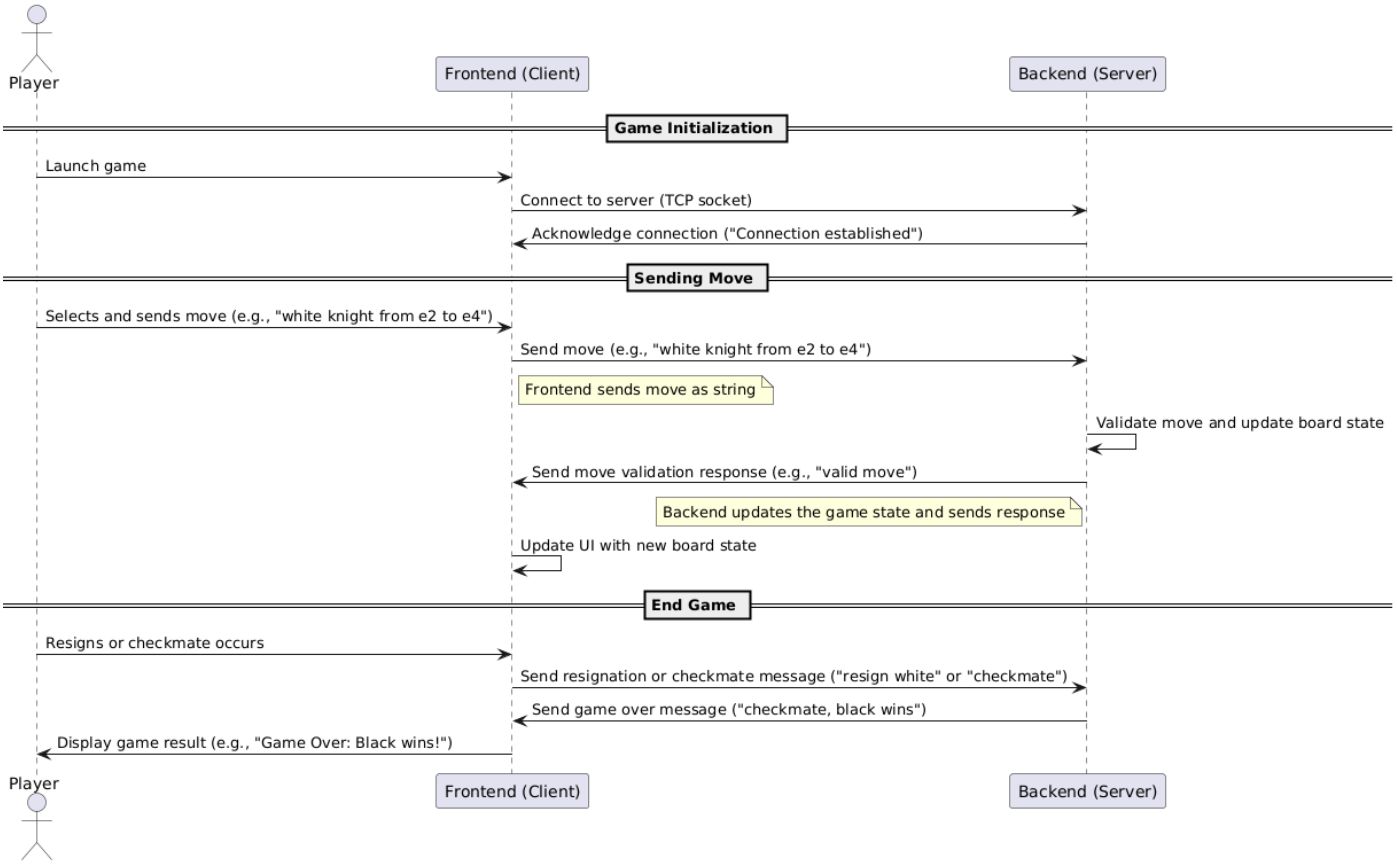
2. Frontend


ChessLauncher
socket: Socket
out: PrintWriter
start(primaryStage: Stage): void
validateIP(ip: String): boolean
showError(message: String): void
main(args: String[]): void
1
1
ChessBoard
socket: Socket
in: BufferedReader
out: PrintWriter
tileSize: int
boardSize: int
pieceManager: PieceManager
gameStatusLabel: Label
ChessBoard()
ChessBoard(socket: Socket, out: PrintWriter)
ChessBoard(socket: Socket)
start(primaryStage: Stage): void
sendMove(move: String): void
listenToServer(): void
processServerMessage(message: String): void
showAlert(title: String, content: String): void
main(args: String[]): void
1
1
1
1
PieceManager
grid: GridPane
tileSize: int
boardState: String[][]
selectedPiece: ImageView
initialRow: int
initialCol: int
moveSender: Consumer<String>
isMyTurn: boolean
PieceManager(grid: GridPane, tileSize: int, moveSender: Consumer<String>)
initializePieces(): void
setMyTurn(myTurn: boolean): void
addPiece(imageName: String, row: int, col: int): void
onPiecePressed(event: MouseEvent, piece: ImageView): void
onPieceDragged(event: MouseEvent, piece: ImageView): void
onPieceReleased(event: MouseEvent, piece: ImageView): void
isValidMove(row: int, col: int): boolean
handlePieceCapture(newRow: int, newCol: int, capturingPiece: ImageView): void
snapBackSelectedPiece(): void
movePiece(piece: ImageView, newRow: int, newCol: int): void
snapBack(piece: ImageView): void
updateBoardFromOpponentMove(move: String): void
updateBoard(pieceType: String, fromCol: int, fromRow: int, toCol: int, toRow: int): void
1
8x8
1
ChessTile
square: Rectangle
isLight: boolean
row: int
col: int
ChessTile(isLight: boolean, size: int, row: int, col: int)
getRow(): int
getCol(): int

10.2 Sequence Diagram

The **sequence diagram** will depict the interaction flow between the frontend and backend during various game phases, such as:

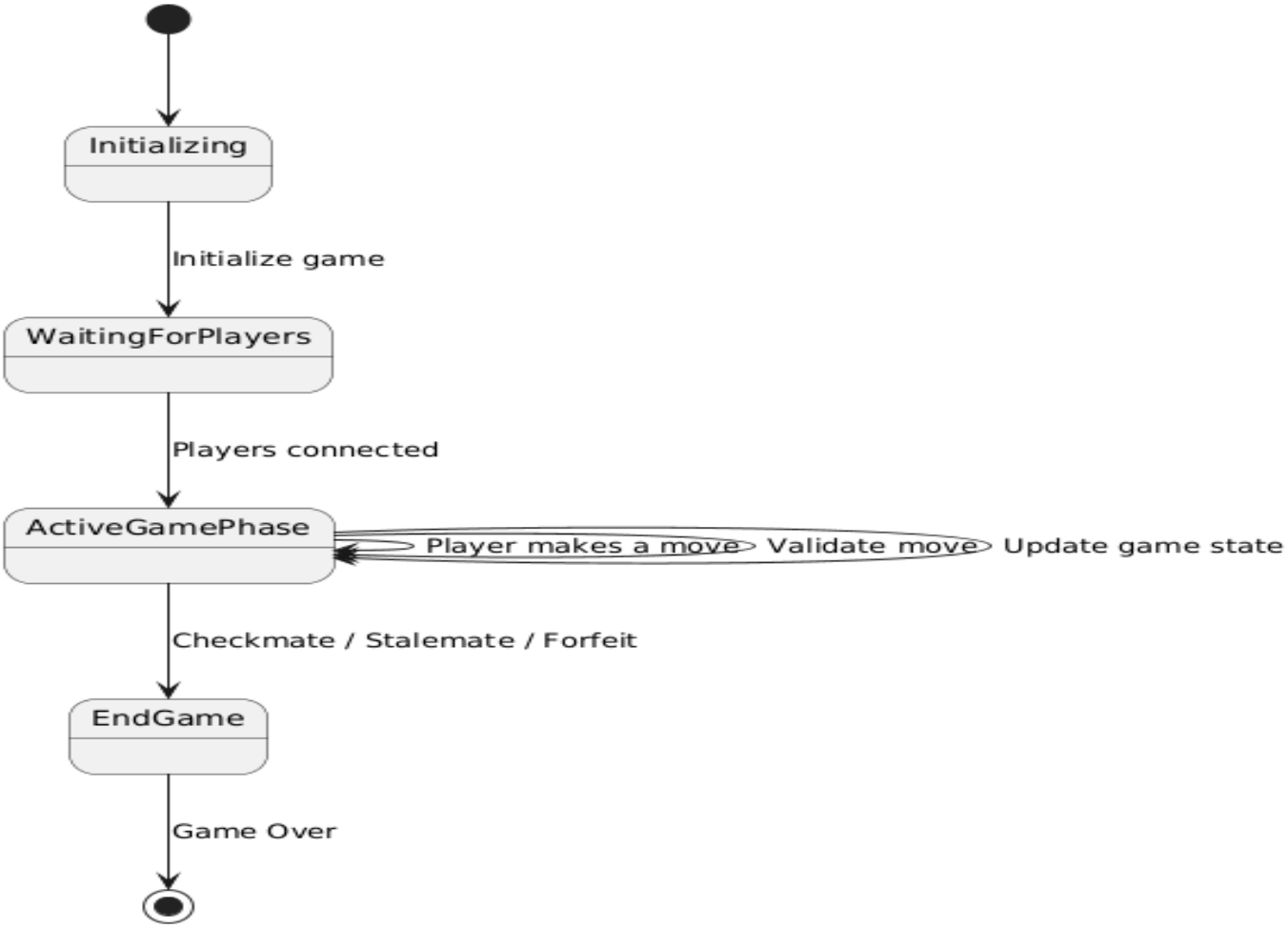
- Game initialization.
- Sending and receiving moves via sockets.
- Synchronization during gameplay.



10.3 State Diagram

The **state diagram** will represent the various states of the chess game and transitions between them, such as:

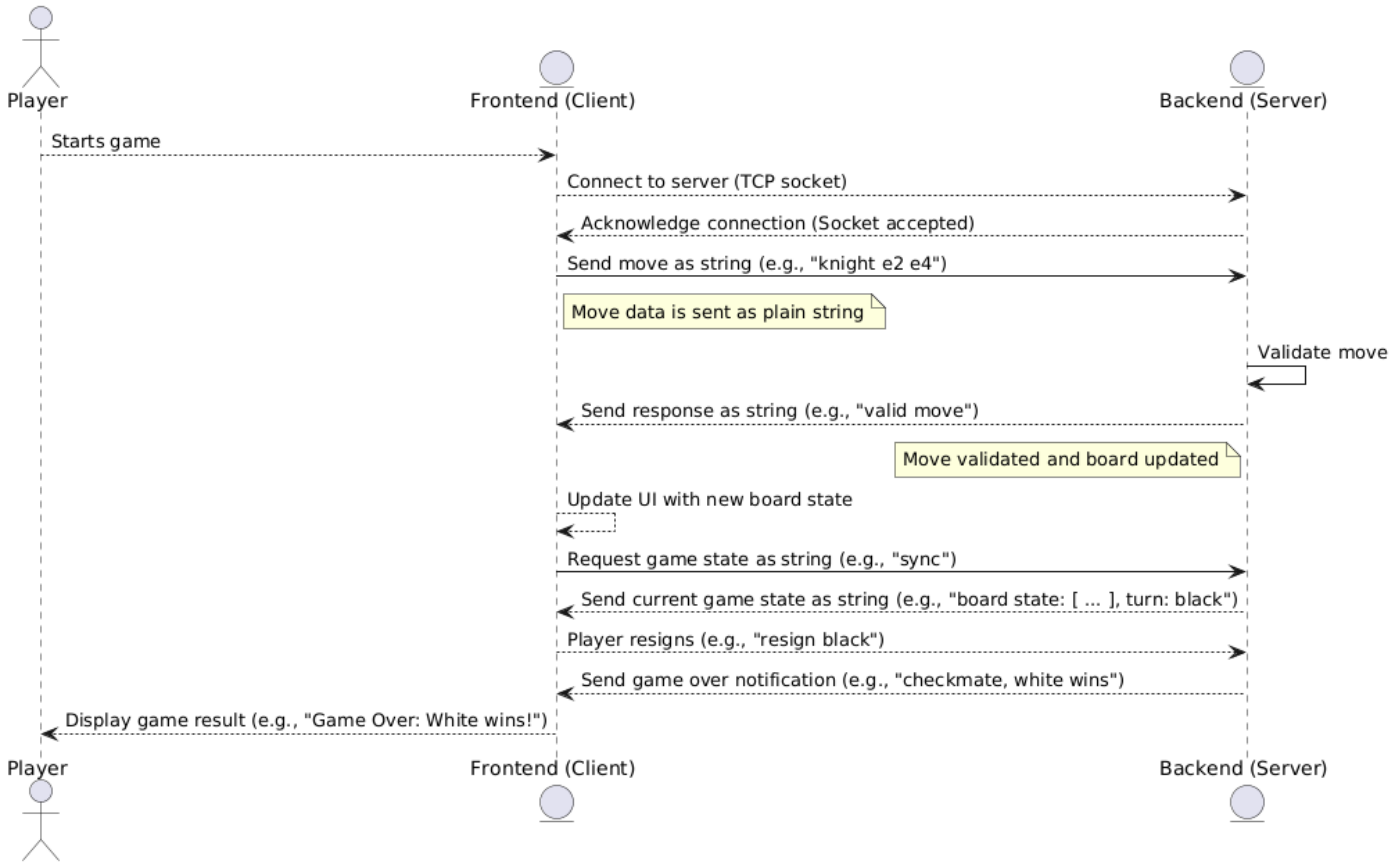
- Initializing.
- Waiting for players.
- Active game phase.
- End game (checkmate, stalemate, or forfeit).



10.4 Socket Communication Diagram

The **socket communication diagram** will show the data flow between the frontend (client) and backend (server), specifically focusing on:

- Connection establishment.
- Message serialization and deserialization.
- Real-time move exchange and state updates.



11. Conclusion

This updated design document reflects the implementation changes, emphasizing a simplified structure with five Java frontend files using JavaFX and a single backend file for C++. The client-server-client architecture ensures consistency, reliability, and a smooth multiplayer experience.