SOFTWARE DESIGN DOCUMENT

for

Checkers

Version 2.0.0 approved

Prepared by:
John Zlotek
Matt Horger
Jake Carfagno
Preet Patel

Team: Big Chungus

July 29, 2019

Contents

Revision History

Name	Date	Reason For Changes	Version
1.0.0	19–07–23	Initial Structure	mhorger
1.0.1	19–07–24	Added content to Introduction, System Overview, Architecture	mhorger
1.0.2	19–07–25	Added assignments for group distribution	mhorger
2.0.0	19–07–29	Finished Document	mhorger, jcarfagno

1 Introduction

1.1 Purpose of Document

This document's purpose is to detail the design implementations of our Checkers program as described on the higher-level by the requirements specification document. This includes all data flows and a lower-level explanation of programming constructs and language mechanics used. We will include diagrams and other sample images that will be used as guidelines for our design process as well.

1.2 Scope of Document

In this document, we split our design implementations into separate categories that are each unique parts in our overall program. We cannot have a program without an underlying architecture or tools defined. We abstract our components into a separate category in order to distinguish them for reuse throughout other parts of our program. We also abstract out our UI design to make the distinction between menus and components in regards to our program and data flows. This document will try to justify that our design choices will satisfy the requirements laid out for our program. We will not prove in this document that these choices are tested and correct.

1.3 Definitions, Acronyms, and Overview of Document-Specific Language

1.3.1 Programming Specifics

1. UI / UX - An acronym for *User Interface* and *User Experience*, respectfully.
2. Swing - A term for the library used for composition of graphical components in Java, built-in.
3. IDE - An acronym for *Integrated Development Environment*, which is a piece of software used to easily format, compile, run code
4. Tux - An acronym for the server provided by the College of Computing and Informatics with an address of *tux.cs.drexel.edu*
5. Async - An acronym used for Asynchronous execution of processes in our program.
6. Process - A term used interchangeably with a thread in Java.

1.3.2 Game Specifics

1. Player - A term used interchangeably with a Client (connection), who plays the game.
2. Lobby - A term used for the data structure used to store client connection and game information.
3. Board - A term used for the 8 by 8 matrix area that players interact with.
4. Piece - A term used loosely to define a checker and any association with it.

1.3.3 Architecture Specifics

1. Client - A term used interchangeably for the port connection a player opens when interfacing with our server. Can also mean the executable program that end-users will run to play our game.
2. Server - A term used for the executable program that consistently handles all client interactions, information, and connections.
3. Heartbeat - A term used for the process that checks to make sure the server executable program is still running.

2 System Overview

2.1 Description of Software

Checkers is designed to be a simple checkers game that can be only played between two players, with a fixed amount of games that a certain number of clients can play concurrently. Each player will have a list of games to join or create and will only be able to interact with the other player via game moves. There will be no spectating games, there will be no chat functionality between clients, and there will be no saved statistics for clients determining their win / lost ratio, games played, etc.

2.2 Technologies Used

Checkers will be exclusively a desktop application, requiring a keyboard and mouse in order to properly input moves and play the game. Our target operating system will be Windows desktop machines, but this program will be able to run on any other operating systems with Java Runtime Environment 12+ installed as long as they have enough storage space and memory to store and run the program on the machine. Our technological stack consists of Java 12 as our programming language with Eclipse Java Neon 3 as our primary IDE. We implement libraries inside of Java in order to provide capabilities that aren't necessarily easy to access using Java. We utilize UNIX servers provided to us by the College of Computing and Informatics in order to serve content from our server application over a secure network.

3 System Architecture

3.1 Architecture Design Components

3.1.1 Networking

1. **Protocols** There will be a couple of protocols being used in our application. In order for clients to connect to the server, they must first provide credentials in order to pass the WPA2-Enterprise security of the Drexel network. There are no trusted CAs on this network. Most network traffic transported between the server and client will be over TCP/IP. We will not be implementing SSL for game data because the data is not necessarily critical to secure for the scope of this project.

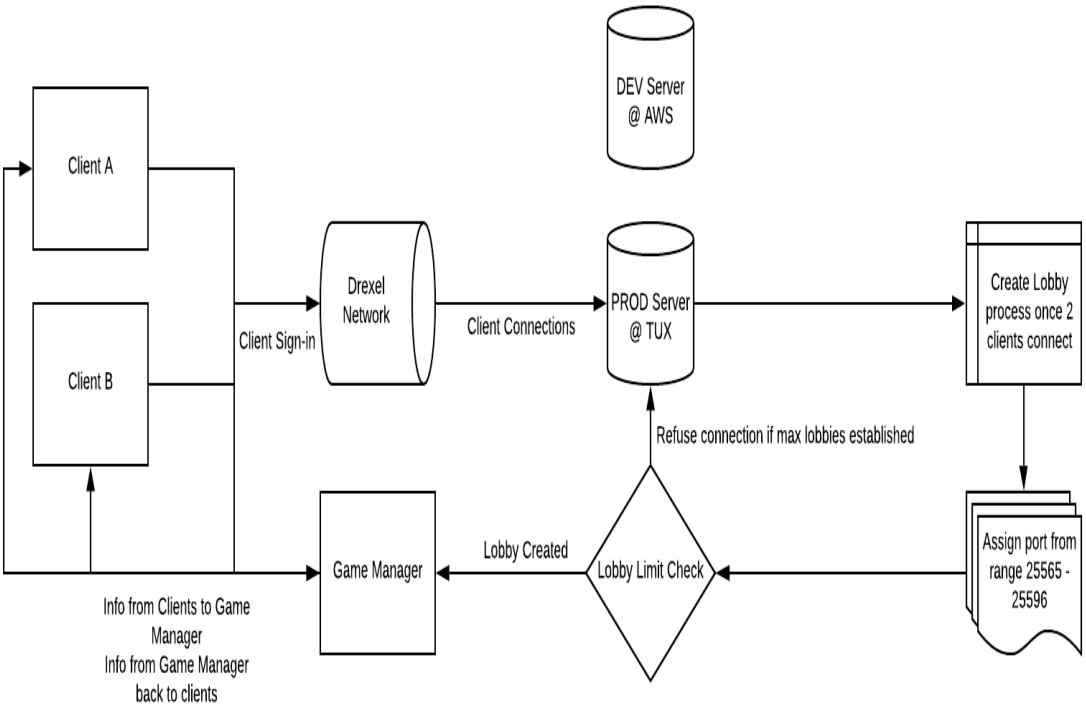
2. **Port Ranges** We will be limiting the port ranges that the server can assign lobbies to within a range of 30 ports. The port range will be from 25566 - 25596. Any ports assigned outside this range can only be attached via a client connection.

3. **Client** Clients will be able to connect to a server with any given port. The client themselves are tied to one player object in terms of game purposes, so they will have control over game pieces, game logic, etc. The bare bones client will only send and receive messages from the server rather than processing heavy amounts of data.

4. **Server** The server will have the priority game engine and processing unit. Clients will send a specific object serialized for the server to validate, and send an appropriate message back to the client. This data flow is appropriate for this project because we shouldn't let clients maliciously edit their code and have them implement exploits. The server will be responsible for maintaining the validity of the game manager / data.

5. **Lobbies** Lobbies will be limited to 5 randomized ports from the given ranges. Lobbies are responsible for maintaining client connections and sending / receiving data from the game manager hosted on the server. Lobbies can be terminated when a game ends, the server will not terminate when a game ends.

6. **Deployment** We will deploy two servers: one DEV and one PROD instance. The PROD instance will be hosted on Drexel's network at tux.cs.drexel.edu, while the DEV instance will be hosted locally for testing on a private AWS reservation, which we will not be sharing the details here. The reason for this is because we don't want clients connecting to DEV and breaking mechanics while we are developing them.

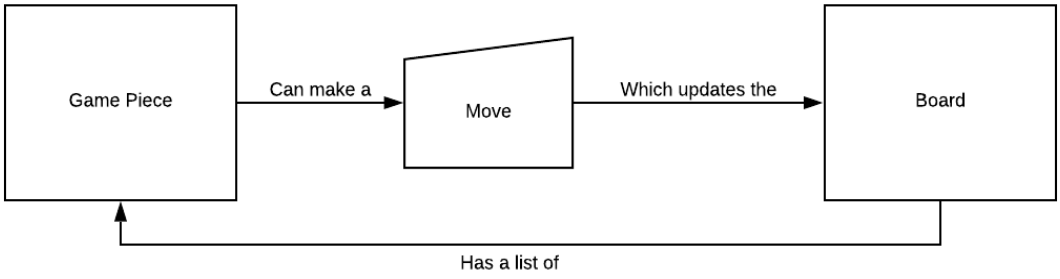


3.1.2 Gameplay

1. **Game Piece** A game piece is the color assignment for a client. A client cannot have more than 12 pieces assigned to them or more than 1 color. A board cannot have more than 24 game pieces active. A game piece is either a standard piece or a king piece, since we simply place a designation on the token that is a king.

2. **Move** A move is a valid operation which a game piece can perform. A game piece cannot have a move, it can only perform a move and have the server check if it is a valid move. Moves are the operational logic to progress the game just as we progress through a state machine in a deterministic automaton; there's always an ending state for a move that determines a winning or losing move.

3. **Board** A board can have up to 24 game pieces, but only stores the information about the color and position they are in. The board is also there for graphical input and output, as a user interacts with a game piece to move it to another position on the board.



3.2 Rationale

3.2.1 Networking

1. **Why host on Drexel's network? Why not use a dedicated EC2 @ AWS instance?**

   We need to host our server on Drexel's network in order to promote security for our users. Clients will have to connect to dragonfly3 or another similar network in order for tux.cs.drexel.edu to validate your Drexel credentials. We chose to implement our server side on Drexel's network instead of AWS for two reasons. The first being cost. Drexel CCI provides these servers for students to use free of charge for classwork. Amazon charges roughly 10 dollars a month USD in order just to reserve an instance, let alone all the requests and run-time usage we would incur charges on. The second reason is security. We could have implemented SSO in AWS or a similar encryption method to validate users, but that is outside the scope of this project. Since we are hosting on Drexel's network, we are given security "free of charge", meaning that we don't have to take extra time out of our project to implement it ourselves.

2. **Why assign ports from a specific range?**

   We need to assign ports from a specific range to not overflow tux.cs.drexel.edu with random ports across a huge range. By limiting our server to only assign lobbies a specific port within a respectable range, this helps us as developers quickly check ports that are open in that specific range, rather than ping the entire ports to see which ones we are using. We won't be necessarily storing active port connections, so by defining a specific range, we will guarantee that we can deduct where a port could be open, and how many theoretically could be open. This design choice of not having a spread of ports is also good for other Drexel users because we will select a port range that is obscure and shouldn't affect other applications.

3. **Why limit the amount of lobbies to be created / joined for clients?**

This one is pretty straightforward: we don't want to crash Drexel's network. By limiting the amount of open ports, in this definition lobbies, we are ensuring that we don't flood our server with too much data to cause interruptions in the network. Therefore, we are going to limit a client to either create a lobby, or join one of the 5 maximum lobbies that can be available at any given moment. If there are more than 5 lobbies attempting to control resources from Drexel's network, the server will automatically shutdown in order to prevent extreme resource allocation in terms of network traffic.

4. **Why implement a heartbeat for the server when you could just run other internal checks?**

   This design piece was hard to agree upon among our team members. Some team members suggested using the screen command in order to run our server 24/7. Others suggested we implement a heartbeat function that checks to make sure that server is still operating. We decided on the second feature for two reason. The first being alerting purposes. When we have a heartbeat that doesn't make a connection, we can use that time to send a warning email to developers instructing them to restart the server. With the screen command, the server might died and we will not know about it until we target it again. The second reason being network resources. By having a heartbeat built into the server, we can easily have a direct access point to check if the network is still communicating with the server. Instead of having a separate client try to ping the server, we have the server handle itself and not use more network resources than needed.

3.2.2 Gameplay

1. **Why create a unique object when simple coordinates can be used?** We do in fact use coordinates in our design of handling moves, but we simply did not want to solely rely on them. By encapsulating these coordinates only if the move is valid, we would be able to first check if a move is valid first before actually utilizing the coordinates. 90% of the time, moves will be valid, but if a user enters an invalid move, we want to process it first before we even need the coordinate to update the board.

2. **Why use an ArrayList¡Vector¿ when you have to convert it to a Vector[]?** For serialization, it is easier to construct an ArrayList of the vector type for our server to process the information. Then, we can always send it back in whatever format the client needs for ease of processing. However, for the sake of server calculations, an ArrayList of valid moves, considering that a move is an object, is a great design choice for serialization and proper handling of data.

3. **Why use a smaller 8 by 8 board?** The more popular versions of checkers have bigger boards to accommodate for bigger pieces, and therefore more in-depth strategies. However, we wanted to stick with the classic design for simplicity.

4 Component Design

4.1 Overview

This section details the underlying implementation of our game components, which will be reused throughout our application. This section will detail how we we implement the requirements of the game logic as well as board reconciliation and player moves using a UML diagram. We will also display some mock-ups of game pieces and the board using different styles.

4.2 Lobby / Game Manager Object

1. Contains

   ArrayList playerPieces

   EventHandler checking if the game has been won

   Active player color playing during the turn

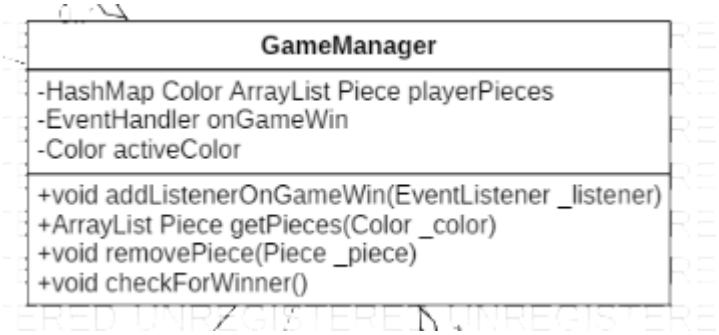
2. Important Functions

   Adds an event listener to check for a game win

   Gets all the piece locations for a given color

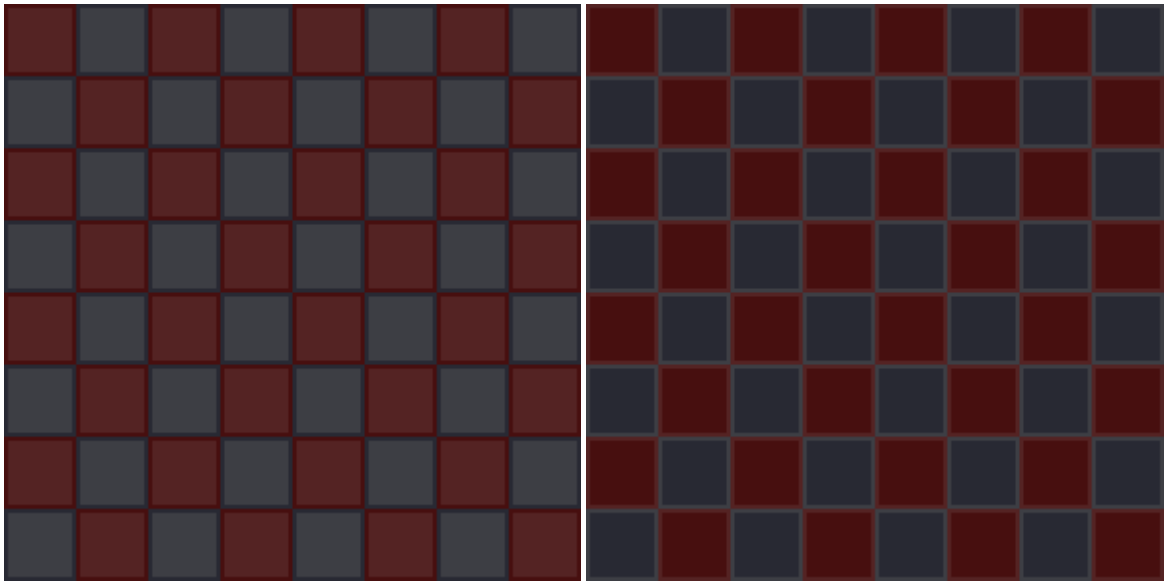
   Removes a piece from the board

   Can hard check for a winner



4.3 Board Design

Below are some style choices for our board. One image is a darker variation, and this could be implemented in further releases for users to customize their board styles.



1. Contains

    Board size

    Matrix of spaces

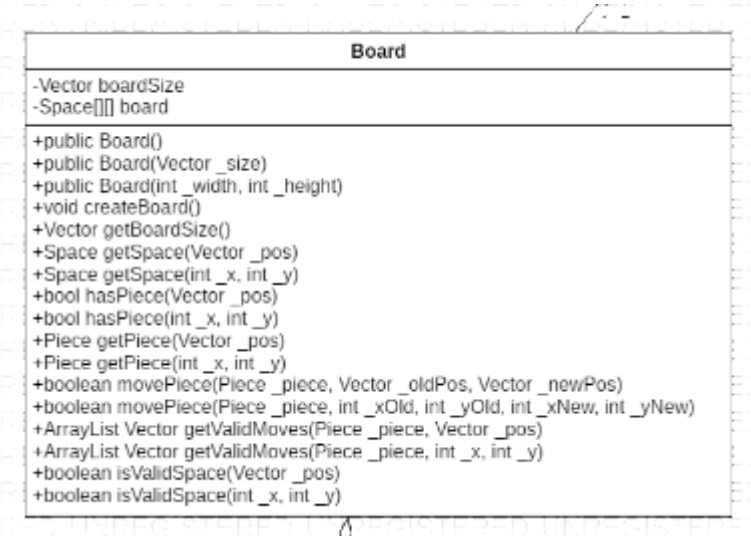
2. Important Functions

    Checks if a space has a piece

    Moves a piece

    Gets valid moves

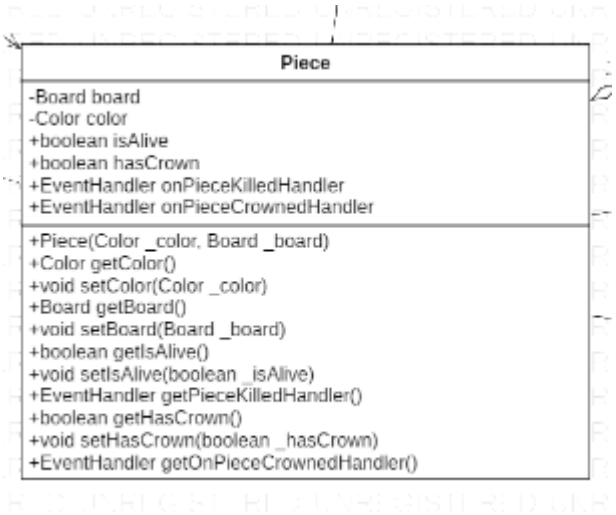
    Checks valid spaces for moves



4.4 Piece Design

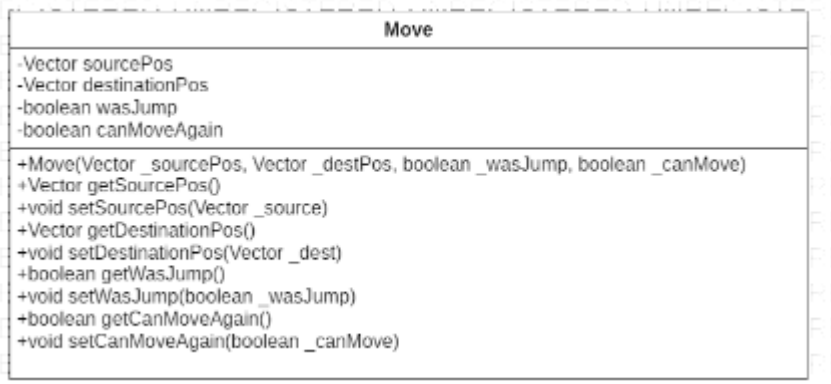
Color choices for our pieces

1. Contains
    - Color
    - Crown Status
    - Two event handlers for crowning / destruction
2. Important Functions
    - Gets active status
    - Gets crown status
    - Gets color status



4.5 Player Move Design

1. Contains
    - Source Position
    - Destination Position
    - Jump Ability
2. Important Functions
    - Gets / Sets Source and Destination Position
    - Gets / Sets Jump abilities
    - Checks if piece can move again



4.6 Player Move Logic

For a player to move, the below steps are taken:

1. The player selects a space on the client-side

   If the space selected does not contain one of their own pieces, then nothing happens

2. The client sends the server the x and y coordinate of the space selected on the board

3. The server uses the Board to create a list of all valid moves from that location

4. This list of possible moves, created as an ArrayList¡Vector¿ is then converted to a Vector[] for sending back to the client over the network

   The Vector class is simply a structure containing an x and y coordinate

   The conversion ensures minimal data transfer for greatest reliability and least waiting time between actions

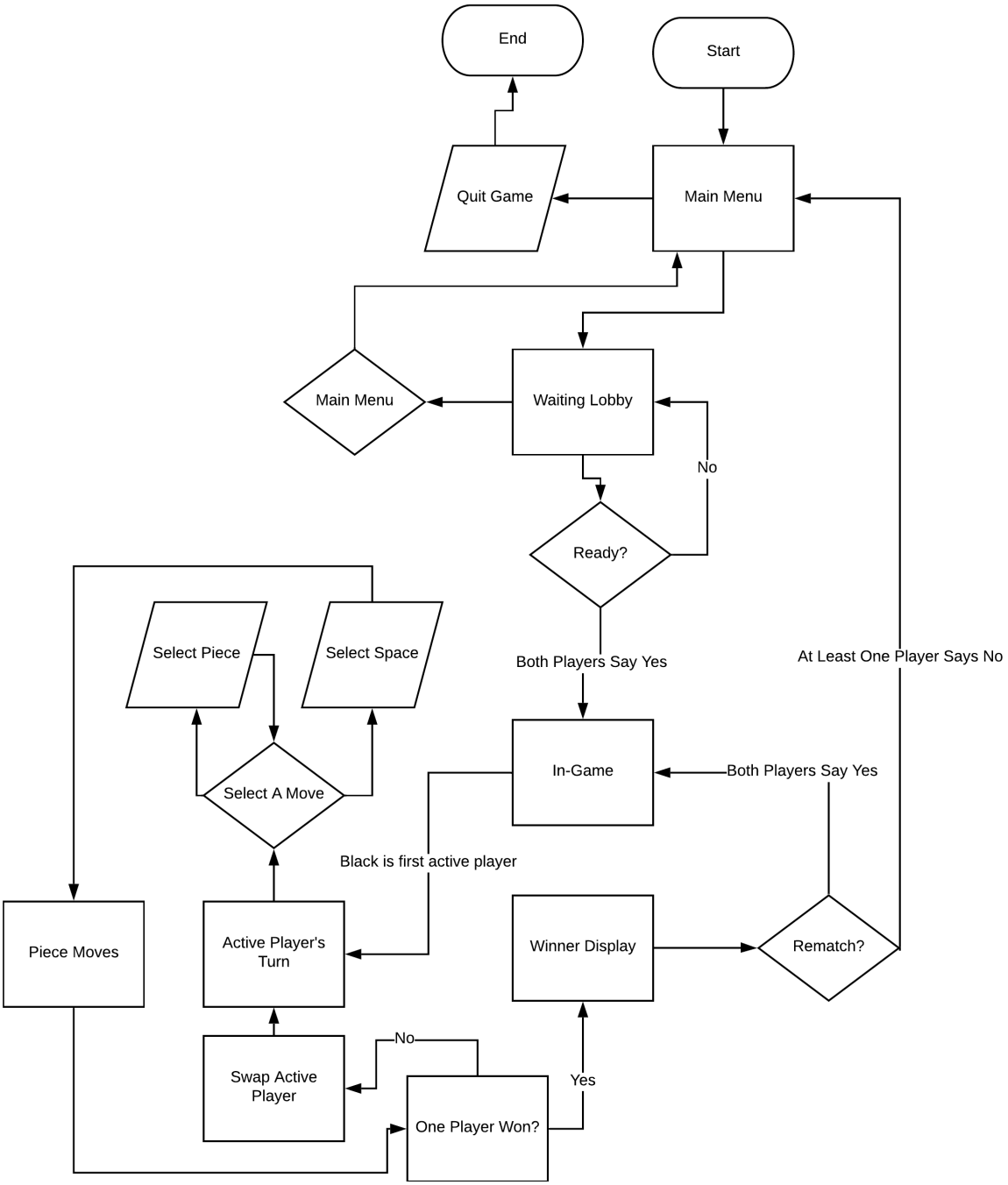
5. The client's display highlights all possible moves the player can make from that selected space

6. The player may click on a highlighted space to finalize that move

   If the player instead clicks on a non-highlighted space, the new space is selected, repeating the process

7. Once the move is confirmed, the client sends the server the x and y coordinates of the source and destination space

8. The server once again validates the move, checks if any additional moves can be made (in the case of a completed jump), and returns to the moving player the two Vectors and a boolean of whether an additional move can be made



Graphic detailing the logic behind player moves

Complete Design UML for Components

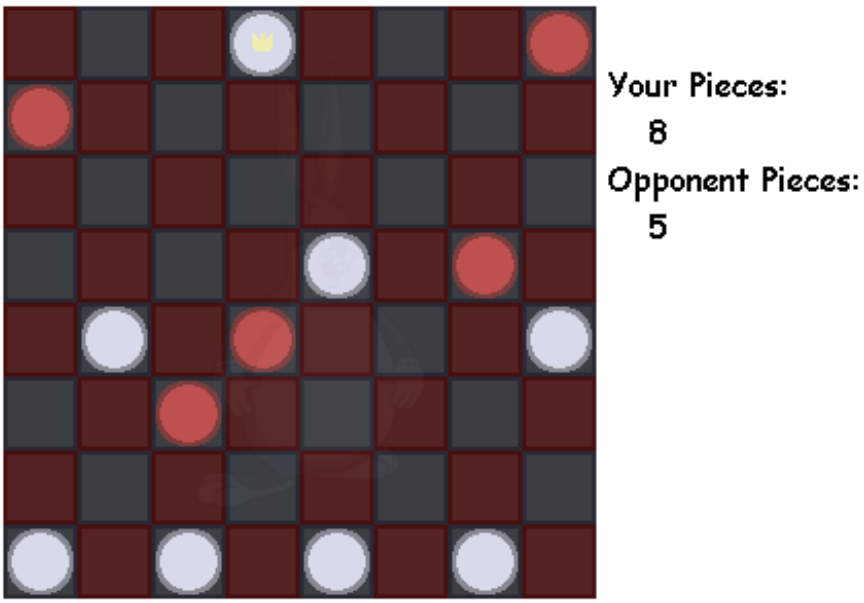
5 UI Design

5.1 Overview

This section details the design flow between screens and menus. We also include screen mockups using the tool *Paint 3D*. These mock-ups are not necessarily final, because the implementation in Swing might differ due to styling and functionality. Nonetheless, we would like to implement the same color schemes and menu design logic to provide a seamless user experience when using our application. This section covers the requirements of game logic and application flow.

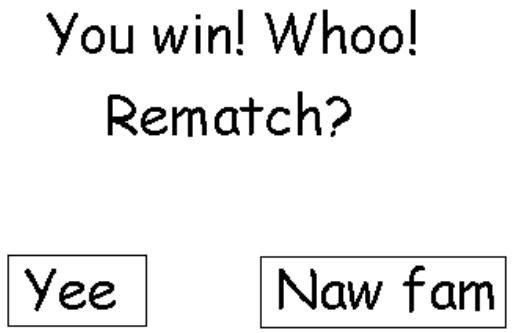
5.2 Screens

1. **Game Screen** The game screen will just simply consist of the board object, with a side menu detailing how many pieces are active on the board. The pieces will be broken down into categories: your pieces and your opponent's pieces.

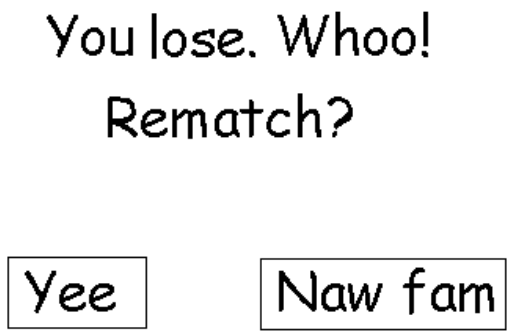


2. **End of Game Screen** There will be only two popup screens at the end of the game: either a player won the game or lost the game.

   Victory Screen The victory screen will prompt the client if they want to rematch the current lobby participant, or to bring them back to the main lobby menu.



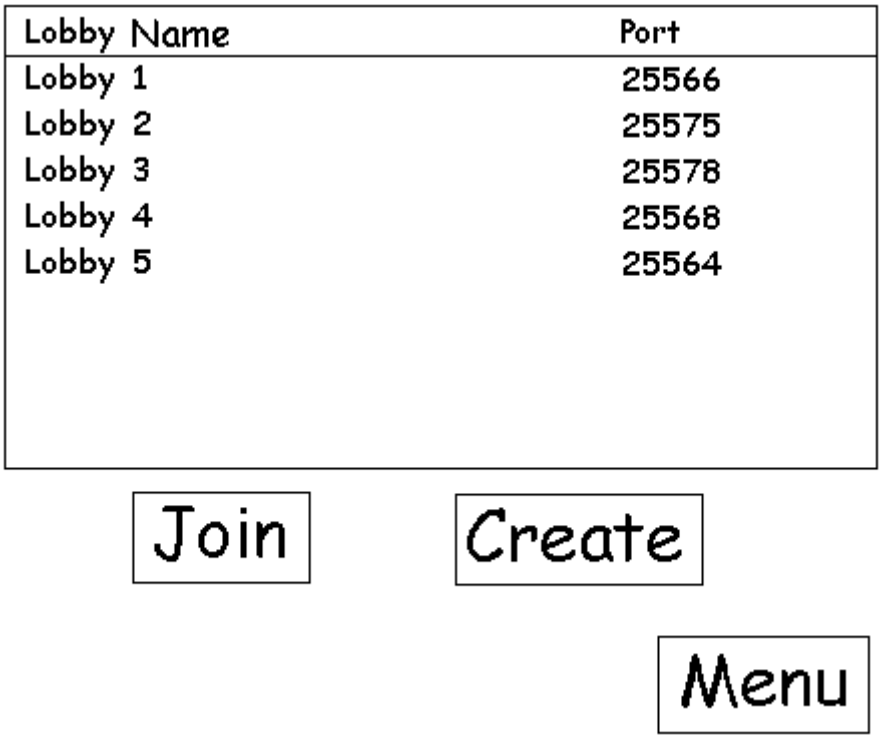
   Defeat Screen The defeat screen is very similar to the victory screen in that it prompts a rematch or return back to the main lobby menu.



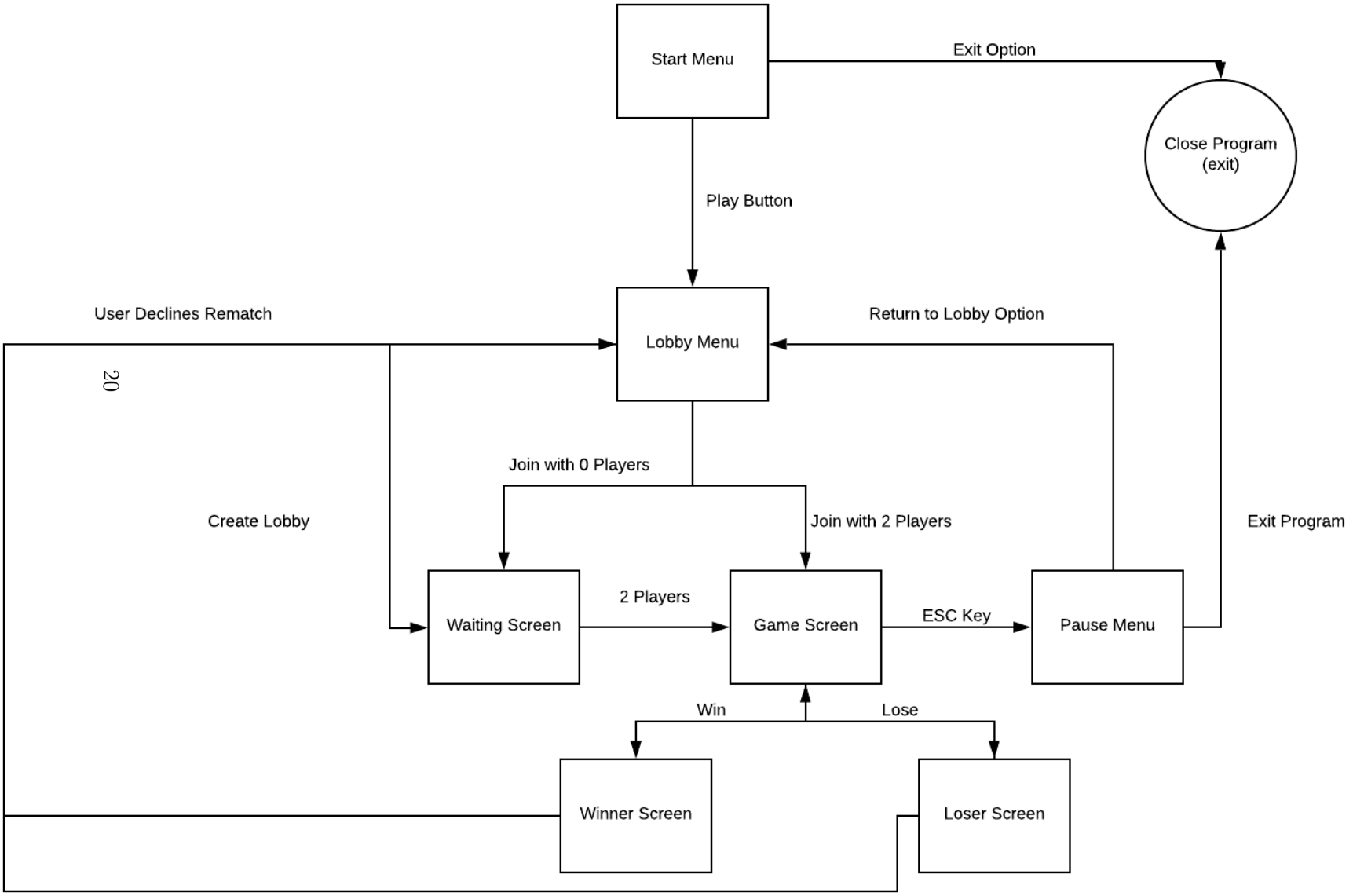
5.3 Menus

1. **Start Menu** This menu will only have two options; a play button and an exit button. The play button will launch the lobby menu if and only if the player can connect to the server. The exit button will close out of the program entirely.

2. **Lobby Menu** This menu will have the list lobbies available for the client to join. If there are 5 active lobbies, the user will be unable to click on Create Lobby. If a user does click on Create Lobby, they will be prompted with a waiting screen for more players to join. If a player joins a lobby without any players, they will see the same screen.



3. **Pause Menu** This menu will have two options; either to exit out of the current game back to the lobby menu or to quit out of the program entirely. This menu option can only be activated when a client is in a created lobby playing the game.


Start Menu
Exit Option
Close Program
(exit)
Play Button
User Declines Rematch
Lobby Menu
Return to Lobby Option
Join with 0 Players
Create Lobby
Join with 2 Players
Exit Program
Waiting Screen
2 Players
Game Screen
ESC Key
Pause Menu
Win
Lose
Winner Screen
Loser Screen

Bibliography

[1] Software Requirements Specifications, *Team Big Chungus*, CS 451-002 Submission 1, 2019.