Remote_Checkers
Software Design Specification

Christopher Macco
Khang Nghe
Zakary Olyarnik
Dana Thompson

Revision History

Member	**Date**	**Change**
Olyarnik	2/13/17	Completed outline of all sections (r0.1.0)
Macco	2/13/17	First draft of System Architecture section (r0.2.0)
Macco	2/14/17	Added Figures 3 and 4: Low Level Class Diagrams (r0.2.1)
Olyarnik	2/15/17	First draft of Introduction and Interface Design sections (r0.3.0)
Nghe, Olyarnik, Thompson	2/18/17	Final draft of Introduction, System Overview, and Interface Design sections (r0.4.0)
Nghe, Olyarnik, Thompson	2/19/17	Completion of Submission Draft (r1.0.0)

Table of Contents

1. Introduction

1.1 Purpose of Document

This document shall describe the Java implementation of Remote_Checkers. It is to be used as a reference by developers when coding the system. The completed application shall meet all of the requirements laid out in the Requirements Specification.[1]

1.2 Scope of Document

This document contains enough information that a developer could implement the Remote_Checkers system unambiguously as described. The provided class diagrams are intended to provide a complete design specification, showing each class' attributes, methods, and relationships to the other classes in the system. This document also includes a brief design rationale, control flowcharts, and an overview of the user interface design.

2. System Overview

2.1 Product Perspective

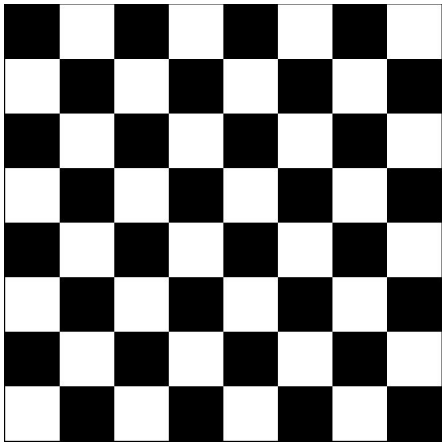

Figure 1: The checkerboard layout used by Remote_Checkers

Remote_Checkers is a game application for two Players to play Checkers from two different locations. The game requires a Server to be running, which can be started by either of the two Players or a third party. The game requires two Players to connect to the Server in order to create a Game session. The game is top-down and follows the standard rules and regulations of Checkers. Players will take turn making Moves and trying to capture all of their opponent's Pieces. When a winner is declared, Players can vote for a rematch, or go back to the main screen and connect to a new Game.

2.2 Control Flow Diagram

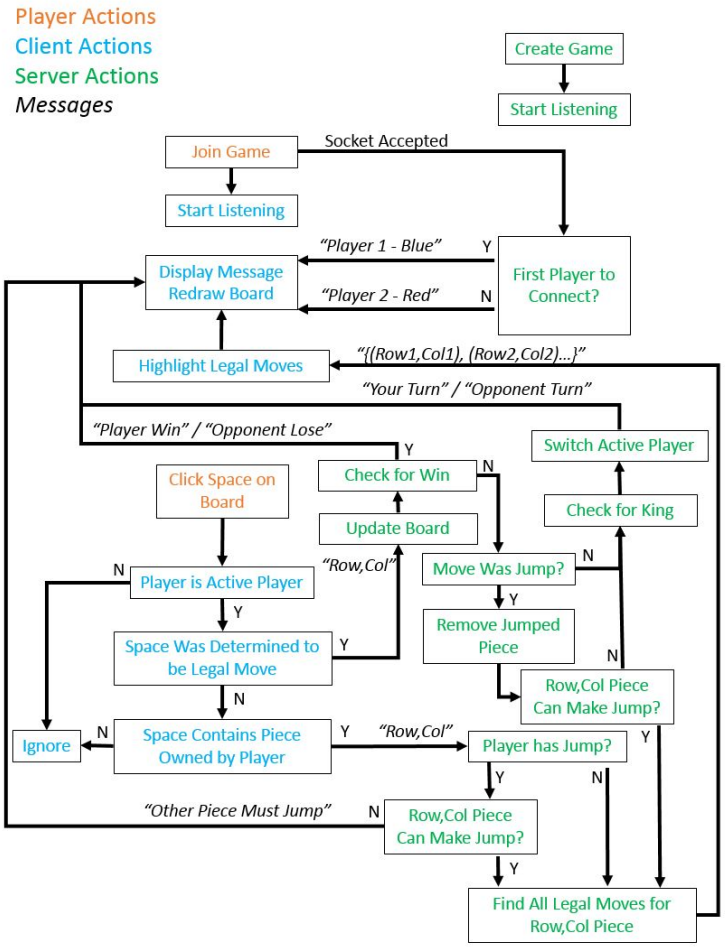

Figure 2: Control Flow Diagram

This diagram shows the general control flow of a game of Remote_Checkers. The user link-in points are as follows:

- Starting the Server to create a Game, which must be done before any other actions
- A Player joining a Game, which launches a Client and starts communication
- A Player clicking a Space on the Board, which will either be validated as a Piece they wish to move or a Space they want to move into or an invalid Move and ignored

See **Figure 7: Screen Interaction Diagram** for more details about starting and ending games.

2.3 Technologies Used

Remote_Checkers is implemented using the Client/Server pattern. The Clients are meant to be run on separate computer systems, while the Server can be hosted on either Player's computer or a third party computer. All programs must communicate over a shared local wireless network. Remote_Checkers is meant to be played using mouse and keyboard input.

Remote_Checkers is implemented in Java 1.8 using Eclipse. Static analysis is provided by Checkstyle. Code coverage is provided by EclEmma. Bug reporting is provided by Jira. Version control is handled using GitHub.

3. System Architecture

3.1 Architectural Design Components

3.1.1 Server

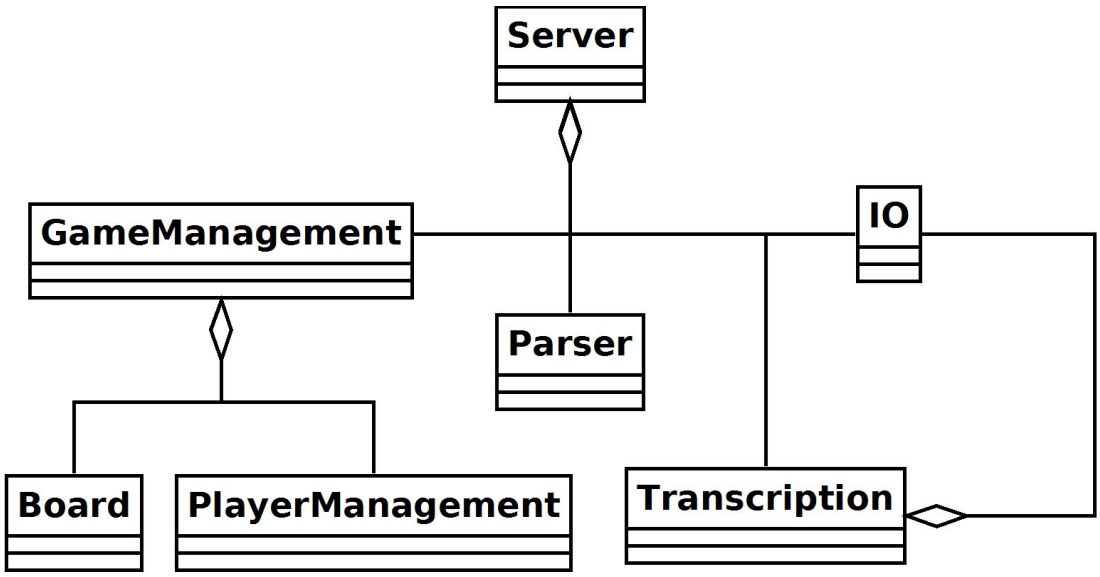

Figure 3: High Level Class Diagram for the Server

Server - The Server system facilitates game management, Player management, Player Moves, end-game conditions, and creation/storage of the gameboard. The Server also handles connections between Clients and matchmaking.

Board - This subsystem keeps track of the Pieces on the **b**oard, which Pieces are King Pieces, and which Player they belong to. It also keeps track of possible “green spaces”, or Spaces available for the Active Player to move to.

Game Management - This subsystem is used to manage the game. This includes creating the Game, facilitation of Moves, Move validity checking, executing rules of the game, calculating possible Moves or “green spaces”, calculating possible Jumps, and controlling the Game’s status.

IO - The IO (Input Output) subsystem is responsible for handling any input or output to files, the console, or the Client. This subsystem only handles the facilitation of

Messages, and does not create them. This subsystem looks similar to the IO subsystem on the Client side.

Parser - This subsystem is used to ensure the Messages to and from the Client are properly handled by parsing them and storing values.

Player Management - This subsystem is used to manage the matchmaking part of the system. It determines which Client get paired for a Game and assigns those Players their color and turn order.

Transcription - This subsystem is used to ensure a connection is made to Clients coming to the system, and is used to open Ports for Clients to connect to in order to send Messages back and forth for the Game.

3.1.2 Client

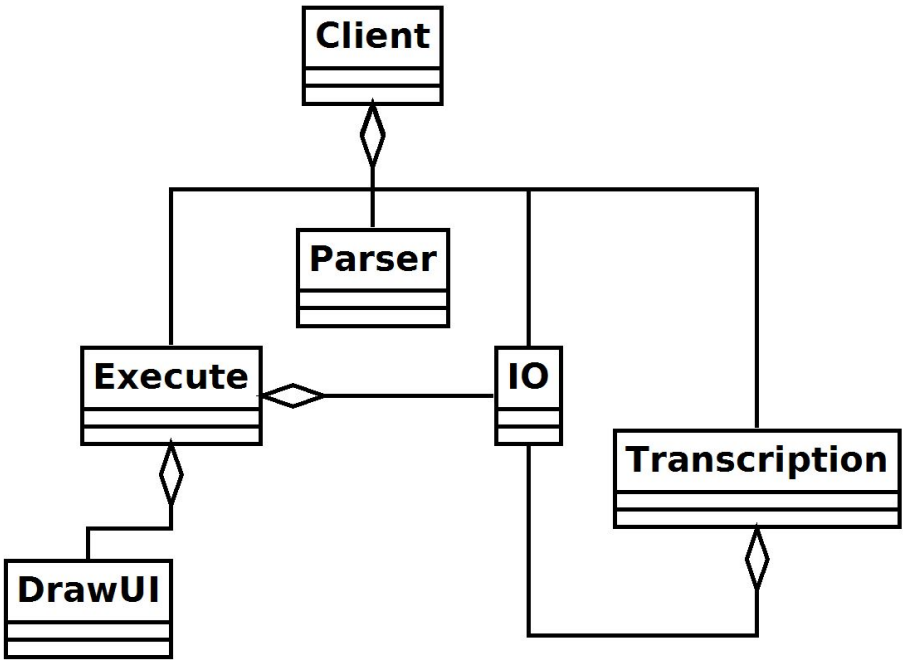

Figure 4: High Level Class Diagram for the Client

Client - The Client system handles drawing the Board (including flipping its orientation depending on the Player), facilitating sending Moves to the Server, displaying Messages (both informational and error), parsing Messages to and from the Server, and using basic Move validation to stop unnecessary Server calls.

DrawUI - This subsystem is solely responsible for drawing all graphical user interfaces, most notably the gameboard.

Execute - This subsystem is used to execute Game and Settings logic. Settings can be both retrieved from file and resaved, and the Client-side Game performs basic Move validation before calling the Server.

IO - The IO (Input Output) subsystem is responsible for handling any input or output to files, the console, or the Server. This subsystem only handles the facilitation of Messages, and does not create them. This subsystem looks similar to the IO subsystem on the Server side.

Parser - This subsystem is used to ensure the Messages to and from the Server are properly handled by parsing them and storing values.

Transcription - This subsystem is used to make a connection to the Server, and keep track of the open IP Addresses and Ports for the active connection.

3.2 Design Rationale

3.2.1 Separate Client/Server Architecture

We chose to design separate programs for our Client/Server architecture in order to mimic the traditional online gaming pattern, where the Server is a standalone system that an unlimited number of Clients can connect to. Of course, the Server is still able to be run on a Client's computer (as a separate program), so the game can still be played with whatever resources are available.

3.2.2 Server Processing

We decided to have the Server do most of the game processing in order to take the load off of the Client computers. To accomplish this, we decided to have the Client send only necessary information twice per turn to the Server, once to select a Piece to move and once to select the Space to move to. The Server calculates all possible Moves for the selected Piece, makes the chosen Move, and then calculates all legal Pieces the opponent Player can make in preparation for their turn. The result is that the

Client simply relays the Messages it receives, while the Server truly implements Checkers.

3.2.3 Board Representation

In order to keep track of the Board, the Pieces, and legal selections, we chose to implement three parallel 8x8 arrays of enums, one located on each Client and one on the Server. We have defined enum values for empty Spaces, Pieces and Kings of both colors, legally-selectable Spaces to move into, and legally-selectable Pieces of all types. The Server always controls the most up-to-date Board, and only sends updates to the Clients as necessary. For example, the Active Player will receive a Board update whenever they select a Piece to move (highlighting that Piece's legal Moves), but the non-Active Player will not get this update. The Active Player will also receive an update at the end of their turn, showing the result of their Move. The Server then calculates all of the possible Pieces the opponent can legally select to move and sends a new Board update including this information to the opponent at the start of their turn. These possible Piece selections are not displayed visually to the Payer as the possible Moves for a selected Piece are, but the Client having knowledge of them allows it to do basic Move validation without having to Message the Server every time.

3.2.4 Board Flipping

We chose to flip the displayed Board depending on the Player in order to keep metaphor with real life Checkers, where Players control the Pieces that start closest to them on the Board. Since our Boards are represented as 8x8 arrays, we use simple math on the indexes of Player 2's Moves to translate their selected Pieces/Moves to the correct positions on the Server's Board. When the Server sends updates back, it sends the whole Board, so we just read the Json string in reverse when checking for Spaces to update.

3.2.5 Modularization and Abstraction

We wanted to separate the graphical user interface from the backend implementation, the backend implementation from the Server, and any data transfer or facilitation of data from the implementation. We chose to do this in order to create modular software that can be more easily supported or updated in the future.

4. Component Design

4.1 Server Architecture

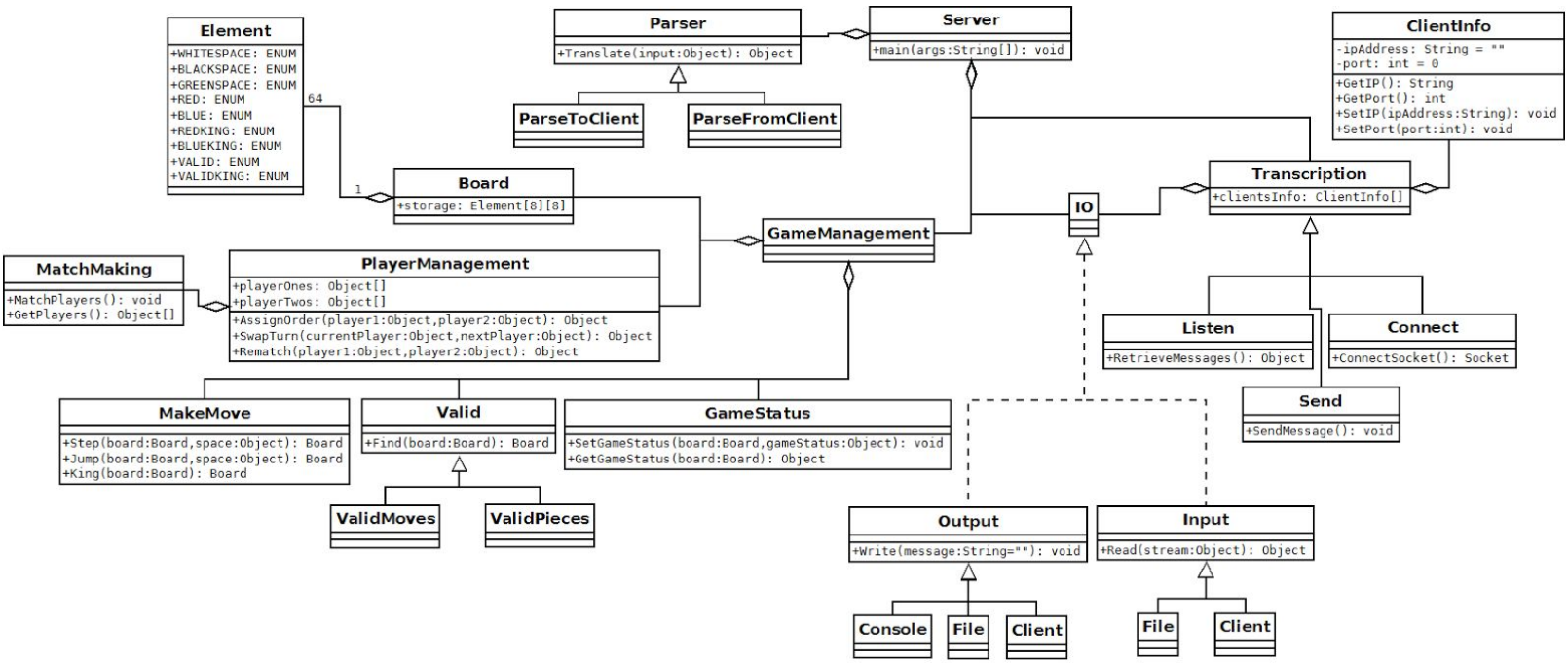

Figure 5: UML Class Diagram for Server

4.1.1 Server

The Server class contains the main() function for the Server system, which coordinates communication with Clients and creation and management of Games.

Server
- gM: GameManagement - parser: Parser - transcription: Transcription - io: IO[]
+ main(args: String[]): void

4.1.1.1. Server Attributes

Name	Type	Description
gM	GameManagement	Manages and runs the Game by managing Players, storing Board states, enacting Moves, checking for validity, and controlling Game status.
parser	Parser	Parses Messages to and from the Client into a useable format.
transcription	Transcription	Connects Clients to the Server and sends and receives Messages from them.
io	IO[]	Handles input and output.

4.1.1.2 Server Methods

Name	Input	Output	Description
main()	String[]	void	Runs the program and coordinates Client communication and Game management.

4.1.2 IO

The IO system is responsible for handling input and output to different IO streams. The Input and Output classes implement the IO interface, with their subclasses handling reading and writing to specific streams.

4.1.2.1 Output

Output
+ Write(message: String = “”): void

Name	Input	Output	Description
Write()	String	void	Writes a Message to an output stream.

4.1.2.2 Input

Input
+ Read(stream: Object): Object

Name	Input	Output	Description
Read()	String	void	Reads and retrieves data from input streams.

4.1.3 GameManagement

The GameManagement system is responsible for coordinating all the different elements of a Game, such as matching and managing Players, storing Boards and applying Moves, finding all valid selectable Pieces, finding all valid Moves, and checking/verifying the status of the Game.

GameManagement
+ singleton: GameManagement + playerManagement: PlayerManagement + boards: Board[] + makeMove: MakeMove + validPiece: ValidPiece + validMove: ValidMove + gameStatus: GameStatus

4.1.3.1 GameManagement Attributes

Name	Type	Description
singleton	GameManagement	Handles input and output
playerManagement	PlayerManagement	Matches and manages Players.
boards	Board[]	An array holding the Boards for all running Games.

makeMove	MakeMove	Applies Moves to a Board.
validPiece	ValidPiece	Finds all valid pieces a Player can select to Move.
validMove	ValidMove	Finds all valid Moves of a selected piece.
gameStatus	GameStatus	Determines the status (running or ended) of a Game.

4.1.3.2 MakeMove

The MakeMove class handles applying different types of Move to the gameboard.

MakeMove
+ Step(board: Board, destination: Object): void + Jump(board: Board, destination: Object): void + King(board: Board): void

Name	Input	Output	Description
Step()	Board, Object	void	Handles Step Moves, where a Piece moves one Space diagonally.
Jump()	Board, Object	void	Handles Jump Moves, where a Piece jumps two Spaces diagonally and captures an opponent's Piece.
King()	Board	void	Handles Crowning, where a Man becomes a King when it reaches the opponent's end of the Board.

4.1.3.3 Valid

The Valid class system is responsible for validating Piece selections and Moves from Clients. It does this by pre-emptively evaluating the Board for valid Moves (ValidMove subclass) or valid Pieces (ValidPiece subclass) and sending those to the Clients so they can do the on-click validation themselves without having to ask the Server.

Valid
+ Find(board: Board): Board

Name	Input	Output	Description
Find()	Board	Board	Find either valid selectable Pieces or valid Moves for a selected Piece, and update the Board.

4.1.3.4 GameStatus

The GameStatus class is responsible for maintaining and checking the state of the Game and handling the game accordingly. There are two states: running and ended. The ended state is achieved if one of the Players wins or the game draws.

GameStatus
+ SetGameStatus(board: Board, gameStatus: Object) : void + GetGameStatus(board: Board): Object

Name	Input	Output	Description
SetGameStatus()	Board, Object	void	Sets the Game's status to running or ended.
GetGameStatus()	Board	Object	Checks the state of the Game.

4.1.3.5 Board

The Board system handles keeps track of the state of gameboards through an 8x8 array of Elements. The different possible value of an Element enum correspond to different possible states of a Space on the gameboard.

Board
+ storage:Element[8][8]

Name	Type	Description
storage	Element[8][8]	An 8x8 array of Elements representing the 64 Spaces on the game board.

4.1.3.5.1 Element Enum

Element (enum)
+ WHITESPACE: Element + BLACKSPACE: Element + GREENSPACE: Element + RED: Element + BLUE: Element + REDKING: Element + BLUEKING: Element + VALID: Element + VALIDKING: Element

Name	Type	Description
WHITESPACE	Element	A white Space on the Board.
BLACKSPACE	Element	A black Space on the Board.
GREENSPACE	Element	A black Space with a green border indicating a valid Move for a currently selected Piece.
RED	Element	A black Space with a red Man.
BLUE	Element	A black Space with a blue Man.
REDKING	Element	A black Space with a red King.
BLUEKING	Element	A black Space with a blue King.
VALID	Element	A valid Man for selection to move. The color will be determined by the Client.
VALIDKING	Element	A valid King for selection to move. The color will be determined by the Client.

4.1.3.6 PlayerManagement

The PlayerManagement system is responsible for matching Players against each other, sharing information, assigning color, turn order, Active Player switching, scoring, and rematching.

PlayerManagement
+ playerOnes: Object[] + playerTwos: Object[]
+ AssignOrder(player1:Object, player2:Object): Object + SwapTurn(currentPlayer:Object, nextPlayer:Object): Object + Rematch(player1:Object, player2:Object): Object

4.1.3.6.1 PlayerManagement Attributes

Name	Type	Description
playerOnes	Object[]	Array of Player ones for all active Games.
playerTwos	Object[]	Array of Player twos for all active Games.

4.1.3.6.2 PlayerManagement Methods

Name	Input	Output	Description
AssignOrder()	Object, Object	Object	Assign the turn order for Players.
SwapTurn()	Object, Object	Object	Swap the Active Player in the Game.
Rematch()	Object, Object	Object	Initiate a rematch for the two Players

4.1.3.6.3 MatchMaking

The MatchMaking class is responsible for matching every two Clients together to form a Game.

MatchMaking
+ MatchPlayers(): void + GetPlayers(): Object[]

Name	Input	Output	Description
MatchPlayers()	void	void	Match two Clients together to form a Game.
GetPlayers()	void	Object[]	Get the Players who have been matched together.

4.1.4 Parser

The Parser system handles the translation of Messages to and from the Clients to make sure they are in the intended format and are handled appropriately. The subclasses handle translation in both directions.

Parser
+Translate(input: Object): Object

4.1.4.1 Parser Methods

Name	Input	Output	Description
Translate()	Object	Object	Parent method that the subclasses will override to implement their translation algorithm.

4.1.5 Transcription

The Transcription system makes the connection to Clients and facilitates the sending and receiving of Messages.

Transcription
+ clientsInfo: ClientInfo[] - io: IO[]

4.1.5.1 Transcription Attributes

Name	Type	Description
io	IO[]	Handles input and output.
clientInfo	ClientInfo[]	Contains information about the IP Address and Port of the Clients.

4.1.5.2 Transcription Subclasses

The Connect subclass is responsible for opening sockets to allow for connections from the Clients. The Listen subclass is responsible for listening for Messages from the Clients. The Send subclass is responsible for sending Messages to the Clients.

4.1.5.3 ClientInfo

Stores information about each Client that connects.

ClientInfo
- ipAddress: String = “” - port: int = 0
+ GetIP(): String + GetPort(): int + SetIP(ipAddress:String): void + SetPort(port:int): void

Name	Type	Description
ipAddress	String	The Client’s IP Address.
port	int	The Client’s Port.

Name	Input	Output	Description
GetIP()	void	String	Gets the Client's IP Address.
GetPort()	void	int	Gets the Client's Port.
SetIP()	String	void	Records the Client's IP Address in this list.
SetPort()	int	void	Records the Client's Port in this list.

4.2 Client Architecture

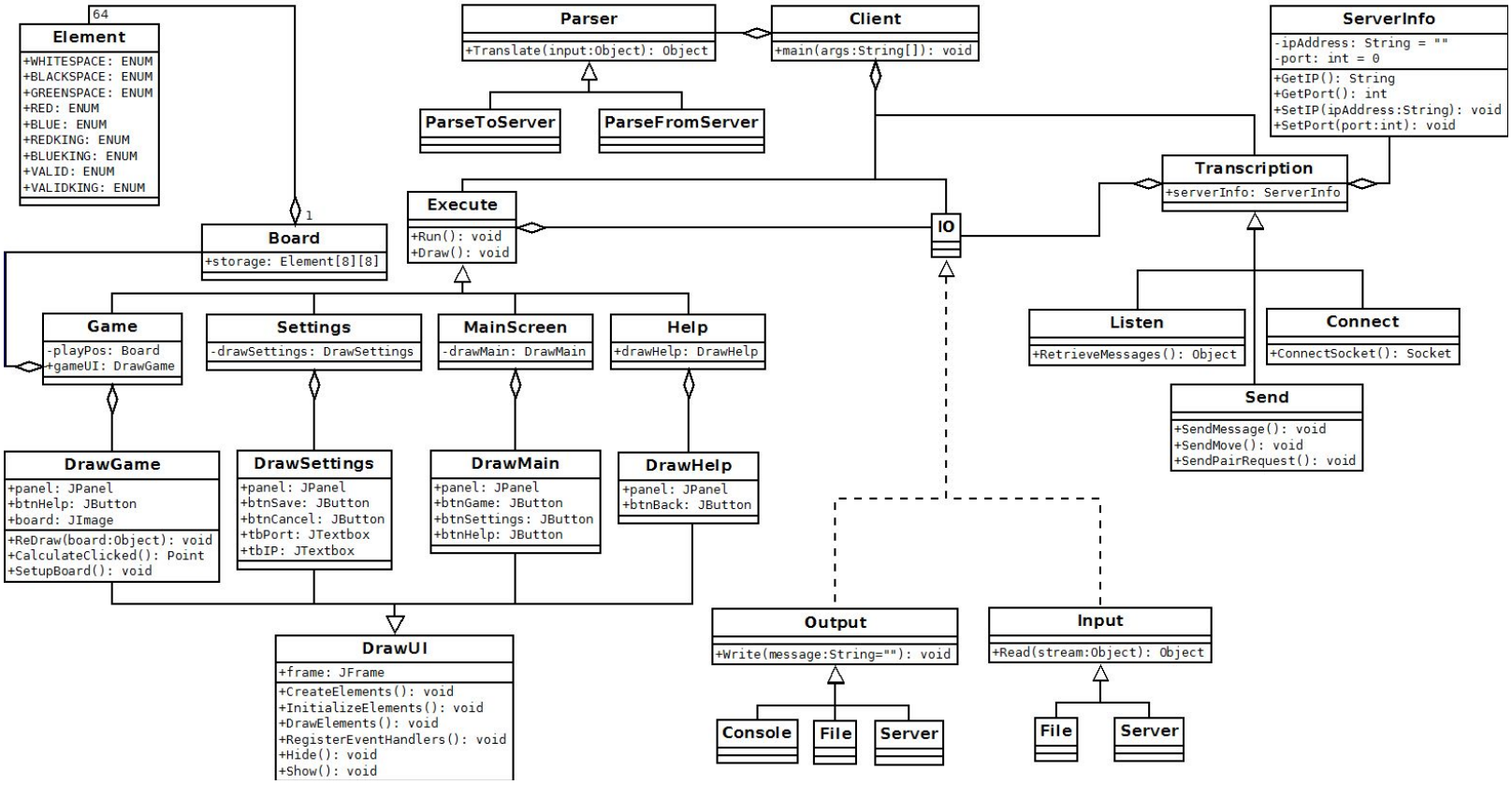

Figure 6: UML Class Diagram for Client

4.2.1 Client

Client
+ parser : Parser + transcribe : Transcription + io : IO + ex : Execute
+ main(args::String[]) : void

4.2.1.1 Client Attributes

Name	Type	Description
parser	Parser	Parses Messages to and from the Server into a readable format.
transcribe	Transcription	Connects to the Server and sends and receives Messages from it.
io	IO	Handles input and output.
ex	Execute	Responsible for running the Game.

4.2.1.2 Client Methods

Name	Input	Output	Description
main()	String[]	void	Runs the program and coordinates Server communication.

4.2.2 Execute

Execute
+ io : IO
+ Run() : void + Draw() : void

4.2.2.1 Execute Attributes

Name	Type	Description
io	IO	Handles input and output.

4.2.2.2 Execute Methods

Name	Input	Output	Description
Run()	void	void	Runs any logic required by the screen of the inheriting subclass.
Draw()	void	void	Links into a DrawUI element to draw the UI of the screen of the inheriting subclass.

4.2.2.3 Execute Subclasses

The Game subclass stores the Board as an 8x8 array and draws a graphical representation of it. The implementation of the Board is the same as on the Server side (see **Section 4.1.3.5 Board** for details). The Settings subclass reads Server connection settings from the config file and draws the components necessary to display and edit them. The MainScreen subclass draws the main screen and the Help subclass draws the help window. The latter two do not require any heavy coding logic to facilitate.

4.2.3 DrawUI

DrawUI
+ frame : JFrame
+ CreateElements() : void + InitializeElements() : void + DrawElements() : void + RegisterEventHandlers() : void + Hide() : void + Show() : void

4.2.3.1 DrawUI Attributes

Name	Type	Description
frame	JFrame	Displays all of the UI for the Java application.

4.2.3.2 DrawUI Methods

Name	Input	Output	Description
CreateElements()	void	void	Creates the components to draw the UI that the subclasses need.
InitializeElements()	void	void	Instantiates UI objects for the subclasses.
DrawElements()	void	void	Draws the UI objects for the subclasses.
RegisterEventHandlers()	void	void	Sets up button-click handlers.
Hide()	void	void	Hides UI when the screen switches.
Show()	void	void	Displays UI when the screen switches.

4.2.3.3 DrawUI Subclasses

The DrawSettings, DrawMain, and DrawHelp subclasses contain all of the Java Swing elements necessary to build the desired graphical user interface. They overwrite the DrawUI methods with behavior unique to the screen they are responsible for displaying. The DrawGame subclass additionally contains methods to do an initial setup of the Board and redraw it as necessary.

4.2.4 Transcription

Transcription
+ io : IO + serverInfo : ServerInfo

4.2.4.1 Transcription Attributes

Name	Type	Description
io	IO	Handles input and output.
serverInfo	ServerInfo	Contains information about the IP Address and Port of the Server.

4.2.4.2 Transcription Subclasses

The Connect subclass is responsible for connecting to the Server socket. The Listen subclass is responsible for listening for Messages from the Server. The Send subclass is responsible for sending Messages to the Server.

4.2.4.3 ServerInfo

ServerInfo
- ipAddress: String = “” - port: int = 0
+ GetIP(): String + GetPort(): int + SetIP(ipAddress:String): void + SetPort(port:int): void

Name	Type	Description
ipAddress	String	The Server’s IP Address.
port	int	The Server’s Port.

Name	Input	Output	Description
GetIP()	void	String	Gets the Server’s IP Address.
GetPort()	void	int	Gets the Server’s Port.
SetIP()	String	void	Records the Server’s IP Address.
SetPort()	int	void	Records the Server’s Port.

4.2.5 IO

The Client-side IO system is extremely similar to the Server-side implementation, with the only difference being that Input and Output contain subclasses for Server streams. See **Section 4.1.2 IO** for more information.

4.2.6 Parser

The Client-side Parser system is extremely similar to the Server-side implementation, with the only difference being that subclasses parse Messages to and from the Server. See **Section 4.1.4 Parser** for more information.

5. Interface Design

5.1 Screen Overview

This section describes the different interfaces found in the Remote_Checkers Client application. (The Server has no graphical user interface; it shall be run from the console/terminal of the computer and only display a simple message on successful launch.) The UI shall be implemented in Java Swing. Each new “screen” shall be a JPanel which is switched in and out at runtime, and each “window” shall be a modal JDialog. More details can be found in the Requirements Specification document.[1]

Main Screen - Displayed on application launch. Contains buttons which shall switch to the Game Screen and Settings Screen, launch the Help Window, and launch the Exit Confirmation Window to exit the application.

Game Screen - Contains the interactive Checkers gameboard. The Player’s Pieces shall be displayed at the bottom edge of the Board, the one closest to them. The Game Screen shall also include a message display area and buttons to launch the Help Window and the Forfeit Confirmation Window.

Settings Screen - Allows the user to view and edit the IP Address and Port number of the Server they wish to connect to hosting Remote_Checkers Games.

Help Window - Displays the basic rules of Checkers and instructions how to use the application. Can be viewed from either the Main Screen or Game Screen.

Exit Confirmation Window - Asks the user to confirm their desire to exit the application.

Forfeit Confirmation Window - Asks the user to confirm their desire to forfeit the current Game.

Opponent Disconnect Notification Window - Displayed when the opponent disconnects from the Game session unexpectedly. The Player shall be returned to the Main Screen.

Rematch Prompt Window - Asks Players if they want a rematch after a win, loss, draw, or forfeit. If either Player refuses the rematch, both shall be returned to the Main Screen.

5.2 Screen Interaction

The following diagram shows the flow from one screen or window to the next within the application. Most transitions are caused by clicking the corresponding on-screen button, and all are designed with error prevention/recovery in mind.

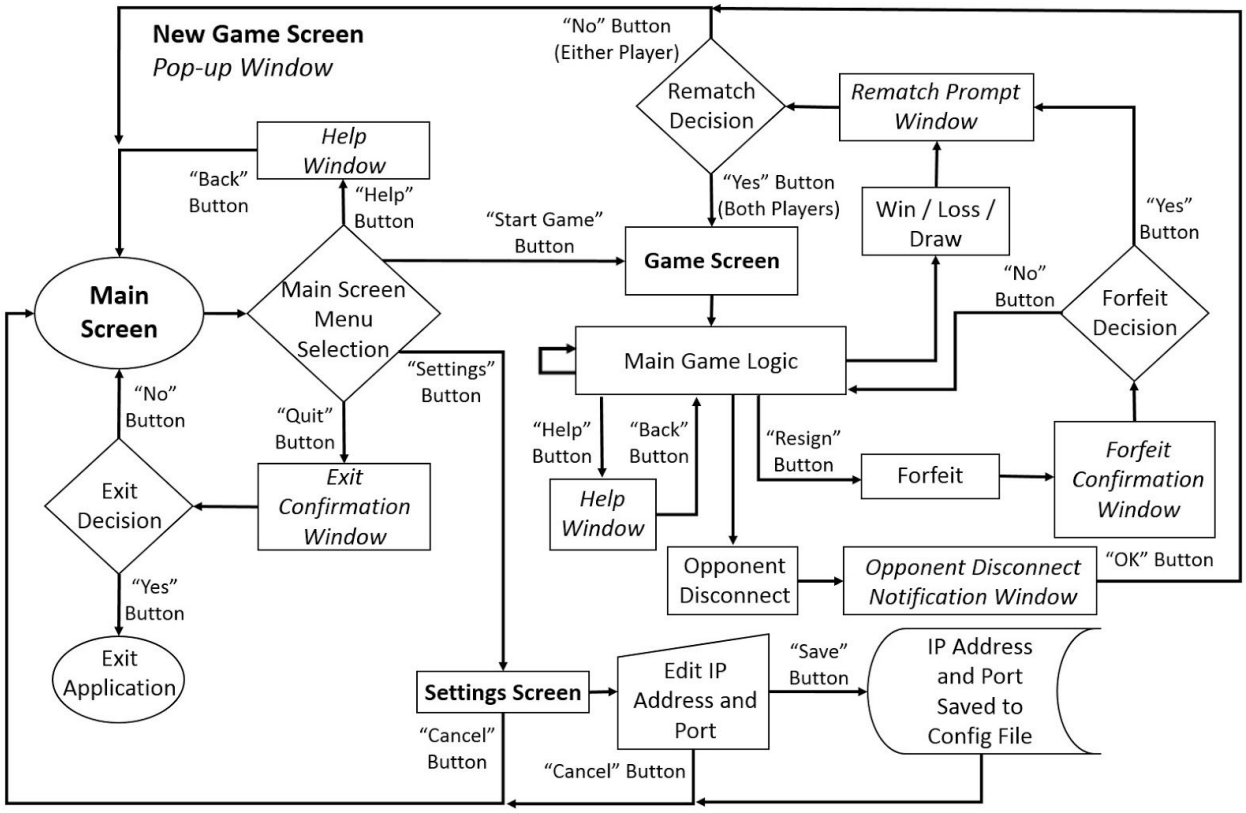

Figure 7: Screen Interaction Diagram

6. Glossary

Active Player - The Player whose turn it is.

Board - Either the Server's abstract model representation of the current Game's checkerboard, or the Client's graphical representation of the same.

Checkers - The game being emulated. Players move their Pieces diagonally across the board to try and capture all of their opponent's Pieces.[2]

Client - Program which the Player interacts with. Displays a graphical representation of the Server's gameboard, which Players can click to make Moves. Player actions will be sent in Message form to the Server for validation, at which point the Board will be updated.

Crowning - The act of a Player moving one of their Pieces all the way to the opposite end of the Board, at which time it becomes promoted to a King.

Game - A single instance of a two-Player session of Checkers, created, stored, and updated by the Server. Holds the state of the Board and implements the rules.

Internet Protocol (IP) Address and Port - Information that a Client needs about a Server in order to make a network connection. The IP Address uniquely identifies the machine running Server code, and the Port specifies a channel where the Server is listening to accept potential connections.

Java - The coding language used for development of Remote_Checkers.

Jump - A Checkers Move where one Player "jumps" their Piece over an opponent's Piece in the Space directly diagonal. The opponent's Piece is captured. If a Player can make a Jump on their turn, they must do so.

King - A promoted Piece which can move backwards.

Man - A normal Checkers Piece.

Message - Text sent from Server to Client or Client to Server. The receiver parses the text and decides how to act on it.

Move - The act of a Player choosing a Piece and then a Space to move that Piece into. Either a Step or a Jump.

Piece - A single Checker, either a Man or a King, which a Player controls and moves around the board.

Player - A physical human playing the game, done by interacting with a Client instance unique to them.

Remote_Checkers - The game being designed, allowing two Players to play a game of Checkers from separate locations over a network connection.

Server - Program which handles main Game facilitation. Starts a Game and allows Clients to connect to it to play. Manages any running Games.

Space - A single one of the 64 squares making up the Board. A Space can either be white, empty black, black with a blue Man, black with a red Man, black with a blue King, or black with a red King. Remote_Checkers also defines green Spaces as Spaces marked as valid Moves for the Active Player.

Step - The basic Move of moving a Piece into an empty forward-diagonal Space.

7. References

[1] Macco, et. al. Remote_Checkers Requirements Specification. Philadelphia, PA, United States, 2017.

[2] https://www.itsyourturn.com/t_helptopic2030.html#helpitem1197