Checkers Application
Design Document

Group 4:

Christopher Fosythe
Samuel Platek
Robert Roche
Justin Roszko

Table of Contents

1. Introduction

1.1 Purpose of Document

This document is to describe the implementation of our SAFC as described in the Checkers Application Requirements document. This Checkers application is designed to be a recreation of the famous 2 player checkers game played on two different computers.

1.2 Scope of Document

This document describes the implementation details of the SAFC. The software will consist of several systems that are mainly split by their function. The major systems are the Networking State, Game State, and Player state. Code for networking is designed to work with the other two states in order to pass the information between the two players. Code for the Game state is designed to track the state of the game board, including the different pieces, as well as track which player's turn it is. Code for the player state is designed to enforce valid moves for the player. This document does not cover the testing for the application.

1.3 Definitions

1.3.1 Game State

Main Game Screen - Refers to the screen that both the client and server see that includes the game board, the pieces, and a button for either player to forfeit

Turn - Defines who is allowed to move a piece at any given moment during a game, if it is one players turn, only that player may move a piece, the other players pieces will be locked at this point in time

Win Conditions - The circumstances under which a game ends and one player wins. The win conditions are one player completely eliminating the other's pieces, in which case they win, or if one of the players forfeit, in which case the other player wins

1.3.2 Pieces

Normal - Normal pieces are the pieces that are put on the board at the beginning of a game. They are represented by circles of a certain color depending on the player that they belong to, either white or black, with the host being the white pieces and the client being the black pieces. The white team will go first. The normal pieces will only be able to move diagonally in the direction away from the side they start on. They will also be able to **jump** over and remove

pieces that belong to the other player if one of their pieces is in an area that they would normally be able to move and there is an open space the space behind the enemy's piece in a straight line from player's piece to the enemy's piece. If there is one or more enemy's piece(s) able to be jumped on a player's turn, the player must jump an enemy's piece.

King - A king is just like a normal piece but the image used has a gold crown in the middle of the circle and the king has the ability to move diagonally towards the area that they originated as well as away.

1.3.3 Acronyms

SAFC - "**S**oftware **A**pplication **F**or **C**heckers"

2. System Overview

2.1 Description of Software

Our checkers application is designed to support 2 players to play from different computers. Users will be able to host or connect to a game with no bearing on gameplay. They will compete in a classic game of checkers with the game ending if either player forfeits or a win condition is met. We are emulating the traditional checkers game and including all of the basic rules.

2.2 Technologies Used

This checkers application will use Computers as their input devices. The computers will communicate over a local network (both must be on the same network).

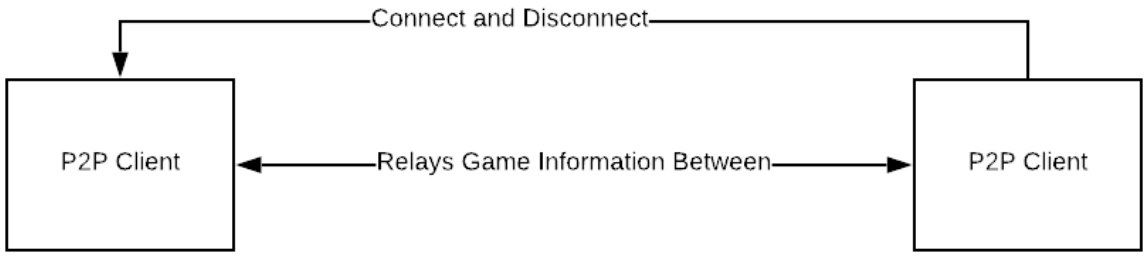
Target platform is Microsoft Windows 10 for the computer. The development environment is PyCharm. Version Control is handled through Git, hosted by Github. We will use Coverage.py for test Coverage and Pylint for static code analysis.

3. System Architecture

3.1 Architectural Design Components

Game State - This system keeps track of the state of the current game. This includes the pieces position on the board, checks if any player has reached the win condition, and determines whose turn it is. It is updated on the P2P Client.

Two P2P Clients - The client system is largely composed of the interface that the player will interact with in order to see the state of the game and make moves. Each client will be tied to a single player. The client will send their move information to the server.



3.2 Design Rationale

Why Peer to Peer Model?

As we only have two clients that need to be connected at any time, we chose to allow the clients to communicate directly with each other, and pass along the information between the clients. This keeps the code much simpler, and will allow two players to play on the same computer easily just by launching two applications.

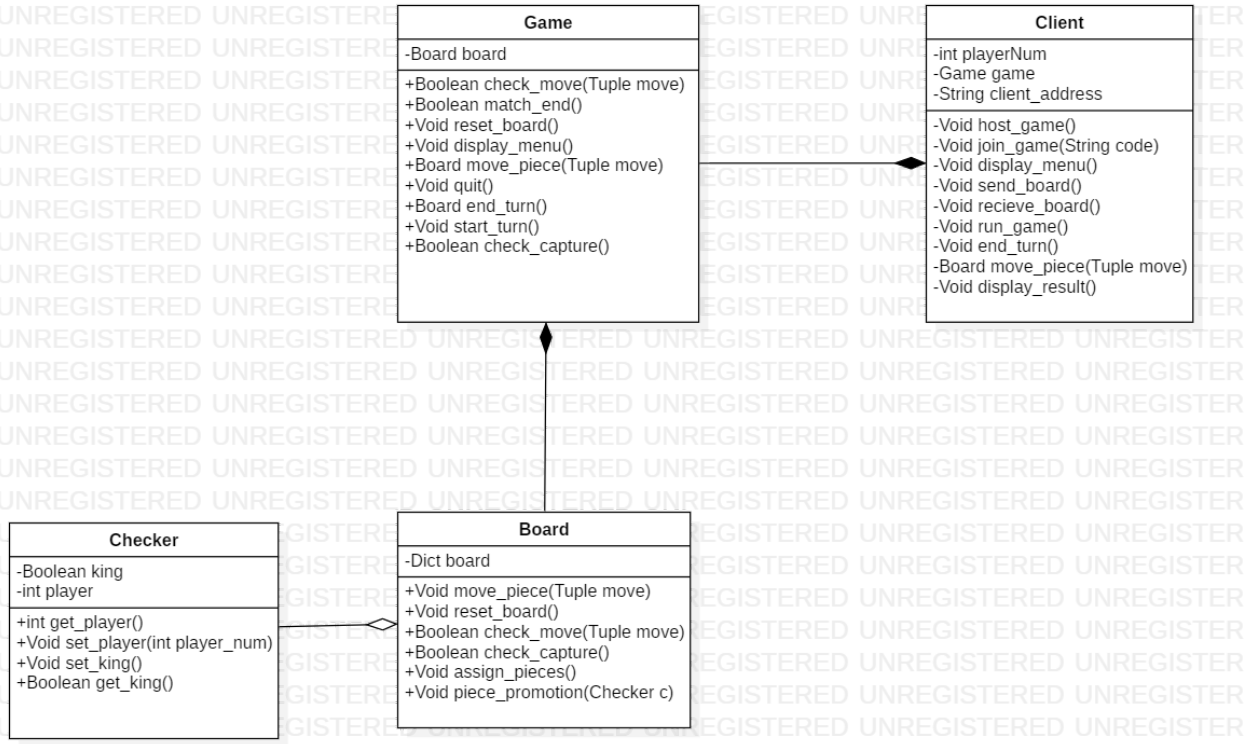
Why only local area network?

By keeping it contained to only working on a Local Area Network (LAN), it will make development less complex to implement. In addition, opening up the game to the internet would create additional networking and security problems that would push the entire project's timeline out of scope.

4. Component Design

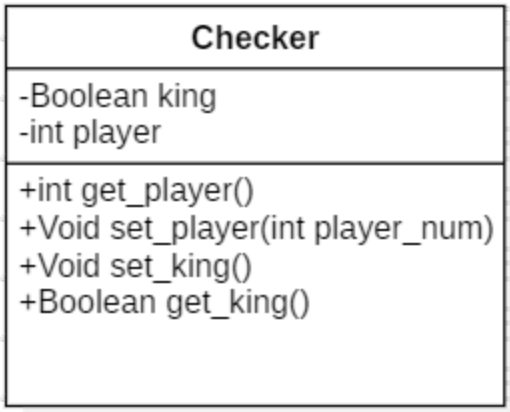
4.1 Overview

In this section, more details on each component are given. For every component listed, a UML and a brief description is given related to its functionality.



4.2 Checker

This is a checker piece. It contains information on which player owns the piece as well as if it is a king or not.



4.2.1 Attributes

Name	Type	Description
king	Boolean	Tracks whether the piece has been kinged
player	int	The player which the piece belongs to

4.2.2 Methods

int get_player()	
Input: None	Void
Output: player_num	int
Description: Tells which player the piece belongs to, 0 for host or 1 for client.	

Void set_player()	
Input: player_num	int
Output: None	Void
Description: Used by the board when initialized.	

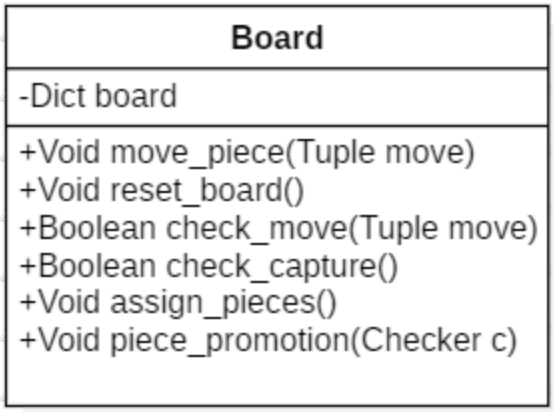
Void set_king()	
Input: None	Void
Output: None	Void
Description: Changes king variable to true. Will not be able to change king variable to	

false.	

Boolean get_king()	
Input: None	Void
Output: True/False	Boolean
Description: Returns the value of the king variable	

4.3 Board

This is the game board which holds all of the checkers and displays them to the client. Checks all moves, and if there's any moves that allows for a capture, forces the current client to make that move. This component is contained within the Game component. Most of the gameplay and logic for it is contained within Board.



4.3.1 Attributes

Name	Type	Description
board	Dict of maps	Keeps track of piece positions on the board.

4.3.2 Methods

Void move_piece()	
Input: move	Tuple
Output: None	Void
Description: Alters the game board to represent a valid move.	

Void reset_board()	
Input: None	Void
Output: None	Void
Description: sets board back to initialization state, with no pieces removed and all pieces in their respective starting positions.	

Boolean check_move()	
Input: move	Tuple
Output: True/False	Boolean
Description: If the move is good, it calls move_piece() to execute the move. Otherwise it returns false, forcing the player to choose a different move/piece..	

Boolean check_capture()	
Input: None	Void
Output: True/False	Boolean
Description: Checks if any capture moves are	

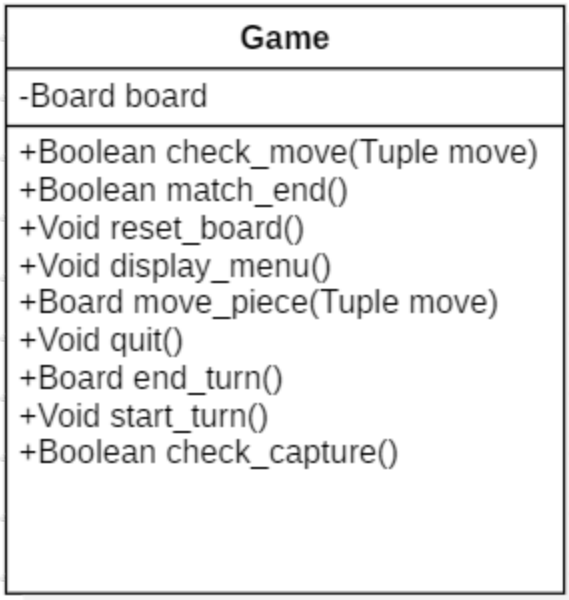
available.	

Void assign_pieces()	
Input: None	Void
Output: None	Void
Description: Assigns the piece to one of the players, either 0 for host 1 for client.	

Void piece_promotion()	
Input: Checker	Checker
Output: none	Void
Description: Changes the value of the king boolean of the Checker given as a parameter to true.	

4.4 Game

The game object contains a board object which contains all the information about the positions of the checkers. It is mostly a wrapper for Board. It is updated after every move, and is passed from client to client.



4.4.1 Attributes

Name	Type	Description
board	Board	Contains the board component

4.4.2 Methods

Boolean check_move	
Input: move	Tuple
Output: true/false	Boolean
Description: Calls Board.check_move() function	

Boolean match_end	

Input: none	Void
Output: true/false	Boolean
Description: Checks if the move that was made was a winning move and returns the result as a bool	

Void reset_board()	
Input: none	Void
Output: none	Void
Description: Calls the Board.reset_board() method	

Void display_menu()	
Input: none	Void
Output: none	Void
Description: Displays the main menu to allow players to connect to each other and start a game	

Board move_piece	
Input: move	Tuple
Output: board	Board
Description: Takes a tuple of the move's coordinates and returns the updated board object. Calls the Board objects move_piece function to do this.	

Void quit	

Input: none	Void
Output: none	Void
Description: Quits the application, closing the program.	

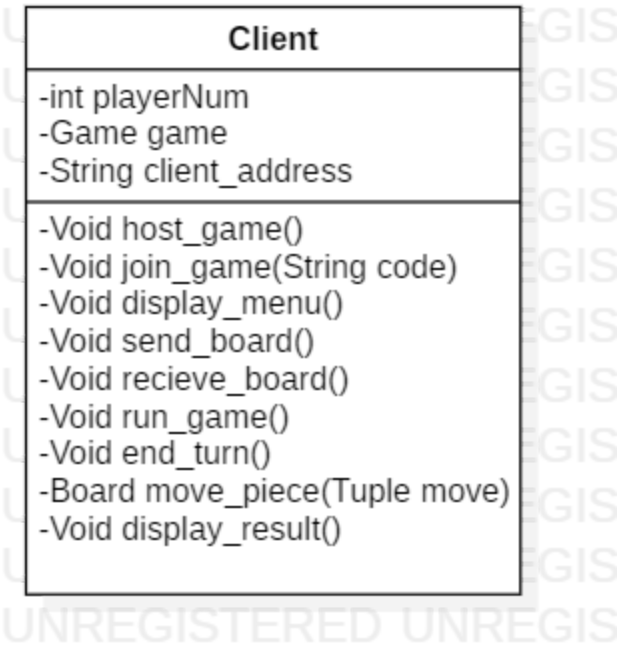
Board End_turn()	
Input: none	Void
Output: board	Board
Description: Called when the player made a move and no more moves are available. Returns the board to be passed to the other client.	

Void start_turn()	
Input: none	Void
Output: none	Void
Description: Starts the player's turn by calling Board.check_capture() to see the available moves for the player then displays it to them.	

Boolean check_capture	
Input: none	Void
Output: true/false	Boolean
Description: Calls the Board.check_Capture() function to see if the player has to make a jump.	

4.5 Client

Connects to another client to pass a Game object back and forth. Most of the Client's functions are wrappers to call the Game component's methods.



4.5.1 Attributes

Name	Type	Description
playerNum	int	Indicates player1 and player2
game	Game	Game object
client_address	String	Address for peer to peer connection

4.5.2 Methods

Void host_game()	
Input: None	Void

Output: None	Void
Description: Creates a client to host a game	

Void join_game()	
Input: code	String
Output: None	Void
Description: Allows a player to join the game	

Void display_menu()	
Input: None	Void
Output: None	Void
Description: Displays menu on player screen	

Void send_board()	
Input: None	Void
Output: None	Void
Description: Sends updated board to other player	

Void recieve_board()	
Input: None	Void
Output: None	Void
Description: Updates board for player	

Void run_game()	

Input: None	Void
Output: None	Void
Description: Initializes a game	

Void end_turn()	
Input: None	Void
Output: None	Void
Description: Concludes a players turn makes calls to update board and send it to other player	

Board move_piece()	
Input: move	Tuple
Output: updated board	Board
Description: Moves piece and returns updated board	

Void display_result()	
Input: None	Void
Output: None	Void
Description: Displays the result of the game once the game ends	

5. Human Interface Design

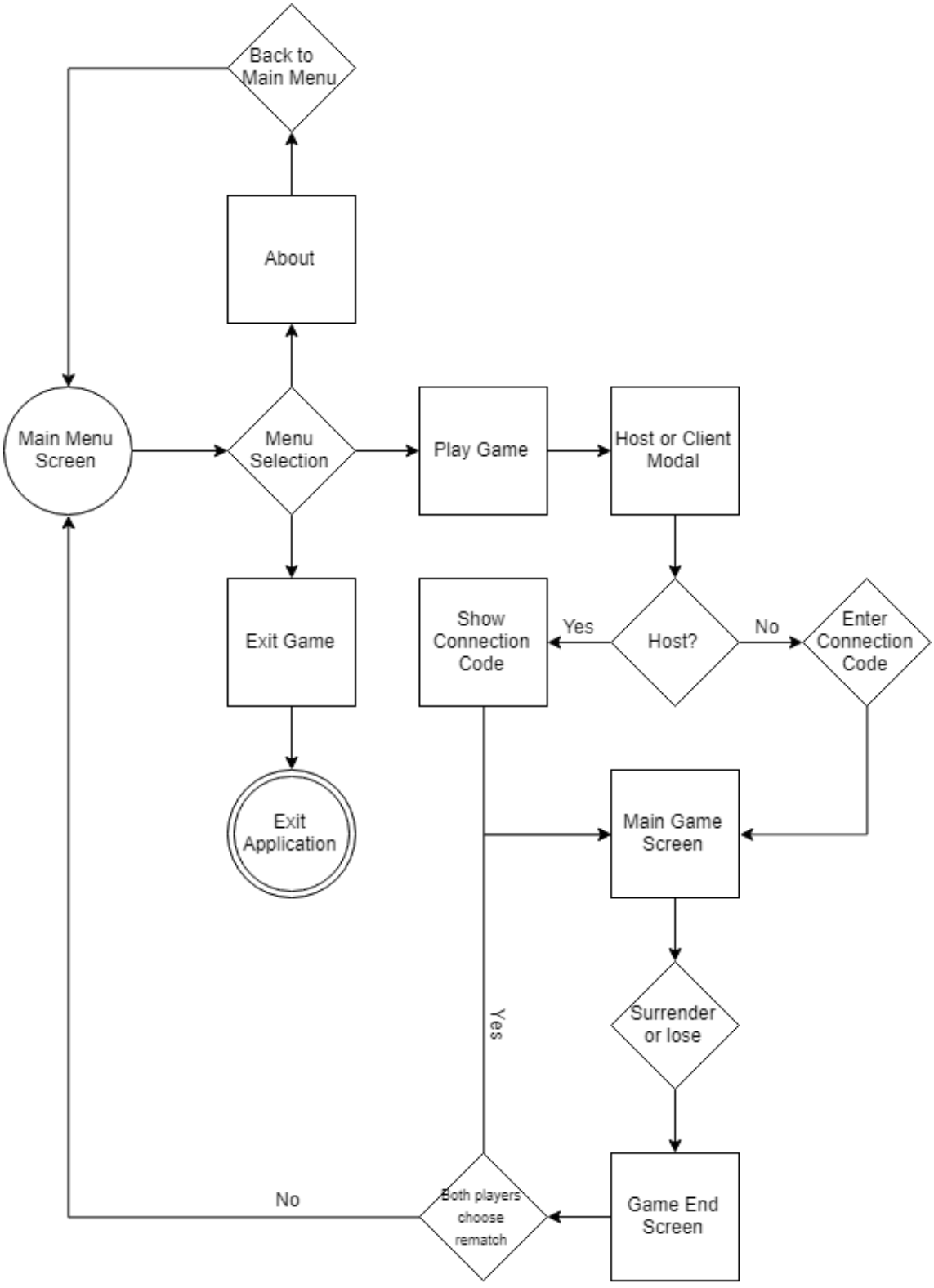
5.1 Overview of User Interface

The UI is explained more in depth in this project's Requirements Document [1]. We will be using the open-source Kivy GUI package to implement our UI [4].

5.2 Screen Objects and Actions

Client - The intended input method for use on the client is a mouse. Additional care should be made to ensure the gameboard is responsive to allow players to make moves that feel right.

5.3 Server and Client Menu Flow



6. References

https://docs.google.com/document/d/1tlQl__QgNoCikX2eYpTMEk8zDbBSKc3feafCkaj7tC0/edit?usp=sharing

https://www.fgbradleys.com/rules/Checkers.pdf

https://www.draw.io/

https://kivy.org/#home