CHESS OVER A NETWORK

By Thomas Clarke

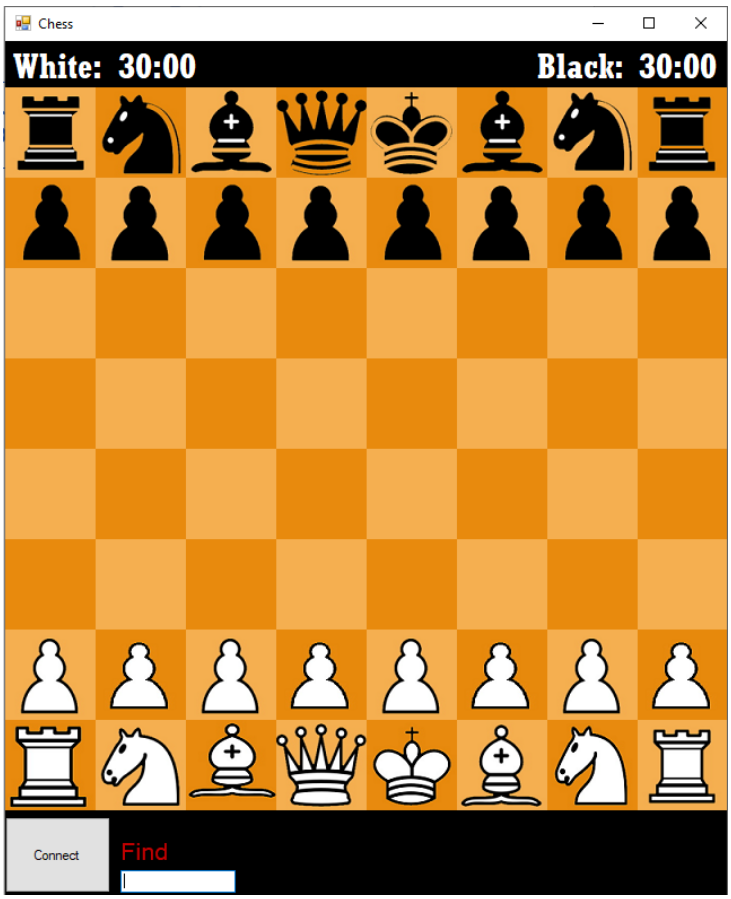


Table of contents

Analysis 5

Client 5

Background to the Problem 5

Interview with client 5

Investigation of current applications 6

Chess.com 6

Ranking 7

Rules of Chess 7

Standard Moves 7

Pawn 7

Rook 8

Knight 8

Bishop 8

King 8

Queen 8

Blocked/Taking 9

Special Moves 9

Pawn Promotion 9

En passant 9

Castling 9

Special Board states 10

Check 10

Checkmate 10

Stalemate 10

Winning, Losing and Drawing 10

Conditions for a win/loss 10

Conditions for a Draw 11

Clocks 11

Design Specification & Brief 11

Design Brief 12

Design Limitations 14

Time constraints 14
Server Hosting 14
Design 15
Overview of system 15
OOP Class Design 18
Data flow between classes 19
Description of Classes 19
Drawing the Board 20
Representing the chessboard as a multi-dimensional array 20
Table of constants 21
Moving Algorithms 23
Legal & Potential Moves 23
Finding all legal moves 23
Highlighting the squares 27
Getting potential moves 27
Pawn 27
Rook 28
Knight 28
Bishop 29
Queen 29
King 30
Making the move 31
Flow of game 34
Determining Check, Checkmate & Stalemate 36
GUI/HCI 41
Server-Client Model 45
Technical solution 46
Data & Modules in classes 46
Techniques used 47
Styles 47
Key Modules and variables 50

Client Side Code 51
Game Class 51
Chess class 62
Client Class 67
Server Side code 69
Server class 69
Testing 72
Testing moves. 72
PAWN 72
Link to test: https://youtu.be/pRzhX8CImeY 72
Rook 73
Link to video: https://youtu.be/ErhIahTt87I 73
Knight 74
Link to video: https://youtu.be/3Df5uIp8CFI 74
Bishop 74
Link to video: https://youtu.be/4LdwaRhWe8M 74
King 75
Link to video: https://youtu.be/GOeCa8BmsLc 75
Queen 76
Link to video: https://youtu.be/l4rnuMiU2M0 76
General (All pieces) 77
Board States 77
Board States 77
Connection with server 79
Connection: see testing screenshots at end of document for proof. 79
Holistic tests 80
Game 1: Adolf Anderssen – Lionel Kieseritzky, 1851 80
Game 2: Paul Morphy – Duke Karl/Count Isouard, 1858 80
Game 3: Mikhail Botvinnik – José Raul Capablanca, 1938 80
Game 4: Donald Byrne – Robert James Fischer, 1958 81
Game 5: Anatoly Karpov – Veselin Topalov, 1994 81
Game 6: Magnus Carlsen – Sergey Karjakin, 2012 81
Evaluation 83

User feedback 85
Response to user feedback / Ideas for improvements 85
Test Screenshots 87

Analysis

Client

Name: Eve Clarke

Occupation: University Student

Contact: 33clarkee@gmail.com

Background to the Problem

My client is Eve Clarke, an enthusiastic chess player. She plays for leisure, with family and friends, and competitively at her university's chess society.

She is currently unable to play chess with her friends, after moving back home due to the COVID-19 pandemic. Instead, she simply plays in person with her family using a physical chessboard.

Programs allowing two users to play chess against one another have existed for a long time. Many of the applications that allow network play are hosted on a website, meaning there is no guarantee that these services will exist forever. Whilst most of the websites allow users to play free of charge, they often have a monthly subscription model, restricting the number of games the user can play in a certain time period, to incentivise payment from the user.

I have asked my client to use a free web application called "Chess.com", so I can ask her what she thinks about it in an interview.

Interview with client

Q: What was your overall experience with the application "Chess.com"?

A: I think it was a good application, I used it to play chess with my friends still in Birmingham from Hertfordshire. The connection was quick & the design made it easy to use.

Q: What are some of the application's notable qualities?

A: I like that you can change the design of the chess pieces. I like that the design is simplistic and not too complicated to use.

Q: Was there anything you did not like about the application?

A: The program highlights squares with a marker, I found this quite difficult to see as it is a transparent circle, I would prefer the program to highlight the square entirely to make it easier for me to see.

Q: What features would you add to the program?

A: I would like a system that used our society's ranking system, as the website had its own.

Investigation of current applications

Chess.com

The chess game is well-made. Players select a piece and are shown the squares that it is possible for it to move to. It is proprietary software; therefore, I am unable to access the source code to figure out how the algorithms involved in moving the chess pieces works.

The GUI is very clean. I think the spots highlighting the possible moves should be made bigger and darker, so they have a larger contrast with the background. This would allow people with visual impairments to better be able to use the application.

Users can resign or offer a draw to the other player if they do not want to play the rest of the game.

We are told lots of extra information about the game, including the full list of moves that have been made up to that point, as well as how much time each player has left before a "flag fall".

Ranking

In the interview with my client, she mentioned that the chess society have their own ranking system, in which everyone's rank is reset each month. The following formula is used:

$$Ranking = \frac{\text{Sum of Opponent's rankings} + 400(\text{Win} - \text{Losses})}{\text{Total Number of Games Played}}$$

Credit: https://en.wikipedia.org/wiki/Elo_rating_system#Performance_rating

This algorithm is a variant on one of the many FIDE ranking system. It is taken from an article on Wikipedia which simplifies the official FIDE ranking system.

The society is considering moving to another ranking system, that would allow them to carry over their scores each month. The system will use the following formula:

$$Rank_{New} = Rank_{Old} + X * (Result_{Actual} - Result_{Predicted})$$

$$Result_{Predicted} = \frac{1}{(10^{\left(-\frac{dR}{600}\right)} + 1}$$

$$dR = Rank_{Old} - Rank_{Opponent}$$

Rules of Chess / Current System

Below I have detailed the rules of chess that are set out by the FIDE handbook, as this is the rules by which the chess society play by.

This document will be sent to my client to ensure that the descriptions of the laws are correct according to the Warwick Chess Society.

Standard Moves

This diagram shows how the chessboard should initially be laid out. Any underlined words are special moves that the piece can make.

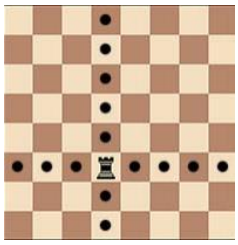
Each player takes one move per turn with white always starting the game. On the diagrams the circles show where a piece can move to and the crosses show where it can move to, if it is able to take a piece, by moving to that square.

Pawn

- Moves forward – towards the opponent's side – by one square.
- On its first move, a pawn may move forward two squares.
- Can move forward by one diagonal to take a piece of the opposition.
- May use Pawn promotion.
- May use En passant.

Rook

- Moves in any direction (except diagonally) any number of squares.
- Is required for Castling.



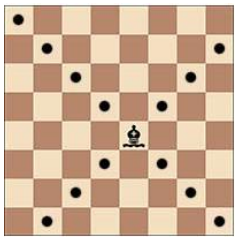
Knight

- Moves in an "L-shape"
- Can "hop" over other pieces.
- Moves to any squares, inside its 5x5 grid, except any squares that are on the same row or column or are diagonal.

5x5 grid

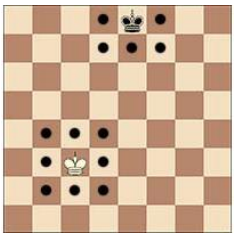
Bishop

- Moves diagonally in any direction, any number of squares.



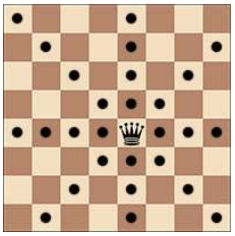
King

- Moves one square in any direction.
- Can use Castling.



Queen

- Moves in any direction, any number of squares.



Blocked/Taking

- If any piece moves to a square that is occupied by an opponent's piece then it captures the piece, removing it from the board.
- No piece can move to a square occupied by a piece of the same colour.
- A piece also cannot "hop" over any other piece unless it is a knight. (or a rook Castling).

Special Moves

Pawn Promotion

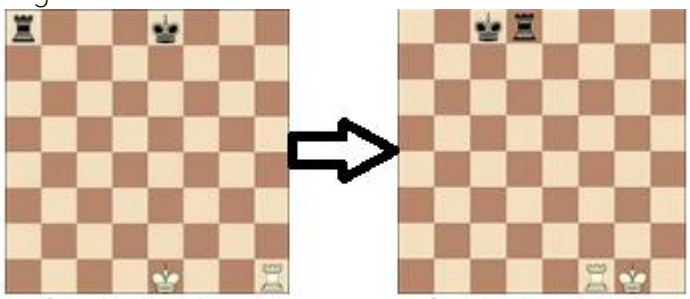
- When a pawn reaches the opponent's end of the board- meaning it can no longer move forward - it must be exchanged for another piece, excluding a king or another pawn. After it has been "promoted", the player's turn ends.

En passant

- An example of En passant is:
- Black's Move: A Black pawn moves forward two places on its first go.
  En passant is now possible.
  White's Move: If white has a pawn horizontally adjacent to the black pawn that was just moved, it can take it as though it only the black pawn moved only one square.
- *Note: The reverse is also possible with white moving their pawn forward first.*
- The two moves MUST be consecutive i.e. If black did not use en passant to take the pawn on the next move, they cannot take the pawn via En passant in later on.

Castling

- A King, if unmoved, maybe advance two spaces horizontally towards a rook that has also not been moved.
- The Rook will move, hop over the king, and be positioned on the square horizontally adjacent to the king.



Special Board states

Check

- When the current player's king is under attack – able to be captured – by an opponent's piece, it is said that the king is in check.
- A player cannot make a move that would put them in check, or a move that would keep them in check.

Checkmate

- When the current player's king is in check and no legal move will allow an escape. At this point the game is over and the opponent wins.

Stalemate

- When the current player's king **is not** in check and no legal moves can be made. This will end the game and the result will be a draw.

Winning, Losing and Drawing

Conditions for a win/loss

- Checkmate

A player's king is in check and no legal move will them to escape. The player loses and the opponent wins.

- Flag fall

A player runs out of time, and there is no ***dead position.*** *The player who ran out of time loses the game.*

- Resignation

A player forfeits the game, allowing the opposition to win.

Conditions for a Draw

- Dead position

Neither player can win the game by any legal sequence of moves. This occurs when:

Player A pieces	*Player B pieces*
King	*King*
King	*Bishop*
King	*Knight*
*King & Bishop**	*King & Bishop**

*****provided Bishops are both on the same colour tile.*

- Stalemate

This occur when the current player has no legal move to make, for example every move would put them in check.

- Draw by agreement.

Both players agree to draw.

- ***Draw due to board repetition.***

Once the same board has been detected 3 times, a player may request that a draw is made. After the same board has been seen 5 times, the result is automatically a draw.

- ***Draw due to no pieces taken or pawns moved.***

After 50 moves without the movement of a pawn or any pieces being taken, a player may request a draw. After this has exceed to 75 moves, the game is automatically ended in a draw.

Rules are from: https://www.fide.com/FIDE/handbook/LawsOfChess.pdf
Upon sending this list of rules to my client, she stated that she was not familiar with these rules & requested that they be omitted from her application.

Clocks

- Each player has their own clock.
- The clocks count down on each player's turn.
- Players cannot let their clocks run out of time or they will lose the game.

Design Specification & Brief

Design Brief

I will create an application that allows two people to play a remote game of chess over a long physical distance. The application should also allow two users to play locally on the same machine.

There will be three main parts to my project:

- Game Logic – Code handling moving of pieces.
- Connection. – Code handling connection
- User Interface. – Code handling human & computer interaction

1. The Project must allow two clients to play chess.

- High Priority – The Project needs/must…
  - To highlight moves that the player can make under the rules of chess.
  - Not allow the player to make any move that would contradict the rules of chess.
  - Be able to detect when a Checkmate has occurred.
  - Pawn Promotion.
  - Tell the player if they are in check.

- Medium Priority – The Project should…
  - Regulate the order of turns – white then black.
  - Be able to detect when a Stalemate has occurred – Medium Priority due to rareness.
  - Have En passant.
  - Detect when a player has run out of time – flag fall.
  - Allow Castling.

- Low Priority – The Project would ideally…
  - Detect if a dead position has occurred.
  - Allow the two players to agree to a draw.

2. The Project must allow a connection between two clients.

- High Priority – The Project needs/must…
  - Connect two players over a wireless network.
  - Allow two clients to update a chessboard.

- Medium Priority – The Project should…
  - Connect two players over two networks.
  - Handle a player leaving unexpectedly.
  - Have a connection time of less than 30 seconds.

- Low Priority – The Project would ideally…
  - Transfer metadata – name of player, etc…

3. The Project must have a good User Interface.

- High Priority – The Project needs/must…
  - A virtual chessboard that allows the player to move pieces.
  - Inform the user which colour they are.

- Medium Priority – The Project should…
  - Tell each player how much time they have left.
  - Have a menu system.
  - Have a clean and minimalistic design.

- Low Priority – The Project would ideally…
  - Display information about the person they are playing – IP, Name, etc…
  - Record the moves made during the game.

Design Limitations

Time constraints

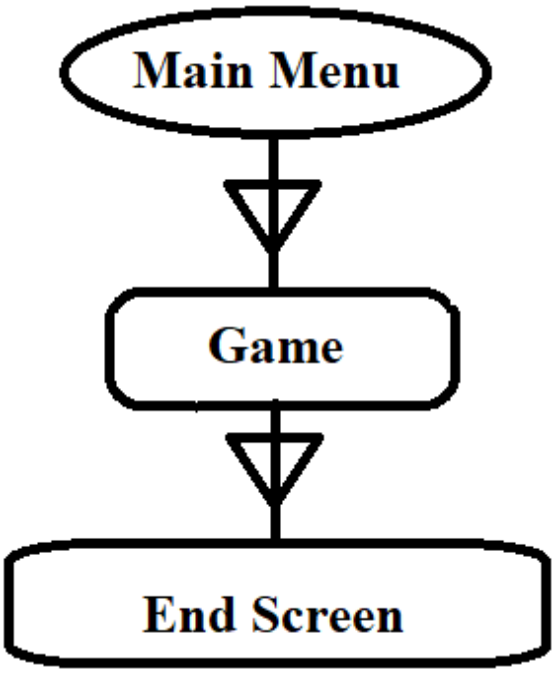
This is a large project and, since the application must be made by the 19th March 2021 at the latest, the constraints are largely time-based. I have spoken to Miss. Clarke and informed her that implementation of a database system that will store & update the rankings of players according to a complex formula would be its own additional project. I would be happy to do this another time; however, it will not be part of this application.

Server Hosting

I do not have the capabilities to permanently host my own dedicated server for chess. I will therefore have to rely on the client being able to host a private server and dealing with the setup of a VPN or port-forwarding on their end.

Design

Overview of system



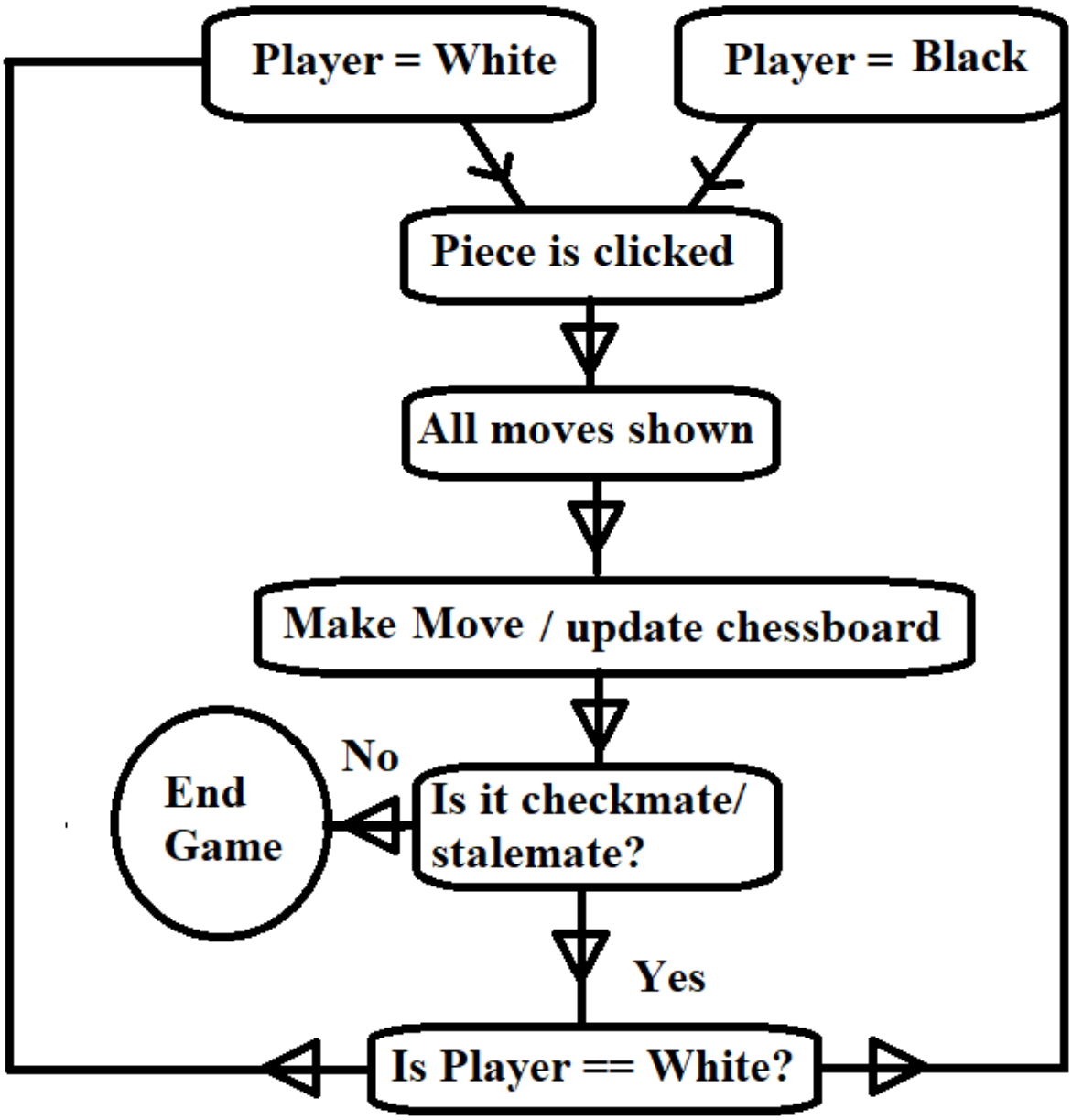
This is a diagram showing the form view of the program. The first form will be the main menu, which will then load into the game – the second form. When the game has ended due to a win loss or draw, it will show an end screen.

My Program consists of two modules: Server & Client. Two clients can connect to a server to play a game of chess over a network or, two people can play with one client on a local machine.

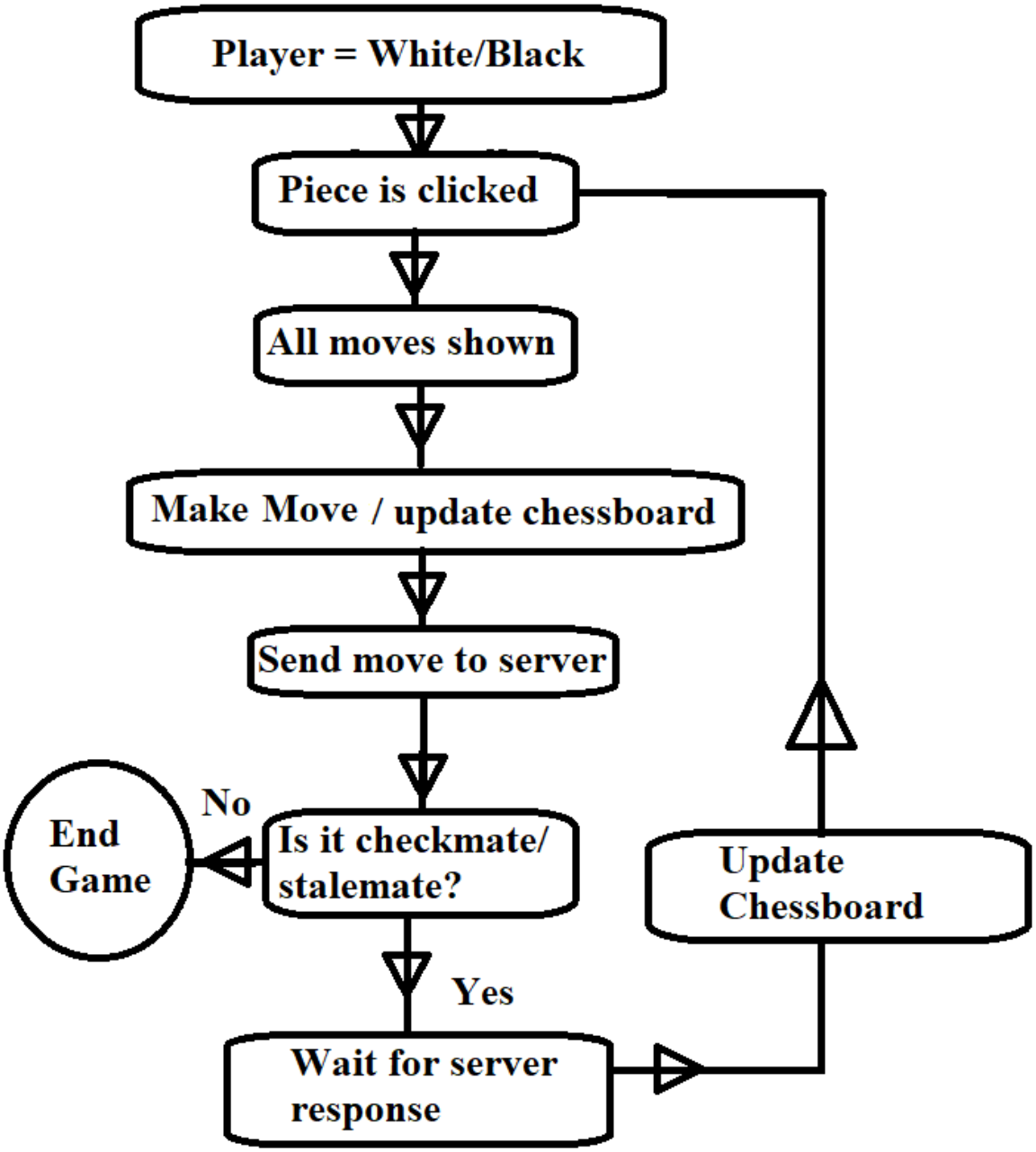
This means the code enforcing that only legal moves are made will be done client-side, creating a potential security risk.

However, I think the chances of exploitation are low since:

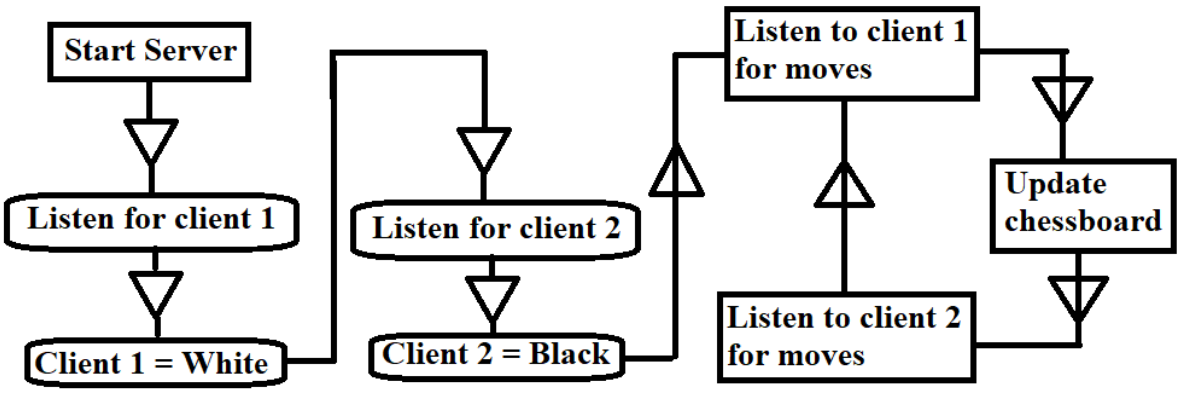
- The client does not allow its user to give the server data containing an illegal move. Anyone wanting to exploit this flaw in the program would have to first figure out the layout of the data sent to the server, and then create a separate program to send malicious data to the server.
- The users are friends, so it is unlikely they would create a "cracked client" allowing them to make illegal moves.



This diagram is an overview of how the client will work when played locally.

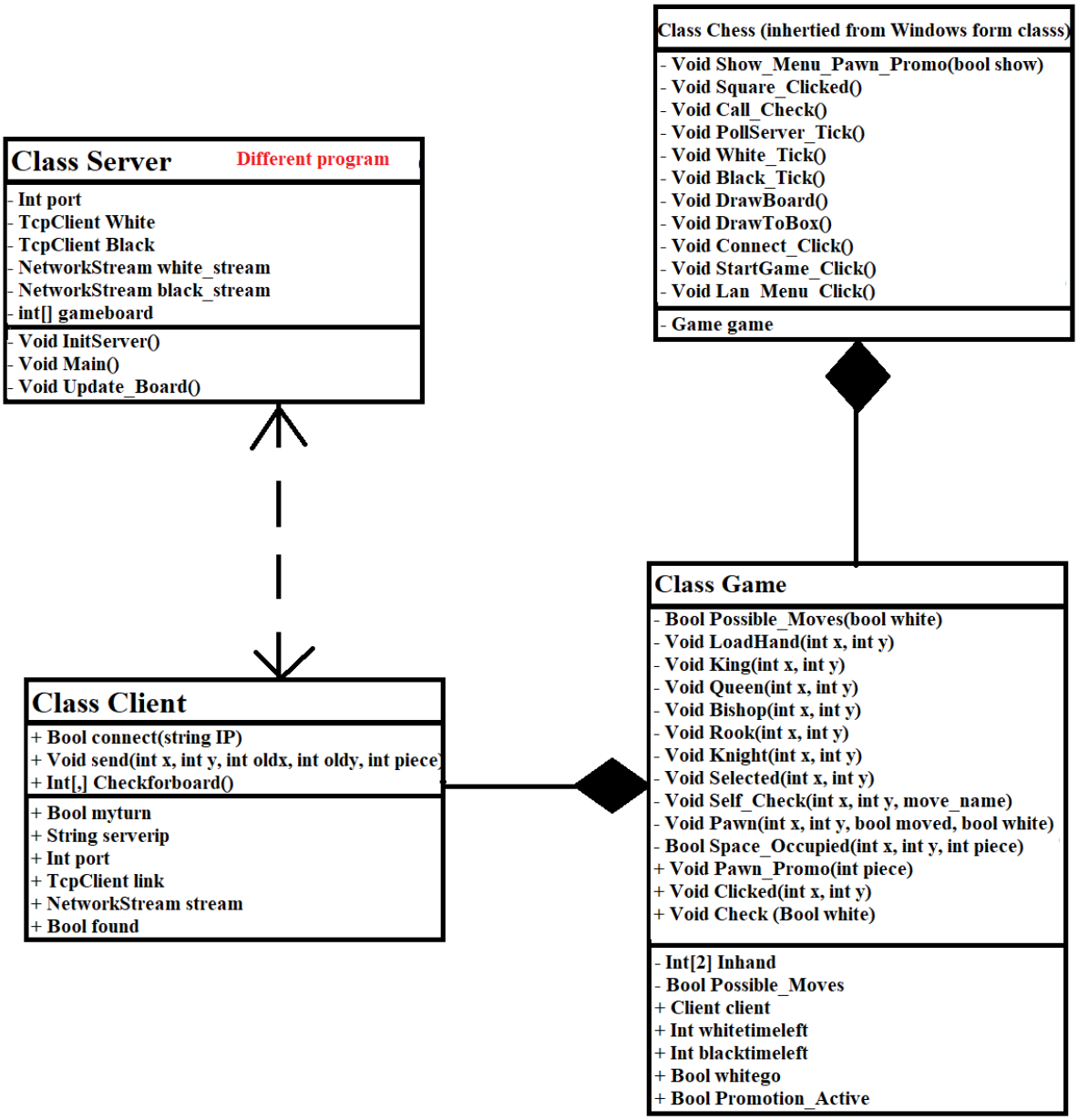


This diagram is an overview of how the client will work when playing over a network.



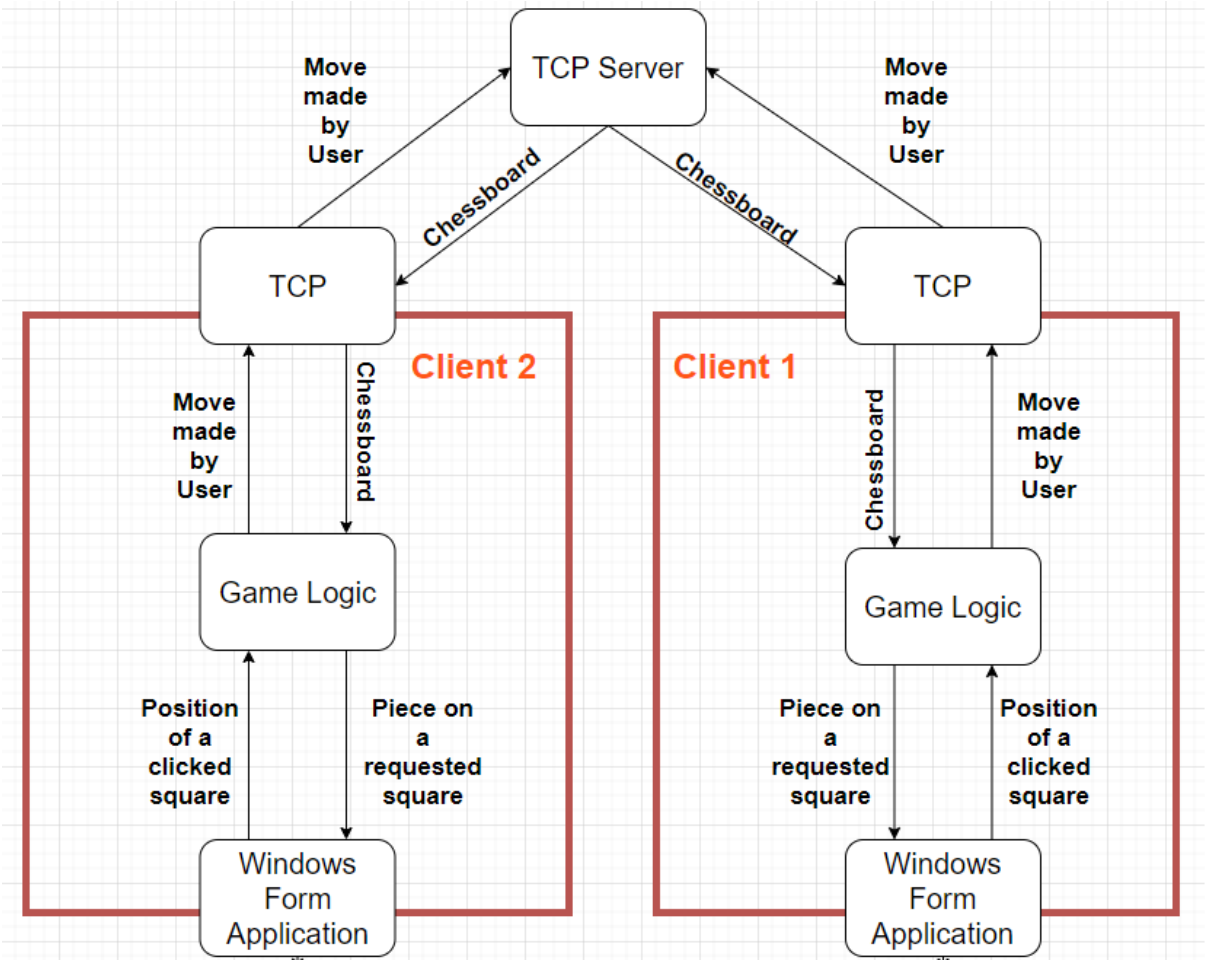
This diagram is an overview of how the server will work to connect the two clients.

OOP Class Design



This is a UML model of the program. The lines with black diamonds represent composite classes. Game cannot exist without Chess and Client cannot exist without Game. Server & Client model a mutually dependant relationship, since the client depends on information from the server, and the server depends on information from the client.

Data flow between classes



Description of Classes

The Client Application has three Classes:

- Chess (Handles user interface e.g. drawing the board). (Windows Form Application)
- Game (Handles Logic for chess game). (Game Logic)
- Client (Handles Connection to and from the server). (TCP)

The Server Application is simply one class.

- Server (Handles Connections from both clients).

This is class is static which means, we do not need to create an instance of the class.

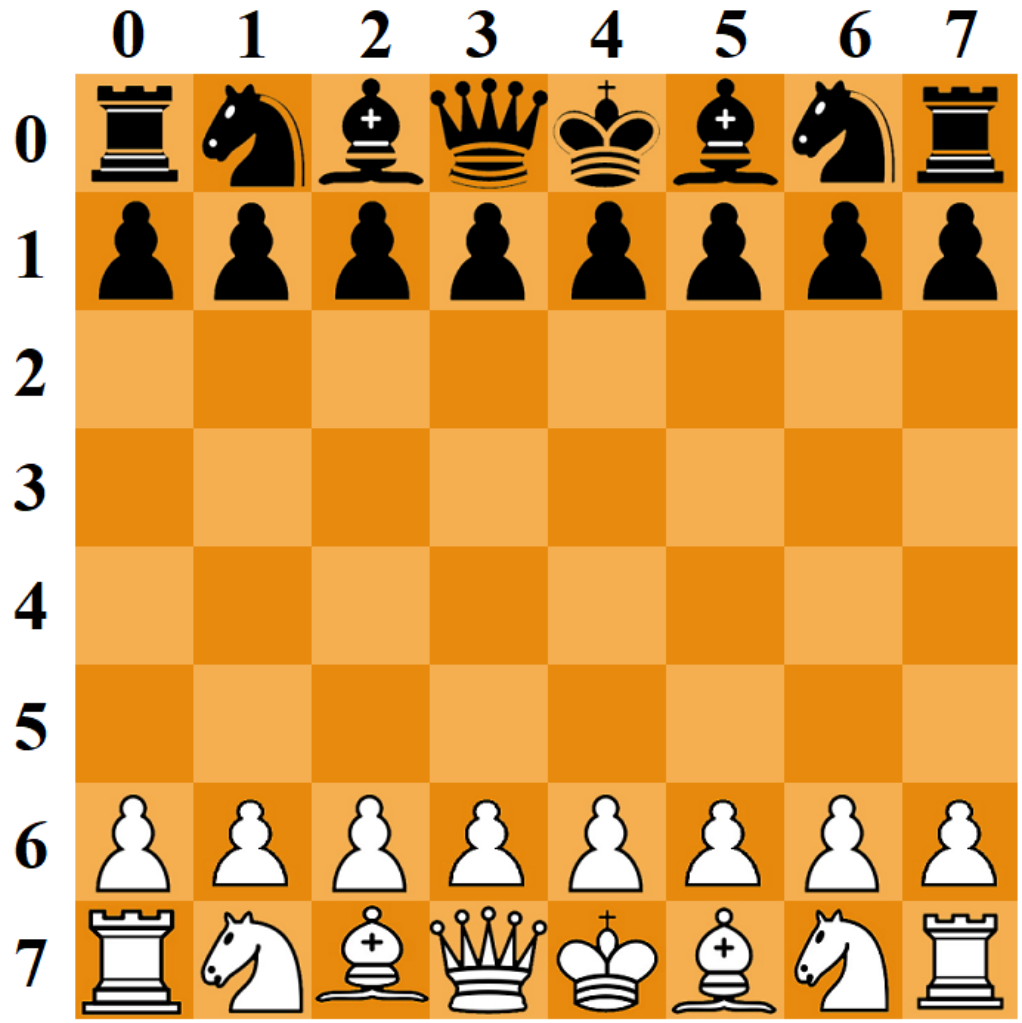
Drawing the Board

I have used windows forms to create my GUI. This is because

The GUI has 64 picture boxes, each representing one of the squares on an 8x8 chessboard. The name property of the pictures is set to:

- p + (X Coordinate) + (Y Coordinate)

Coordinates are shown by the following system:



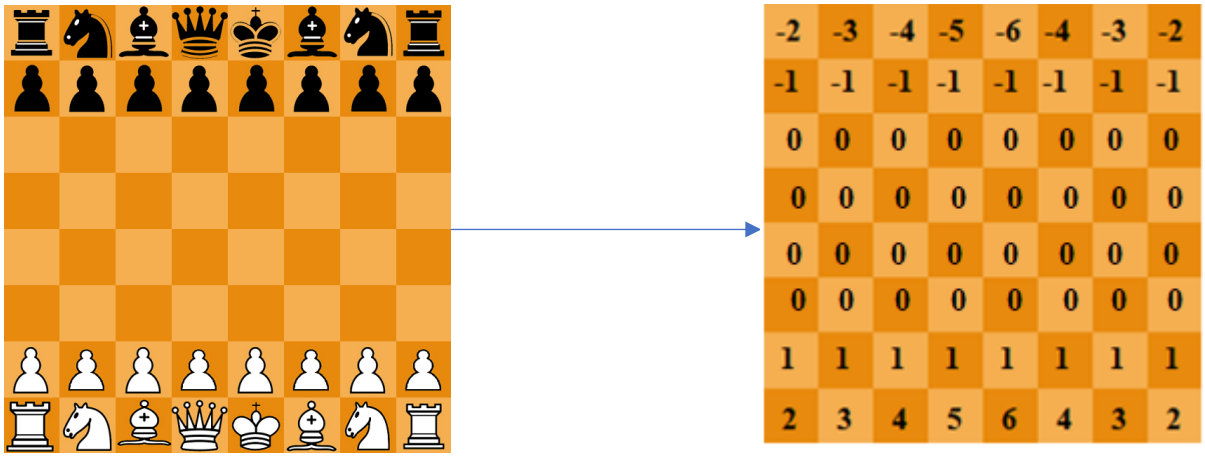
Representing the chessboard as a multi-dimensional array

A 2D integer array is created at the start of the game. The positions in the array correspond to the positions of the pieces on the board. The value of the integers refer to the pieces that occupy that position. To make it easier to program the following constants were defined:

Table of constants

Name of Constant	Value
Empty_Space	0
Pawn	1
Rook	2
Knight	3
Bishop	4
Queen	5
King	6
Pawn_Moved	7
King_Moved	8
Rook_Moved	9
Pawn_enpass	10
BLACK	-1

A black piece is stored as the negative value of the white piece, e.g. a Black Queen would be -5, which is also: BLACK * Queen. An example of the board and its corresponding array is below:



Storing the black pieces as negatives makes it easy to check whether two pieces are of the same colour. We find the product of two pieces, if they product is negative, the pieces are different colour. If the product is positive, the pieces are the same colour.

After a piece is clicked, its possible moves are highlighted. The array stores highlighted squares as (50 + the previous value of the square). This allows it to reverse engineer the square to find out what value it had before it was highlighted.

The following algorithm is part of the GUI class, it shows how the 2D array can be used to draw a virtual chessboard:

```
Function DrawBoard(){
    For(int x = 0, x < 8, x++) {
        For(int y = 0, y < 8, y++) {
            Int piece_value = Gameboard.returnpiece(x, y);
            switch(Abs(piece){
            case 1:
            case 7:
            case 10:
                piece_name = "pawn";
            case 2:
                piece_name = "rook";
            ……
            Default:
                piece_name = "selected";
            }
            If(piece_value < 0){ //If the piece is black
                piece_name += "_black";}
            if((x+ y) % 2 == 0 && piece_name <> "selected") {
                piece = "light" + piece;
                //This gives every piece a light and dark background
                //alternatively giving off a chessboard effect.
                }
            Draw(x, y, piece_name);
        }
    }
}
```

The Draw() algorithm simply finds the picture box at coordinates (x,y), then loads the image from a filename – specified by piece_name – into that box.

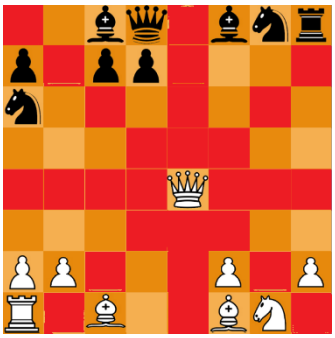
Moving Algorithms

Legal & Potential Moves

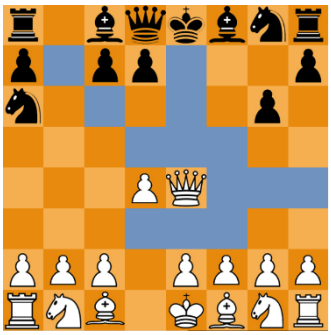
The notion of Legal and Potential moves will be discussed in this document. This is an overview of all the types of moves.

Consider this board, think about all the moves the white queen can make.

Potential moves are defined as the set of moves a piece can make given the laws of their movement. I.e. all the places a piece could move to if the board was empty. Below is all of the queen's potential moves



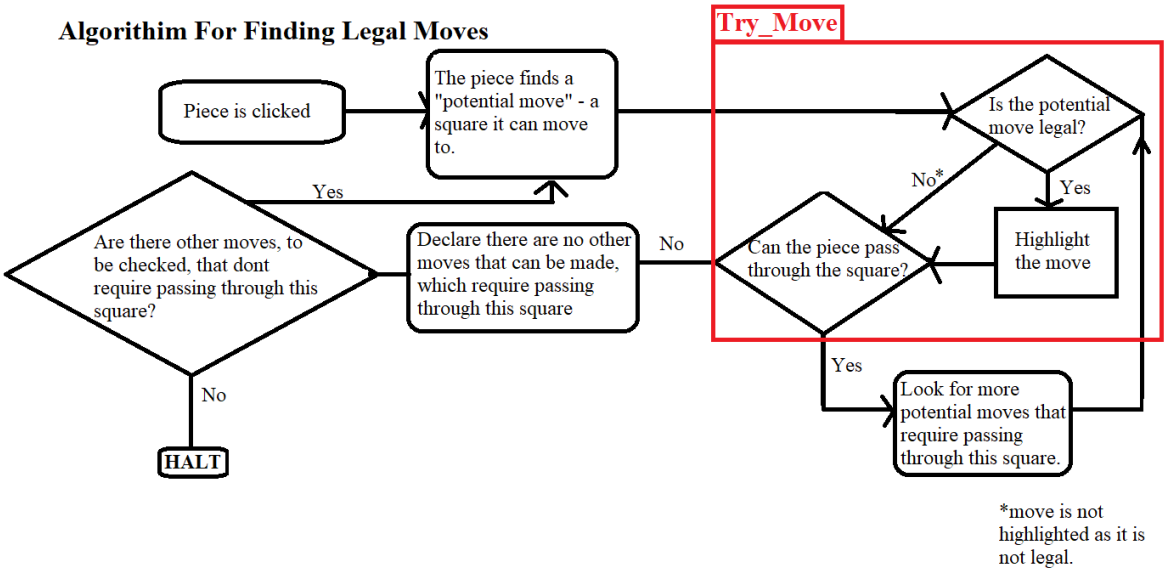
Legal moves are defined as the subset of the potential moves that are not illegal. A move is illegal if it: puts your own side in check, moves to a square occupied by a piece of the same colour or hops over any other piece*. Below are all of the queen's legal moves.



*There are exceptions to this rule for knights & kings castling.

Finding all legal moves

When the player clicks on a piece it shows all the legal moves for that piece. This is done by checking the set of potential moves for legal moves. The flowchart below, shows how we can find all the legal moves in a set of potential moves, without having to check all the potential moves.



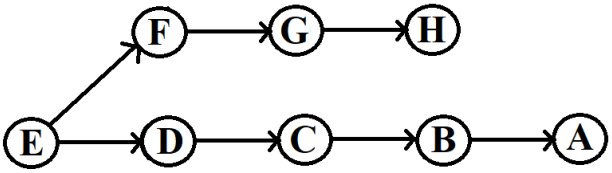
Try_Move – Used to check if a potential move is legal.

```
Function bool Try_Move(int X, int Y, string move = "normal"){
    If Out_Of_Bounds(X, Y)
    {
        Return False;
    }
    If (Gameboard[X,Y] * Gameboard[inhand[0],inhand[1]] > 0)
    {
        Return False;
    }
    If(Self_Check(y, x, move))
    {
        Return Space_Occupied(X,Y);
    }
    Gameboard[Location_X, Location_Y] += 50;
    Global possible_move = true;
    If(move == "enpass")
    {
        Gameboard[x, y] = 12;
    }
    If(move == "castle_right")
    {
        Gameboard[x,y] = 11;
    }
    If(move== "castle_left")
    {
        Gameboard[x,y] = 13;
    }
    return(Space_Occupied(X,Y));
}
```

This is key algorithm in the project, it determines if a potential move to a square is legal – highlighting it if it is. It then returns a Boolean declaring whether the piece can pass through the square. Thus, reducing the amount of moves the program must check. This idea is further explained down below.

Consider this horizontal row on a chessboard.

The following potential squares must be checked.



A square is a child of another square if you have to pass through the parent square to get to the child (e.g. you must pass through square F to get to G, so G is a child of F).

Tracing the Algorithm

First, we check square D: (Note: "Checked" simply refers to whether we have determined if passing through or moving to the square is possible – i.e. does NOT refer to any kind of checkmate or board position.)

Square	Move to Square	Pass through square	Checked
A	-	-	No
B	-	-	No
C	-	-	No
D	No	No	Yes
E	Yes	Yes	No
F	-	-	No
G	-	-	No
H	-	-	No

Square	Move to Square	Pass through square	Checked
A	No	No	No (results inferred)
B	No	No	No (results inferred)
C	No	No	No (results inferred)
D	No	No	Yes
E	Yes	Yes	No
F	-	-	No

G	-	-	No
H	-	-	No

This table shows us that passing through square D is not possible, therefore all children of D, do not need to be checked as we can infer that moving to them is impossible.

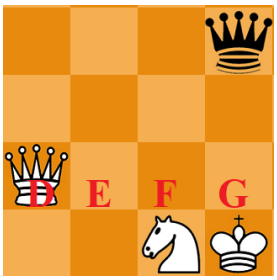
We also check F, seeing it is both possible to move to and pass through the square, we then check G. We can move to G but cannot pass through it, so moving to any child of G is impossible.

Square	Move to Square	Pass through square	Checked
F	Yes	Yes	Yes
G	Yes	No	Yes
H	No	No	No (results inferred)

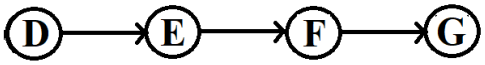
This shows that by checking the legality of three potential moves we can infer the legality of another four.

Tracing the algorithm - For cases of check/checkmate

Finally, consider this board. It is white's turn, and they must get themselves out of check.



This is a graph of possible moves.



First, we check if we can move to E

Square	Move to Square	Pass through square	Checked
E	No	Yes	Yes
F	-	-	No
G	-	-	No

We learn that we can pass through E, despite not being able to move to it – as this would put us in checkmate. So, we must also check F, and then G.

Square	Move to Square	Pass through square	Checked
E	No	Yes	Yes
F	No	Yes	Yes
G	Yes	Yes*	Yes

This demonstrates an example in which moving to a parent square is an illegal move, but moving to a child of it is a legal move.

*You may notice that this would then check a square which is off the board – which would be declared as illegal bringing the algorithm to an end.

Highlighting the squares

For a normal move we add 50 to the value of the square we wish to highlight. This means that if a square is deselected it is easy to find out what was originally on the square since all we must do to get back to the original value is subtract 50.

If the move is complex – consists of more than moving a value from one position in the array into another position in the array – it is highlighted with a specific number, all complex moves are such that the square it highlights will always have an original value of 0 i.e., it will be an Empty Space.

Getting potential moves

Below are the algorithms showing how each piece generates its own potential moves and then sends them to try_move to determine if they are legal.

Pawn

```
Function Pawn(int Location_X, int Location_Y, bool first_move, bool black){
Int direction = -1;
If (Black) {direction = 1;}
// Taking
If (Space_Occupied((Location_X - 1, Location_Y + direction){
Try_Move(Location_X - 1, Location_Y + direction);
}
If (Space_Occupied(Location_X + 1, Location_Y + direction)){
Try_Move(Location_X + 1, Location_Y + direction);
}
// Moving
If (first_move AND Try_Move(Location_X, Location_Y + direction){
    Try_Move(Location_X, Location_Y + (2 * direction));
}
// En passant
If (Space_Occupied(Location_X + 1, Location_Y){
    Try_Move(Location_X + 1, Location_Y + direction, "enpass");
}
If (Space_Occupied(Location_X + 1, Location_Y){
    Try_Move(Location_X + 1, Location_Y + direction, "enpass");
}
```

Rook

```
Function Rook(int X, int Y){
For(int I = X + 1; I < 8; I++){
    If !(Try_Move(I,Y)) {HALT}
    // Checks all east moves
    }
For(int I = X - 1; I > -1; I--){
    If !(Try_Move(I,Y)) {HALT}
    // Checks all West moves
    }
For(int I = Y + 1; I < 8; I++){
    If !(Try_Move(X,I)) {HALT}
    // Checks all South Moves
    }
For(int I = Y + 1; I > -1; I--){
    If !(Try_Move(X,I)) {HALT}
    // Check all North Moves
    }
}

```

Knight

```
Function Knight(int X, int Y){
    Try_Move(X - 2 , Y + 1);
    Try_Move(X - 2, Y - 1);
    Try_Move(X + 2, Y - 1);
    Try_Move(X + 2, Y + 1);
    Try_Move(X + 1, Y - 2);
    Try_Move(X + 1, Y + 2);
    Try_Move(X - 1, Y + 2);
    Try_Move(X - 1, Y - 2);
}
```

Bishop

```
Function Bishop(int X, int Y){
    For(int I = 1; I < 8;I++){
        If !(Try_Move(X + I, Y + I)){ HALT }
    }
    For(int I = 1; I < 8;I++){
        If !(Try_Move(X - I, Y + I)){ HALT }
    }
    For(int I = 1; I < 8;I++){
        If !(Try_Move(X + I, Y - I)){ HALT }
    }
    For(int I = 1; I < 8;I++){
        If !(Try_Move(X - I, Y - I)){ HALT }
    }
}
```

Queen

```
Function Queen(int X, int Y){
    Rook(X, Y);
    Bishop(X, Y);
}
```

King

```
Function King(int X, int Y){
    Try_Move(X - 1, Y);
    Try_Move(X - 1, Y - 1);
    Try_Move(X - 1, Y + 1);
    Try_Move(X, Y + 1);
    Try_Move(X, Y - 1);
    Try_Move(X + 1, Y);
    Try_Move(X + 1, Y - 1);
    Try_Move(X + 1, Y + 1);
    //CASTLING
    If (Space_Occupied(X,Y,KING) OR Space_Occupied(X,Y,-KING))
    //Makes sure the king hasn't moved
    {
        //RIGHT SIDE CASTLE
        If (Space_Occupied(0,Y,ROOK) OR Space_Occupied(0,Y,-ROOK))
        //Makes sure there is a rook in the corner.
        // that hasn't been moved.
        {
            If NOT(Space_Occupied(X-1, Y) OR
                   Space_Occupied( X - 2, Y) OR
                   Space_Occupied( X-3, Y))
            //checks no pieces are blocking the castle
            {
                Try_Move(x - 2, y, "castle_left")
            }
        }
        //LEFT SIDE CASTLE
        If (Space_Occupied(7,Y,ROOK) OR Space_Occupied(7,Y,-ROOK))
        //Checks for rook
        {
            If NOT(Space_Occupied(X+1, Y) OR
                    Space_Occupied( X + 2, Y))
                //checks for blocking.
            {
                Try_Move(x + 2, y, "castle_right")
            }
    }
}

```

Making the move

Now that the player can see all the legal moves, they can select one. For this, the following algorithm is used:

```
Function Selected(int x, int y, int select){
    Bool endturn = true;
    Remove_EnPassant();
    Int piece_inhand = gameboard[global inhand[0], global inhand[1]];
    Switch(piece_inhand){
        Case PAWN:
        Case -PAWN:
            If( Absolute(y-inhand[1] == 2){
                Piece_inhand *= PAWN_ENPASS;
            }
            Else{
                Piece_inhand *= PAWN_MOVED;
            }
        CASE PAWN_MOVED:
        CASE -PAWN_MOVED:
            If(y == 0 OR y == 7){
                Endturn = false;
            }
        CASE ROOK:
            Piece_inhand = ROOK_MOVED;
        CASE -ROOK:
            Piece_inhand = -ROOK_MOVED;
        CASE KING:
            Piece_inhand = KING_MOVED;
        CASE -KING:
            Piece_inhand = -KING_MOVED;
    }
    Gameboard[x, y] = piece_inhand;
    Gameboard[inhand[0],inhand[1]] = EMPTY_SPACE;
    If(select == 12){
        Gameboard[x, y + (piece_inhand / 7)] = 0;
    }
    If(select == 13){
        Gameboard[x+1,y] = gameboard[y,0];
        Gameboard[y,0] = EMPTY_SPACE;
    }
    If(select == 11){
        Gameboard[x-1, y] = gameboard[7, y];
        Gameboard[7, y] = EMPTY_SPACE;
    }
    If(endturn){
        Global Whitego = NOT(Whitego);
        if(select == 12){
            Client.send(x, y, inhand[0], inhand[1], 14);
        }
        Else if(select == 11){
            Client.send(x, y, inhand[0], inhand[1], 12);
```

```
48          }
49          Else if(select == 13){
50              Client.send(x, y, inhand[0], inhand[1], 3* piece_inhand)
51          }
52          Else{
53              Client.send(x, y, inhand[0], inhand[1], piece_inhand);
54          }
55      }
56      Else{
57          Global Pawn_Promo = true;
58          Global Pawnx = x;
59          Global Pawny = y;
60      }
61  }
```

This subroutine is responsible for moving the piece from one part of the board to another; it also relays the move to the server. This algorithm does many things, so we will break it down piece by piece.

Lines: 2-4

"Endturn" is set to true, meaning that at the end of the algorithm the players turn is over and the opponent can now make their move.

"Remove_EnPassant()" is called, changing all pieces with a value of "PAWN_ENPASS" to "PAWN_MOVED", removing the ability to take them via en passant.

Inhand is an array holding the coordinates of the last piece the player clicked, Inhand[0] refers to the x coordinate and inhand[1] refers to the y coordinate. "piece_inhand" is set to the value of the piece at this position.

Lines: 5-27

Most of the time when a move is made, we simply transfer the piece to the destination square and set the initial square to "EMPTY_SPACE". There are three exceptions:

- Changing the value of a piece to signify that the piece has been moved – for rooks, kings, and pawns.
- Changing the value of a pawn to show that it can be captured via en passant.
- Changing the value of a pawn that has been promoted to another piece.

Line 8 checks if a pawn has moved two squares and sets its value to PAWN_ENPASS if so.

Line 16 checks if a pawn has reached the end of the board and sets "endturn" to false allowing the player to promote their pawn.

All the other checks are simply changing the value of a piece to show whether it has moved. This is done since some pieces have certain moves restricted if it is not their first go.

Lines: 28-29

Line 28 places the piece into its destination square and line 29 clears its initial square.

Lines: 30-40

This determines whether the selected square represents a complex move.

Lines 30 to 32 handle en passant. This is responsible for setting the square at [x, y+1] or [x,y-1] to an empty space, for a white and black move respectively. To determine which square it must clear, it divides the value of the piece being moved by 7, giving 1 or -1 – for a white or black pawn, respectively.

Lines 33 to 36 handle castling to the right. It is only responsible for moving the rook as the king is already moved by lines 28-29.

Lines 37 to 40 are similar to lines 33-36, handles castling to the left.

Lines: 41-54

Tells the server the move made when the players turn is over.

It tells the server:

x – the piece's new x coordinate

Y – the piece's new y coordinate

Inhand[0] – the piece's old x coordinate

Inhand[1] – the piece's old y coordinate

Piece_inhand – the value of piece (variable in subroutine).

If a complex move has been made then it will send a specific value to the server, rather than the value of piece_inhand, telling it to adjust its board for the complex move.

Lines: 57-59

Sets Pawn_Promo to true, halting all further moves until the player selects a piece for the pawn to be promoted to. It also records the position of the pawn so the program can access this later.

Flow of game

```
Function Square_Clicked(){
    call_check(global game.whitego);
    if (game.client.found){
        if(game.whitego AND game.client.name == "black") {HALT}
        if(NOT(game.whitego) AND game.client.name == "white") {HALT}
    }
    int x = Get_X_Coords(this);
    int y = Get_Y_Coords(this);
    if(game.Pawn_Promo == true)
    {
        Show_Menu_Pawn_Promotion(true);
        HALT
    }
    else
    {
        game.Getmoves(x, y);
    }
    DrawBoard();
    call_check(global game.whitego);
    }
```

Walking through this algorithm – responsible for setting the flow of the game we see:

- Before any move is made, we see if the current player is in check*.
- If we are playing over a network, we make sure that this client only selects pieces which they are permitted to move.
- We get the coordinates of the piece.
- If we are in the middle of a pawn promotion we force them to promote their pawn instead of making any other move.
- Otherwise, we call getmoves.
- Getmoves will either show all the legal moves for a piece, or call another subroutine to move a piece – if a selected square is chosen.
- The new Board is then drawn.
- We see if the opponent is in check before they make their move.

*This was later changed during the technical solution stage.

```
21  Function Getmoves(int x, int y){
22      int piece = GameBoard[x, y];
23      GameBoard = Deselect_All(GameBoard);
24      if(whitego == false AND (piece > 0 AND piece < 11)) {HALT}
25      if(whitego == true AND (piece < 0) {HALT}
26      switch(Absolute(piece){
27          case EMPTY_SPACE:
28              //Do Nothing.
29          case PAWN:
30              loadhand(x,y);
31              Pawn(x, y, true, (piece < 0);
32              //Pawn(x, y, first_move?, black?)
33          case PAWN_MOVED:
34          case PAWN_ENPASS:
35              loadhand(x,y);
36              Pawn(x, y, false, (piece < 0);
37          case ROOK:
38          case ROOK_MOVED:
39              loadhand(x,y);
40              Rook(x, y);
41          case KNIGHT:
42              loadhand(x,y);
43              Knight(x, y);
44          case BISHOP:
45              loadhand(x,y);
46              Bishop(x, y);
47          case QUEEN:
48              loadhand(x,y);
49              Queen(x, y);
50          case KING:
51          case KING_MOVED:
52              loadhand(x,y);
53              King(x , y);
54          default: // Any Selected piece
55              Selected(x, y, piece);
56          }
57      }
58
```

Determining Check, Checkmate & Stalemate

```
Function bool Check(bool black){
    int prefix;
    if(black) {prefix = 1;}
    else
    {
        prefix = -1;
    }
    int[] pos = findking(black); // pos == [ x , y]
    int x = pos[0];
    int y = pos[1];
    //Do any rooks or queens put the king in check
    bool take = false;
        for(int u = 0; u < 8; u++){
            int piece = gameboard[u, y];
            if(piece == prefix * ROOK OR
               piece == prefix * ROOK_MOVED OR
               piece == prefix * QUEEN)
            {
                take = true;
                if(u > x) { return true;}
            }
            Else if(piece <> EMPTY_SPACE AND
                    piece <> -prefix * KING AND
                    piece <> -prefix * KING_MOVED)
            {
                if(u > x) { break; }
                take = false;
            }
        }
        if(take) { return true;}
        for(int u = 0; u < 8; u++){
            int piece = gameboard[x, u];
            if(piece == prefix * ROOK OR
               piece == prefix * ROOK_MOVED OR
               piece == prefix * QUEEN)
            {
                take = true;
                if(u > y) { return true;}
            }
            Else if(piece <> EMPTY_SPACE AND
                    piece <> -prefix * KING AND
                    piece <> -prefix * KING_MOVED)
            {
                if(u > y) { break; }
                take = false;
            }
        }
```

```
48          if(take) { return true;}
49      //Do any bishops or queens put the king in check
50      for(int i = 1; i < 8; i++)
51      {
52          if(Space_Occupied(x - i, y + i, prefix * BISHOP)) {return true; }
53          if(Space_Occupied(x - i, y + i, prefix * QUEEN))  {return true; }
54          if(Space_Occupied(x - i, y + i)) {break;}
55      }
56      for(int i = 1; i < 8; i++)
57      {
58          if(Space_Occupied(x + i, y - i, prefix * BISHOP)) {return true; }
59          if(Space_Occupied(x + i, y - i, prefix * QUEEN))  {return true; }
60          if(Space_Occupied(x + i, y - i)) {break;}
61      }
62      for(int i = 1; i < 8; i++)
63      {
64          if(Space_Occupied(x + i, y + i, prefix * BISHOP)) {return true; }
65          if(Space_Occupied(x + i, y + i, prefix * QUEEN))  {return true; }
66          if(Space_Occupied(x + i, y + i)) {break;}
67      }
68      for(int i = 1; i < 8; i++)
69      {
70          if(Space_Occupied(x - i, y - i, prefix * BISHOP)) {return true; }
71          if(Space_Occupied(x - i, y - i, prefix * QUEEN))  {return true; }
72          if(Space_Occupied(x - i, y - i)) {break;}
73      }
74  }
```

This is merely a set of tests, to see if the player specified by the parameters is in check.

```
Function bool Self_Check(int x , int y, string move_name) {
    bool incheck;
    int colour = -1;
    if(whitego) {colour = 1;}
    int[,] temp = Backup_To(temp);
    gameboard = Deselect_All(gameboard);
    gameboard[x, y] = gameboard[inhand[0], inhand[1]];
    gameboard[inhand[1], inhand[0]] = 0;
    if (move_name == "enpass")
    {
        gameboard[y + colour, x] = 0;
    }
    if (move_name == "castle_right")
    {
        gameboard[y, x - 1] = gameboard[y, 7];
        gameboard[y, 7] = 0;
    }
    if (move_name == "castle_left")
    {
        gameboard[y, x + 1] = gameboard[y, 0] = 0;
        gameboard[y, 0] = 0;
    }
    if(whitego){
        incheck = check(WHITE);
    }
    else{
        incheck = check(BLACK);
    }
    gameboard = temp;
    return incheck;

}
```

To determine whether a move would put your own king into check – and is therefore illegal – we use this algorithm.

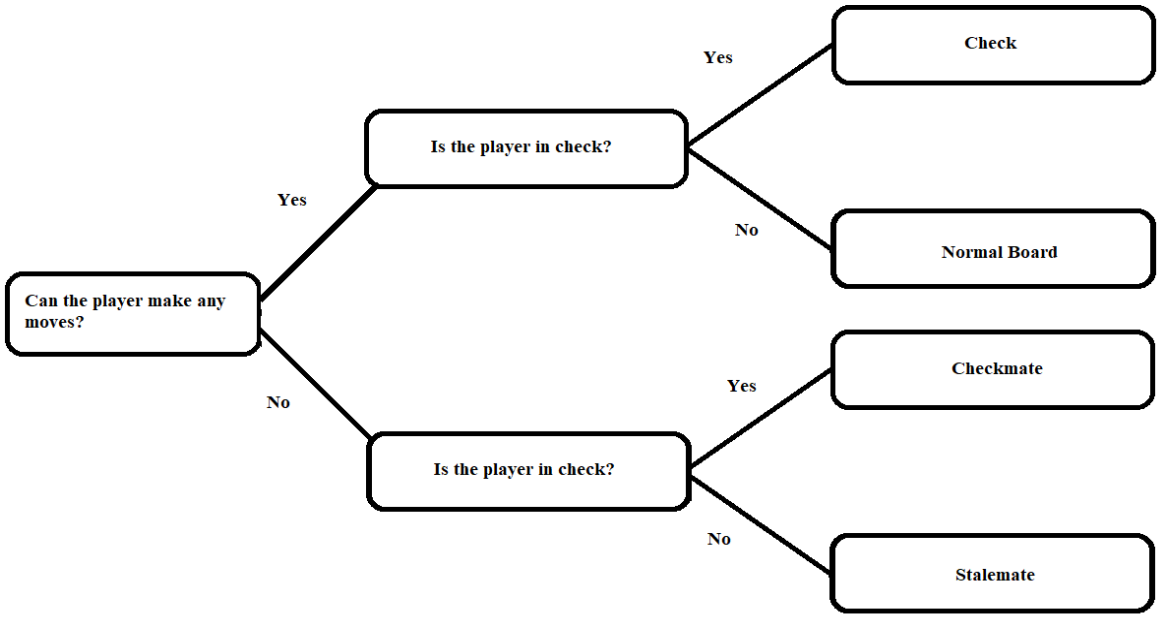
- First the board is backed up to a temporary array.
- We make the move on the original board.
- We then check if the current player, is in check.
- We reset the board to its previous state, using the temporary array.
- If the player was in check then we return saying the move was illegal.

```
Function bool Possible_moves(){
    int colour = -1;
    if(whitego) {colour = 1;}
    int tx = inhand[0];
    int ty = inhand[1];
    int [,] temp = new int[8,8];
    temp = Backup_To(temp);
    gameboard = Deselect_All(gameboard);
    global possible_move = false;
    for(int y = 0; y < 8; y++)
    {
        for(int x = 0; x < 8; x++)
        {
            if (gameboard[x, y] * colour > 0)
            {
                clicked(x,y);
                if(possible_move == true)
                {
                    loadhand(tx, ty);
                    gameboard = temp;
                    return true;
                }
            }
        }
    }
    loadhand(tx, ty);
    gameboard = temp;
    return false;
}
```

This is to determine whether or there are any possible moves.

- First the board is backed up to a temporary array.
- Then we back up the value of the inhand array.
- We go through the entire board, and if we find a piece that is owned by the player, we find all the legal moves associated with it.
- If we find one legal move, we can end the subroutine as we know there are possible moves for the player.
  Otherwise, we go through the whole board and find no legal moves.

Call_Check

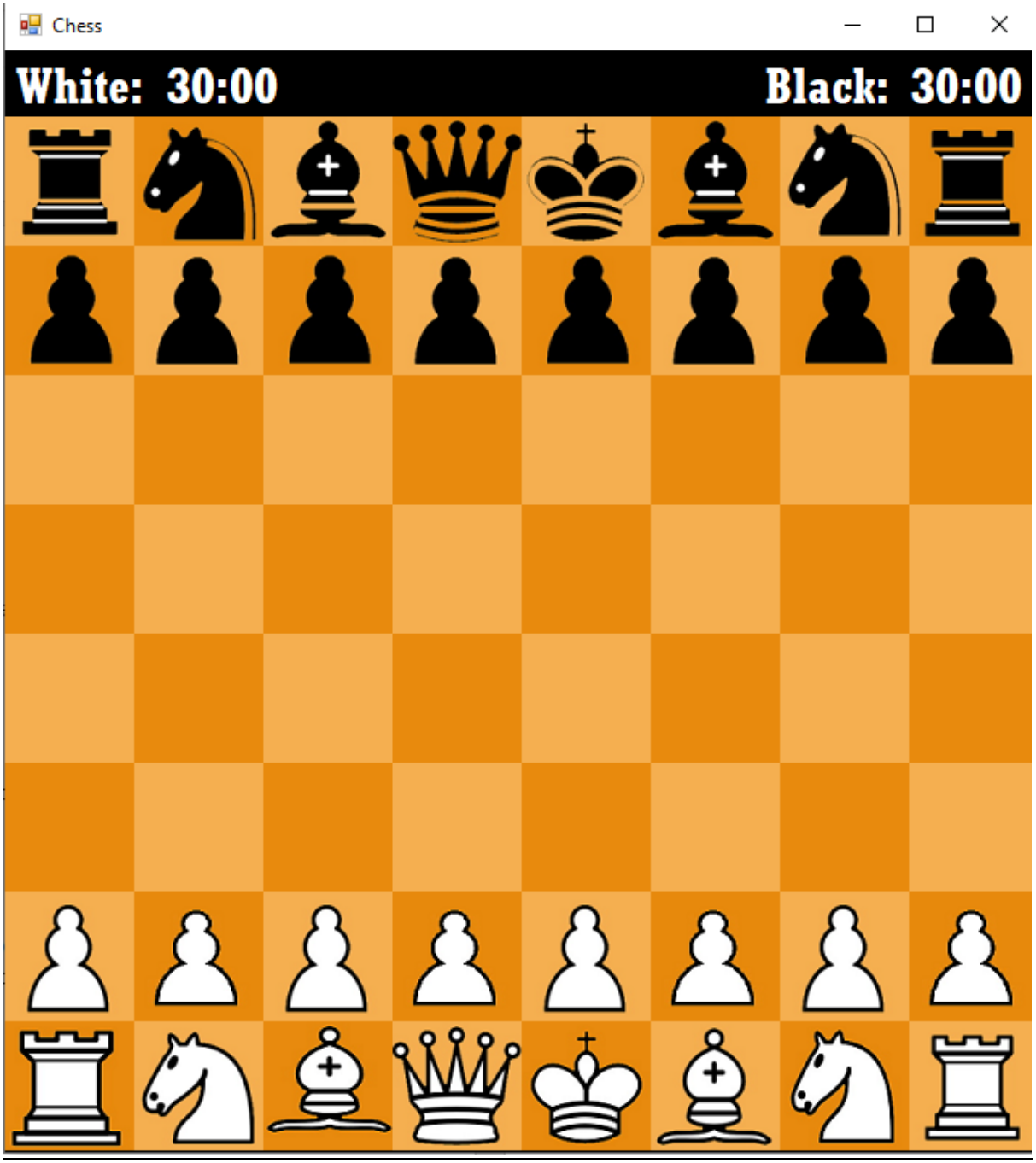


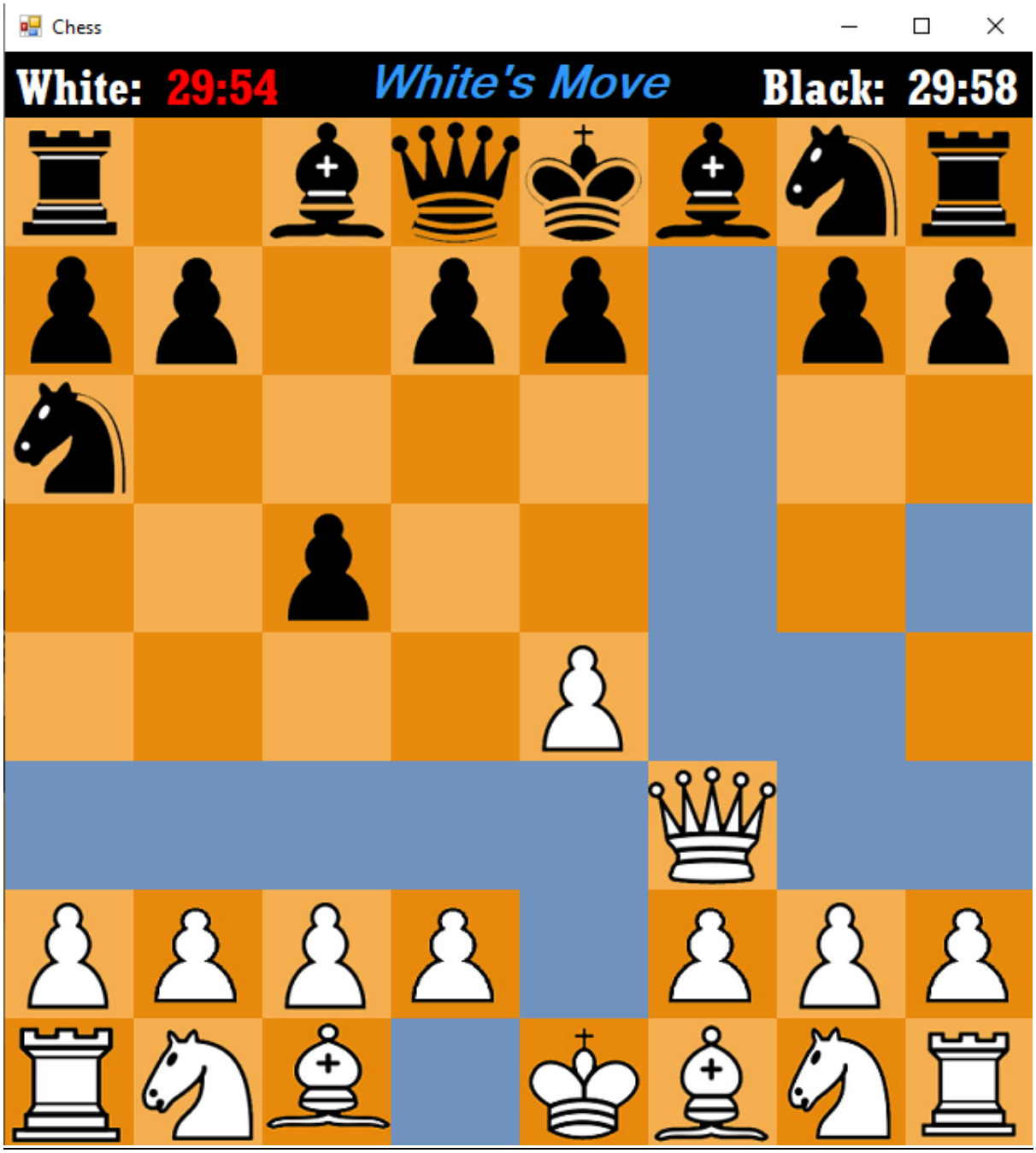
We use the following flowchart to determine the current state that the board is in.

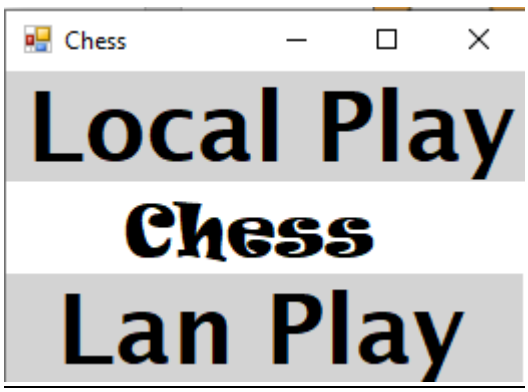
This can be reflected by the following algorithm:

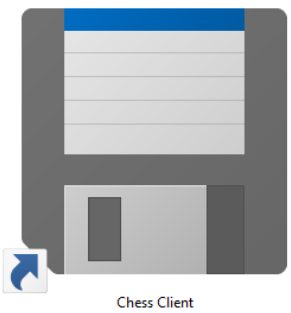
```
Function Call_Check(Bool White){
    bool incheck = false;
    string name;
    int colour;
    if (White){
        name = "White";
        colour = 1;
    }
    else{
        name = "Black";
        colour = -1;
    }
    if(game.check(colour)){
        Notice.Text = name + "Check";
        incheck = true;
    }
    else
    {
        Notice.Text = name + "'s Move";
    }
    if(game.Possible_Moves()){
        if(incheck)
        {
            Notice.Text = name + " Checkmate";
        }
        else
        {
            Notice.Text = name + " Stalemate";
        }
    }
}
```

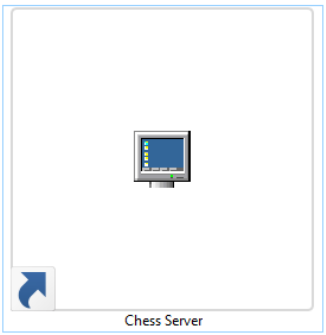
GUI/HCI

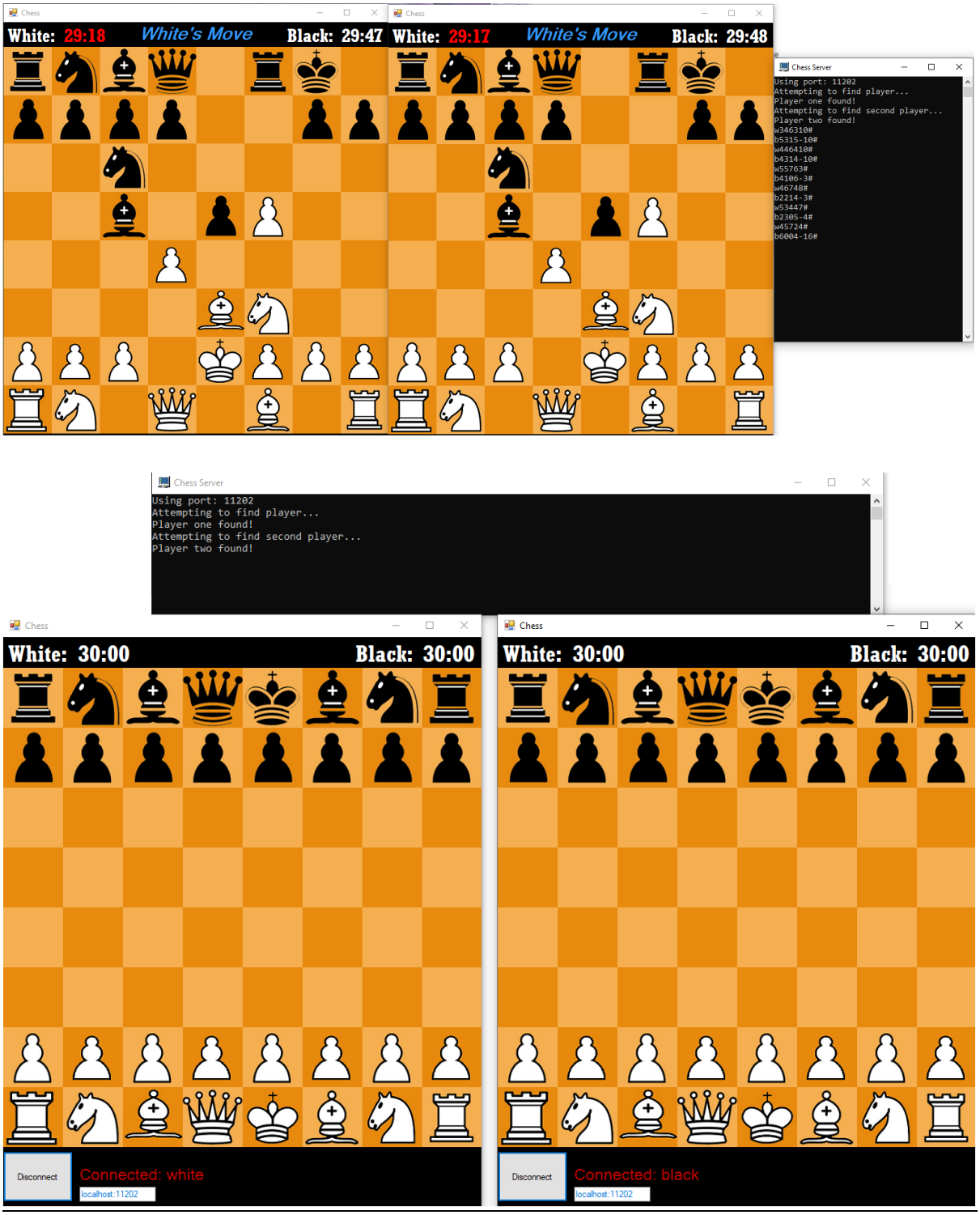


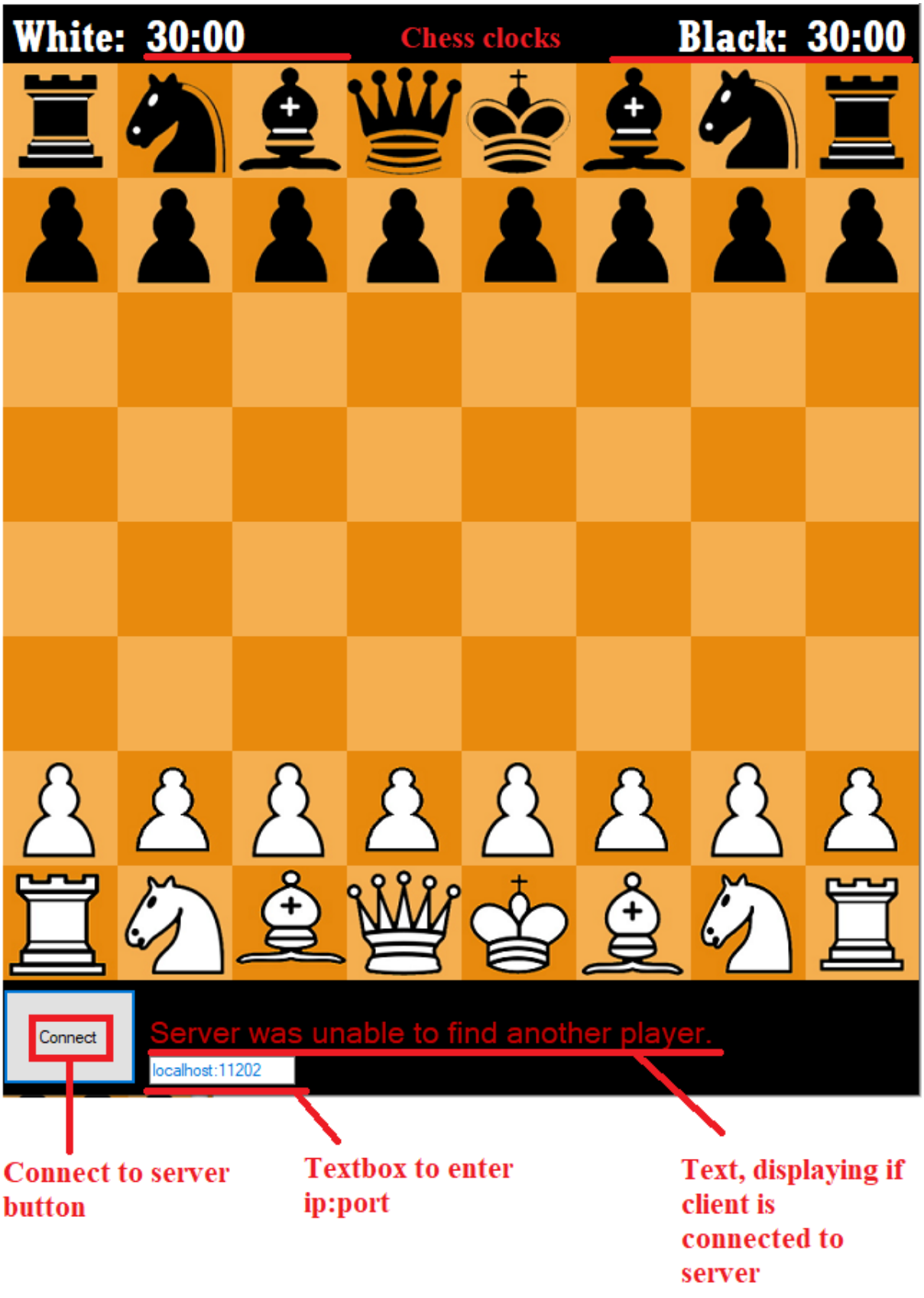

Chess
White: 29:54
White's Move
Black: 29:58


Chess
Local Play
Chess
Lan Play


Chess Client


Chess Server


Chess
White: 29:18
White's Move
Black: 29:47
Chess
White: 29:17
White's Move
Black: 29:48
Chess Server
Using port: 11202
Attempting to find player...
Player one found!
Attempting to find second player...
Player two found!
w346310#
b5315-10#
w446410#
b4314-10#
w55763#
b4106-3#
w46748#
b2214-3#
w53447#
b2305-4#
w45724#
b6004-16#
Chess Server
Using port: 11202
Attempting to find player...
Player one found!
Attempting to find second player...
Player two found!
Chess
White: 30:00
Black: 30:00
Disconnect
Connected: white
localhost:11202
Chess
White: 30:00
Black: 30:00
Disconnect
Connected: black
localhost:11202


White: 30:00
Chess clocks
Black: 30:00
Connect
Server was unable to find another player.
localhost:11202
Connect to server button
Textbox to enter ip:port
Text, displaying if client is connected to server

Server-Client Model

I have used TCP (Transmission Control Protocol) for my connection between server and client. To do this I have used the library System.Net.Sockets.

My two choices were UDP (User Datagram Protocol) & TCP. I picked TCP as it allows for a more stable connection, and UDP did not offer any additional benefit.

For this model I will need to make use of Async sockets, these are sockets which do not need a receiver and transmitter to be receiving and transmitting simultaneously.

There were many ways to do this. I went with using .Poll(), this is a method, which sees if the stream is ready to transfer any data and returns positive if it is.

Technical solution

Data & Modules in classes

Class Game:

Private:

```
const int WHITE = 1;
const int BLACK = -1;
const int EMPTY_SPACE = 0;
const int PAWN = 1;
const int ROOK = 2;
const int KNIGHT = 3;
const int BISHOP = 4;
const int QUEEN = 5;
const int KING = 6;
const int PAWN_MOVED = 7;
const int KING_MOVED = 8;
const int ROOK_MOVED = 9;
const int PAWN_ENPASS = 10;
bool possible_move;
int pawnx;
int pawny;
Int[2] inhand;
int[8,8] gameboard;
int[,] Deselect_All();
void loadhand();
void King();
Void Pawn();
Void Bishop();
Void Knight();
Void Queen();
Void Rook();
Bool IsOccupied();
Int[,] backup();
Void Selected();
Void Remove_EnPassant();
Bool allowmove();
Public:
Game();
Bool IsItMyGo();
Int[,] findking();
Int returnpiece();
Void PawnPromo();
Int whitetime;
Int blacktime;
Bool whitego;
Bool promotion_Active;
Client client;
Void Connect();
Void Reload();//
Bool check();
Bool Any_Legal_Moves();
Void Clicked();
Bool Self_Check();
```

Class Chess (inherits form Class Form)

Private:
Game game;
Void TextBoxIP_Click();
Void Show_Menu_Pawn_Promotion();
Void Square_Clicked();
Void Call_Check();
Void PollServer_Tick();
PictureBox getbox();
Void DrawBoard();
Void Drawtobox();
Void Connect_Click();
Void White_Tick();
Void Black_Tick();
Void StartGame_Click();
Void Lan_Click();
Void Promobox_Click();
Public:
Chess()

Class Client:

Private:

Public:

Bool myturn;
String serverip;
Int port;
TcpClient link;
NetworkStream stream;
Bool found;
Client();
Bool connect();
Void send();
Int[,] Checkforboard();

```
Class Server:

Private:

Static int port;
Static TcpClient White;
Static TcpClient Black;
Static NetworkStream white_stream;
Static NetworkStream black_stream;
Static int[8,8] gameboard;
Static void update();
Static string boardasstring();
Static void InitServer();
Static byte[] bites();
Static void Main();
Public:
```

Techniques used

Complex User Defined Algorithms

Complex User-defined use of object orientated programming (OOP) including inheritance, composition, and dependency.

Complex client-server model

Multi-dimensional arrays

Use of external libraries

Styles

Excellent

Modules have appropriate interfaces, parameters have default values, all parameters are necessary.

Example: Seen throughout code, Game class line 600 is one example.

Modules interact with one another via their interfaces.

Example: Seen throughout code, Game class line 485 subroutine, Any_Legal_Move has good examples of this.

All Modules are cohesive, each carry out one task.

Example: The Key Data structures & Variable sheet show all modules purpose.

Subroutines with common purposes put together.

Example: In Game class:

```
//Additional Tools
3 references
private int[,] Deselect_All(int[,] board)...
1 reference
public int returnpiece(int x, int y)...
1 reference
private void Remove_EnPassant()...
1 reference
private int[] findking(bool black)...
2 references
private int[,] backup(int[,] temp)...
1 reference
public void reload() // resets gameboard
{
    whitego = true;
    gameboard = new int[,]...;
}
50 references
private bool IsOccupied(int x, int y, int piece = 0)...
9 references
private void loadhand(int x, int y)...
//Excellent coding style, Modules with default parameters.
```

```
//Finding & Selecting Moves
32 references
private bool allowmove(int x, int y, string move_name = "normal")...
//Group A Model = Complex User Defined algorithm.
1 reference
private void King( int x , int y)...
2 references
private void Bishop(int x, int y)...
1 reference
private void Knight(int x, int y)...
2 references
private void Rook(int x, int y)...
2 references
private void Pawn(bool first, int x, int y, bool black)...
1 reference
private void Selected(int x, int y, int select)...
```

```
//Program Flow
1 reference
public void Pawnpromo(int piece)...
2 references
public void clicked(int x, int y) //decides piece on sq
//Server Handling
1 reference
public bool IsItMyGo()...
1 reference
public bool connect() // connects to client...
//Check, Checkmate & Stalemate
1 reference
private bool Self_Check(int x, int y, string move_name)
//GROUP A Model - Complex Algorithm
3 references
public bool check(int prefix)...
2 references
public bool Any_Legal_Moves(int colour_king)...
```

Note how modules are grouped by their purpose.

Source code is comprehensible.

Example: Chess Class, all controls on the form are relevantly labelled.

Defensive programming is used.

Example: Chess Class line 158, a default case is used as fallback data, if no data is found, it will simply draw a selected square not crashing the program in case of a fault.

There is some exception handling.

Example: Client Class line 29, note how the program is looking for a specific exception to make sure it is not overly defensive.

Good

Code is in modules.

Example: See all of code.

Global variables are not used, all variables exist within a class.

Example: No global variables are used, all variables exist within their classes.

Constants have been well used.

Example: game Class line 10, the constants are used as values for pieces.

Appropriate indentation.

Example: See all of code

Code is documented.

Example: See all of code, Game class line

A consistent style is kept.

Example: See code

Files paths are parametrised.

Example: Game class line 174.

Basic

Meaningful identifier names used

Example: See code

Code is helpfully annotated.

Example: See code

Key Modules and variables

The following are modules that were not covered in the design aspect of this document:

Game

- Any_Legal_Moves() – This is possible_moves()
- IsOccupied() - This is Space_occupied()
- Findking() – finds the king.
- Reload() – resets the board on connection to a server.
- Remove_Enpassant() – takes away en passant from any pawn that are currently selected.

Chess

All of these refer to handling controls of the program.

Client

- Connect() – establishes connection with server.
- Send() – sends information to server.
- Checkforboard – polls the server and returns the board when it gets a response.

Server

- Bites() – Encodes data to be sent via the stream.
- Main() – Handles data from clients.
- Update() – changes the array, server's version of the board.
- InitServer() – Tells the owner of the server the port being used.

Client Side Code

Game Class

```
using System;

namespace ChessPro
{
    public class Game
    {
        //Excellent Coding styles - Modules with common purposes held together.

        //Good Coding Styles - Use of constants
        const int WHITE = 1;
        const int BLACK = -1;
        const int EMPTY_SPACE = 0;
        const int PAWN = 1;
        const int ROOK = 2;
        const int KNIGHT = 3;
        const int BISHOP = 4;
        const int QUEEN = 5;
        const int KING = 6;
        const int PAWN_MOVED = 7;
        const int KING_MOVED = 8;
        const int ROOK_MOVED = 9;
        const int PAWN_ENPASS = 10;
        private bool possible_move = true;
        private int pawnx;
        private int pawny;
        public int whitetime = 1800;
        public int blacktime = 1800;
        public bool whitego = true;
        public bool Promotion_Active = false;
        private int[] inhand = { 0, 0 };
        public Client client = new Client();
        //GROUP A MODEL - OOP COMPOSTION
        private int[,] gameboard = new int[,]
        {
            {-2,-3,-4,-5,-6,-4,-3,-2},
            {-1,-1,-1,-1,-1,-1,-1,-1},
            {0,0,0,0,0,0,0,0},
            {0,0,0,0,0,0,0,0},
            {0,0,0,0,0,0,0,0},
            {0,0,0,0,0,0,0,0},
            {1,1,1,1,1,1,1,1},
            {2,3,4,5,6,4,3,2}
        };
        //Group B Model - Multi-dimensional array
        public Game()
        {
```

```
47
48          }
49          //Finding & Selecting Moves
50          private bool allowmove(int x, int y, string move_name = "normal")
51          {
52              if (y < 0 || x < 0 || y > 7 || x > 7) { return false; } //Checks if square is
53  out of bounds
54              int spot = gameboard[x, y] * gameboard[inhand[1], inhand[0]]; //Checks square
55  is either free or taken by another piece.
56              if (spot <= 0)
57              {
58                  if (Self_Check(y, x, move_name)) { return spot == 0; }
59                  gameboard[x, y] += 50; // "Hashes" highlighted square.
60                  possible_move = true;
61                  if (move_name == "enpass")
62                  {
63                      gameboard[x, y] = 12; //Special case for en passant.
64                  }
65                  if (move_name == "castle_right")
66                  {
67                      gameboard[x, y] = 11;
68                  }
69                  if (move_name == "castle_left")
70                  {
71                      gameboard[x, y] = 13;
72                  }
73                  return (spot == 0);
74              }
75              return false;
76          }
77          //Group A Model = Complex User Defined algorithm.
78          private void King( int x , int y)
79          {
80              allowmove(y - 1, x);
81              allowmove(y - 1, x - 1);
82              allowmove(y - 1, x + 1);
83              allowmove(y, x + 1);
84              allowmove(y, x - 1);
85              allowmove(y + 1, x);
86              allowmove(y + 1, x - 1);
87              allowmove(y + 1, x + 1);
88              // Castling
89              // Left_side
90              if (gameboard[y, x] == KING || gameboard[y,x] == -KING)
91              {
92                  if (gameboard[y, 0] == ROOK || gameboard[y, 0] == -ROOK)
93                  {
94                      if(!(IsOccupied(y, x-1) || IsOccupied(y, x-2) || IsOccupied(y, x -
95  3)))
96                          {
97                          allowmove(y, x - 2, "castle_left");
98                      }
99                  }
100                 // Right_Side
101                 if (gameboard[y, 7] == ROOK || gameboard[y,7] == -ROOK)
102                 {
```

```
103                     if (!(IsOccupied(y, x + 1) || IsOccupied(y, x + 2))){
104                         allowmove(y, x + 2, "castle_right");
105                     }
106                 }
107             }
108         }
109         private void Bishop(int x, int y)
110         {
111                 for (int i = 1; i < 8; i++)
112                 {
113                     if(!allowmove(y + i, x + i)) { break; }
114                 }
115                 for (int i = 1; i < 8; i++)
116                 {
117                     if (!allowmove(y + i, x - i)) { break; }
118                 }
119                 for (int i = 1; i < 8; i++)
120                 {
121                     if (!allowmove(y - i, x + i)) { break; }
122                 }
123                 for (int i = 1; i < 8; i++)
124                 {
125                     if (!allowmove(y - i, x - i)) { break; }
126                 }
127 
128         }
129         private void Knight(int x, int y)
130         {
131             allowmove(y -2 , x + 1);
132             allowmove(y -2, x - 1);
133             allowmove(y + 2, x - 1);
134             allowmove(y + 2, x + 1);
135             allowmove(y + 1, x - 2);
136             allowmove(y + 1, x + 2);
137             allowmove(y - 1, x + 2);
138             allowmove(y - 1, x - 2);
139         }
140         private void Rook(int x, int y)
141         {
142             for (int i = x + 1; i < 8; i++)
143             {
144                 if (!allowmove(y, i)) { break; }
145             }
146             for (int i = x -1; i > -1; i--)
147             {
148                 if (!allowmove(y, i)) { break; }
149             }
150             for (int i = y + 1; i < 8; i++)
151             {
152                 if (!allowmove(i, x)) { break; }
153             }
154             for (int i = y -1; i > -1; i--)
155             {
156                 if (!allowmove(i, x)) { break; }
157             }
158         }
```

```
159         private void Pawn(bool first, int x, int y, bool black)
160         {
161             int direction = -1;
162             if (black) { direction = 1; } //Gets Direction that pawn is going
163             if (IsOccupied(y + direction, x - 1))
164             {
165                 allowmove(y + direction, x - 1);
166             }
167             if (IsOccupied(y + direction, x + 1))
168             {
169                 allowmove(y + direction, x + 1); // Handles taking pieces diagonally
170             }
171             if (IsOccupied(y, x- 1, direction *10)) {
172                 allowmove(y + direction, x -1, "enpass"); }
173             if (IsOccupied(y, x +1, direction * 10)) {
174                 allowmove(y + direction, x + 1, "enpass"); } // Checks for en passant
175             if (IsOccupied(y + direction, x)) { return; } //Checks piece above for
176 obstructions
177             if(!allowmove(y + direction , x)) { return; } //Highlight piece aboce it, ends
178 routine if blocked
179             if (first)
180             {
181                 if (IsOccupied(y + direction * 2, x)) { return; } //Checks two pieces up
182 for obstruction
183                 allowmove(y + (2 * direction), x); //Highlights piece above it.
184             }
185 
186         }
187         private void Selected(int x, int y, int select)
188         {
189             bool endturn = true;
190             Remove_EnPassant();
191             int piece_inhand = gameboard[inhand[1], inhand[0]];
192             switch (piece_inhand)
193             {
194                 case -PAWN:
195                 case PAWN:
196                     if (Math.Abs(y - inhand[1]) == 2)
197                     {
198                       piece_inhand *= PAWN_ENPASS; // select the piece for en passant IF
199 (It is a pawn) && (It has moved two places).
200                     }
201                     else
202                     {
203                         piece_inhand *= PAWN_MOVED;   //else set it to a regular moved
204 pawn.
205                     }
206                     break;
207                 case PAWN_MOVED:
208                 case -PAWN_MOVED:
209                     if(y == 0 || y == 7)
210                     {
211                         endturn = false;
212                         //If a pawn is at the end of the board,
213                         //it needs to be promoted ergo the turn is not over yet.
214                     }
```

```
215                     break;
216                 case ROOK:
217                     piece_inhand = ROOK_MOVED;
218                     break;
219                 case -ROOK:
220                     piece_inhand = -ROOK_MOVED;
221                     break;
222                 case KING:
223                     piece_inhand = KING_MOVED;
224                     break;
225                 case -KING:
226                     piece_inhand = -KING_MOVED;
227                     break;
228                 default:
229                   break;
230 
231             }
232             gameboard[y, x] = piece_inhand;
233             gameboard[inhand[1], inhand[0]] = EMPTY_SPACE;
234             if (select == 12)// en pass
235             {
236                 gameboard[y + (piece_inhand / 7), x] = 0;
237             }
238             if (select == 13)//castle left
239             {
240                 gameboard[y, x + 1] = gameboard[y, 0];
241                 gameboard[y, 0] = 0;
242             }
243             if (select == 11)//castle right
244             {
245                 gameboard[y, x - 1] = gameboard[y, 7];
246                 gameboard[y, 7] = 0;
247             }
248             if (endturn)
249             {
250                 whitego = !whitego;
251                 if (select == 12)
252                 {
253                     client.send(x, y, inhand[1], inhand[0], 2 * piece_inhand);
254                 }
255                 else if (select == 11)
256                 {
257                     client.send(x, y, inhand[1], inhand[0], 2 * piece_inhand);
258                 }
259                 else if (select == 13)
260                 {
261                     client.send(x, y, inhand[1], inhand[0], 3 * piece_inhand);
262                 }
263                 else
264                 {
265                     client.send(x, y, inhand[1], inhand[0], piece_inhand);
266                 }
267             }
268             else
269             {
270                 Promotion_Active = true;
```

```
271                 pawnx = x;
272                 pawny = y;
273             }
274         }
275         //Program Flow
276         public void Pawnpromo(int piece)
277         {
278             piece *= (gameboard[pawny, pawnx] / 7);
279             gameboard[pawny, pawnx] = piece;
280             client.send(pawnx, pawny, inhand[1], inhand[0], piece);
281             Promotion_Active = false;
282             gameboard[inhand[1], inhand[0]] = 0;
283             whitego = !whitego;
284         }
285         public void clicked(int x, int y) //decides piece on square and calls function for
286 it
287         {
288             int piece = gameboard[y, x];
289             gameboard = Deselect_All(gameboard);
290             if (whitego == false && (piece > 0 && piece < 11)) { return; }
291             if (whitego == true && (piece < 0)) { return; }
292             switch (Math.Abs(piece))
293             {
294                 case 0: //free space
295                     break;
296                 case PAWN:
297                     loadhand(x, y);
298                     Pawn(true, x, y, (piece < 0));
299                     break;
300                 case ROOK_MOVED:
301                 case ROOK:
302                     loadhand(x, y);
303                     Rook(x, y);
304                     break;
305                 case KNIGHT:
306                     loadhand(x, y);
307                     Knight(x, y);
308                     break;
309                 case BISHOP:
310                     loadhand(x, y);
311                     Bishop(x, y);
312                     break;
313                 case QUEEN:
314                     loadhand(x, y);
315                     Bishop(x, y);
316                     Rook(x, y);
317                     break;
318                 case KING:
319                 case KING_MOVED:
320                     loadhand(x, y);
321                     King(x, y);
322                     break;
323                 case PAWN_MOVED:
324                 case PAWN_ENPASS:
325                     loadhand(x, y);
326                     Pawn(false, x, y, (piece < 0));
```

```
327                     break;
328                 default:
329                     Selected(x, y, piece);
330                     break;
331             }
332         }
333         //Server Handling
334         public bool IsItMyGo()
335         {
336             if (client.found)
337             {
338                 int[,] newboard = new int[8, 8];
339                 newboard = client.Checkforboard(); //Change, so one doesn't need knowledge
340 of how client.Checkforboard works
341                 if (newboard != null)
342                 {
343                     gameboard = newboard;
344                     whitego = !whitego;
345                     return true;
346                 }
347                 return false;
348             }
349             return false;
350         }
351         public bool connect() // connects to client
352         {
353             return client.connect();
354         }
355         //Check, Checkmate & Stalemate
356         private bool Self_Check(int x, int y, string move_name)
357         {
358             bool incheck;
359             int colour = BLACK;
360             if (whitego)
361             {
362                 colour = WHITE;
363             }
364             int[,] temp = new int[8, 8];
365             temp = backup(temp);
366             gameboard = Deselect_All(gameboard);
367             gameboard[y, x] = gameboard[inhand[1], inhand[0]];
368             gameboard[inhand[1], inhand[0]] = 0;
369             if (move_name == "enpass")
370             {
371                 gameboard[y + colour, x] = 0;
372             }
373             if (move_name == "castle_right")
374             {
375                 gameboard[y, x - 1] = gameboard[y, 7];
376                 gameboard[y, 7] = 0;
377             }
378             if (move_name == "castle_left")
379             {
380                 gameboard[y, x + 1] = gameboard[y, 0] = 0;
381                 gameboard[y, 0] = 0;
382             }
```

```
383             if (whitego)
384             {
385                 incheck = check(WHITE);
386             }
387             else
388             {
389                 incheck = check(BLACK);
390             }
391             gameboard = temp;
392             return incheck;
393         }
394         //GROUP A Model - Complex Algorithm
395         public bool check(int prefix)
396         {
397             int[] pos;
398             pos = findking(prefix == -1);
399             prefix *= -1; //prefix refers to piece which can take you.
400             if (pos == null) { return false; }
401             int y = pos[0];
402             int x = pos[1];
403             if (IsOccupied(y + prefix, x + 1, prefix * (PAWN)))
404             {
405                 return true;
406             }
407             if (IsOccupied(y + prefix, x - 1, prefix * (PAWN)))
408             {
409                 return true;
410             } //Can pawns take the king.
411             bool take = false;
412             for (int u = 0; u < 8; u++)
413             {
414                 if (IsOccupied(y, u, prefix * ROOK) || IsOccupied(y, u, prefix * QUEEN))
415                 {
416                     take = true;
417                     if (u > x) { return true; }
418                 }
419                 else if (IsOccupied(y, u) && !IsOccupied(y, u, -prefix * KING))
420                 {
421                     if (u > x) { break; }
422                     take = false;
423                 }
424             }
425             if (take) { return true; }
426             for (int u = 0; u < 8; u++)
427             {
428                 if (IsOccupied(u, x, prefix * ROOK) || IsOccupied(u, x, prefix * QUEEN))
429                 {
430                     take = true;
431                     if (u > y) { return true; }
432                 }
433                 else if (IsOccupied(u, x) && !IsOccupied(u, x, -prefix * KING))
434                 {
435                     if (u > y) { break; }
436                     take = false;
437                 }
438             }
```

```
439             if (take) { return true; }
440             { //knights
441                 if (IsOccupied(y - 2, x + 1, prefix * 3)) { return true; }
442                 if (IsOccupied(y - 2, x - 1, prefix * 3)) { return true; }
443                 if (IsOccupied(y + 2, x - 1, prefix * 3)) { return true; }
444                 if (IsOccupied(y + 2, x + 1, prefix * 3)) { return true; }
445                 if (IsOccupied(y + 1, x - 2, prefix * 3)) { return true; }
446                 if (IsOccupied(y + 1, x + 2, prefix * 3)) { return true; }
447                 if (IsOccupied(y - 1, x + 2, prefix * 3)) { return true; }
448                 if (IsOccupied(y - 1, x - 2, prefix * 3)) { return true; }
449             }// bishops and queens
450             for (int i = 1; i < 8; i++)
451             {
452                 if (IsOccupied(y + i, x - i, prefix * BISHOP)) { return true; }
453                 if (IsOccupied(y + i, x - i, prefix * QUEEN)) { return true; }
454                 else if (IsOccupied(y + i, x - i)) { break; }
455             }
456             for (int i = 1; i < 8; i++)
457             {
458                 if (IsOccupied(y - i, x + i, prefix * BISHOP)) { return true; }
459                 if (IsOccupied(y - i, x + i, prefix * QUEEN)) { return true; }
460                 else if (IsOccupied(y - i, x + i)) { break; }
461             }
462             for (int i = 1; i < 8; i++)
463             {
464                 if (IsOccupied(y + i, x + i, prefix * BISHOP)) { return true; }
465                 if (IsOccupied(y + i, x + i, prefix * QUEEN)) { return true; }
466                 else if (IsOccupied(y + i, x + i)) { break; }
467             }
468             for (int i = 1; i < 8; i++)
469             {
470                 if (IsOccupied(y - i, x - i, prefix * BISHOP)) { return true; }
471                 if (IsOccupied(y - i, x - i, prefix * QUEEN)) { return true; }
472                 else if (IsOccupied(y - i, x - i)) { break; }
473             }
474             //king
475             if (IsOccupied(y - 1, x + 1, prefix * 6)) { return true; }
476             if (IsOccupied(y - 1, x - 1, prefix * 6)) { return true; }
477             if (IsOccupied(y - 1, x, prefix * 6)) { return true; }
478             if (IsOccupied(y + 1, x + 1, prefix * 6)) { return true; }
479             if (IsOccupied(y + 1, x - 1, prefix * 6)) { return true; }
480             if (IsOccupied(y + 1, x, prefix * 6)) { return true; }
481             if (IsOccupied(y, x + 1, prefix * 6)) { return true; }
482             if (IsOccupied(y, x - 1, prefix * 6)) { return true; }
483             return false;
484         }
485         public bool Any_Legal_Moves(int colour_king)
486         {
487             int tx = inhand[0];
488             int ty = inhand[1];
489             int[,] temp = new int[8, 8];
490             temp = backup(temp);
491             gameboard = Deselect_All(gameboard);
492             possible_move = false;
493             for (int i = 0; i < 8; i++)
494             {
```

```
                for (int k = 0; k < 8; k++)
                {
                    if (gameboard[i, k] * colour_king > 0)
                    {
                        clicked(k, i);
                        if (possible_move == true)
                        {
                            gameboard = temp;
                            loadhand(tx, ty);
                            return false;
                        }
                    }
                }
            }
            loadhand(tx, ty);
            gameboard = temp;
            return true;
        }
        //Additional Tools
        private int[,] Deselect_All(int[,] board)
        {
            for(int i = 0; i< 8; i++)
            {
                for(int k = 0; k < 8; k++)
                {
                    if (Math.Abs(board[i,k]) > 13) // A Selected number: 50 + piece_num
                    {
                        board[i, k] -= 50;
                    }
                    if(board[i,k] == 12 || board[i,k] == 11 || board[i,k] == 13) //
Handles highlightedd squares under en passant.
                    {
                        board[i, k] = 0;
                    }
                }
            }
            return gameboard;
        }
        public int returnpiece(int x, int y)
        {
            return gameboard[y, x];
        }
        private void Remove_EnPassant()
        {
            for (int i = 0; i < 8; i++)
            {
                for (int k = 0; k < 8; k++)
                {
                    if (gameboard[i, k] == 10)
                    {
                        gameboard[i, k] = 7;
                    }
                    if (gameboard[i, k] == -10)
                    {
                        gameboard[i, k] = -7;
                    }
```

```
551                 }
552             }
553         }
554         private int[] findking(bool black)
555         {
556             int prefix = 1;
557             int[] pos = new int[2];
558             if (black) { prefix *= -1; }
559             for (int i = 0; i < 8; i++)
560             {
561                 for (int k = 0; k < 8; k++)
562                 {
563                     if (IsOccupied(i,k,prefix*KING))
564                     {
565                         pos[0] = i;
566                         pos[1] = k;
567                         return pos;
568                     }
569                 }
570             }
571             return null;
572         }
573         private int[,] backup(int[,] temp)
574         {
575             for (int i = 0; i < 8; i++)
576             {
577                 for (int k = 0; k < 8; k++)
578                 {
579                     temp[i, k] = gameboard[i, k];
580                 }
581             }
582             return temp;
583         }
584         public void reload() // resets gameboard
585         {
586             whitego = true;
587             gameboard = new int[,]
588         {
589             {-2,-3,-4,-5,-6,-4,-3,-2},
590             {-1,-1,-1,-1,-1,-1,-1,-1},
591             {0,0,0,0,0,0,0,0},
592             {0,0,0,0,0,0,0,0},
593             {0,0,0,0,0,0,0,0},
594             {0,0,0,0,0,0,0,0},
595             {1,1,1,1,1,1,1,1},
596             {2,3,4,5,6,4,3,2}
597             //GROUP B Model - Multidimensional array
598         };
599         }
600         private bool IsOccupied(int x, int y, int piece = 0)
601         {
602             int altpiece = 400; //Arbitary number.
603             int altpiece_two = 400;
604             if (y >= 0 && x >= 0 && y <= 7 && x <= 7)
605             {
606                 if (piece == PAWN) { altpiece = PAWN_MOVED; altpiece_two = PAWN_ENPASS; }
```

```
607                 if (piece == -PAWN) { altpiece = -PAWN_MOVED; altpiece_two = PAWN_ENPASS;
608 }
609                 if (piece == KING) { altpiece = KING_MOVED; }
610                 if (piece == ROOK) { altpiece = ROOK_MOVED; }
611                 if (piece == -KING) { altpiece = -KING_MOVED; }
612                 if (piece == -ROOK) { altpiece = -ROOK_MOVED; }
613                 if (piece != 0)
614                 {
615                     return (gameboard[x, y] == piece ||
616                             gameboard[x, y] == altpiece ||
617                             gameboard[x, y] == altpiece_two); ;
618                 }
                    return (gameboard[x, y] != 0) && (gameboard[x, y] != 50);
                }
                return false;
            }
            private void loadhand(int x, int y)
            {
                inhand[0] = x;
                inhand[1] = y;
            }
            //Excellent coding style, Modules with default parameters.
        }
    }
```

Chess class

```
 1 using System;
 2 using System.Drawing;
 3 using System.Linq;
 4 using System.Windows.Forms;
 5
 6 namespace ChessPro
 7 {
 8     public partial class Chess : Form
 9         //GROUP A MODEL - INHERITANCE
10     {
11         Game game = new Game();
12         //GROUP A MODEL - OOP COMPOSITION
13         public Chess()
14         {
15             InitializeComponent();
16             this.Size = new Size(275, 195);
17             //Load elements for main menu.
18             MainMenu_Background.BringToFront();
19             Lan_Menu.BringToFront();
20             StartGame.BringToFront();
21             Logo_Text.BringToFront();
22             //prevents us having to create a seperate form, which will hog resources.
23             DrawBoard(); // Draw the board.
24             Show_Menu_Pawn_Promotion(false); //Hide pawn promotion menu
25         }
26         private void Show_Menu_Pawn_Promotion(bool Show) // Shows or hides pawn promo menu
27         {
```

```
28              if (Show)
29              {
30                  Q5.Show();
31                  K3.Show();
32                  R2.Show();
33                  B4.Show();
34                  Pawn_Menu_Background.Show();
35              }
36              else
37              {
38                  Pawn_Menu_Background.Hide();
39                  Q5.Hide();
40                  K3.Hide();
41                  R2.Hide();
42                  B4.Hide();
43              }
44          }
45          private void Square_Clicked(object sender, EventArgs e) //Handles selecting square
46  // rename
47          {
48              if (game.client.found)
49              {
50                  if (game.whitego && game.client.name == "black") { return; }
51                  if (!game.whitego && game.client.name == "white") { return; }
52              }
53              PictureBox me = sender as PictureBox;
54              int x = Int32.Parse(me.Name[1].ToString());
55              int y = Int32.Parse(me.Name[2].ToString());
56              if (!game.Promotion_Active)
57              {
58                  game.clicked(x, y);
59              }
60              if (game.Promotion_Active)
61              {
62                  Show_Menu_Pawn_Promotion(true);
63                  return;
64                  //This forces the player to promote their pawn.
65              }
66              DrawBoard(); //Draw the new board.
67              call_check(game.whitego);
68          }
69          private void call_check(bool forwhite)
70          {
71              bool incheck = false;
72              string name;
73              int colour;
74              if (forwhite)
75              {
76                  name = "White";
77                  colour = 1;
78              }
79              else
80              {
81                  name = "Black";
82                  colour = -1;
83              }
```

```
            //BLACK
            if (game.check(colour))
            {
                label2.Text = name + " Check";
                incheck = true;
            }
            else { label2.Text = name+ "'s Move"; }
            if (game.Any_Legal_Moves(colour))
            if (game.Any_Legal_Moves(colour))
            {
                label2.Location = new Point(200, 3);
                if (incheck)
                {
                    label2.Text = name + " Checkmate";
                }
                else {
                    label2.Text = name + " Stalemate";
                }
            }
        }
        private void PollServer_Tick(object sender, EventArgs e) //Rename- make clear what
timer1 ticks for
        {
            if (game.IsItMyGo())
            {
                DrawBoard();
                call_check(game.whitego);
            }
        }
        private PictureBox getbox(int[] coord) //return a picturebox given location -
finds by name.
        {
            string name = "p" + coord[0].ToString() + coord[1].ToString();
            PictureBox square = this.Controls.Find(name, true).First() as PictureBox;
            return square;

        }
        private void DrawBoard() // Draws new board. Rename
        {
            string piece = "";
            for (int x = 0; x < 8; x++)
            {
                for (int y = 0; y < 8; y++)
                {
                    int piecenum = game.returnpiece(x, y);
                    switch (Math.Abs(piecenum))
                    {
                        case 7:
                            piece = "pawn";
                            break;
                        case 10: // En Passant
                        case 1:
                            piece = "pawn";
                            break;
                        case 9:
                        case 2:
```

```
                            piece = "rook";
                            break;
                        case 3:
                            piece = "knight";
                            break;
                        case 4:
                            piece = "bishop";
                            break;
                        case 5:
                            piece = "queen";
                            break;
                        case 8:
                        case 6:
                            piece = "king";
                            break;
                        case 0:
                            piece = "clear";
                            break;
                        default:
                            piece = "selected";
                            break;
                    }
                    if (piecenum < 0) { piece += "b.png"; }
                    else { piece += ".png"; }
                    if (((x + y) % 2 == 0) && !piece.Contains("selected")) { piece =
"light" + piece; }
                    int[] box = { x, y };
                    Drawtobox(getbox(box), piece);
                }
            }
        }
        private void Drawtobox(PictureBox set, string filename) // Changes images on box
        {
            PictureBox draw = set as PictureBox;
            draw.Load("art/" + filename);
        }
        private void Connect_Click(object sender, EventArgs e) // connects to server
        {
            if (TextBoxIP.Text[0].ToString() != "E") { game.client.serverip =
TextBoxIP.Text; }
            game.reload();
            game.whitetime = 1801;
            game.blacktime = 1801;
            DrawBoard();
            if (game.connect()) { label1.Text = "Connected: " + game.client.name;
Connect.Text = "Disconnect"; }
            else { label1.Text = "Server was unable to find another player."; }

        }
        private void Promobox_Click(object sender, EventArgs e) // handles pawn promotion
        {
            PictureBox me = sender as PictureBox;
            game.Pawnpromo(Int32.Parse(me.Name[1].ToString()));
            Show_Menu_Pawn_Promotion(false);
            DrawBoard();
        }
```

```
196         private void White_Tick(object sender, EventArgs e)
197         {
198             if (game.whitego)
199             {
200                 Black_Clock.ForeColor = Color.White;
201                 White_Clock.ForeColor = Color.Red;
202                 game.whitetime--;
203                 string seconds = (game.whitetime % 60).ToString();
204                 if (seconds.Length == 1) { seconds = "0" + seconds; }
205                 White_Clock.Text = (game.whitetime / 60).ToString() + ":" + seconds;
206             }
207         }
208 
209         private void Black_Tick(object sender, EventArgs e)
210         {
211             if (!game.whitego)
212             {
213                 White.Enabled = true;
214                 Black_Clock.ForeColor = Color.Red;
215                 White_Clock.ForeColor = Color.White;
216                 game.blacktime--;
217                 string seconds = (game.blacktime % 60).ToString();
218                 if (seconds.Length == 1) { seconds = "0" + seconds; }
219                 Black_Clock.Text = (game.blacktime / 60).ToString() + ":" + seconds;
220             }
221         }
222         private void StartGame_Click(object sender, EventArgs e)
223         {
224             this.Size = new Size(655, 720);
225             StartGame.Dispose();
226             MainMenu_Background.Dispose();
227             Lan_Menu.Dispose();
228             Logo_Text.Dispose();
229         }
230         private void TextBoxIP_Click(object sender, EventArgs e)
231         {
232             TextBoxIP.Text = "";
233         }
234         private void Lan_Click(object sender, EventArgs e)
235         {
236             this.Size = new Size(655, 800);
237             StartGame.Dispose();
238             MainMenu_Background.Dispose();
                Lan_Menu.Dispose();
                Logo_Text.Dispose();
            }

        }
    }
```

Client Class

```
1  using System;
2  using System.Text;
3  using System.Net.Sockets;
4  namespace ChessPro
5  {
6      public class Client
7      {
8          public bool myturn;
9          public string serverip = "192.168.0.87";
10         public int port = 11202;
11         public string name = "";
12         TcpClient link = new TcpClient();
13         NetworkStream stream = default(NetworkStream);
14         public bool found = false;
15         //private string hostip;
16         public Client()
17         {
18             link.ReceiveTimeout = 10000;
19         }
20         public bool connect(string ip = "127.0.0.1")
21         {
22             string[] portandip = new string[2];
23             if (serverip.Contains(":"))
24             {
25                 portandip = serverip.Split(':');
26                 serverip = portandip[0];
27                 port = Int32.Parse(portandip[1]);
28             }
29             try
30             {
31                 link.Connect(serverip, port);
32                 stream = link.GetStream();
33                 byte[] buffer = new byte[5];
34                 stream.Read(buffer, 0, 5);
35                 name = Encoding.ASCII.GetString(buffer);
36                 if(name == "black") { myturn = false; }
37                 else { myturn = true; }
38                 found = true;
39                 return true;
40             } //Excellent coding style - Good exception handling.
41             catch (System.Net.Sockets.SocketException)
42             {
43                 found = false;
44                 return false;
45             }
46         }
47         public void send(int x, int y, int oldx, int oldy, int piece)
48         {
49             if (found)
50             {
51                 string message =
52                     name[0].ToString()
```

```
                    + x.ToString()
                    + y.ToString()
                    + oldx.ToString()
                    + oldy.ToString()
                    + piece.ToString()
                    + "#";
                byte[] send = (Encoding.ASCII.GetBytes(message));
                stream.Write(send, 0, send.Length);
                stream.Flush();
            }
        }
        public int[,] Checkforboard()
        {
            byte[] board_buffer = new byte[250];
            if (link.Client.Poll(2500, SelectMode.SelectRead))
            {
                stream.Read(board_buffer, 0, board_buffer.Length);
            }
            else { return null; }
            myturn = true;
            int[,] boardint = new int[8, 8];
            string stringboard = Encoding.ASCII.GetString(board_buffer, 0,
board_buffer.Length);
            int x = 0, y = 0;
            string piece = "";
            for (int i = 0; i < stringboard.Length; i++) //expects 2#3#4#5#6#4#3#2#
            {
                if (stringboard[i].ToString() == "#" && y < 8)
                {
                    boardint[y, x] = Int32.Parse(piece);
                    x++;
                    if (x > 7)
                    {
                        y++;
                        x = 0;
                    }
                    piece = "";
                }
                else
                {
                    piece += stringboard[i].ToString();
                }
            }
            return boardint;
        }
    }
}
```

Server Side code

Server class

```
using System;
using System.Text;
using System.Net;
using System.Net.Sockets;
namespace ChessProServer
{
    class Server
    {
        static int port = 11202;
        static TcpClient White = default(TcpClient);
        static TcpClient Black = default(TcpClient);
        static NetworkStream white_stream = default(NetworkStream);
        static NetworkStream black_stream = default(NetworkStream);
        //GROUP A MODEL - Dependancy, complex OOP
        static int[,] gameboard = new int[,]
{
            {-2,-3,-4,-5,-6,-4,-3,-2},
            {-1,-1,-1,-1,-1,-1,-1,-1},
            {0,0,0,0,0,0,0,0},
            {0,0,0,0,0,0,0,0},
            {0,0,0,0,0,0,0,0},
            {0,0,0,0,0,0,0,0},
            {1,1,1,1,1,1,1,1},
            {2,3,4,5,6,4,3,2}};
        private static void update(string msg)
        {
            Remove_EnPassant();
            int piece = 0;
            msg = msg.Substring(1, msg.IndexOf("#") - 1);
            int x = Int32.Parse(msg[0].ToString());
            int y = Int32.Parse(msg[1].ToString());
            int oldx = Int32.Parse(msg[2].ToString());
            int oldy = Int32.Parse(msg[3].ToString());
            piece = Int32.Parse(msg.Substring(4));
            if(piece == -14 || piece == 14)
            {
                piece /= 2;
                gameboard[y + (piece / 7), x] = 0;
            }
            if(piece == -16 ||  piece == 16)
            {
                piece /= 2;
                gameboard[y, x - 1] = gameboard[y, 7];
                gameboard[y, 7] = 0;
            }
            if(piece == -24 || piece == 24)
            {
                piece /= 3;
                gameboard[y, x + 1] = gameboard[y, 0];
                gameboard[y, 0] = 0;
```

```
51              }
52              gameboard[y, x] = piece;
53              gameboard[oldx, oldy] = 0;
54          }
55          private static void Remove_EnPassant()
56          {
57              for (int i = 0; i < 8; i++)
58              {
59                  for (int k = 0; k < 8; k++)
60                  {
61                      if (gameboard[i, k] == 10)
62                      {
63                          gameboard[i, k] = 7;
64                      }
65                      if (gameboard[i, k] == -10)
66                      {
67                          gameboard[i, k] = -7;
68                      }
69                  }
70              }
71          }
72          private static string boardasstring()
73          {
74              string boardstring = "";
75              for (int i = 0; i < 8; i++)
76              {
77                  for (int k = 0; k < 8; k++)
78                  {
79                      boardstring += gameboard[i, k].ToString() + "#";
80                  }
81              }
82              return boardstring;
83          }
84          private static void InitServer()
85          {
86              Console.WriteLine("Using port: " + port);
87          }
88          private static byte[] bites(string message)
89          {
90              return Encoding.ASCII.GetBytes(message);
91          }
92          private static void Main()
93          {
94              byte[] board = new byte[250];
95              InitServer();
96              IPAddress ip = System.Net.IPAddress.Any;
97              TcpListener server = new TcpListener(ip, port);
98              try
99              {
100                 server.Start();
101                 Console.WriteLine("Attempting to find player...");
102                 White = server.AcceptTcpClient();
103                 white_stream = White.GetStream();
104                 white_stream.Write(bites("white"), 0, 5);
105                 white_stream.Flush();
106                 Console.WriteLine("Player one found!");
```

```
107                 Console.WriteLine("Attempting to find second player...");
108                 Black = server.AcceptTcpClient();
109                 black_stream = Black.GetStream();
110                 black_stream.Write(bites("black"), 0, 5);
111                 black_stream.Flush();
112                 /*white_stream.Write(bites("start"), 0, 5);
113                 white_stream.Flush();
114                 black_stream.Write(bites("start"), 0, 5);
115                 black_stream.Flush();*/
116                 Console.WriteLine("Player two found!");
117                 White.ReceiveTimeout = 5000;
118                 Black.ReceiveTimeout = 5000;
119              }
120             catch (Exception ex)
121             {
122                 return;
123             }
124             while(Black.Connected && White.Connected)
125             {
126                 byte[] buffer = new byte[10];
127                 if (White.Client.Poll(100, SelectMode.SelectRead))
128                 {
129                     if (White.Connected)
130                     {
131                         white_stream.Read(buffer, 0, buffer.Length);
132                     }
133                     else
134                     {
135                         Disconnect_Client(ref White);
136                         break;
137                     }
138                     Console.WriteLine(Encoding.ASCII.GetString(buffer));
139                     update(Encoding.ASCII.GetString(buffer));
140                     string boardstring = boardasstring();
141                     board = Encoding.ASCII.GetBytes(boardstring);
142                     black_stream.Write(board, 0, board.Length);
143 
144                 }
145                 if (Black.Client.Poll(100, SelectMode.SelectRead))
146                 {
147                     black_stream.Read(buffer, 0, buffer.Length);
148                     Console.WriteLine(Encoding.ASCII.GetString(buffer));
149                     update(Encoding.ASCII.GetString(buffer));
150                     string boardstring = boardasstring();
151                     board = Encoding.ASCII.GetBytes(boardstring);
152                     white_stream.Write(board, 0, board.Length);
153                 }
154 
155             }
156 
157         }
158         private static void Disconnect_Client(ref TcpClient client)
159         {
160             client.Dispose();
161             Console.WriteLine("A Client has disconnected");
162 
```

Testing

Testing is vital as it allows us to have quality control. I have tested this program both whilst creating it and at the end. My final testing plan will consist of four parts:

- Testing moves.
- Testing detection of board states.
- Testing connection with server.
- Holistic testing.

Each test will have three outcomes:

- SUCCESS
  - ❖ Criteria is fully met.
- PASS
  - ❖ Criteria is met to a viable standard.
- FAIL
  - ❖ Criteria is not met to a viable standard.

- **WARNING: COMMENTS ON THE YOUTUBE VIDEOS HAVE LINKS TO MALCIOUS WEBSITES, DO NOT CLICK ON ANY OF THESE LINKS.**

Testing moves.

PAWN	Link to test: https://youtu.be/pRzhX8ClmeY		
Test ID	Criteria	Outcome	Notes
1	Legal moves are shown for a pawn's first move.	SUCCESS	Works for both black and white pieces. Pawn is unable to move diagonally if not capturing or en passant.
2	Legal moves are shown for a pawn's non-first move.	SUCCESS	Works for both black and white pieces. Pawn is unable to move diagonally if not capturing or en passant. The pawn cannot move vertically by two squares.
3	Menu for promotion is seen when pawn reaches final rank.	SUCCESS	Works for both black and white pieces. Only white symbols are used for promotion choices.

4	A Pawn uses pawn promotion to change into a queen.	SUCCESS	Works for both black and white pieces.
5	A pawn uses pawn promotion to change into a rook.	SUCCESS	Works for both black and white pieces.
6	A pawn uses pawn promotion to change into a bishop	SUCCESS	Works for both black and white pieces.
7	A pawn uses pawn promotion to change into a knight	SUCCESS	Works for both black and white pieces.
8	Legal moves are shown for a pawn able to move via en passant.	SUCCESS	Works for both black and white pieces. Pawn cannot capture opposition pawn after another move.
9	When a pawn moves via en passant, opponent piece is captured.	SUCCESS	Works for both black and white pieces.
10	No illegal moves can be made	SUCCESS	Works for both black and white pieces.

Rook	Link to video: https://youtu.be/ErhlahTt87I		
Test ID	Criteria	Outcome	Notes
1	All Legal Rook moves – in north direction shown.	SUCCESS	Works for both black and white pieces.
2	All Legal Rook moves – in south direction shown.	SUCCESS	Works for both black and white pieces.
3	All Legal Rook moves – in west direction shown.	SUCCESS	Works for both black and white pieces.

4	All Legal Rook moves – in east direction shown	SUCCESS	Works for both black and white pieces.
5	No illegal moves can be made	SUCCESS	Works for both black and white pieces.
5	A rook is moved to the correct position when west side castling is done.	SUCCESS	Works for both black and white pieces. Rook moves three squares towards the west.
6	A rook is moved to the correct position when east side castling is done.	SUCCESS	Works for both black and white pieces. Rook moves two squares towards the east.
7	No illegal moves can be made	SUCCESS	Works for both black and white pieces.

Knight	Link to video: https://youtu.be/3Df5ulp8CFI		
Test ID	Criteria	Outcome	Notes
1	All Legal moves Knight moves are shown.	SUCCESS	Works for both black and white pieces.
2	Knight can "hop" over pieces.	SUCCESS	Works for both black and white pieces.
3	No illegal moves can be made	SUCCESS	Works for both black and white pieces.

Bishop	Link to video: https://youtu.be/4LdwaRhWe8M		
Test ID	Criteria	Outcome	Notes
1	All Legal moves – in north-east direction are shown.	SUCCESS	Works for both black and white pieces.

2	All Legal moves – in north-west direction are shown.	SUCCESS	Works for both black and white pieces.
3	All Legal moves – in south-east direction are shown.	SUCCESS	Works for both black and white pieces.
4	All Legal moves – in south-west direction are shown.	SUCCESS	Works for both black and white pieces.
5	No illegal moves can be made.	SUCCESS	Works for both black and white pieces.

King	Link to video: https://youtu.be/GOeCa8BmsLc		
Test ID	Criteria	Outcome	Notes
1	All Legal moves shown – not including castling.	SUCCESS	Works for both black and white king.
2	All Legal castling moves shown, for first move.	PASS	Works for both black and white king. Moves not shown if it is not the kings or the rook's first move. King can castle through check.
3	King is moved to correct position for west side castling.	SUCCESS	Works for both black and white king. King is moved two pieces to the west.
4	King is moved to correct position for east side castling.	SUCCESS	Works for both black and white king. King is moved two pieces two the west.
5	No illegal moves are highlighted	PASS	The king can castle through check. I found no other illegal moves that were allowed.

Queen	Link to video: https://youtu.be/l4rnuMiU2M0		
Test ID	Criteria	Outcome	Notes
1	All Legal moves – in north direction shown.	SUCCESS	Works for both black and white pieces.
2	All Legal moves – in south direction shown.	SUCCESS	Works for both black and white pieces.
3	All Legal moves – in west direction shown.	SUCCESS	Works for both black and white pieces.
4	All Legal moves – in east direction shown.	SUCCESS	Works for both black and white pieces.
5	All Legal moves – in north-west direction shown.	SUCCESS	Works for both black and white pieces.
6	All Legal moves – in north-east direction shown.	SUCCESS	Works for both black and white pieces.
7	All Legal moves – in south-east direction shown.	SUCCESS	Works for both black and white pieces.
8	All Legal moves – in south-west direction shown.	SUCCESS	Works for both black and white pieces.
9	No illegal moves are highlighted	SUCCESS	Works for both black and white pieces.

General (All pieces)			
Test ID	Criteria	Outcome	Notes
1	When a highlighted square is chosen the relevant piece is loaded into the square.	SUCCESS	Works for both black and white moves. Seen in many tests.
2	When the contents of one square are loaded into another square, the initial square is cleared.	SUCCESS	Works for both black and white moves. Seen in many tests.
3	Left Side castling can be done	SUCCESS	Works for both black and white moves. Link: https://youtu.be/AiDjzD7tVqQ
4	Right Side castling can be done	SUCCESS	Works for both black and white moves. Link: https://youtu.be/mtsV-PcKYlc
5	En passant can be done	SUCCESS	Works for both black and white moves. Seen in pawn tests.

Board States

Board States			
Test ID	Criteria	Outcome	Notes

1	The game can determine when a player is in check. It can determine which player.	SUCCESS	The game is aware of whether black or white is in check. Link: https://youtu.be/otOlgREmZBc
2	The game can determine when a player is in stalemate. It can determine which player.	SUCCESS	The game is aware of whether black or white is in stalemate. Link: https://youtu.be/TwMj-Tbh948
3	The game can determine when a player is in checkmate. It can determine which player.	SUCCESS	The game is aware of whether black or white is in checkmate. Link: https://youtu.be/ts-iPlmPHgM
4	The game can determine when a player is out of time.	FAIL	The game has a timing system however it is purely cosmetic and time limits not enforced.
5	The game can determine a dead position.	FAIL	The game is unable to do this.
6	The game can determine when there is no special board position.	SUCCESS	The game is aware when there no is no special board position. Seen in many tests.
7	The game can determine when a player is out of time	FAIL	Clocks are purely cosmetic, both locally and over network. See testing screenshots. After time runs out they display negative times.

Connection with server

Connection: see testing screenshots at end of document for proof.			
Test ID	Criteria	Outcome	Notes
1	Both players can connect to the server.	SUCCESS	Both players can connect to the server. Link: https://youtu.be/XlTsvafz9mY
2	Each client is given a player identity – white or black.	SUCCESS	The players are given a identify based on the order they join, i.e. first connected player is white. Seen in the last two holistic tests, Game 5 & 6.
3	A client disconnecting can be handled by the other client.	PASS	If Client A disconnects, client B is given a warning that there is an issue. They are given the option to close the program or carry on. Should they carry on they will find the game does not run as intended & will eventually crash. Link to Testing Screenshot:
4	The server can handle a disconnect from a client.	FAIL	The server will crash and close.
5	The server transfers metadata	PASS	The server informs the user of what colour they will be.
6	The server does not allow the white client, to move black pieces. For both	SUCCESS	The server will only allow the clients to move their own pieces.
7	The server allows each client to take turns moving.	SUCCESS	A client cannot move a piece if it is not their go.

8	A Client can run entirely independent of the server.	SUCCESS	All moving and checking is done client-side, meaning an internet connection is not required, and games can be played locally.
9	The Client can connect to a specific IP address, at a specific port.	SUCCESS	Both the port and the IP address can be entered into the text box. Erroneous data entered is rejected.

Holistic tests

Games 1 – 5: https://www.ichess.net/blog/famous-chess-games/

Game 6: https://www.chess.com/article/view/carlsen-checkmates

For a holistic test I played 6 famous chess games. Games 1 to 4 are played locally – on the same application. Game 5 is played over a LAN & Game 6 is played over a WAN.

Game 1: Adolf Anderssen – Lionel Kieseritzky, 1851

Link to test video: https://www.youtube.com/watch?v=on4FYiMaLSo

Played Locally

Features/ Complex moves: N/A

Outcome: SUCCESS

Notes: Test Successful.

Game 2: Paul Morphy – Duke Karl/Count Isouard, 1858

Link to test video: https://www.youtube.com/watch?v=v9QlftsbaZc

Played Locally

Features/ Complex moves: Left Side Castling (White)

Outcome: SUCCESS

Notes: Test Successful.

Game 3: Mikhail Botvinnik – José Raul Capablanca, 1938

Link to test video: https://www.youtube.com/watch?v=UfBo_N5DyF4

Played Locally

Features/ Complex moves: Right Side Castling (White & Black), En passant (White)

Outcome: PASS

Notes: The game was ended by a resignation; this is not possible using the application.

Game 4: Donald Byrne – Robert James Fischer, 1958

Link to test video: https://www.youtube.com/watch?v=G6dC1xTUZSM

Played Locally

Features/ Complex moves: Right Side Castling (Black)

Outcome: SUCCESS

Notes: Test Successful.

Game 5: Anatoly Karpov – Veselin Topalov, 1994

Link to test video: https://www.youtube.com/watch?v=o_MhQHTZQxE

Played over LAN

Features/ Complex moves: Right Side Castling (White & Black)

Outcome: PASS

Notes: The Game was ended by a resignation; this is not possible using the application

Game 6: Magnus Carlsen – Sergey Karjakin, 2012

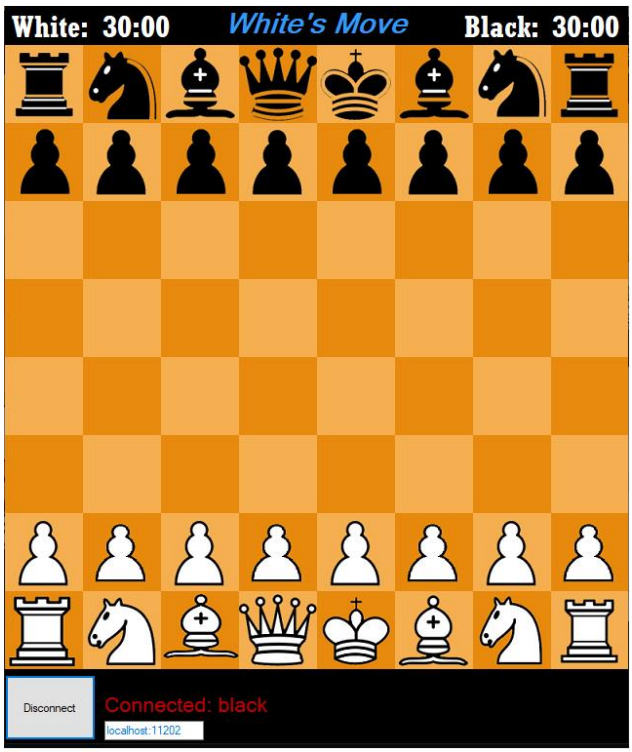
Link to test video: https://www.youtube.com/watch?v=K26JQjgjrrQ

Played over WAN

Features/ Complex moves: Right Side Castling (White & Black), en passant (Black)

Outcome: SUCCESS

Notes: Test Successful.



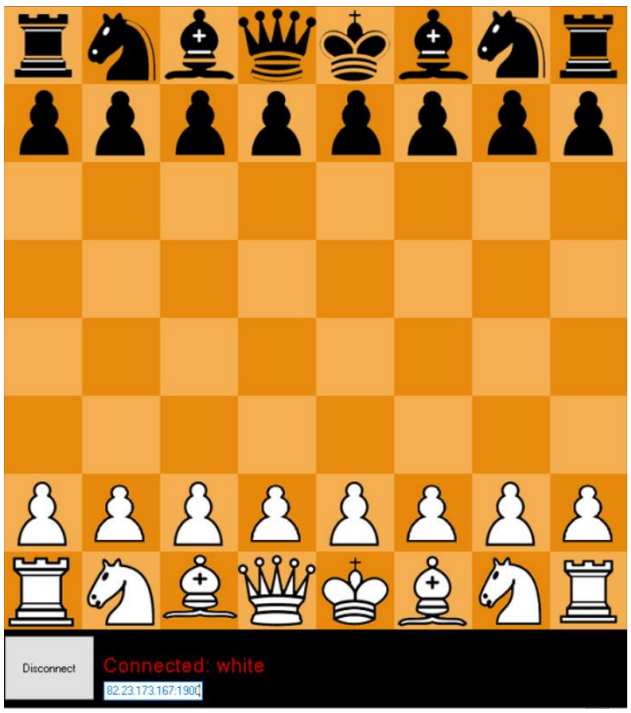
My Public IPv4 is:

82.23.173.1

67

My Public IPv6 is: Not Detected

My IP Location: Baldock, ENG GB



My Public IPv4 is:

213.107.64.158

My Public IPv6 is: Not Detected

My IP Location: Manchester, ENG GB

82.23.173.167:1900

To test the connection over a WAN, I used two laptops, connected one to my home wi-fi and the other to my mobile phone hot spot. Using port forwarding, I was able to connect the two clients, to carry out this test.

Evaluation

Specification	Final System	Outcome
Connect two players over a wireless network.	Two players are connected over a network.	Success
Allows the two clients to interact	The two clients can send information back and forth to the server.	Success
Allow two clients to play over different networks	The clients can play over two networks.	Success
Handle a player leaving unexpectedly.	When a client disconnects the server crashes, shutting down. The other client is then given an error message, giving the option to close the program or continue running.	Pass
Have a connection time of less than 30 seconds.	Connection time is rarely ever more than 4 to 5 seconds – over WANs.	Success
Transfer metadata about the player	Minimal metadata is transferred, about which side the player is on.	Pass
All Legal moves for each piece highlighted	In all tests every single Legal move for a player has been seen.	Success
No non-Legal moves are highlighted or can be made.	The king can castle through check. This is the only illegal move I found which is possible using the program.	Pass
The Program detects a checkmate	The program has detected all checkmates tested.	Success
Pawn Promotion	The program has pawn promotion.	Success
The Program can detect a check	The program has detected all possible checks.	Success

The program enforces turn order – white then black.	The program enforces turn order, in both local and LAN play.	Success
The Program can detect a stalemate	The program detects all stalemates.	Success
The Program allows for en passant movement	The programs highlights and allows moves that are possible via en passant.	Success
The Program detects when a player has run out of time	The clocks are purely cosmetic.	Fail
The Program allows for castling	The Program highlights and allows moves that would only be possible via castling.	Success
The Program detects if a dead position has occurred	The Program does not detect any dead positions.	Fail
The Program allows the two users to agree to a draw	The Program has no way for the two players to officially agree to a draw.	Fail
The Program's GUI displays a chessboard	A coloured chessboard is displayed and used to play chess.	Success
The Program's GUI tells the user what colour they are – if playing over a network.	The program informs the user which colour they are. They are assigned this when they connect to the server.	Success
The Program's GUI tells the user how much time they have left.	The users are told how much time they have left; however nothing will happen if they let this time run out. The server also does not ensure that the clocks of the clients are in sync. It starts timing when the first move is made, to prevent the clocks initially being out of sync.	Pass

There is a menu system	There is a basic menu system that the client can use to choose whether to play locally or over a network.	Pass
The program has a clean and easy to use design	It is a well-designed chessboard GUI however, when a piece's legal moves are shown, they obscure pieces that are on the squares which can be taken.	Pass
Display information about the person they are playing	No information is displayed about the character that they are playing.	Fail
Record moves made during the game.	Moves are not recorded by the client; they are however recorded by the server. The list of moves is unable to be saved to a file.	Fail

User feedback

In an email my client stated: "

I am happy with the overall design of the application. The Program itself was easy to use, however sometimes it was a bit of a hassle to run the server as well. I did not find any problems with the server.

It would be nice to be able to change the style and look of the chess pieces. I would also like for the program to end once a game has ended or have something other than text to simply declare victory.

"

Response to user feedback / Ideas for improvements

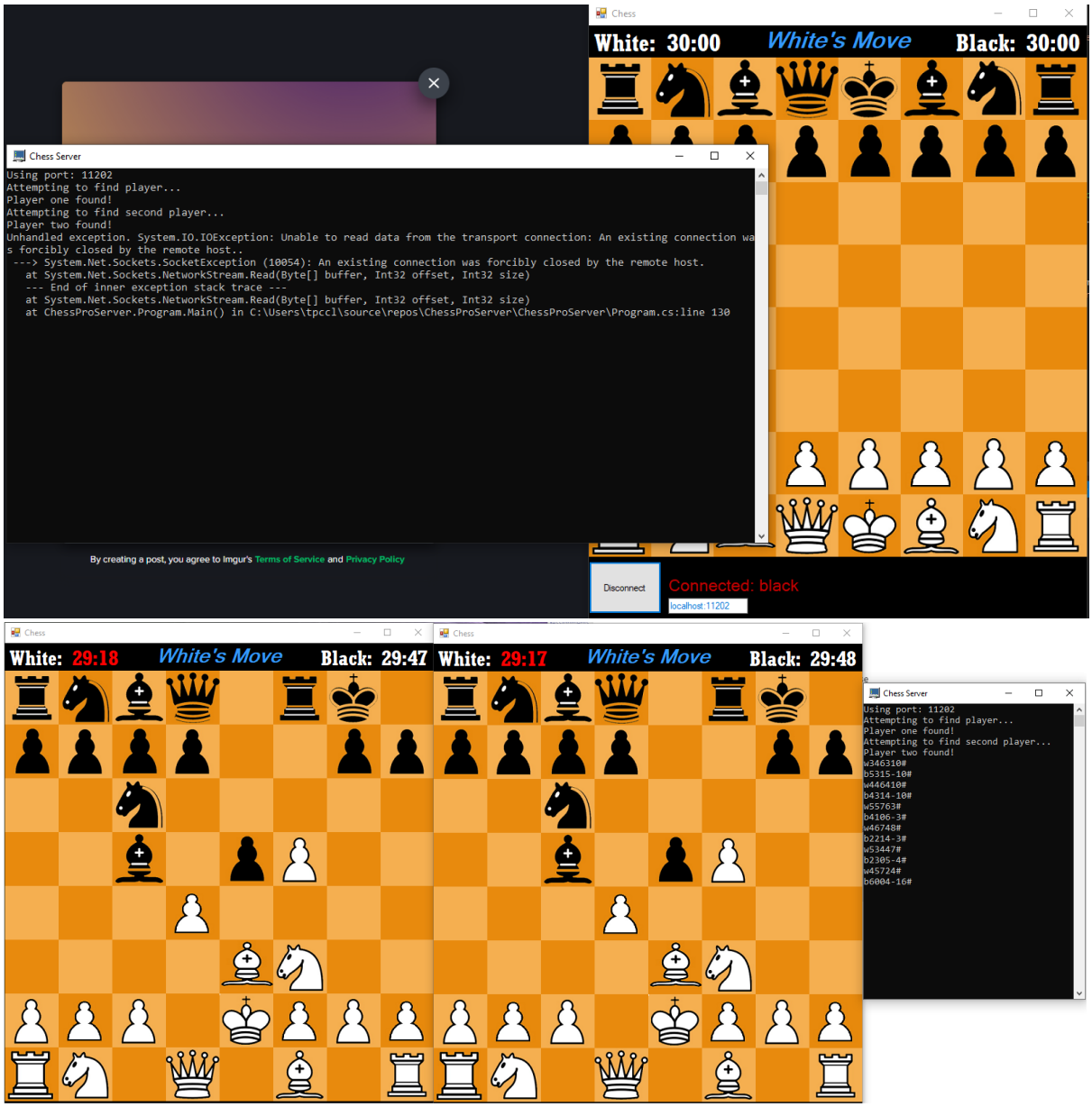
If I were to further develop my project, I would host the server on a dedicated machine, meaning it is not the client's responsibility to handle setting up a VPN or port-forward. I would most likely do this by buying computing on a cloud platform such as the ones offered by google and amazon.

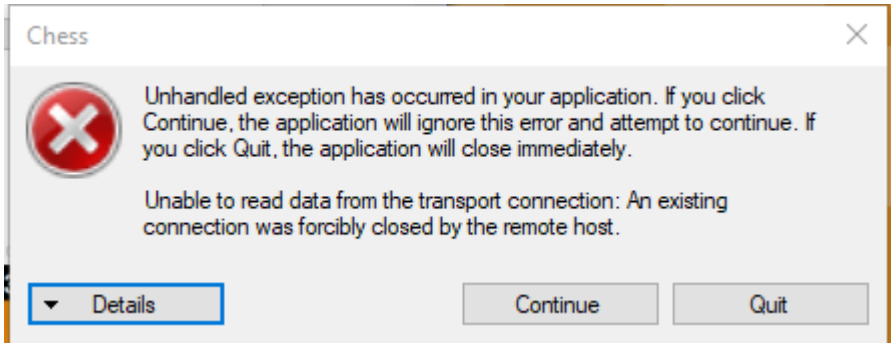
I would also add more textures to my game, so that the clients may have a choice of style. I would probably hire a graphic designer for this.

Given more time, I would reconfigure the server code, so it was better equipped to handle a player leaving the game. I would also make it so that the clients can see who they are playing against.

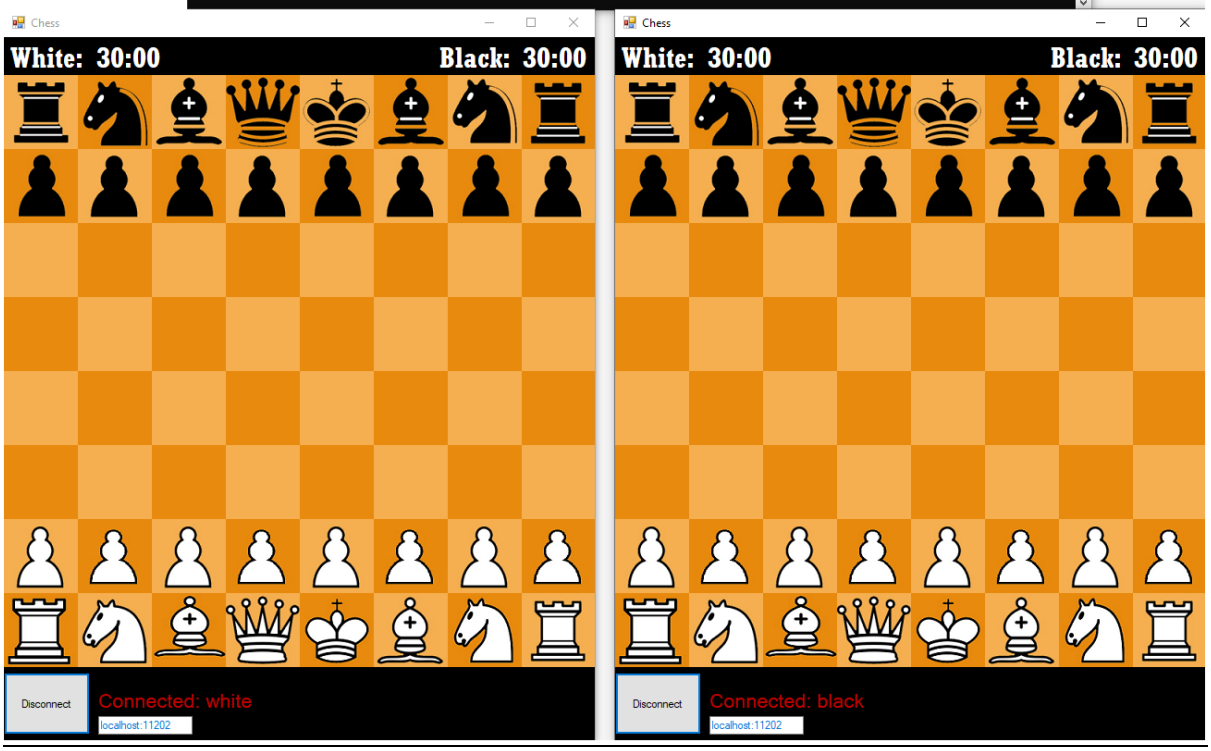
I would also fix the clocks so that they run in sync. I would also add “flag fall” to the game to gives the clocks a purpose.

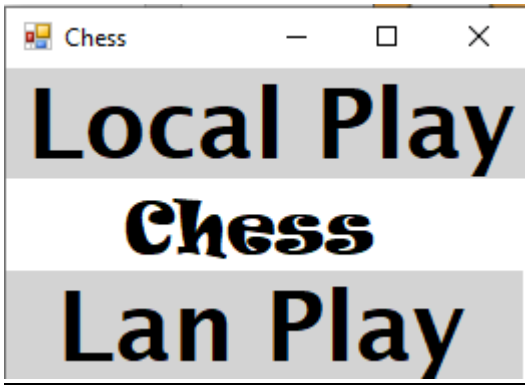
Test Screenshots

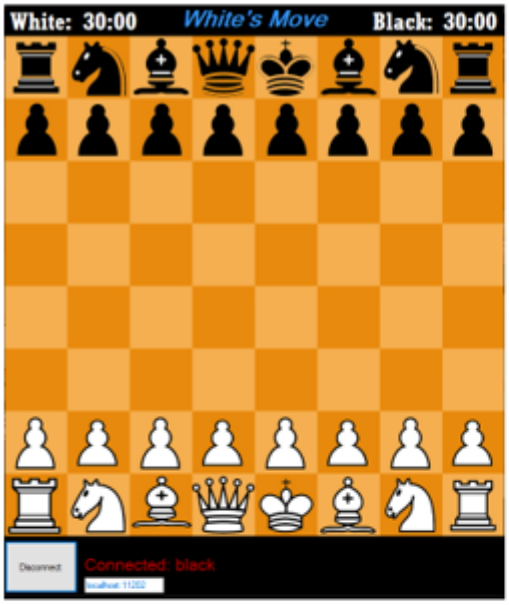



Chess
Unhandled exception has occurred in your application. If you click Continue, the application will ignore this error and attempt to continue. If you click Quit, the application will close immediately.
Unable to read data from the transport connection: An existing connection was forcibly closed by the remote host.
Details
Continue
Quit


Chess Server
Using port: 11202
Attempting to find player...
Player one found!
Attempting to find second player...
Player two found!


Chess
White: 30:00
Black: 30:00
Disconnect
Connected: white
localhost:11202
Chess
White: 30:00
Black: 30:00
Disconnect
Connected: black
localhost:11202


Chess
Local Play
Chess
Lan Play



My Public IPv4 is:

82.23.173.167

My Public IPv6 is: Not Detected

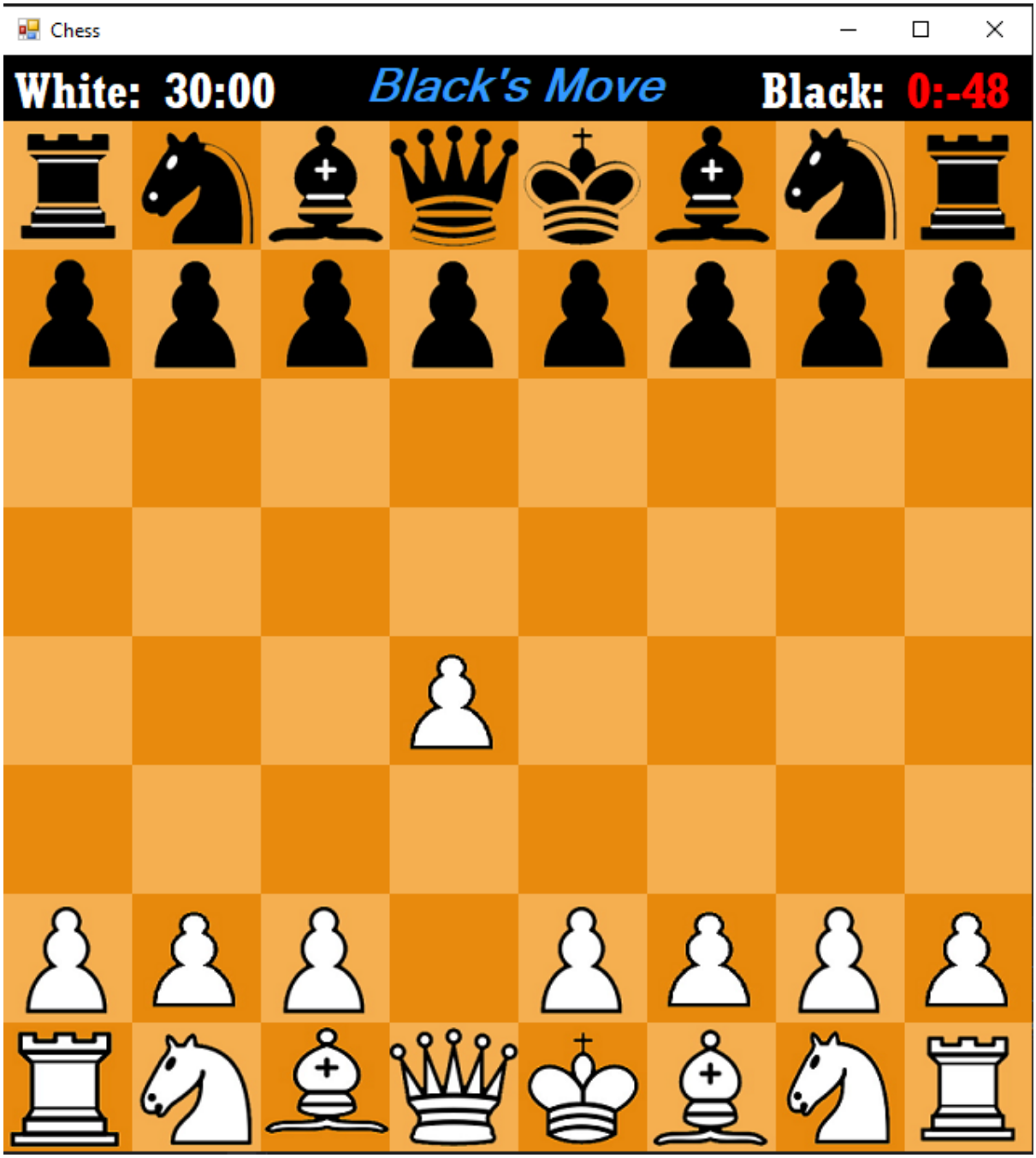
My IP Location: Baldock, ENG GB

My Public IPv4 is:

213.107.64.158

My Public IPv6 is: Not Detected

My IP Location: Manchester, ENG GB



This occurs when time runs out, the clocks simply turn negative.