FATİH SULTAN MEHMET VAKIF ÜNİVERSİTESİ

Students:

Name Surname: MHD ALHABEB ALSHALAH
ID Number: 2221251360
Department: Software Engineering

Project:

Topic: Networked Chess Game
Github Link:
https://github.com/Habeeb-sh/Networked_Chess_Game

Course:

Name: Computer Networks and Design
Instructor: Dr. Samet Kaya

1) Chess Game Logic:

Chessboard Setup:

It's an 8x8 board. Each player has a white square on their bottom-right. Back row: Rooks (corners), Knights, Bishops, Queen (on her color), King. Front row: 8 pawns.

Piece Movement:

Each piece moves differently. Knights can jump over others. No piece can land on a friendly piece. Capturing means landing on an opponent's square. White always goes first.

Special Moves

1) Pawn Promotion: When a pawn reaches the opposite end, it can become a Queen, Rook, Bishop, or Knight (usually a Queen). Doesn't require a captured piece.

2) En Passant: If a pawn moves 2 squares forward and lands beside an enemy pawn, the enemy pawn can capture it as if it moved 1 square. Must be done immediately after.

3) Castling: A move involving the King and Rook. The King moves 2 squares toward the Rook, and the Rook jumps over beside the King.
Conditions:

- Neither the King nor the Rook has moved.
- No pieces between them.
- The King is not in check or moving through check.

Game Ending:

Checkmate: You win by putting the enemy King in check with no escape.
Draw (Stalemate): Happens when a player has no legal moves and is not in check, or other draw rules (e.g., repetition, no material).

2) Chess Game Flow:

The networked chess game follows a simple turn-based loop:

Lobby & Pairing:

When you launch the client, you're presented with a little "home" window where you enter your name and hit the "Connect" button.
The client (`CClient`) class opens a Socket to the server and sends a `JOIN_GAME#YourName` message.
The server (`Server`) class collects joining clients in `GameRooms` class; as soon as two players enter the same room, it sends each a `GAME_START#Color,OpponentName` message.

Board Initialization

On the client side, once `GAME_START` arrives, we spin up a `NetworkGame` class window.
Internally `NetworkGame` class creates a `GameController` class instance, which does:

```
board = new ChessBoard();
board.initializeBoard();  // places Rooks, Knights, Bishops, Queen, King, Pawns
currentTurn = "W";         // White always starts
```

The 8×8 grid of `JButtons` is drawn and populated with Unicode symbols (e.g. ♖, ♘, etc.).

Turn-Based Play

Local Move

You click once to select a square, then click again on a valid target.
Under the hood, `NetworkGame.handleClick(...)` function calls:

```
if (controller.movePiece(from, to)) {
  client.sendMove(from, to);}
```

The `GameController.movePiece(...)` function does:

1. Rule-check (via each piece's `isValidMove`)
2. "Would move cause check?" simulation
3. Special logic (castling, en passant, promotion)
4. Board update (`ChessBoard.placePiece(...), removePiece(...)`)
5. View update (`view.updateBoard(), view.updateTurn(...)`)
6. Check for checkmate/stalemate, pop up dialogs if game over
7. Switch turn (`currentTurn = (currentTurn.equals("W") ? "B" : "W")`)

Remote Move

Your opponent's move arrives as a `MOVE#from,to` message in your `CClient.run()` function's loop, parsed in `MsgParser` function, which calls the function `gameFrame.receiveMove(from, to);` where it applies it locally via the same function `controller.movePiece(...)` and updates the UI.

Game End & Cleanup

If someone disconnects mid-game, the server informs the other with `DISCONNECT#Name`. The client shows a "Opponent Disconnected" dialog, then returns to the lobby.
And if a checkmate or draw occurs, both clients get a `GAME_OVER#winner` message, pop up the result, and go back to the home screen.

3) Connection Handling

A brief breakdown of how sockets and threads are used on both sides to set up, manage, and tear down the network link.

Server-Side Connection Handling:

Listening for new clients:

ServerSocket: The Server constructor creates a listening socket on port 6000:

```
public Server(int port) throws IOException {
    this.ssocket = new ServerSocket(port);
    this.clients = new ArrayList<>();
    this.gameRooms = new ArrayList<>();
    this.clientId = 0;
}
```

StartAcceptance & run(): Calling `StartAcceptance()` spins up the server thread, which loops on `accept()`:

```
public void StartAcceptance() throws IOException {
    this.start();   // Thread.start() → invokes run()
}

@Override
public void run() {
    System.out.println("Chess server started on port 6000");
    while (!ssocket.isClosed() && isRunning) {
```

```
        Socket csocket = ssocket.accept();             // blocking: wait for client
        SClient newClient = new SClient(csocket, this); // wrap in handler
        newClient.StartListening();                     // spawn its thread
        clients.add(newClient);
        System.out.println("New client connected: " + newClient.id);
    }
}
```

Per-Client Thread (SClient):

Handshake & ID assignment: the SClient constructor takes:

```
this.id = server.clientId++;
this.coutput = csocket.getOutputStream();
this.cinput  = csocket.getInputStream();
```

Receiving messages: `StartListening()` function simply does `this.start()`, kicking off the `run()` function:

```
public void run() {
    while (!csocket.isClosed() && isConnected) {
        int bsize = cinput.read();             // read 1-byte length
        byte[] buf = new byte[bsize];
        cinput.read(buf);                       // read payload
        String msg = new String(buf);
        MsgParser(msg);                         // route JOIN_GAME, MOVE, etc.
    }
}
```

Shutdown: When the admin types “quit,” `Server.shutdown()` function gets called broadcasting a `DISCONNECT` message to each SClient, which closes their sockets, and closes the ServerSocket.

Client-Side Connection Handling:

Establishing the link (CClient):

```
public CClient(String ip, int port, String playerName) throws IOException {
    this.csocket    = new Socket(ip, port);        // connect to server
    this.coutput    = csocket.getOutputStream();
    this.cinput     = csocket.getInputStream();
    this.playerName = playerName;
    this.isConnected = true;
}
```

Starting the listener & joining a game:

The function `Listen()` starts listening for server messages and sends initial join request

```
public void Listen() throws IOException {
    this.start();       // spawn the background thread → run()
    joinGame();         // immediately ask to join with "JOIN_GAME#Name"
}
```

And the `run()` loop: continuously listens for server messages in separate thread to avoid blocking UI

```
public void run() {
    while (!csocket.isClosed() && isConnected) {
        int bsize = cinput.read();
        byte[] buf = new byte[bsize];
        cinput.read(buf);
        String msg = new String(buf);
        MsgParser(msg);   // handle GAME_START, MOVE, etc.
    }
}
```

Message framing & parsing

All messages use the pattern: `[length:1 byte][ASCII payload]` where payload is always `TYPE#data` (e.g. `MOVE#e2,e4`).

Message types:

```
   public enum Type {
        NONE, // No message
        JOIN_GAME, // Player joining a game
        GAME_START, // Game starting
        MOVE, // Player making a move
        GAME_OVER, // Game has ended
        WAITING, // Waiting for opponent
        ERROR, // Error occurred
        DISCONNECT, // Player disconnected
        SERVER_SHUTDOWN, // Server shutting down
        OPPONENT_DISCONNECTED   // Opponent has disconnected
    }
```

MsgParser() function:

A function that is on both sides splits on `#` and dispatches by enum:

```
String[] tokens = msg.split("#");
```

```
Message.Type mt = Message.Type.valueOf(tokens[0]);
switch (mt) {
  case GAME_START: ...
  case MOVE:       ...
  case DISCONNECT: ...
}
```

Clean disconnect:

Client: When you close the window or call `disconnect()` function, the client sends:
Server: Detects EOF or IOException in SClient.run()'s read loop, triggers
And with this structure:
Server → one thread accepts connections + one SClient thread per player.
Client → one background thread reads server messages + the Swing UI thread.

4) Class Structure & Design Architecture

This project is cleanly separated into three domains:

1. **Server-Side (server package)**
2. **Client-Side (Client package)**
3. **Shared Chess Logic (com.mycompany.chess package)**

Server Side– The Backend Controller:

This part is responsible for handling all networking, player pairing, and game state tracking between clients.

Server Class:

The Server class acts as the central logic for managing the chess application's network communication. It listens on port 6000 using a `ServerSocket` and handles incoming client connections. For each new connection, it creates and starts a new SClient thread to manage communication with that particular user. The server maintains a list of currently connected users through `List<SClient> clients` and a list of ongoing or pending games using `List<GameRoom> gameRooms`. It also handles different types of client requests by routing them appropriately:

- `JOIN_GAME` requests are handled by the `addPlayerToGame(...)` method,
- `MOVE` requests trigger the `handleMove(...)` method,
- The client disconnections are managed via the `handleDisconnect(...)` method.

SClient Class:

The SClient class extends Thread and is responsible for managing an individual client connection. Each instance maintains its own input and output streams for communication. Within its `run()` method, it continuously listens for and reads messages from the client's socket. Upon receiving a message, it uses the `MsgParser(...)` function to interpret and act on the message contents. Depending on the message type, it may call back to the Server to facilitate tasks such as player matchmaking, handling disconnections, or processing moves within a game.

GameRoom Class:

The GameRoom class represents a single one-on-one game session between two players. It holds references to two SClient instances: `player1` and `player2`. The class is responsible for managing turn-based control, forwarding moves between the players, and handling the overall lifecycle of the game through methods like `startGame()` and `endGame()`. It can determine whether the room is full using the `isFull()` method or whether it is waiting for a second player with the `needsPlayer()` method. If one player leaves the game, the GameRoom class ensures the remaining player is declared the winner by default.

Client Package – The Frontend + Networking

home Class:

The home class extends JFrame and serves as the lobby window for the application. It provides a simple interface where the user can enter their name and initiate a connection to the server. Upon a successful connection, the application transitions from the lobby to the main game interface by launching the `NetworkGame` window.

CClient Class:

The CClient class extends Thread and is responsible for managing all socket communication with the server on the client side. It provides methods such as `sendMove(...)`, `joinGame(...)`, and `disconnect()` to handle various client-server interactions. Within its `run()` method, it continuously reads messages from the server, parses them using the `MsgParser()` function, and appropriately routes the information to the graphical user interface. Communication is directed back to the home window for events like `GAME_START` and `WAITING`, while gameplay-related messages such as `MOVE`, `DISCONNECT`, and `GAME_OVER` are forwarded to the `NetworkGame` class.

NetworkGame Class:

The NetworkGame class extends `JFrame` and implements the `GameView` interface, serving as the main game window where the chessboard (an 8x8 grid) and game status are displayed. It handles user interactions such as clicking on pieces, updating the UI, and managing pawn promotions. `NetworkGame` integrates with the `GameController` class to enforce game rules and manage the turn flow, and with the `CClient` to send and receive moves over the network. By implementing the `GameController.GameView` interface it responds to game state changes through methods like `updateBoard()`, `updateTurn(...)`, `showMessage(...)`, and `showPromotionDialog(...)`, ensuring the interface reflects the current game state in real time.

Chess Package – Game Logic Core:

This package is the engine of your chess logic.

Piece & Its Subclasses:

The Piece class and its subclasses (Rook, Bishop, Queen, Pawn, King, and Knight) form a polymorphic design that leverages the Strategy Pattern. At its core, the abstract Piece class defines a method `isValidMove(...)` which each specific chess piece overrides to implement its unique movement logic.
This design enables clean and dynamic behavior assignment during runtime. For example, when retrieving a piece from the board like this:

```
Piece p = board.getPieceAt("e2");
```

You can call `p.isValidMove("e2", "e4", boardState)` without needing to know the exact type of the piece—each subclass handles its own validation logic. For instance, the Rook class overrides the method to allow only horizontal or vertical movement as long as the path is clear:

```
@Override
public boolean isValidMove(...) {
  .// horizontal or vertical only, path must be clear
}
```

The Queen combines movement rules from both the Rook and Bishop, delegating its logic as:

```
@Override
public boolean isValidMove(...) {
  return new Rook(...).isValidMove(...) || new Bishop(...).isValidMove(...);
}
```

Promoting code reuse, clarity, and flexibility, allowing each piece to encapsulate its own behavior.

GameController Class:

The GameController class serves as the core controller that bridges the game's logic with its user interface. It manages the overall game state and enforces the rules of chess. Internally, it maintains a `ChessBoard` instance representing the current layout of pieces, a `currentTurn` to track whose turn it is (white or black), and a `List<Move> moveHistory` to record all moves made during the game.

Key responsibilities of the class include handling piece movements via `movePiece(...)`, managing pawn promotions with `checkForPromotion(...)`, and determining game-ending conditions through methods like `switchTurns()`, `isCheckmate()`, and `isStalemate()`.

To keep the UI in sync with the game state, the GameController interacts with the user interface through the `GameView` interface, which is implemented by the `NetworkGame` class. This allows the controller to notify the UI when the board should be updated, a message should be displayed, or a promotion dialog needs to appear.

Interface: GameView:

```
public interface GameView {
    void updateBoard();
    void updateTurn(String turn);
    void showMessage(...);
    Piece showPromotionDialog(String color);
}
```

Making a bridge between the controller logic and the view.

ChessBoard Class:

The `ChessBoard` class is responsible for maintaining the state of the chessboard throughout the game. It uses a `Map<String, Piece> boardState` to represent the position of pieces on the board, with keys like "e2" and "d4" corresponding to specific squares.

The class provides several essential methods to manage and query the board. The `initializeBoard()` method sets up all 32 pieces in their standard starting positions. Utility methods such as `getPieceAt(...)`, `placePiece(...)`, `removePiece(...)`, and `movePiece(...)` are used to interact with individual pieces and update their positions.

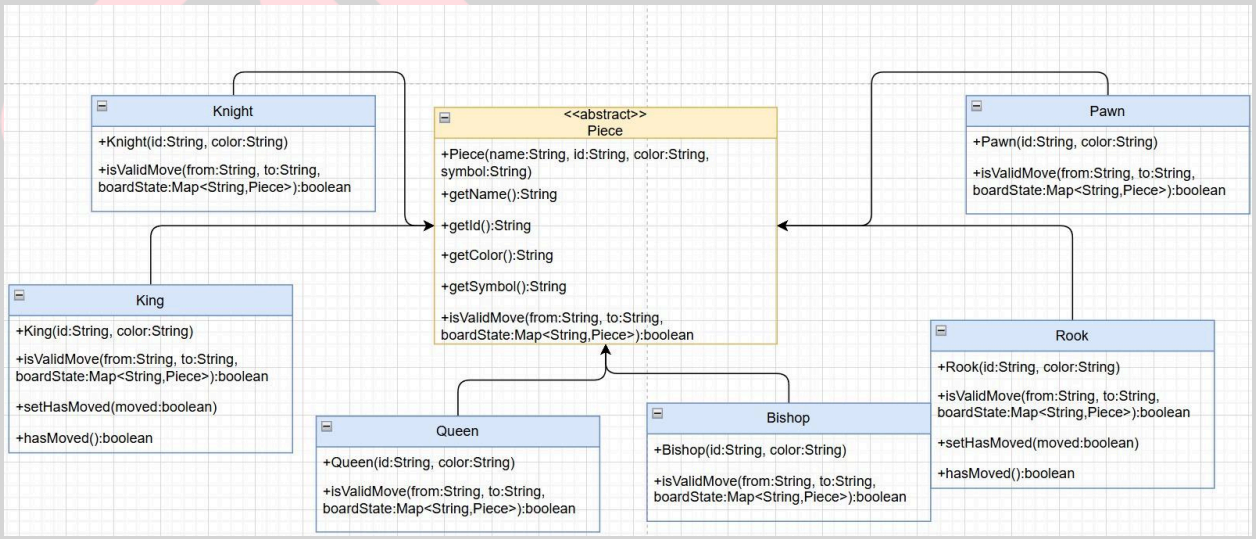
Additionally, ChessBoard includes logic for rule enforcement, such as checking whether a king is currently in check via `isKingInCheck(...)`, determining if a player has any legal moves left through `hasLegalMoves(...)`, and validating if a proposed move

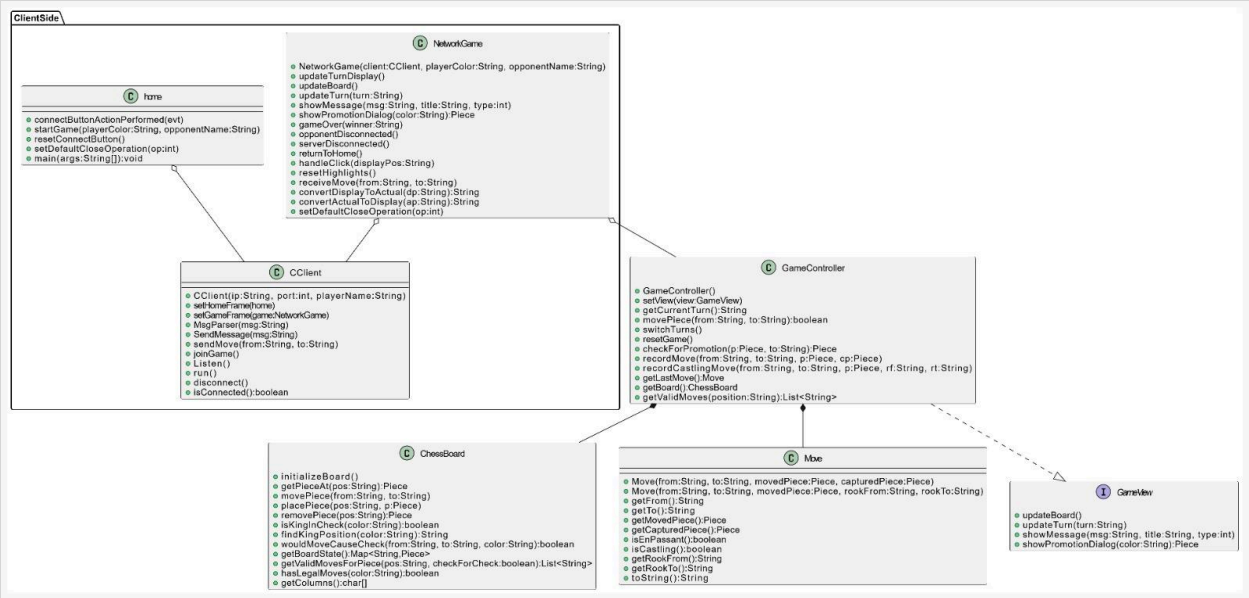
would place the player's own king in check using `wouldMoveCauseCheck(...)`. This simulation-based approach helps ensure that all moves comply with the rules of chess.

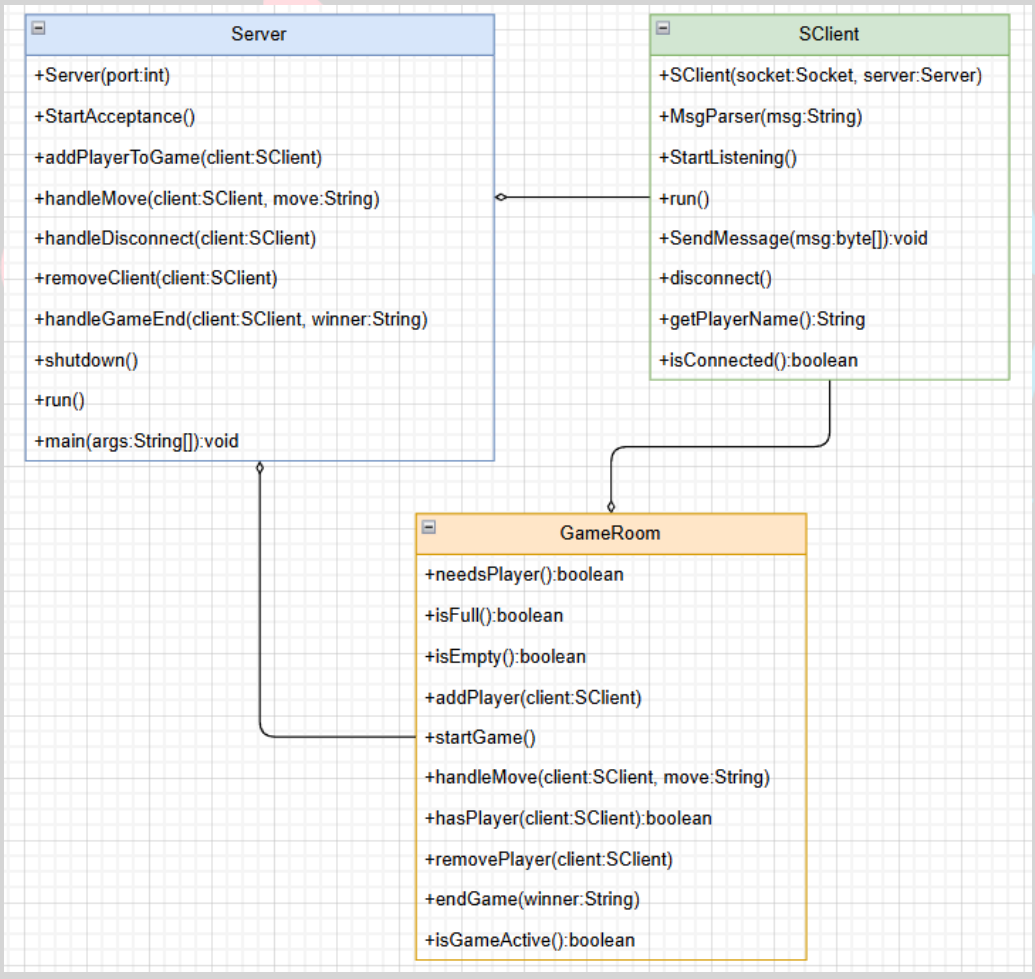
Move Class:

The Move class serves as an immutable record that captures the details of a single chess move. It stores the origin (from) and destination (to) squares, the piece that was moved, any piece that was captured, and flags indicating special move types such as castling (`isCastling`) and en passant (`isEnPassant`).
This class is primarily used by the `GameController` to maintain a complete history of all moves through the `moveHistory` list, which supports features like move validation, undo logic, and rule enforcement. It also plays a role in rule-specific checks—for example, the Pawn class consults the last recorded Move to determine if an en passant capture is valid. By encapsulating all relevant move data, the Move class helps maintain the integrity and traceability of game actions.

Project’s Class Relations & Functions:




ClientSide
NetworkGame
+ NetworkGame(client:CClient, playerColor:String, opponentName:String)
+ updateTurnDisplay()
+ updateBoard()
+ updateTurn(turn:String)
+ showMessage(msg:String, title:String, type:int)
+ showPromotionDialog(color:String):Piece
+ gameOver(winner:String)
+ opponentDisconnected()
+ serverDisconnected()
+ returnToHome()
+ handleClick(displayPos:String)
+ resetHighlights()
+ receiveMove(from:String, to:String)
+ convertDisplayToActual(dp:String):String
+ convertActualToDisplay(ap:String):String
+ setDefaultCloseOperation(op:int)
home
+ connectButtonActionPerformed(evt)
+ startGame(playerColor:String, opponentName:String)
+ resetConnectButton()
+ setDefaultCloseOperation(op:int)
+ main(args:String[]):void
CClient
+ CClient(ip:String, port:int, playerName:String)
+ setHomeFrame(home)
+ setGameFrame(game:NetworkGame)
+ MsgParser(msg:String)
+ SendMessage(msg:String)
+ sendMove(from:String, to:String)
+ joinGame()
+ Listen()
+ run()
+ disconnect()
+ isConnected():boolean
GameController
+ GameController()
+ setView(view:GameView)
+ getCurrentTurn():String
+ movePiece(from:String, to:String):boolean
+ switchTurns()
+ resetGame()
+ checkForPromotion(p:Piece, to:String):Piece
+ recordMove(from:String, to:String, p:Piece, cp:Piece)
+ recordCastlingMove(from:String, to:String, p:Piece, rf:String, rt:String)
+ getLastMove():Move
+ getBoard():ChessBoard
+ getValidMoves(position:String):List<String>
ChessBoard
+ initializeBoard()
+ getPieceAt(pos:String):Piece
+ movePiece(from:String, to:String)
+ placePiece(pos:String, p:Piece)
+ removePiece(pos:String):Piece
+ isKingInCheck(color:String):boolean
+ findKingPosition(color:String):String
+ wouldMoveCauseCheck(from:String, to:String, color:String):boolean
+ getBoardState():Map<String,Piece>
+ getValidMovesForPiece(pos:String, checkForCheck:boolean):List<String>
+ hasLegalMoves(color:String):boolean
+ getColumns():char[]
Move
+ Move(from:String, to:String, movedPiece:Piece, capturedPiece:Piece)
+ Move(from:String, to:String, movedPiece:Piece, rookFrom:String, rookTo:String)
+ getFrom():String
+ getTo():String
+ getMovedPiece():Piece
+ getCapturedPiece():Piece
+ isEnPassant():boolean
+ isCastling():boolean
+ getRookFrom():String
+ getRookTo():String
+ toString():String
GameView
+ updateBoard()
+ updateTurn(turn:String)
+ showMessage(msg:String, title:String, type:int)
+ showPromotionDialog(color:String):Piece


Server
+Server(port:int)
+StartAcceptance()
+addPlayerToGame(client:SClient)
+handleMove(client:SClient, move:String)
+handleDisconnect(client:SClient)
+removeClient(client:SClient)
+handleGameEnd(client:SClient, winner:String)
+shutdown()
+run()
+main(args:String[]):void
SClient
+SClient(socket:Socket, server:Server)
+MsgParser(msg:String)
+StartListening()
+run()
+SendMessage(msg:byte[]):void
+disconnect()
+getPlayerName():String
+isConnected():boolean
GameRoom
+needsPlayer():boolean
+isFull():boolean
+isEmpty():boolean
+addPlayer(client:SClient)
+startGame()
+handleMove(client:SClient, move:String)
+hasPlayer(client:SClient):boolean
+removePlayer(client:SClient)
+endGame(winner:String)
+isGameActive():boolean