Project Plan: NexusBoard

1 Project Vision

To develop a robust, server-authoritative framework in Java for hosting two-player, turn-based grid games, enabling networked gameplay. The initial implementation will support Chess, with a modular design to accommodate future games like Checkers, Tic-Tac-Toe, or Go.

2 Core Concept

NexusBoard is a networked application using a client-server architecture. The server acts as the central authority, managing game state, validating moves, and enforcing rules, while the client provides a graphical interface built with Java Swing for players to interact with the game. This separation ensures consistency, prevents cheating, and keeps the client lightweight. The framework's modularity allows easy integration of additional grid-based games.

3 Key Technologies

- **Language:** Java – Chosen for platform independence, robust networking, and mature GUI libraries.
- **Networking:** Java Sockets API (`ServerSocket`, `Socket`) – Ensures reliable TCP/IP communication.
- **Data Transfer:** Java Object Serialization (`ObjectInputStream`, `ObjectOutputStream`) – Facilitates sending complex game objects.
- **Concurrency:** Java Threads – Handles multiple client connections and game sessions.
- **User Interface:** Java Swing – Provides a lightweight, customizable GUI for rendering the game board and capturing user inputs.

4 Architectural Overview

4.1 Server (The Referee)

- **Authoritative Game State:** Maintains the definitive `GameState`, ensuring all clients display consistent data.
- **Rule Enforcement:** Executes all game logic and move validation.
- **Matchmaking:** Pairs two clients into a game session; initially supports one game, with potential for multiple concurrent sessions.
- **State Broadcaster:** Sends updated `GameState` to both clients after each valid move.

4.2 Client (The Player Interface)

- **Dumb Terminal:** Contains no game logic; renders the state received from the server using Swing components.
- **Input Handler:** Captures user actions via mouse clicks on a Swing-based board.
- **Move Transmitter:** Sends `Move` objects to the server for validation.

5 Communication Protocol

The server and clients communicate by serializing the following Java objects:

- `GameState`: Represents the current game state.
  - `Piece[][] board`: 2D array for the game board (8x8 for Chess).
  - `PlayerColor currentPlayerTurn`: Enum (`WHITE`, `BLACK`).
  - `GameStatus status`: Enum (`IN_PROGRESS`, `CHECK`, `CHECKMATE`, `STALEMATE`).
- `Move`: Represents a player's action.
  - `BoardPosition from`: Starting coordinates (row, column).
  - `BoardPosition to`: Destination coordinates.
  - `String specialMove`: Optional field for Chess-specific moves (e.g., "castle", "promotion").
- `ServerResponse`: Handles general communication.
  - `ResponseType type`: Enum (`WAITING_FOR_OPPONENT`, `GAME_START`, `INVALID_MOVE`, `GAME_OVER`).
  - `String message`: Human-readable text for the client UI.

6 Development Roadmap & Milestones

6.1 Milestone 1: The Core Game Engine (Offline)

- **Objective:** Build a functional Chess game playable in the console without networking.
- **Tasks:**
  1. Define `GameEngine` interface with methods: `getInitialState()`, `validateMove(Move)`, `updateState(Move)`, `checkGameOver()`.
  2. Implement `ChessEngine` class to handle Chess-specific logic.
  3. Create `Piece` class with attributes: `type` (e.g., pawn, rook), `color` (white/black).
  4. Develop `Board` class for an 8x8 grid, managing piece positions.
  5. Build `GameState` class to track board, turn, and status.
  6. Implement move validation for all Chess pieces, including special rules (castling, en passant, pawn promotion).
  7. Create a console-based UI: display the board using ASCII, prompt moves (e.g., "e2 to e4"), and show status messages.
  8. Test scenarios: valid/invalid moves, check, checkmate, stalemate.
- **Estimated Duration:** 2 weeks

6.2 Milestone 2: The Network Foundation

- **Objective:** Establish reliable server-client communication.
- **Tasks:**
  1. Implement `Server` class with `ServerSocket` on a configurable port (e.g., 8080).
  2. Create `ClientHandler` class extending `Thread` for each client connection.
  3. Set up `ObjectInputStream` and `ObjectOutputStream` for serialized communication.
  4. Develop `Client` class to connect via `Socket`, with configurable IP and port.
  5. Implement matchmaking to pair two clients into a session, assigning `WHITE` and `BLACK`.
  6. Test by connecting two clients to a localhost server, verifying message exchange.
- **Estimated Duration:** 1.5 weeks

6.3 Milestone 3: Server-Client Integration

- **Objective:** Integrate the game engine with networking and render the initial board using Swing.
- **Tasks:**
  1. Integrate `ChessEngine` into the server; initialize `GameState` for paired clients.
  2. Ensure `GameState`, `Piece`, and related classes implement `Serializable`.
  3. Develop a Swing-based client UI using `JPanel` with a `GridLayout` (8x8 for Chess).
  4. Load and display chess piece images (e.g., PNGs) using `ImageIcon` on `JLabel` components.
  5. Send initial `GameState` from server to clients upon game start.
  6. Implement client-side rendering to display the starting Chess position.
  7. Test by launching server and two clients, ensuring both display the initial board correctly.
- **Estimated Duration:** 2 weeks

6.4 Milestone 4: The Interactive Gameplay Loop

- **Objective:** Enable real-time, turn-based gameplay over the network.
- **Tasks:**
  1. Add `MouseListener` to Swing `JPanel` to detect piece selection and movement.
  2. Construct `Move` objects from user clicks (e.g., clicking source and destination squares).
  3. Send `Move` objects from client to server via `ObjectOutputStream`.
  4. On the server, validate `Move` objects, checking turn order and legality using `ChessEngine`.
  5. Broadcast updated `GameState` to both clients if valid; send `ServerResponse` with error message if invalid.
  6. Implement client-side thread to listen for server updates and refresh the Swing UI.
  7. Add turn indicators in the UI (e.g., `JLabel` showing "Your Turn" or highlighting the active player's board).
  8. Test a full game cycle: move pieces, handle invalid moves, and synchronize both clients.
- **Estimated Duration:** 2.5 weeks

6.5 Milestone 5: Final Polish and Features

- **Objective:** Enhance usability and complete the application.
- **Tasks:**
  1. Implement end-game detection in `ChessEngine` (checkmate, stalemate, resignation).
  2. Send `ServerResponse` with game outcome; display result on client UI using `JOptionPane`.
  3. Enhance UI with status messages (e.g., `JLabel` for "Check!" or "Opponent's Turn") and visual cues (e.g., border highlight on selected piece).
  4. Add a "Play Again" `JButton` to reset the game with the same opponent.
  5. Handle disconnections: detect client disconnection, notify the remaining player via `JOptionPane`, and end the session.
  6. Test edge cases: mid-game disconnections, rapid move submissions, and UI responsiveness.
- **Estimated Duration:** 2 weeks

7 Definition of Success

The project is successful when two users on separate machines can:

- Launch the Swing-based client and connect to the server.
- Play a complete Chess game with all rules enforced (e.g., legal moves, checkmate).
- Receive clear visual feedback on game status and outcome via the Swing UI.

8 Future Considerations

- **Scalability:** Support multiple concurrent games using a `ThreadPoolExecutor`.
- **Game Variety:** Add new games by implementing additional `GameEngine` classes.
- **UI Enhancements:** Introduce a Swing-based lobby for game selection and opponent matching.
- **Security:** Add basic authentication to prevent unauthorized connections.
- **Accessibility:** Ensure Swing UI supports keyboard navigation for accessibility.