Code Design:

Our code has 9 primary classes:

- For the game logic we have Piece, Player, and Game
- For networking we have ChessClientObj and ChessServerObj
- For GUI we have ChessWidget, GameOverWindow, MainWindow, and MenuWindow

Additionally, we also have classes King, Queen, Bishop, Rook, Pawn, Knight that inherit the main Piece class.

Piece is an abstract base class that contains shared functions such as returning which player the pieces belong to, return back the possible moves vector (for GUI purposes), set if it's captured or not, and getter functions related to the shared values.

Each of the piece's derived classes host their own (overridden) implementations of list_possible_moves - a function that calculates the possible moves for the piece and saves it to a 2D array. This function is called right after a player makes a turn to calculate possible moves (this includes the moves, if made, can be checkmated, but these will be deleted in a further point).

The Game Class consists of most of the logic pertaining to movement validity checks and actual movement on the board. It also covers the Check calculations, identifies players that are currently under Check, and from whom.

The main window is the window that the users will spend the most time on. It consists of the board that the users are playing on, their respective scores, whether they are a bot or not, and whether they are currently playing or waiting, or if they are checked, checkmated, stalemated, or disconnected. Hence, the display update code has been implemented in the main window class. The main window has a pointer to the chess widget, which acts as an interface between the game and the main window. The chess widget is meant to handle the click responses, the structure of the initial map, the creation of the game and so on, and hence plays a major role in our implementation. The chess widget handles clicks, and these clicks trigger a move, and the calculation of the valid moves of the next active player.

The game over window is a screen that displays the current winner or winners, and prompts the user to close the screen.

ChessClientObj:

This class maintains the server connections. See Server Design below for more details, however, to discuss briefly, ChessClientObj has a pointer to the message box that pops up when the server connections, the MainWindow to the are finalized and can close the popup and open the MainWindow which hosts the ChessWidget in it.

ChessServerObj:

This class maintains the server connections on the server side. See Server Design below for more details.

Server Connection Design:

During connection initialization with the client, the server sends in the number of players currently within the server. Based on that number, the client completes the init_connection function and based on the value received, the game asks if they want to create bots (if they're the first person connecting to the server) or just join the server as an individual player.

Based on the value received, the client pre-registers the number of players to the server to keep a track of the number of players connected. For the computer hosting bots, the players are also registered at this point.

Once the server reaches 4 players, the server broadcasts "START" as a signal to start the MainWindow.

The next instructions come from clients - when they move a piece, they send a move message to the server, which it broadcasts to everyone.

When a user disconnects from the server, the server deletes the socket from it's containers and broadcasts "dead x" where x is the player id number.

There is a disconnect function for cases where an user wants to play in networking again, to force reset the server.

Each message has a signature at the end of the message- with the IP address that they're connecting from and a random message (based on the Mersenne generator). This is saved in the client, which checks if the message is from them or not. If it is, then they don't process that message.

Code Components, Integration, and Functionality:

Our code is compartmentalised, but comes together effectively to form a functioning end product. Our network features exist almost independently from the rest of the game. However, the GUI and the backend logic have ties at various points.
Furthermore, many of the classes have references to each other, in order to access elements from one another. For example, pieces have a reference to the player object to whom they belong, and the game object that they are on. The players have an array of pointers to the 16 pieces that they own, and a reference to the game object that they are playing on. Lastly, the game object has an array of pointers to player objects that are playing on them, and a board of Tile objects which have pointers to the pieces that are currently on them.

The individual piece derived classes store the possible valid moves that they can make at any time. They have individual functions to take account of their different movement mechanisms. For example, the King class computes whether or not it can castle depending on the player it belongs to, and whether or not it has moved. The game object has the role of

governing the different moves, and invalidating any illegal ones before the player makes a move. The game is also responsible for the movement of pieces, and tracking captures. The Game class is designed in order to best capture the ruleset of the 4-Player Chess game. Upon starting, the first player is allowed to make any move that their list possible moves function allows them to. Based on this move, the next player's possible moves are calculated, and control of the board is granted to the next player in line, who is limited to the legal moves that were calculated before the start of their turn. If the next player is a bot, their move is randomly chosen using a pseudo-random number generator (PRNG), and carried out on the board as usual.

The valid moves for a player are identified using an algorithm that simulates all possible moves of that player on a less memory intensive copy of the board, and checks whether the king is under attack in any of those board variations. If the king is under attack after any possible move of a piece, that move is deemed invalid and removed from that piece's candidate moves list. Finally, the number of valid moves is also kept in track, and if the current player has no legal moves remaining, the current board is analysed to determine whether the player is currently under check or not. If the player is under check, they are checkmated, otherwise they are stalemated.

The game object also plays the role of tying up the frontend with the backend, since the game itself is created in the MenuWindow class instance. It has a pointer that points to the main window, since it needs to access and modify the various UI elements in the MainWindow form. The game also has a pointer to the client.

The game server runs completely independently from the rest of the code. It, however, is integral to the networking aspect of our program, which further enhances the separation of this code with regards to functionality.

Reusability of Code:

Server and Client classes can be reused in any 4-player move-enabled game, so long as the necessary functions in game and windows are implemented and pointers to those are passed to the Client Class

The GUI implementation can be reused so long as the board size is edited. In fact, editing the board to match an 8x8 instead of a 14x14 can make the GUI reusable for a 2 player Chess game instead. Furthermore, this board can also be reused for other tile-based games like Checkers.

The menu window can be directly reimplemented in a 2-player chess game without any major revisions, so long as the game constructor is modified to suit a 2-player format instead. Even most of the assets are reusable, and slight modifications to the code can resize the board as appropriate.

The piece objects can also be reused in corresponding 2-player Chess game implementations so long as the possible moves array is resized. All of this makes for code that is moderately reusable for similar projects in the future. In fact, apart from the pawn and king, none of the piece classes require any major revisions to implement in a corresponding 2-player Chess game.

Moreover, the logic to verify whether a certain orientation of the board leads to a check for the current player can essentially remain constant. The only changes needed would be to tweak the encoding of the copy of the board.

The game over screen can also largely remain the same, and hence the source file of the game over window needs simple tweaks to make it eligible for a 2-Player outcome. The logic for deciding the winner of a game, however, is different for 2-Player chess.

ChessServerObj:
This class is the most reusable class in the project and can be applied to any 4-person board game that requires piece movements.

References

- The chess pieces used were received from https://www.figma.com/file/BKcHIJHKvDL8hopfygWZzT/Chess-Simple-Assets-(Community)?node-id=2%3A331
- The random number generator is based on https://en.cppreference.com/w/cpp/numeric/random/mersenne_twister_engine
- The tutorials that we used to learn Qt and Networking are
- Game Programming using Qt 5 Beginner's Guide, Strakhov, Packt Publishing
- https://www.youtube.com/watch?v=BSdKkZNEKlQ
- The rules of the game were adopted from Chess.com's implementation
- We also extensively used Qt's documentation sites for self-help.