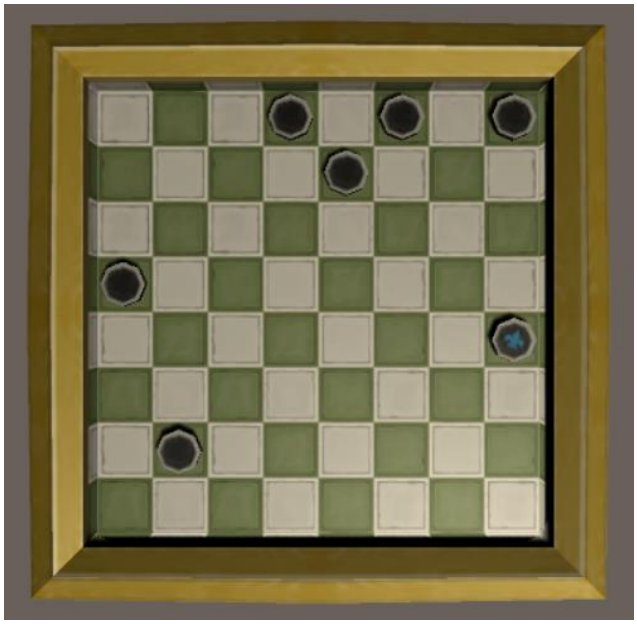


Just van der Linde

Klas - GDV2

Datum – 17-06-2018

Game – Chess Multiplayer

Document – Technical document

# The Game

*Introduction*

This assignment is about making a functional turn-based multiplayer game.
This has proven to be much harder than expected. Before thinking about the multiplayer mechanics, I assembled a game with great visuals and concept, which was working great.

'The only thing left right now is implementing the network functionality, that can't be that hard right?'

It turned out implementing network functionality on an already existing game without thinking about the multiplayer is harder than expected. I got everything working except for a few mechanics on which I struggled for quite some days.
I had little knowledge about what everything the Photon add-on was taking care of was actually doing in the background, which made it hard for me to get the logic behind it all.

While doing some research, I discovered a course on Unity multiplayer using only C# and TCP, which was great for understanding the basics of network communication, so I proceeded to start from scratch and follow the course.
The course also pointed out how to think of network purposes by adjusting your scripts towards it.

*Gameplay*

The gameplay doesn't need much explanation, as it is chess and most people know all rules, but here are the basics:

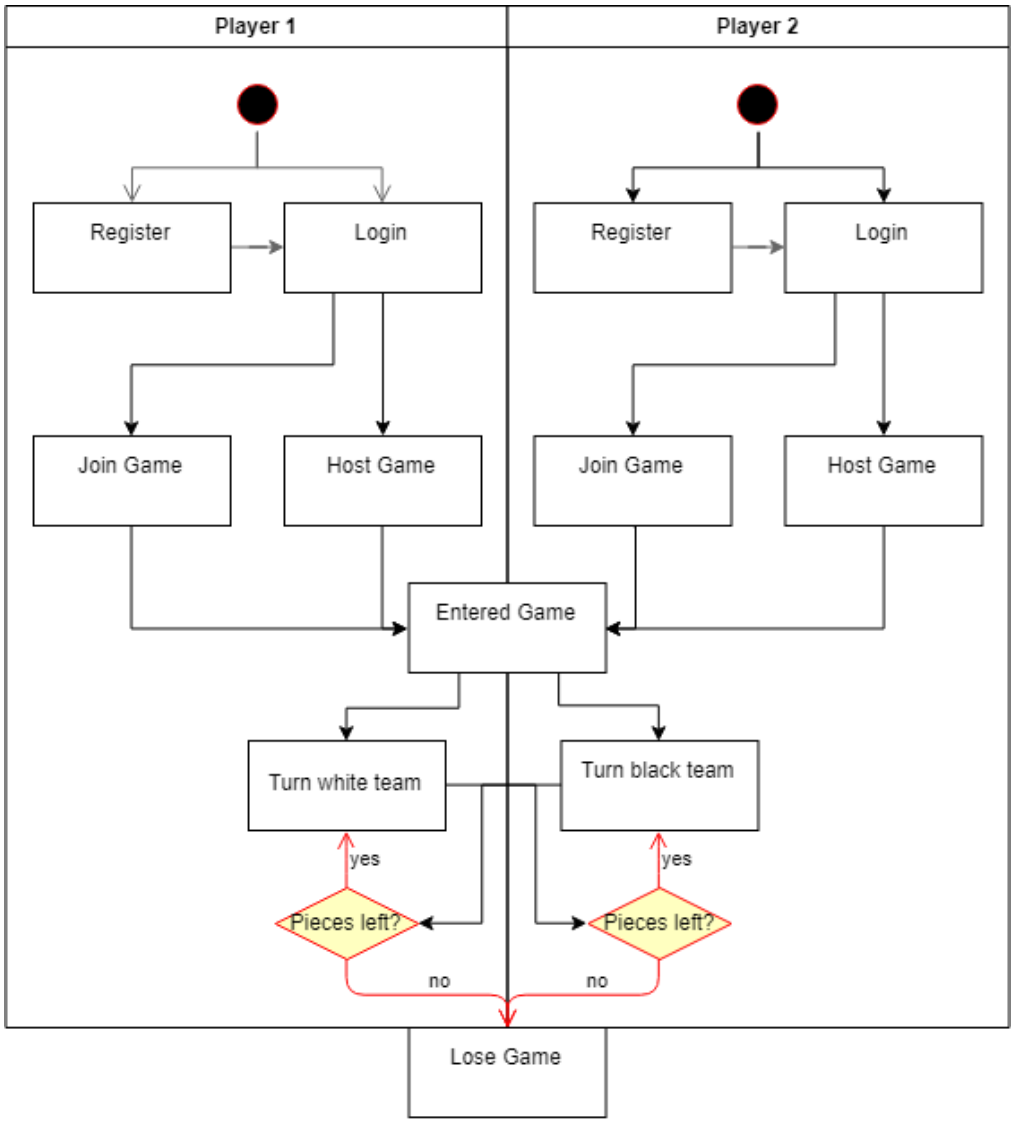
The host of the game starts as the white pieces, while the other team using the black pieces has to wait. After a turn is made, the black team can make a move.
If a piece is in position to 'kill' another piece, the move has to be made, all other moves are blocked out. If a player reaches the opposite end of the board, the piece turns into a 'king piece', making it able to move in all directions. The player to lose all pieces first loses the game.

If a player managed to win the game, a win score is submitted to their account which can be found on the website.

*Improvement*

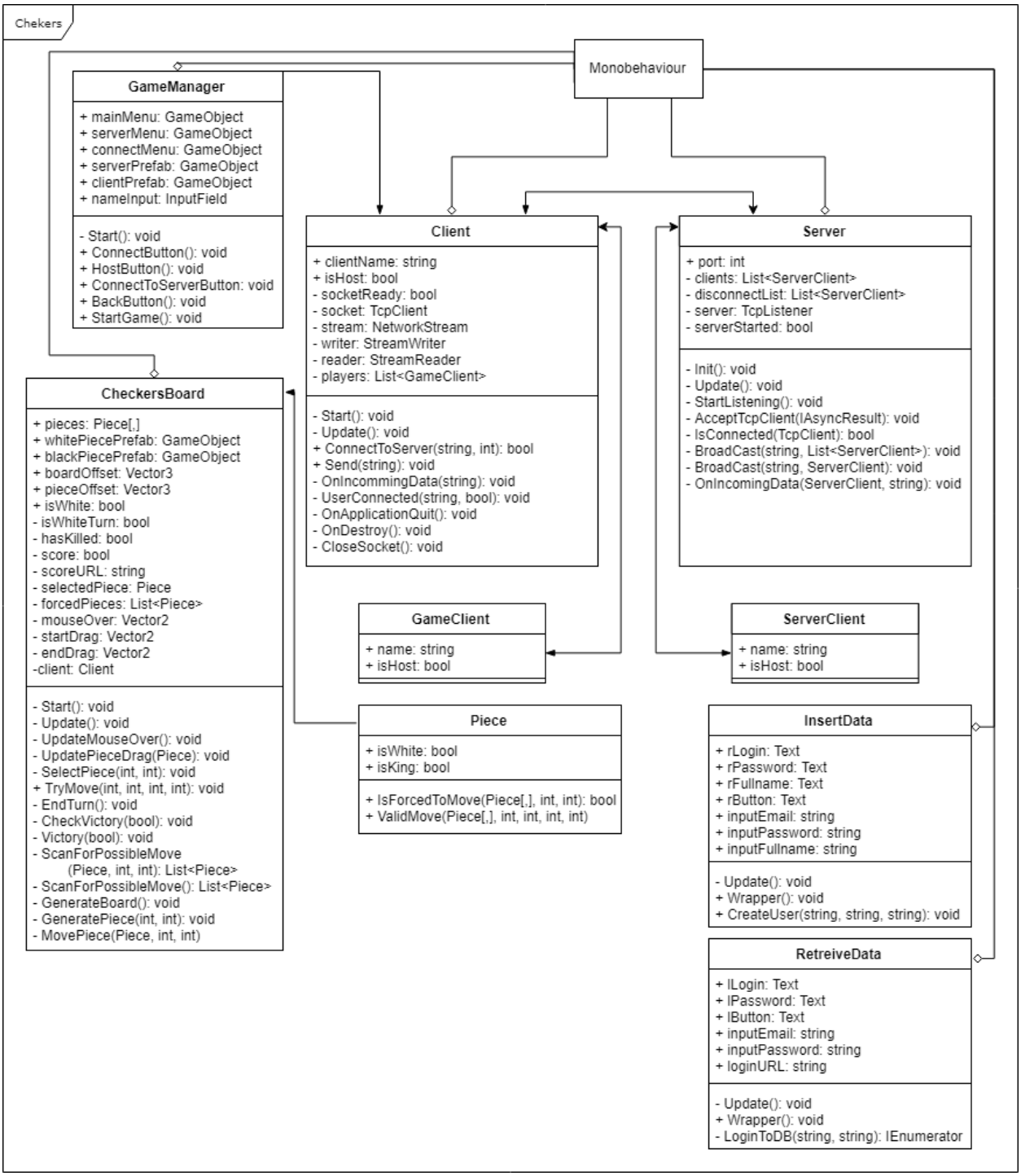
One point of improvement worth noting is the communication between Unity and my database. This implementation is old and can be much better/safer. It's because of the scarcity of time that I chose to use this old implementation.

# Activity Diagram




When starting the game, the player is given the opportunity to login to their account or register a new one. Once logged in the user can host a new game or join an existing one. The player hosting the game starts as the white team, which means getting the first turn. Once a player has no pieces left, he/she loses the game and a lose score is submitted to their account. A win score is submitted to the opposing teams' account.

# UML Diagram


Chekers
Monobehaviour
GameManager
+ mainMenu: GameObject
+ serverMenu: GameObject
+ connectMenu: GameObject
+ serverPrefab: GameObject
+ clientPrefab: GameObject
+ nameInput: InputField
- Start(): void
+ ConnectButton(): void
+ HostButton(): void
+ ConnectToServerButton: void
+ BackButton(): void
+ StartGame(): void
Client
+ clientName: string
+ isHost: bool
- socketReady: bool
- socket: TcpClient
- stream: NetworkStream
- writer: StreamWriter
- reader: StreamReader
- players: List<GameClient>
- Start(): void
- Update(): void
+ ConnectToServer(string, int): bool
+ Send(string): void
- OnIncommingData(string): void
- UserConnected(string, bool): void
- OnApplicationQuit(): void
- OnDestroy(): void
- CloseSocket(): void
Server
+ port: int
- clients: List<ServerClient>
- disconnectList: List<ServerClient>
- server: TcpListener
- serverStarted: bool
- Init(): void
- Update(): void
- StartListening(): void
- AcceptTcpClient(IAsyncResult): void
- IsConnected(TcpClient): bool
- BroadCast(string, List<ServerClient>): void
- BroadCast(string, ServerClient): void
- OnIncomingData(ServerClient, string): void
CheckersBoard
+ pieces: Piece[,]
+ whitePiecePrefab: GameObject
+ blackPiecePrefab: GameObject
+ boardOffset: Vector3
+ pieceOffset: Vector3
+ isWhite: bool
- isWhiteTurn: bool
- hasKilled: bool
- score: bool
- scoreURL: string
- selectedPiece: Piece
- forcedPieces: List<Piece>
- mouseOver: Vector2
- startDrag: Vector2
- endDrag: Vector2
-client: Client
- Start(): void
- Update(): void
- UpdateMouseOver(): void
- UpdatePieceDrag(Piece): void
- SelectPiece(int, int): void
+ TryMove(int, int, int, int): void
- EndTurn(): void
- CheckVictory(bool): void
- Victory(bool): void
- ScanForPossibleMove
(Piece, int, int): List<Piece>
- ScanForPossibleMove(): List<Piece>
- GenerateBoard(): void
- GeneratePiece(int, int): void
- MovePiece(Piece, int, int)
GameClient
+ name: string
+ isHost: bool
ServerClient
+ name: string
+ isHost: bool
Piece
+ isWhite: bool
+ isKing: bool
+ IsForcedToMove(Piece[,], int, int): bool
+ ValidMove(Piece[,], int, int, int, int)
InsertData
+ rLogin: Text
+ rPassword: Text
+ rFullname: Text
+ rButton: Text
+ inputEmail: string
+ inputPassword: string
+ inputFullname: string
- Update(): void
+ Wrapper(): void
+ CreateUser(string, string, string): void
RetreiveData
+ lLogin: Text
+ lPassword: Text
+ lButton: Text
+ inputEmail: string
+ inputPassword: string
+ loginURL: string
- Update(): void
+ Wrapper(): void
- LoginToDB(string, string): IEnumerator

The UML is divided in four areas:

*Network section.*

This includes the server and client, and the communication between each other.
The communication goes through a socket opened by each client.
The client sends information through a StreamWriter. This information contains formatted Vector2 data that represent the move named client made.
The server receives the data, processes it and broadcasts it to all receivers.
The client receives formatted strings which can be translated to moves on the gameboard.

*Checkerboard section.*

This section includes all of the game mechanics, as well as the Vector2 information about each step of a move which can be easily transferred trough the network.

*GameManager.*

The GameManager mainly takes care of the UI element flow and the prefabs.

*Database section.*

This part takes care of the logging in and registering of users.