OCR A Level Computer Science

C03 Programming Project

Candidate Name: Jordan de Burgo

Candidate Number: 7249

Hereford Sixth Form College

Centre Number: 24175

The documentation of the creation of Chess

Contents

Analysis of the problem

Problem Identification

Chess is a two-player strategy-based board game. It is played on a chessboard. The chessboard is a chequered board with 64 squares (an 8 by 8 grid). Casual games will usually last between 10 and 60 minutes whilst tournament games can take anywhere from 10 minutes to 6 hours or more.

In chess, there are nine different types of pieces: pawns, rooks, knights, bishops, a queen and a king. Each piece has a different set of specific movements it can do. For example, the rook can go anywhere in a straight line, but cannot jump over other pieces. There are also three different types of special moves: castling, en passant and promotion. These will also need to be incorporated.

The game will always end in either one person winning or there being a draw. A person can win by putting the other kind in “checkmate”, which is where the king is in check and no moves can be made to get the king out of check. Also, if the game is timed, if a player runs out of time they lose by default.

A stalemate is the causation of a draw, a stalemate will occur when a player can make no moves without putting their king in check but is not currently in check. A stalemate can also occur if both players make the same move 3 times in a row, or one of the players is forced to make the same move three time in a row. The last way a game can end in stalemate is if there is insufficient material on the board for either player to get checkmate (for example, there are only two kings left on the board).

There is also a variety of different styles of chess, these styles vary the amount of time a player has, or the setup of the board. Bullet chess is a style of chess where players only have 1 minute on their game timer, they have 1 minute each to play all of their moves.

Stakeholders

My stakeholders are Martha Baylis, Lewis Clark, Charlie Knapp, Daniel de Burgo, Eachann Bruce, and Elianne Pekoulis David.

Lewis Clark is a 17-year-old boy who does computer science. He plays many games in his spare time and enjoys board games such as chess. My solution would be appropriate for Lewis’ needs because it would allow him to play chess in his spare time, perhaps work on improving his skills against the computer on one of the more challenging ssettings or try out different game modes.

Eachann Bruce is a 17-year-old boy who also does computer science. He does not like to spend too long-playing one game because he gets bored easily. Therefore, having different modes where you can play with different times will be beneficial to him. The solution I have proposed would fit Eachann’s requirements as the different game modes would stop him from being bored doing the same thing all the time.

Martha Baylis is a 17-year-old girl who also does computer science. She does not have a lot of experience with gaming and prefers casual games to games that involve many complicated controls. Therefore, I believe she will enjoy a well explained game of chess. My solution would be appropriate for Martha because the artificial intelligence will be designed in such a way that it will be able to play at a very basic level (as well as challenge more experienced players).

Charlie Knapp is a 16-year-old boy who also does computer science. He prefers casual games, small-scale games. This means that chess would be perfect for his needs. Charlie would enjoy my solution as it can be played completely or more casually, the different levels for the computer can decide if you want to be challenged, or just wants a relaxing game.

Elianne Pekoulis David is a 16-year-old girl who does not do computer science. She has almost no experience with gaming so will prefer the easy to play style of my game. My solution will be appropriate for her as it accounts for people who have no experience with chess, giving Elianne the chance to develop her skills against a lower level computer.

Daniel de Burgo is a 19-year-old boy who does computer science at university. He has some experience with PC games and prefers games that are more complex to the simple style of mine. However, he does enjoy playing chess. My solution will be appropriate for Daniel as he can be challenged against a tough AI opponent, or learn to play different game modes for a bit of complexity.

Why it is suited to a computational solution

The game which I have planned to create is Chess. When the game is opened a menu will first come that allows a user to select which game mode they would like to play.

This problem will be broken down into the several game modes. Each mode will then be broken down into smaller problems that will make it easier to solve. The project will be solved by programming each mode with a centralised class which contains the board and pieces. Each mode will then be broken down into more solvable problems. For example, the central class will be broken down into the board and pieces. The board will be broken down into squares and the pieces broken down into their types by having a class type of pieces and each piece type inheriting their properties from the piece class.

Abstraction will be used in this project. For instance, chess can be broken down into a few basic elements. The board, the pieces and the clock. The board then has squares, the pieces have their moves and the clock contains the time. I think it will be relatively easy to break down the problem into small parts, making this problem easier to solve with code.

Thinking ahead, the input to the algorithm will be a player's move. This will be determined by where they clicked on the board. This will be the user's main interaction with the program, however the user will be able to select a game mode, choose to play a computer or a human and have other options presented. These options will be best formed as buttons, to prevent invalid entries to the system.

Procedurally, the problem will be able to be broken down into many subproblems. To start with, I have my main problem, creating a chess game. This is broken down primarily into three parts. The gameplay itself (how the user interacts with and uses the game), the artificial intelligence (the option to play against the computer) and the game modes. These problems can then be broken down. The gameplay must include a board, some pieces and the ability to move those pieces legally. This can then be broken down further, the pieces have types which all have different sets of moves they can do. The board must be generated using singular squares.

There will also be plenty of opportunity for logical thinking here. There will be plenty of times during the algorithm when it will have to decide. For example, whether a move is legal or not (when a player attempts a move), whether a calculated move is the best move. The algorithm will have to

decide whether a piece is to go on a square when the board is generated, what piece is to go on the square and whether the piece is white or black.

There will be the option for concurrency, the minimax algorithm could be programmed parallelly on different threads to calculate different branches of the tree at the same time. However, this concurrency means introducing alpha-beta pruning would be a lot more difficult, so a decision will have to be made on which one makes the process of calculating a move more efficient.

Some heuristics will be used within the artificial intelligence component of the system. For example, the same piece will have a different value in different positions due to its manoeuvrability at that position. These differences in value will be generated not just from logic, but from what works at making the computer play better.

When the program is loaded, a menu will come up asking the user to input a name in a text box and select a game mode to play. Once the game has been loaded, the gameplay will be dependent on which game mode they decided to play.

All the modes will need to use graphical rendering, which will be processed and added to the game space. Many checks will have to take place within each game to make sure the user can do what they want to do with their input. For example, there are a limited number of legal moves in chess.

Calculations will need to be used mostly in the chess AI, when working out the best possible move to do next and ordering moves which could be used depending on the level AI you want to play. Calculations will also need to be used in my other projects when working out algorithms for movement patterns.

The game will have a main section that will lead into all the game mode, this will be the main game area. Whenever a game is completed, the user should always have the option to return to this main screen or play their current mode again. The user interface will be easy to understand and use, with buttons to play each mode with an instruction manual, easily accessible and easy to find.

Abstraction of the problem

I will need to break down the problem into smaller problems. I will start by breaking it down into the generation of the board, and the making of the pieces. I will have a class to generate the board, the squares and the original set up of the pieces. Then I will have a parent class that sets all attributes of the pieces. Then, as children of that class, I will have each individual piece (this is to separate out unique properties of the pieces, the movement and special moves). I will then need to search for check after every move, and ensure that neither player is in check, and if they are, edit the possible moves appropriately. If, after editing the possible moves appropriately I find that there are no possible moves for the player in check, in must be checkmate and therefore the game will end. After all, of those, I will add the timer, so I will be able to start to add different default modes of chess, like bullet chess, blitz and so on.

Research

Existing programs of similar types

Chess.com

There is already a plethora of games that exist that allow you to play chess, either 2 players online, locally or with an AI. The most popular chess app now would be chess.com. This is a website and an app that allows you to play chess on your phone or PC against other users of your ability. It also allows you to play against friends locally or against an AI.

This site allows users to play online against other people, play locally against other people or even play locally against a computer. I would like to allow my users to play locally against other people or against a computer.

The app also includes a GUI that shows your time, the pieces taken and the possible moves you have. There are also a few different game modes available for this app, but not much variety.

I think that the most relevant and unique feature of this app is the way it decides on your ranking as a chess player and matches you against other players of a similar ranking. I think this would be something interesting to add to my project. However, for a small userbase this feature would be incredibly difficult to implement accurately but could be something I add for players against the computer, perhaps the level of the AI could be determined on your wins and losses against it.

Sparkchess.com

Unlike chess.com, sparkchess.com provides a 3D platform to play chess on. It provides personalities for the different levels of AI you can play against. The interface allows you to see which pieces you and your opponent have taken, shows you the timer and the moves you have made. Sparkchess.com also provides an online feature to let you play against other random people or friends. You can track your progress, how many games you have won and lost, over a set period of time. This game also shows possible moves when you click on a chess piece and provides a feature that analyses the board and can tell you the best move to make. This game also allows you to choose what style of board you want to play on, and what pieces you want to play with. Whilst this is a minor feature, it does add to the games overall design, making it better for more users.

Chess.org

Chess.org is another chess playing game. This one has similar features, likes showing possible moves, giving you a hint, at which is the best move, playing against other people online. You can also play against the computer at a difficulty level of your choosing. The interface also shows the pieces that

have been taken and the clock. You can also undo a move or even add more time. Unlike the others you can choose to watch another person's game live which can be interesting. The giving a player a hint to which move would be best is also a very interesting feature here as it could allow the player to get an idea of what moves are good moves. However, this feature can be abused by players when they get a little board and stop paying attention, they could just keep pressing the "Best Move" button and waiting

Chess by The AI Factory

Here is another example of a chess game. This game is available on the android app store and will be very similar to mine. This app shows you possible moves when you click on a piece, has a user interface for the pieces taken. This app also allows you to choose AI difficult from 1 to 10. However, this game sticks to only traditional chess, with an option of how much time each player gets. In this game you get the option to undo a move. You also can choose which colour you would like to play as, this is a little different to most of the other chess games that I have researched as most of them randomise who is black and who is white. Also, there is no option to play two players, online or locally. This is also unique as all other chess games I have researched give you the option to play against real people. This app has no option to resign from a game formally, although you can just quit. This is also a feature of just this app, as all other apps require a formal resignation as they can be against other people. A game which involves real people must have a clear winner and loser.

Parts I can apply to my own solution

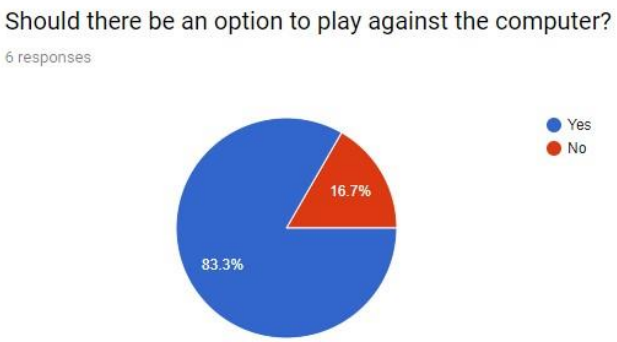
From these different chess games, I can try to incorporate the following features:

- Player vs player local mode
- Player vs player online mode
- Player vs computer mode
- Different levels of AI
- Showing possible moves
- User interface including timer and statistics about the game
- Undo moves
- Functionality of special moves
- Automatic queening
- Randomisation of who is black and white
- No resignation against the computer, just the option to quit or restart
- Hints
- AI Level based on how much you win and lose

Stakeholders Opinions

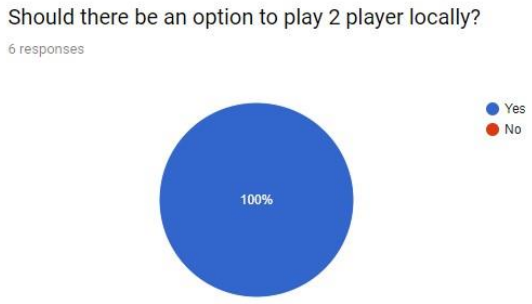
I conducted a survey to find out what my stakeholders thought about these different features, which ones were most important to them, and which ones I should consider not including. I chose to ask about the features of the chess games that I have research that I liked, and thought would be a manageable feature to add to my chess game.

Question 1: Should there be an option to play against the computer?



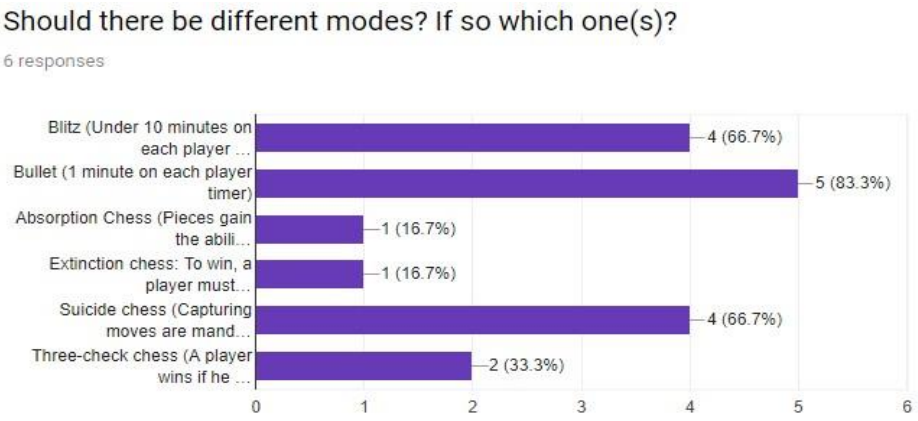
As I expected, the majority of my stakeholders wanted to be able to play against the computer, giving this game a good basic functionality, a feature which I will definitely be including.

Question 2: Should there be an option to play 2 player locally?



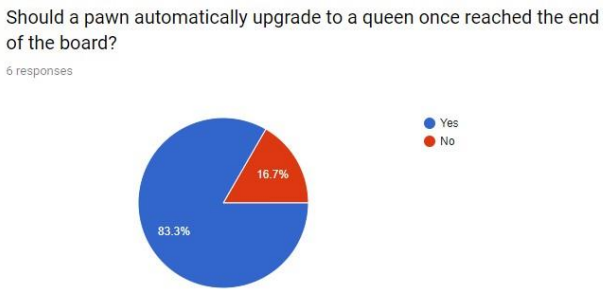
Another very much expected result. All of my stakeholders wanted to be able to play against another person, without this feature I am not sure that you would be able to class it as having any use, another feature that will be included.

Question 3: Should there be different modes? If so which one(S)?



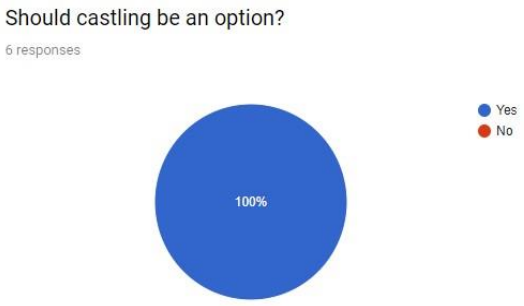
I wasn't entirely sure what my stakeholders would think about the different game modes. You can see here that there are three game modes that have a majority of stakeholders interested. These are the three modes that I will try to include in the game.

Question 4: Should a pawn automatically upgrade to a queen once reached the end of the bored?



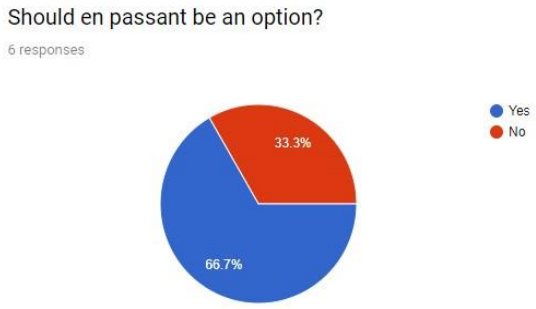
The chess apps I have researched have not seemed to agree on this issue. Many chess apps allow you to decide what you want when a pawn reaches the end, however it was just as common for the game to automatically upgrade your pawn to a queen. My stakeholders have shown that they would like this feature to be automatic.

Question 5: Should castling be an option?



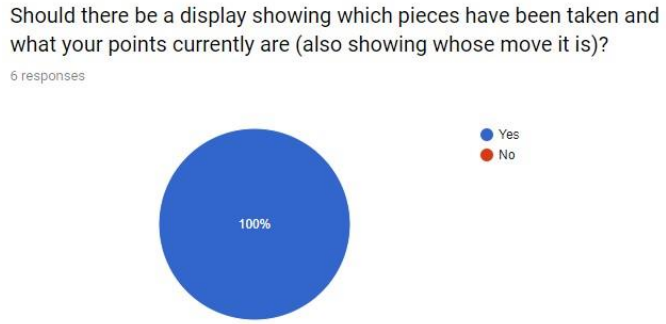
There is a consensus amongst my stakeholders that castling should be an option. I also failed to find a chess app that did not have this option available to users.

Question 6: Should en passant be an option?



Many casual chess players are not aware of this moves existence. For some it can be quite frustrating when someone uses this move and you had no idea that it was possible. However, even some of the casual chess players found themselves intrigued by this move and wanted it as an option.

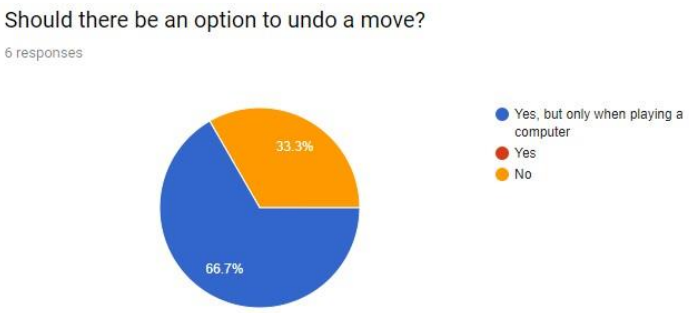
Question 7: Should there be a display showing which pieces have been taken and what your points currently are (also showing whose move it is)?



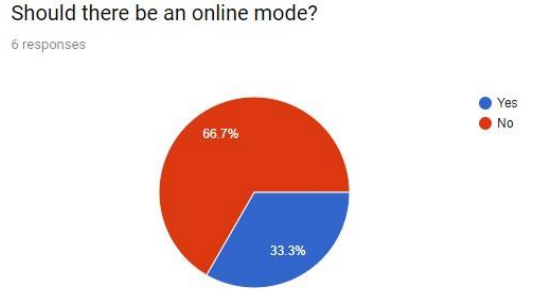
Every chess game I researched had some kind of display that gave you all of the information about the game you were in. My stakeholders like the idea of this and everyone agreed my game should include this feature.

Question 8: Should there be an option to undo a move?

Most of the apps that I've researched have had the option to undo a move but only when you're playing against a computer. Most of my stakeholders agreed with this, although some did feel that once you've made a move that should be the end of it. I will include undoing moves just for when a person is playing against a computer.



Question 9: Should there be an online mode?



This question did not turn out as I expected. I thought that the stakeholders would want the option to be able to play online against other people, however most of my stakeholders did not like the idea of this. One reason given was that "it should be kept on one machine".

Features of my proposed solution

Initial idea considering research

My chess game will include an option to play against another person or against the computer. When playing against the computer, you will be able to pick what level to play against and there will be the option to undo a move. It will offer 3 different modes; blitz chess, bullet chess and suicide chess. All "special" moves will be possible (castling, queening and en passant).

Limitations of the proposed solution

The artificial intelligence implemented in the game will have a limit to how good it can be. The level of AI available will somewhat correlate with the amount of time I can spend developing it, and as there is a finite amount of time, there is a limit to what level of player it could challenge. Also there will be a maximum amount of game modes I can add. I have selected three of the most popular from my stakeholders although it will still leave some disappointment in the stakeholders that voted for the less popular game modes as there is not enough time to develop all of the game modes. Some of my stakeholders also wanted an online mode available, however this will not be included due to a majority not wanting the online feature and the implementation of this taking a long time.

Further input from stakeholders

Message

"Hi,
My initial propositions for the project is "My chess game will include an option to play against another person or against the computer. When playing against the computer, you will be able to pick what level to play against and there will be the option to undo a move. It will offer 3 different modes; blitz chess, bullet chess and suicide chess. All "special" moves will be possible (castling, queening and en passant).". What do you think to this? Any comments and suggestions would be great.
Thanks,
Jordan."

Response

Martha

"Hi Jordan, this seems like a reasonable proposal. Good luck."

Charlie

"This proposition sounds excellent, and exactly fits what I would like to see from the final project. The 'undo' feature sounds particularly useful, although I would like an option that allows more experienced players to disable it, for added challenge.

The AI and multiplayer options both sound particularly exciting, and although I'm slightly concerned about whether you'll be able to implement them in the timeframe. I look forward to the finished product."

Eachann

"That sounds like a good idea. You will be giving the user a lot of choice with how they like to play the game. I would like it if the user had the ability to change the look of the pieces.☺"

Lewis

"I think that this is a fair proposal, balances variation and simplicity well."

Elianne

"Hey Jordan, this proposal sounds like a really nice idea. It seems to fit all aspects of my needs, the different game modes sound really interesting. Can't wait to see the finished product."

Daniel

"I think this is a perfectly good proposal which fits all stakeholder requirements. I would not change anything about the above proposal."

Analysis of Requirements

Software and Hardware Requirements

I am using Visual Studio 2017 for this project; therefore, the users will need the following hardware to play the game:

- 1.6GHz clock speed or higher
- 1GB of RAM
- Integrated graphics
- 5400 RPM HDD with at least 4GB of space
- A mouse
- A monitor

And the following software:

- Windows 7 or later

Requirements Table

Component Number	Name	Description
1.	**User Interface Design**	
1.0	Easy to use	The user needs to be able to use the game without being confused by what they are meant to do. It should be clear what they are supposed to do in order to play how they want.
1.1	Minimalistic	It needs to be kept simple. The user shouldn't feel overwhelmed by too many buttons, menu options or anything of that sort.
1.2	Traditional	The game should be recognisable as a chess app, using traditional colours for the chess board, and standard images for the pieces.
2	**Functionality**	
2.0	Click piece movement	You should be able to click on a piece, see the possible moves for the piece, click on a possible move to move their, or click somewhere else and then go onto a different piece.
2.1	A challenging AI	The chess playing AI should be able to offer a challenge to any user that plays the game.

2.2	Undo button	The undo button should only be available when playing against a computer. When playing against a computer you should be able to undo as much as you want.
2.3	Display possible moves	The game should display all possible moves for a piece when that piece is clicked on and make sure a move is legal when the user attempts to make a move.
2.4	Check/checkmate	The game should check whether a user is in check or checkmate and change what happens next appropriately. If in check, change the legal moves to only ones which get the user out of check. If in checkmate end the game.
2.5	Display pieces taken and timer	The game should display the pieces taken so far. It should also show the amount of time left on each player count. If a timer runs out, it should end the game.
2.6	Special moves	The game should allow for moves like castling, changing a pawn to a queen and en passant.
2.7	Different game modes	There should be 3 different game modes. Blitz, bullet and suicide chess available to the user.
2.8	Board is generated	An 8 by 8 grid should be generated as squares that the game is played on.
2.9	Piece generation	All pieces (black and white) should be generated and placed on the board appropriately.
3	**Usability**	
3.0	Possible moves displayed correctly	The possible moves should be displayed to the user accurately, and they should be displayed in such a way that is clear to the user.
3.1	Rules for each mode available for display	If necessary, the user should be able to check what playing in different game modes implies. Like the Blitz chess mode changing it so that the timer is limited to 10 minutes for each player.
3.2	The pieces move	The user should be able to move the pieces.
3.3	Different levels of AI	The user may be an experienced chess player or be very new to the game. Having different difficulty levels for the AI will give a wider range of user's access to the game and allow them to have a more enjoyable experience.
3.4	Accessible	The game should be easily played by anyone, no matter how much experience they have playing chess. Also, it should be accessible to players with colour blindness, using more easily seen colours.
3.5	Error Tolerant	The game should be able to cope with wrong inputs (like trying to do an “illegal” move) without causing the game to crash.

Success Criteria

Criteria	How to evidence	Justification
An easy to use interface	Screen shots of the interface and reviews from stakeholders.	My program should be accessible, making it easy to use will allow a wider variety of users.
A minimalistic interface	Screen shots and reviews from stakeholders.	A minimalistic design will keep things simple for new users
Traditional design	Screen shots showing the design of the game.	A traditional design will make the came more recognisable.
Click piece movement	Screen shots of before, during and after clicking on a piece and moving it.	This will provide an easy way to interact with the program
AI functionality	Evidence of testing.	An AI will allow people to play by themselves, giving more people the opportunity to use the product
Undo button	Screen shots of the undo button, after a move and then undoing the move.	An undo button whilst playing single player gives people an opportunity to correct their mistakes to learn better moves for a given position
Display possible moves	Screen shots of a piece displaying its possible moves.	This will provide a way for new users to know what they can do before learning all the rules
Check for check and checkmate	Screen shots of someone being in check and the legal moves changing. Also, a screen shot of the game ending in checkmate.	New users will not have to recognise when it is check and checkmate
Display timer, taken pieces and whose turn it is	Screen shots after a piece has been taken. Screen shots showing the timer, and the game after a timer has run out.	Users will easily be able see which of their pieces have been taken and how much time they have left to play their moves
Special moves	Screen shots of the result of each special move.	Users will be able to play all moves, inclusive of the moves that are quite unique and only occur in special cases
Different game modes	Screen shots of the different games modes and how they differ from one another.	This will allow users to try a variety of different game modes to have some variation to standard chess
Pieces move	Evidence of testing.	Pieces must move as they would during chess in a board game
Visible board with recognisable pieces	Screen shots of the board and pieces.	The board should be easy to see with the pieces recognisable so that users can easily tell which piece is which

Invulnerable to errors	Evidence of testing.	This is to ensure the user has a good experience, without the product doing something unexcepted
Accessible to a wide range of people	Stakeholder survey.	This is to make sure my product is accessible
Pieces can take other pieces	Screen shots before and after taking a piece.	This is to allow standard gameplay
Different levels of AI	Screen shots of parts of games from different levels of AI.	This will ensure that the single player mode is accessible to a wide range of users

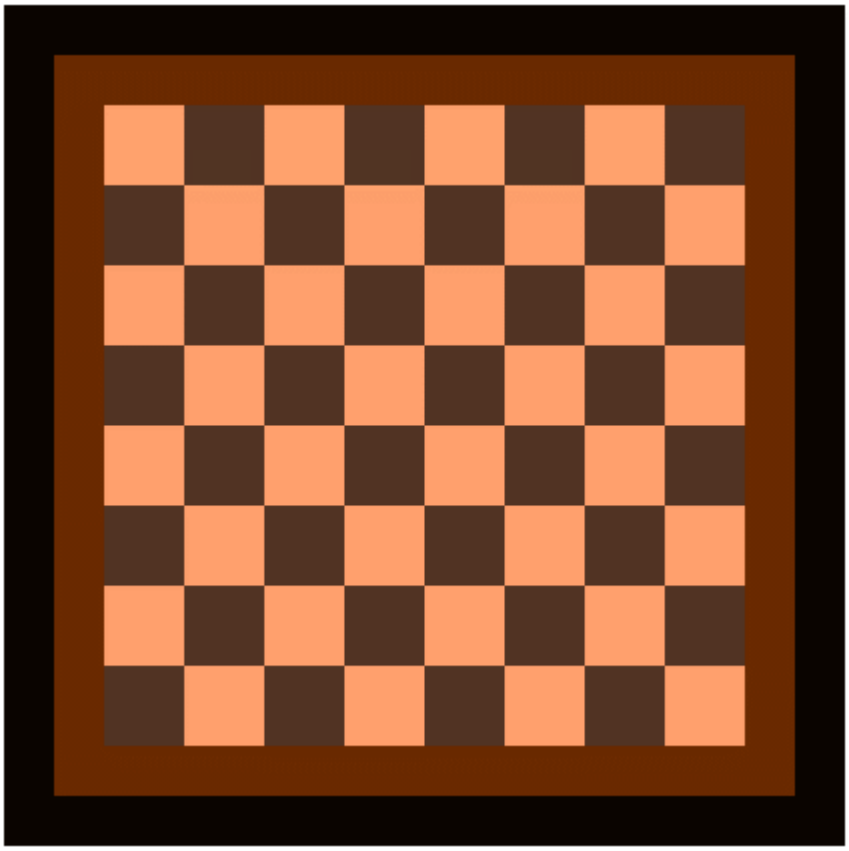
Design

I first designed what the board and pieces would look like. I did a quick mock-up of a board on paint.net and got a set of copyright free images of chess pieces to get some feedback from my stakeholders on what they thought of these preliminary designs. The consensus was that they liked the minimalistic design.

"I really like how simple the design feels, not too complicated" – Charlie Knapp

Board

I took the design of a traditional chess board, the alternating dark brown/ light brown. I originally wanted to ue black and white, however because the pieces must be black and white, this woul have made seeing the pieces quite difficult. My stakeholders agreed that the design was traditional and that it would suffice, however the colours may need to be rethought in a later iteration.



Pieces

I found some examples of classic chess piece designs online and decied to use them in my project. The stakeholders agreed that the pieces were very recognisable and that they were suitable for use in the game.

I then moved onto how the code would work. I started with a simple flow diagram and then moved onto some pseudocode for a generic overview of how the program would run.

Algorithm


```
START
Generate Board
Generate moves for piece
Player1move = TRUE
WHILE NOT checkmate
  INPUT playermove = players move
  IF playermove IS VALID THEN
    Move piece
    IF check
      IF checkmate
        IF Player1move
          Player 1 wins
          checkmate = TRUE
        ELSE
          Player 2 wins
          checkmate = TRUE
    ELSE
      Force next player out of check when their move
    Player1move = NOT Player1move
  ELSE
    Do not move piece
END
```

Classes

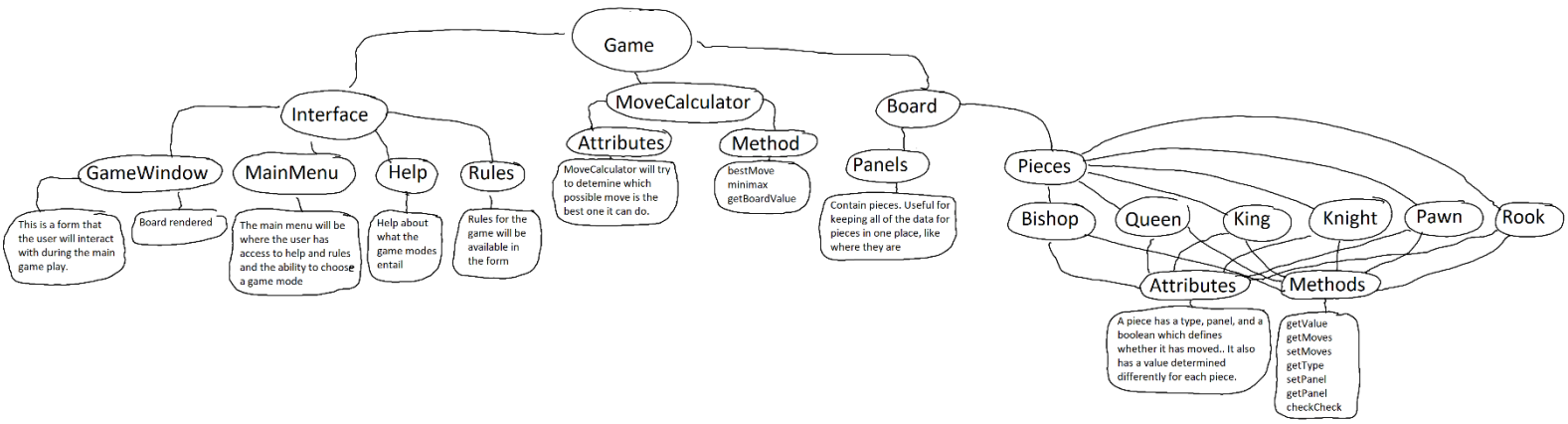
There are some classes that are already obvious that will be needed. There will need to be a piece class, from which each individual piece class will inherit. There will need to be a class for the board and a class that will generate the moves of the computer for single player mode. There will also need to be the classes for the user interfaces: the main menu, the help and rules and the actual gameplay. The piece class wil need to contain common attributes that all pieces share, for example, all pieces will need to have moves (although the actual moves will be different), all pieces need an image, a value, a type... etc. The board will need to contain a method for generating the board and updating it and the move generator will need a method to calculate a best move for a given board position.

Inputs and Outputs

Input	Process	Output
Gamemode buttons on start menu	Set a gamemode type	Open the game in that mode
Help button pressed	Relavent form is made	Opens to form
Quit button pressed	Open the main menu	Main menu is shown
Piece clicked	Moves for that piece calculated	Move for that piece shown
Valid move clicked	Piece is moved to that location	Piece appears where you clicked
Invalid move clicked	Moving piece is set to nothing	Moves for piece disappear
Resign button clicked	Game ends	Says who won

Diagram showing how classes link

The diagram below displays the hierarchical nature of the design of my code. It shows how each class will help to form part of the game.



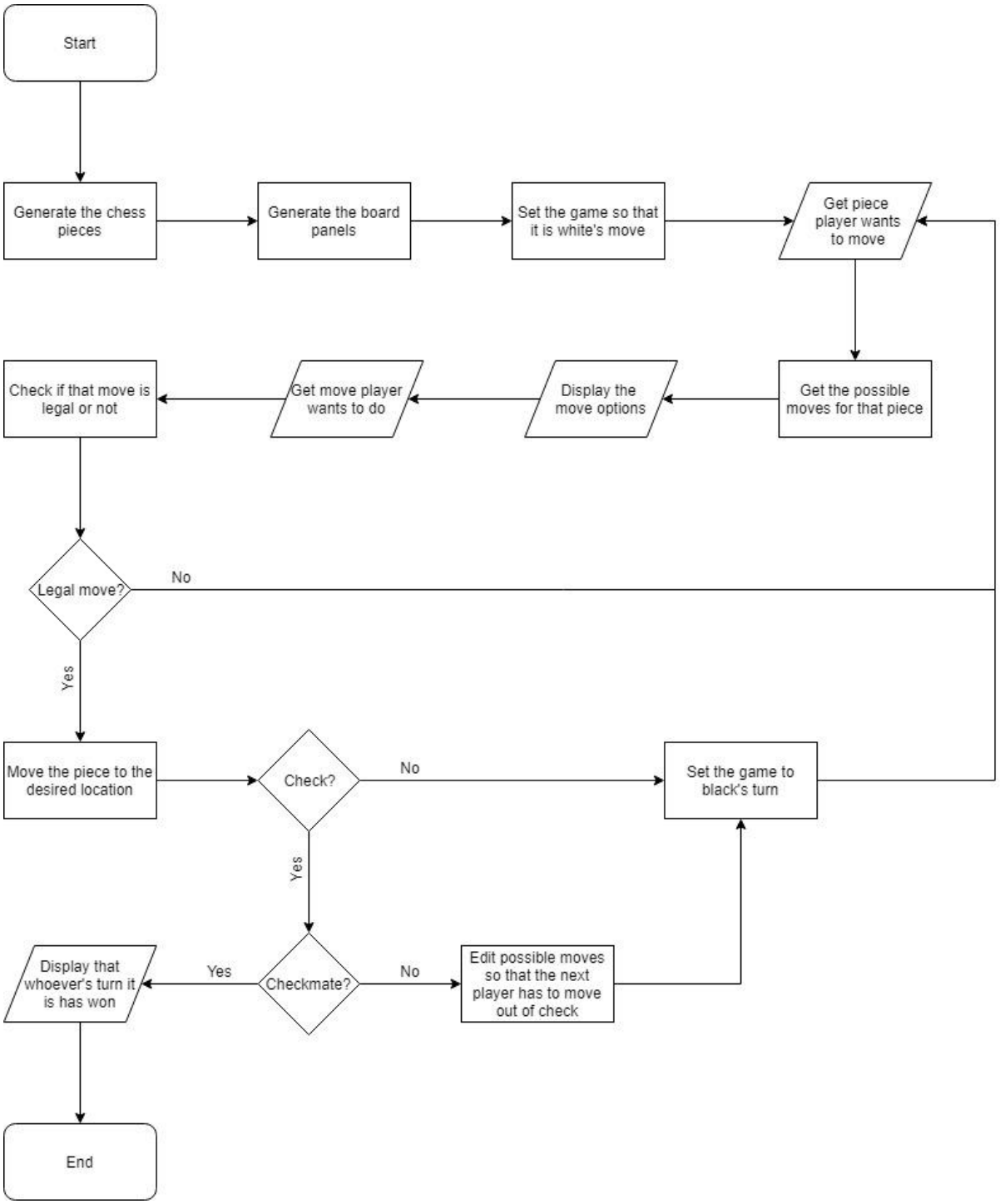
Development

Iteration 1

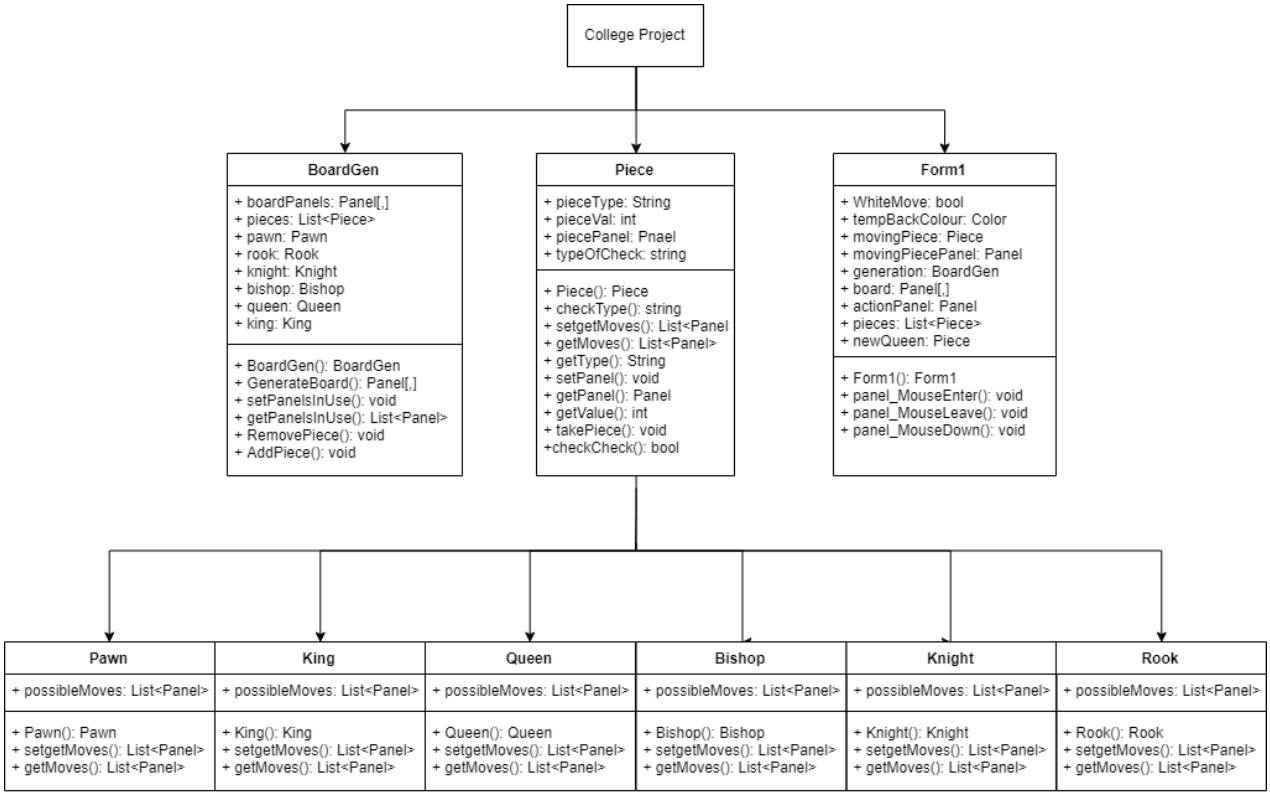
Planning

Iteration design

I have drawn a flow chart below of what the code in this iteration should generally be doing.



UML Class Diagram



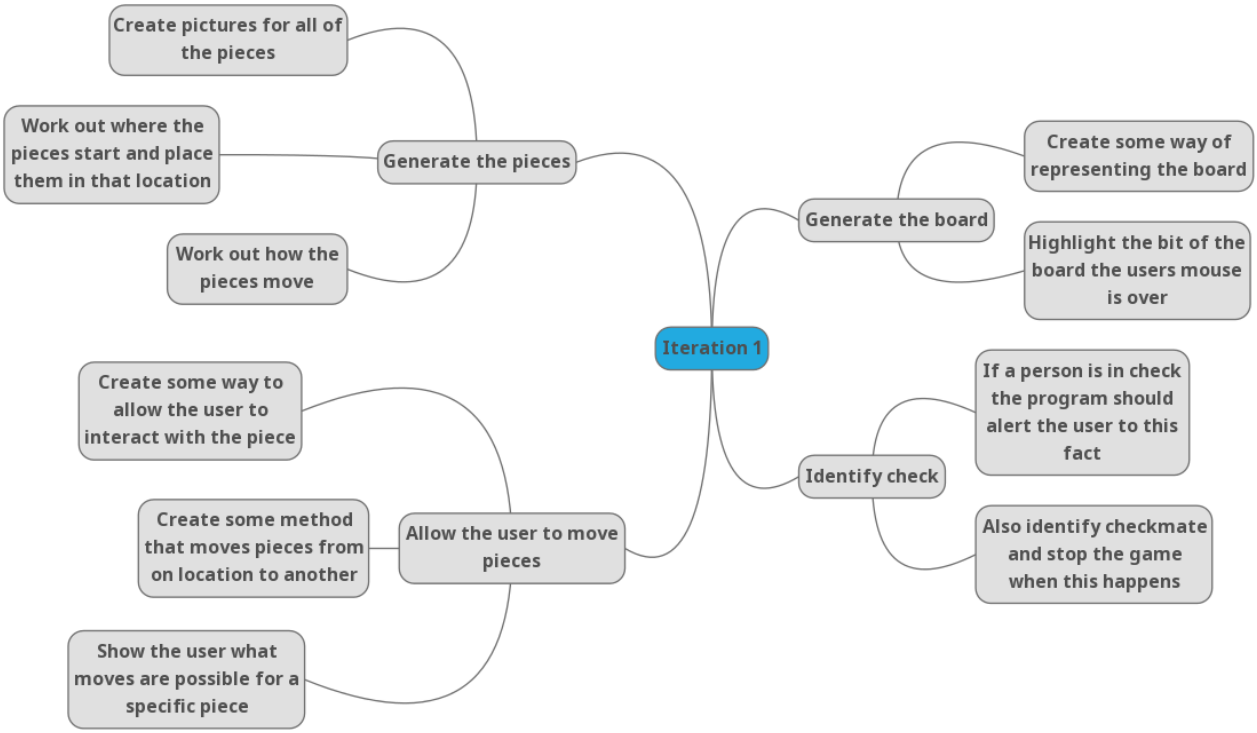
Discussion with Stakeholders

I first turned to my stakeholders to discuss what elements of the game should be incorporated into the first iteration. All my stakeholders said that the most important thing was to have the most basic features of the game, and then I could build from there.

Iteration requirements

ID	Description of component
1.0	Generate the board
1.1	Make sure board is coloured correctly
1.2	Make sure it is reusable throughout the code
2.0	Generate Pieces
2.1	Make sure all pieces appear in the correct starting positions
3.0	Generate the moves for the pieces
3.1	Make the moves displayable
3.2	Limit the moves correctly
3.3	If in check force to move out of check
4.0	Display correct behaviour when check and checkmate occur

Mindmap



Further discussion with stakeholders

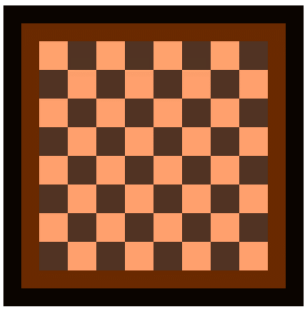
"In this iteration I aim to create a fully functional game of two player chess on a single machine without any "special" moves or the interface allowing you to see what has been taken already. Do you think this is a manageable, realistic but challenging goal?"

Many of the responses indicated some concern about time pressure.

"I do not believe that you will be able to achieve that in the time frame for the first iteration. Maybe you should rethink how much you could get done in one iteration and move some of the criteria to a later iteration? It's up to you really though." – Eachann Bruce

I decided to keep my proposed success criteria and evaluate what needs to be done still at the beginning of the next iteration.

Iteration requirement 1.0 – Generating the board



The board is an eight by eight grid of squares such that the colours of the squares alternate (usually between a dark and light shade of brown). I plan to use this method of generating the board throughout the project. It shall be drawn so that it is like a traditional board, ideally similarly to the design I created earlier.

Iteration requirement 2.0 – Generating the pieces

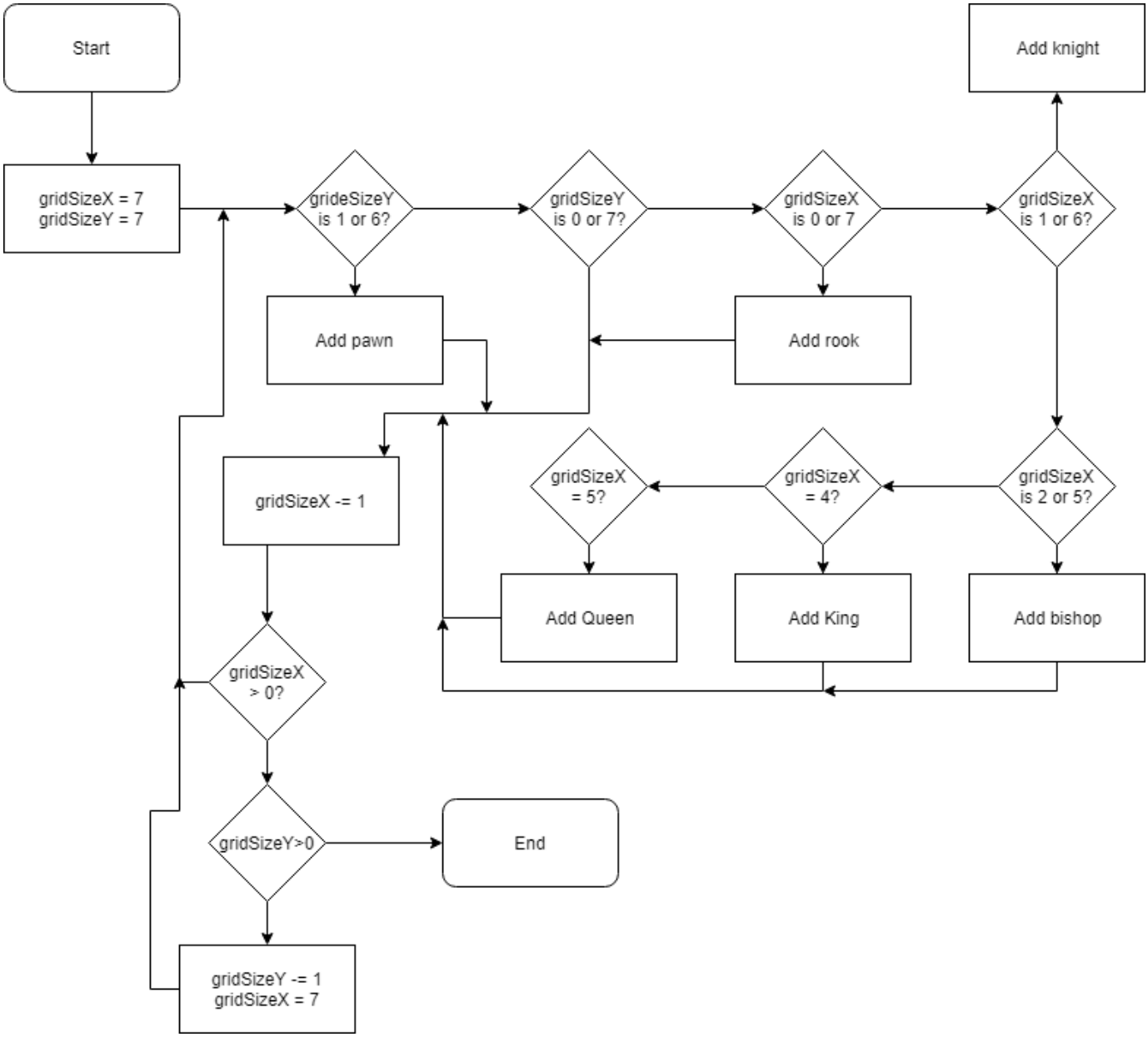
The pieces will need to go on their correct starting squares, which will probably be done whilst the board is being generated, so that generating the board includes setting up all the pieces. I will use the images of chess pieces shown earlier is the report as the pieces during this iteration.

Pseudocode Algorithm – 1.0 and 2.0

```
START
INT tileSize = 40
INT gridSize = 8
color1 = DarkBrown
color2 = LightBrown
boardPanels = new ArrayofSquares[gridSize, gridSize]
FOR i = 0 TO grideSize
    FOR x = 0 TO grideSize
        Square newSquare = new Sqaure
            Size = new Size(tileSize, tileSize),
            Location = new Point(tileSize * i, tileSize * x)
        STRING peiceType = null
        peiceType = The first letter of the side's name
        IF tileSize*x == 40 OR tileSize*x == 240 THEN
             peiceType += "Pawn"
            pawn = new Pawn on this square
        ELSE IF (tileSize*i == 0 OR tileSize*i == 280) AND (tileSize*x == 0 OR tileSize*x == 280) THEN
            peiceType += "Rook"
            rook = new Rook on this square
        ELSE IF (tileSize * i == 40 OR tileSize * i == 240) AND (tileSize * x == 0 OR tileSize * x == 280) THEN
            peiceType += "Knight"
            knight = new Knight on this square
        ELSE IF (tileSize * i == 80 OR tileSize * i == 200) AND (tileSize * x == 0 OR tileSize * x == 280) THEN
            peiceType += "Bishop"
            bishop = new Bishop on this square
        ELSE IF (tileSize * i == 160 ) AND (tileSize * x == 0 OR tileSize * x == 280)
            peiceType += "King"
            king = new King on this square
        ELSE IF (tileSize * i == 120) AND (tileSize * x == 0 OR tileSize * x == 280)
            peiceType += "Queen";
            queen = new Queen(peiceType, 9, newPanel);
        END IF
        boardPanels[i, x] = newSquare
        IF i % 2 == 0 THEN
            newSquare.BackColor = x % 2 != 0 ? color1 : color2
        ELSE
            newSquare.BackColor = x % 2 != 0 ? color2 : color2
        END IF
    END FOR
END FOR
```

This pseudocode just adds a square panel in an 8 by 8 grid. By checking the position of the panel, it can determine whether a piece needs to be placed there. The colour of the board panel is determined by using mod2, this allows the colour to alternate.

END



Iteration requirement 3.0 – Generating the moves for the pieces

The moves will be need to be generated for each type of piece. This will probably mean a separate class for each piece type inheriting from a class with properties they all have, like a colour and type. These will then need to be displayed in a way that is easy to use. Below is a generic way of generating moves for a piece.

Pseudocode Algorithm

START
possibleMoves = new ListofSquares
possibleMovesNoPieces = a list containing the possible moves for a piece if the board was empty
FOR EACH squareOnBoard
    IF theSqaure's location is behind another piece in the direction of movement THEN
        possibleMoves = possibleMovesNoPieces – squareOnBoard

```
        IF theSquare's location leaves my king in check THEN
                possibleMoves = possibleMovesNoPieces – squareOnBoard
        IF theSqaure's location is already the location of one of my pieces THEN
                possibleMoves = possibleMovesNoPieces – squareOnBoard
END FOR
END
```

This is a very simplified and very generic overview of what the code should check for each piece.

Iteration requirement 4.0 – Checking for check

Each time a piece is moved there is going to be a need check if either player is in check, and to stop players moving into check. If there are no moves where a player is not in check and the player is currently in check, it is checkmate, otherwise, stalemate.

Pseudocode Algorithm

A very simplified version of what would happen to check for check. Essentially what it does is it checks every piece on the board. If any of those pieces contain the opposite colour's king in their possible moves, then it is check.

```
BOOL check = FALSE;
Piece piece = NULL;

FOR EACH Piece y IN boardPieces
    List<Panel> possibleMoves = new List<Panel>();
    possibleMoves = y's moves
    IF possibleMoves != NULL THEN
        FOR EACH Sqaure x in possibleMoves
            FOR EACH Piece c in boardPieces
                IF c.getPanel() == x THEN
                    piece = c
                    IF pieceType contains "King" AND y's colour != piece's colour THEN
                        check = TRUE
                    END IF
                END IF
            END FOR
        END FOR
    END IF
END FOR
END
```

Testing Plan

During development each function needs to be fully tested to show that it is a working part of the solution and can be used in the final solution. This should ensure that the final solution works correctly on completion. For this I will use unit testing within the code. All further testing should be designed to highlight any bugs and the programs resilience to invalid inputs.

Testing during development should show what data was inputted, the intended result, the actual result the program gave, and whether this was a match or not. This testing should also include some explanation for why it may not be working as intended, with any console output if there is a crash and any relevant variables with a content analysis.

Some post-development testing will also be necessary to ensure that all the criteria has been sufficiently matched, and all works as described. This will mainly consist of resistance testing and feature analysis. I will state any invalid data put into the program, the desired output, and the actual output. For example, if a player tries to do an invalid move then the desired output would be that the piece didn't move, and it was still that players move.

I will show this evidence of testing to my stakeholders to see if they are content with what has been done and that the program is working as described in the requirements at the start of this iteration.

Usability Testing Plan

	Test	Pass/Fail	Comments
U3.0	Possible moves displayed correctly		
U3.2	The pieces move		
U3.4	Game is easily played		
U3.5	Error Tolerance		

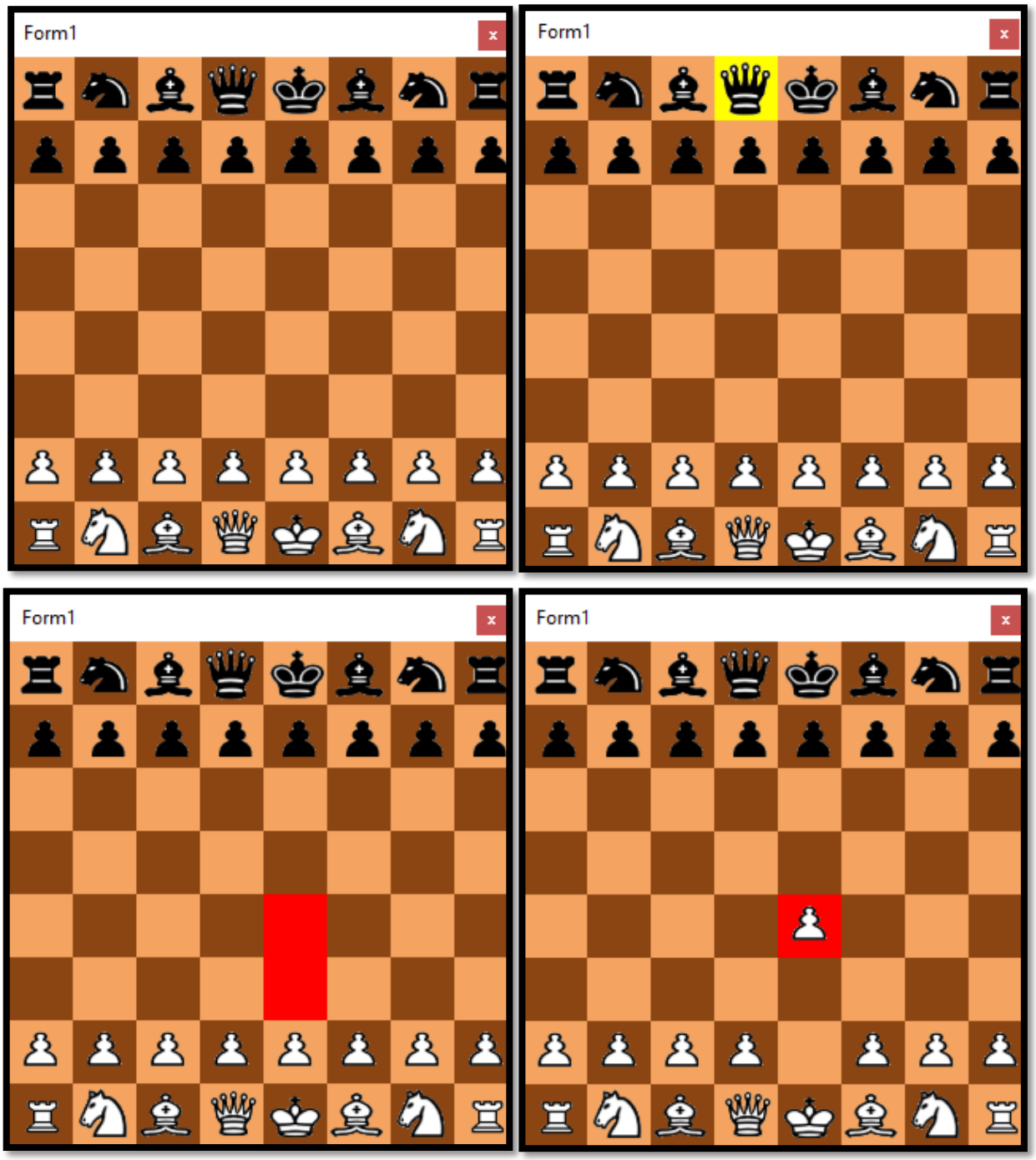
During Development Testing Plan

Test ID	Class being tested	Test Aim	Expected Result	Proof
DD1.0	BoardGen	Make sure the board is the correct size.	Pass	All proof in testing section.
DD1.1		Ensure there are the correct number of pieces.	Pass	
DD1.2		Check the pieces on the board are the correct ones.	Pass	
DD1.3		Make sure the pieces on the board are in the correct position.	Pass	
DD2.0	Piece	Ensure the correct person is in check according to the program.	Pass	
DD2.1		Ensure each piece's type is returned correctly.	Pass	
DD2.2		Ensure each piece's panel is returned correctly.	Pass	
DD2.3		Ensure each piece's value is returned correctly.	Pass	
DD2.4		Ensure the program knows it is check in a position where it is check.	Pass	
DD3.0	Bishop	See if the correct moves are generated for a Bishop.	Pass	
DD4.0	King	See if the correct moves are generated for a King.	Pass	
DD5.0	Knight	See if the correct moves are generated for a Knight.	Pass	
DD6.0	Pawn	See if the correct moves are generated for a Pawn.	Pass	
DD7.0	Queen	See if the correct moves are generated for a Queen.	Pass	
DD8.0	Rook	See if the correct moves are generated for a Rook.	Pass	

Post-development Testing Plan

	Test	Data	Expected Result
F2.0	Test to see if you click on a move that it's moves are displayed correctly	Click on a piece	Moves are displayed for that piece as in the rules
F2.3	Test to see if the pieces move correctly	Click on a piece	A piece moves to a panel if it is moving to a legal space
F2.4	Test if check and checkmate behave correctly	Move to a position where it is check, and then a position of checkmate	A message box appears saying "check" and "checkmate" respectively
F2.8	Test to see if an 8 by 8 grid of squares is generated	Run the game	An 8 by 8 grid of squares is generated
F2.9	Test to see if the pieces are there	Run the game	Pieces are displayed appropriately

Iteration feedback and Analysis



Above are a few images of how the design has turned out. When you are over a panel, the panel is yellow (this aids usability as is helps users to know which panel they are clicking on). The possible moves of a piece are highlighted red, and the most recent move stays highlighted red (I may change this in the next iteration in order to avoid confusion between the most recent move and a possible move). My stakeholders were very satisfied with regards to what I had achieved in this iteration.

"Your project seems to be going really well Jordan. Congratulations, I really like the design, it is very easy to use. Can't wait to see what you come out with after iteration 2." – Elianne Pekoulis David.

Iteration 1 code

BoardGen

```
namespace CollegeProject
{
    public class BoardGen
    {
        Panel[,] boardPanels;
        List<Piece> pieces = new List<Piece>();
        List<Panel> inUse;
        Pawn pawn;
        Rook rook;
        Knight knight;
        Bishop bishop;
        Queen queen;
        King king;

        public BoardGen()
        {

        }


        public Panel[,] GenerateBoard()
        {

            int tileSize = 40;
            int gridSize = 8;
            //Size of squares
            Color color1 = Color.SaddleBrown;
            Color color2 = Color.SandyBrown;
            //Colour of sqaures

            boardPanels = new Panel[gridSize, gridSize];

            //Nested for loop to get the 8 by 8 grid set up
            for (int i = 0; i < gridSize; i++)
            {
                for (int x = 0; x < gridSize; x++)
                {

                    //Set the panel details
                    Panel newPanel = new Panel
                    {
                        Size = new Size(tileSize, tileSize),
                        Location = new Point(tileSize * i, tileSize * x),
                        BackgroundImageLayout = ImageLayout.Stretch,
                    };

                    String peiceType = null;

                    //Set what colour the piece should be
                    peiceType = (tileSize * x <= 40) ? "B" : "W";

                    //Setting what type of pieces go where and geneerating the pieces
                    if(tileSize*x == 40 || tileSize*x == 240)
                    {
                        peiceType += "Pawn";
                        pawn = new Pawn(peiceType, 1, newPanel);
                        pieces.Add(pawn);
```

```
                }
                else if ((tileSize*i == 0 || tileSize*i == 280) &&
                        (tileSize*x == 0 || tileSize*x == 280))
                {
                    peiceType += "Rook";
                    rook = new Rook(peiceType, 3, newPanel);
                    pieces.Add(rook);
                }
                else if ((tileSize * i == 40 || tileSize * i == 240) &&
                        (tileSize * x == 0 || tileSize * x == 280))
                {
                    peiceType += "Knight";
                    knight = new Knight(peiceType, 3, newPanel);
                    pieces.Add(knight);
                }
                else if ((tileSize * i == 80 || tileSize * i == 200) &&
                        (tileSize * x == 0 || tileSize * x == 280))
                {
                    peiceType += "Bishop";
                    bishop = new Bishop(peiceType, 3, newPanel);
                    pieces.Add(bishop);
                }
                else if ((tileSize * i == 160 ) &&
                        (tileSize * x == 0 || tileSize * x == 280))
                {
                    peiceType += "King";
                    king = new King(peiceType, -1, newPanel);
                    pieces.Add(king);
                }
                else if ((tileSize * i == 120) &&
                        (tileSize * x == 0 || tileSize * x == 280))
                {
                    peiceType += "Queen";
                    queen = new Queen(peiceType, 9, newPanel);
                    pieces.Add(queen);
                }

                boardPanels[i, x] = newPanel;

                if (i % 2 == 0)
                    newPanel.BackColor = x % 2 != 0 ? color1 : color2;
                else
                    newPanel.BackColor = x % 2 != 0 ? color2 : color1;
            }
        }

        //update what panels are currently taken
        setPanelsInUse();

        //return the board that has been generated
        return boardPanels;
    }

    public Panel[,] getPanels()
    {
        return boardPanels;
    }

    public List<Piece> getPieces()
    {
        return pieces;
    }
```

```
        public void setPanelsInUse()
        {
            inUse = new List<Panel>();
            foreach (Panel x in boardPanels)
            {
                foreach(Piece p in pieces)
                {
                    if(x == p.getPanel()) {
                        inUse.Add(x);
                    }
                }
            }
        }

        public List<Panel> getPanelsInUse()
        {
            return inUse;
        }

        public void RemovePiece(Piece piece)
        {
            pieces.Remove(piece);
            inUse.Remove(piece.getPanel());
        }

        public void AddPiece(Piece piece)
        {
            pieces.Add(piece);
            inUse.Add(piece.getPanel());
        }
    }
}
```

Form1

```
namespace CollegeProject
{
    public partial class Form1 : Form
    {
        //declaration of variables
        private bool WhiteMove = true;
        private Color tempBackColour;
        private Piece movingPiece = null;
        private Panel movingPiecePanel;
        private BoardGen generation = new BoardGen();
        private Panel[,] board;
        private Panel actionPanel;
        private List<Piece> pieces;
        private Piece newQueen;

        public Form1()
        {
            board = generation.GenerateBoard();
            //Generate the board

            foreach (Panel x in board)
            {
                x.MouseEnter += new EventHandler(panel_MouseEnter);
                x.MouseLeave += new EventHandler(panel_MouseLeave);
                x.MouseDown += new MouseEventHandler(panel_MouseDown);
                Controls.Add(x);
```

```
            }
            //Add the action events to the panels and then render the panels.

            InitializeComponent();
        }

        private void Form1_Load(object sender, EventArgs e)
        {
        }
        private void panel_MouseEnter(object sender, EventArgs e)
        {
            actionPanel = (Panel)sender;
            tempBackColour = actionPanel.BackColor;
            actionPanel.BackColor = Color.Yellow;

            //When mouse is over a panel it appears yellow
        }

        private void panel_MouseLeave(object sender, EventArgs e)
        {
            actionPanel = (Panel)sender;
            actionPanel.BackColor = tempBackColour;

            //Puts panel back to original colour when mouse not over it
        }

        private void panel_MouseDown(object sender, MouseEventArgs e)
        {

            Panel actionPanel = (Panel)sender;

            //This section is for when no piece has been clicked yet
            if (movingPiece == null)
            {
                pieces = generation.getPieces();

                //Gets the piece that was clicked
                foreach (Piece p in pieces)
                {
                    if (p.getPanel() == actionPanel)
                    {
                        movingPiece = p;
                        movingPiecePanel = actionPanel;
                        break;
                    }
                }

                //If the panel that was clicked actually contains a piece
                try
                {
                    //And it's that peice's colours turn
                    if ((WhiteMove &&
                        movingPiece.getType().Substring(0, 1).Equals("W")) ||
                       (!WhiteMove &&
                       movingPiece.getType().Substring(0, 1).Equals("B")))
                    {
                        movingPiece.setgetMoves(generation,  true);
                        //Get that piece's moves

                        foreach (Panel x in movingPiece.getMoves())
                        {
                            x.BackColor = Color.Red;
```

```
                        //And display them
                    }

                }
            }
            catch (NullReferenceException)
            {
                //Could notify the player they clicked an empty sqaure here
            }



        }

        //This section is for when a piece has already been clicked
        else
        {
            movingPiece.setgetMoves(generation, true);
            //Update the moves of that piece for their new position

            //Get the colours of the board back to normal
            for (int k = 0; k <= 7; k++)
            {
                for (int s = 0; s <= 7; s++)
                {

                    if (k % 2 == 0)
                        generation.getPanels()[k, s].BackColor =
                            s % 2 != 0 ? Color.SaddleBrown : Color.SandyBrown;
                    else
                        generation.getPanels()[k, s].BackColor =
                            s % 2 != 0 ? Color.SandyBrown : Color.SaddleBrown;
                }
            }

            //If the panel clicked was not the first panel the person clicked
            if (actionPanel != movingPiecePanel)
            {
                //And the panel is valid
                try
                {
                    if (movingPiece.getMoves().Contains(actionPanel))
                    {

                        if (generation.getPanelsInUse().Contains(actionPanel))
                        {
                            foreach (Piece z in generation.getPieces().ToList())
                            {
                                if (z.getPanel() == actionPanel)
                                {
                                    generation.RemovePiece(z);
                                }
                            }
                            actionPanel.BackgroundImage = null;
                            /*If the panel was a possible move and contains a
                            piece, the piece that it already contains needs
                            to be disposed of*/

                        }
                        if ((WhiteMove &&
                            movingPiece.getType().Substring(0, 1).Equals("W")) ||
                           (!WhiteMove &&
```

```
                movingPiece.getType().Substring(0, 1).Equals("B")))
            {
                actionPanel.BackgroundImage =
                    movingPiece.getPanel().BackgroundImage;
                movingPiece.setPanel(actionPanel);
                generation.setPanelsInUse();
                movingPiecePanel.BackgroundImage = null;
                //As long as the right person (whose turn it is) is
                //moving then this bit above moves it

                if (movingPiece.getType() == "WPawn" ^
                    movingPiece.getType() == "BPawn")
                {
                    if ((movingPiece.getPanel().Location.Y == 0 &&
                        movingPiece.getType() == "WPawn") ^
                       (movingPiece.getPanel().Location.Y == 280 &&
                       movingPiece.getType() == "BPawn"))
                    {
                        generation.RemovePiece(movingPiece);
                        newQueen = new Queen
                            (movingPiece.getType().Substring(0,1) +
                            "Queen", 9, movingPiece.getPanel());
                        generation.AddPiece(newQueen);
                    }
                }
                //If a pawn reaches the end, it becoms a queen

                int totalPossibleMoves = 0;
                foreach (Piece y in generation.getPieces().ToList())
                {
                    y.setgetMoves(generation, true);
                    if ((WhiteMove &&
                       y.getType().Substring(0,1) == "B") ^
                       (!WhiteMove &&
                       y.getType().Substring(0, 1) == "W"))
                    {
                        totalPossibleMoves += y.getMoves().Count;
                    }

                }
                //Make sure there are some possible moves

                if (movingPiece.checkCheck(generation))
                {
                    MessageBox.Show("Check!");
                    //Check check, if it is check, tell the players
                    if(totalPossibleMoves == 0)
                    {
                        //There are no possible moves, it's checkemate
                        MessageBox.Show("Check mate!");
                        //EndGame()
                    }
                }
                WhiteMove = !WhiteMove;
                movingPiece = null;
                //Next players move and a piece has not been clicked

            }
        }
        else
        {
            movingPiece = null;
```

```
                        }
                    }
                    catch(NullReferenceException)
                    {
                        movingPiece = null;
                    }
                }
                else
                {
                    movingPiece = null;
                }
            }
        }
    }
}
```

Piece

```
namespace CollegeProject
{
    public abstract class Piece
    {
        String pieceType;
        int pieceVal;
        Panel piecePanel;
        string typeOfCheck;

        public Piece(String type, int value, Panel image)
        {

            image.BackgroundImage = (Image)
                (Properties.Resources.ResourceManager.GetObject(type));

            pieceType = type;
            pieceVal = value;
            piecePanel = image;

            //Sets original values of the piece
        }

        public string checkType()
        {
            return typeOfCheck;
        }

        public abstract List<Panel> setgetMoves(BoardGen board, bool checkUp);
        //To be overriden by child classes

        public abstract List<Panel> getMoves();
        //To be overriden by child classes

        public String getType()
        {
            return pieceType;
        }

        public void setPanel(Panel newPanel)
        {
            piecePanel = newPanel;
        }

        public Panel getPanel()
        {
```

```
            return piecePanel;
        }

        public int getValue()
        {
            return pieceVal;
        }

        public void takePiece() {

        }

        public bool checkCheck(BoardGen board)
        {
            bool check = false;

            Piece piece = null;

            //Need to check all pieces on board because discovered check exists
            foreach (Piece y in board.getPieces())
            {

                List<Panel> possibleMoves = new List<Panel>();
                possibleMoves = y.getMoves();
                if(y.getMoves() != null)
                {
                    foreach (Panel x in possibleMoves)
                    {

                        foreach (Piece c in board.getPieces())
                        {
                            if (c.getPanel() == x)
                            {
                                piece = c;
                                if (piece.getType().Contains("WKing") &&
                                   (y.getType().Substring(0, 1) !=
                                   piece.getType().Substring(0, 1)))
                                {
                                    typeOfCheck = (piece.getType().Substring(0, 1)
                                        == "W") ? "W" : "B";
                                    return true;
                                }
                            }
                        }
                    }
                }
                //Checks if any piece has a possible move that contains "King",
                //if it does it is check

            }

            return check;

        }
    }
}
```

Bishop

```
namespace CollegeProject
{
    public class Bishop : Piece
    {

        List<Panel> possibleMoves;

        public Bishop(string type, int value, Panel image) :base(type, value, image)
        {
        }
        public override List<Panel> setgetMoves(BoardGen board, bool checkUp)
        {
            possibleMoves = new List<Panel>();
            Piece piece = null;
            Panel tempPan = null;
            //A bishop may move along the diagonal it is on
            foreach (Panel x in board.getPanels())
            {
                for(int i = 0; i < 8; i++)
                {
                    foreach (Piece c in board.getPieces())
                    {
                        if (c.getPanel() == x)
                        {
                            piece = c;
                        }
                    }
                    if ((getPanel().Location.X == x.Location.X + (40 * i) ||
                        getPanel().Location.X == x.Location.X - (40 * i)) &&
                        (getPanel().Location.Y == x.Location.Y + (40 * i) ||
                        getPanel().Location.Y == x.Location.Y - (40 * i)))
                    {
                        if (board.getPanelsInUse().Contains(x))
                        {
                            foreach (Piece y in board.getPieces())
                            {
                                if (y.getPanel() == x)
                                {
                                    if (getType().Substring(0, 1) !=
                                        y.getType().Substring(0, 1))
                                    {
                                        possibleMoves.Add(x);
                                    }
                                }
                            }
                        }
                        else
                        {
                            possibleMoves.Add(x);
                        }
                    }
                }
            }
            /* This bit just gets all of the panels in a diagonal direction from the
            bishop in use */

            List<Panel> panelsRemove = new List<Panel>();
```

```
            foreach (Panel x in board.getPanels())
            {
                for (int i = 0; i < 8; i++)
                {
                    if ((getPanel().Location.X == x.Location.X + (40 * i) ||
                        getPanel().Location.X == x.Location.X - (40 * i)) &&
                        (getPanel().Location.Y == x.Location.Y + (40 * i) ||
                        getPanel().Location.Y == x.Location.Y - (40 * i)) &&
                        board.getPanelsInUse().Contains(x))
                    {
                        if (x.Location.Y < getPanel().Location.Y &&
                            x.Location.X < getPanel().Location.X)
                        {
                            foreach (Panel g in possibleMoves)
                            {
                                if (g.Location.Y < x.Location.Y && g.Location.X <
                                    x.Location.X)
                                {
                                    panelsRemove.Add(g);
                                }
                            }
                        }
                        else if (x.Location.Y < getPanel().Location.Y &&
                            x.Location.X > getPanel().Location.X)
                        {
                            foreach (Panel g in possibleMoves)
                            {
                                if (g.Location.Y < x.Location.Y && g.Location.X >
                                    x.Location.X)
                                {
                                    panelsRemove.Add(g);
                                }
                            }
                        }
                        else if (x.Location.Y > getPanel().Location.Y &&
                            x.Location.X < getPanel().Location.X)
                        {
                            foreach (Panel g in possibleMoves)
                            {
                                if (g.Location.Y > x.Location.Y && g.Location.X <
                                    x.Location.X)
                                {
                                    panelsRemove.Add(g);
                                }
                            }
                        }
                        else if (x.Location.Y > getPanel().Location.Y &&
                            x.Location.X > getPanel().Location.X)
                        {
                            foreach (Panel g in possibleMoves)
                            {
                                if (g.Location.Y > x.Location.Y &&
                                    g.Location.X > x.Location.X)
                                {
                                    panelsRemove.Add(g);
                                }
                            }
                        }
                    }
                }
            }
            /* This part adds all of the panels that are blocked in some way to an
```

```
                array that removes all of those panels from possible moves (the next
                part */
            foreach (Panel x in panelsRemove)
            {
                possibleMoves.Remove(x);
            }
            if (checkUp)
            {
                Piece toRem = null;
                bool removed = false;
                tempPan = getPanel();
                foreach (Panel x in possibleMoves)
                {
                    if (board.getPanelsInUse().Contains(x))
                    {
                        foreach (Piece z in board.getPieces().ToList())
                        {
                            if (z.getPanel() == x)
                            {
                                toRem = z;
                                removed = true;
                                board.RemovePiece(z);
                            }
                        }
                    }
                    setPanel(x);
                    board.setPanelsInUse();
                    foreach (Piece y in board.getPieces().ToList())
                    {
                        if (!y.getType().Contains(getType()))
                            y.setgetMoves(board, false);
                    }
                    if (checkCheck(board) && checkType() ==
                        getType().Substring(0, 1))
                    {
                        panelsRemove.Add(x);
                    }
                    setPanel(tempPan);
                    if (removed)
                    {
                        board.AddPiece(toRem);
                    }
                    board.setPanelsInUse();
                }
                /* This part checks all of the current possible moves for the
                   the bishop to see if any would leave the player in check, if it
                   would, it is not a valid move and is removed */
                foreach (Panel x in panelsRemove)
                {
                    possibleMoves.Remove(x);
                }
            }
            return possibleMoves;
        }
        public override List<Panel> getMoves()
        {
            return possibleMoves;
        }
    }
}
```

Testing

Testing during development (Unit Testing)

Test ID	Class being tested	Function being tested	Pass/Fail	Proof

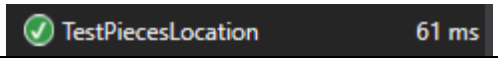
1DD1.0	BoardGen	GenerateBoard()	Pass	`BoardGen gen = new BoardGen();`
				`[TestMethod()]`
				`public void TestBoardSize()`
				`{`
				`  //This procedure ensures the board is an 8 by 8 grid`
				`  Panel[,] board = gen.GenerateBoard();`
				`  if (!(board.GetLength(0) == 8 && board.GetLength(1) == 8))`
				`  {`
				`  Assert.Fail();`
				`  }`
				`}`
				TestBoardSize 49 ms

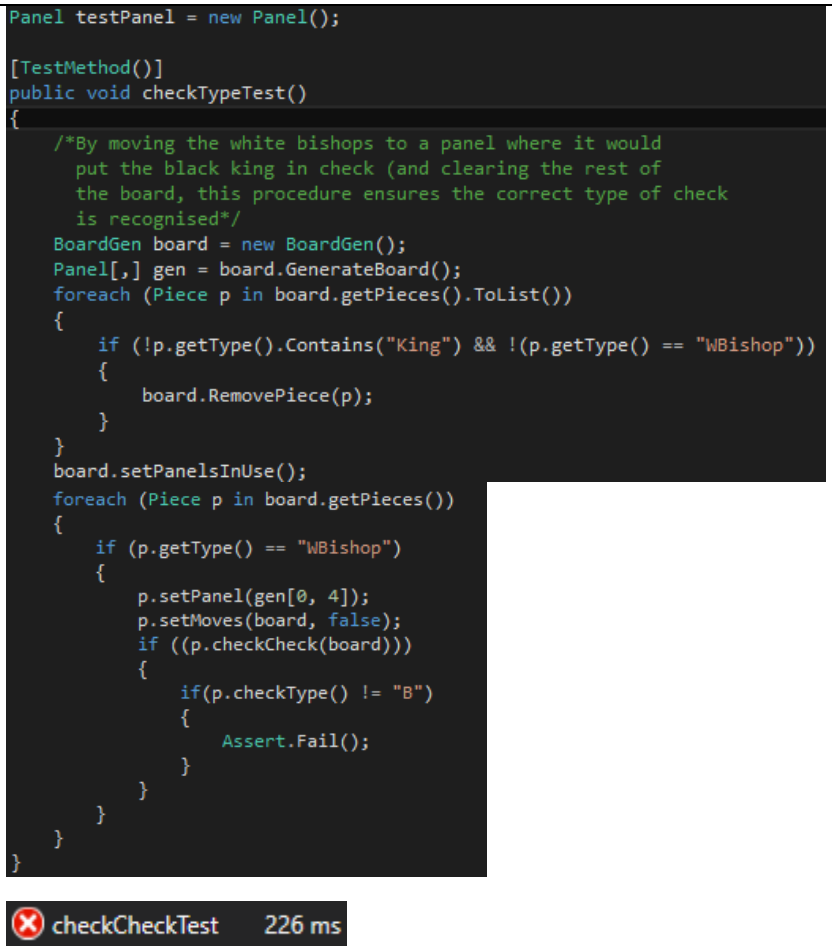
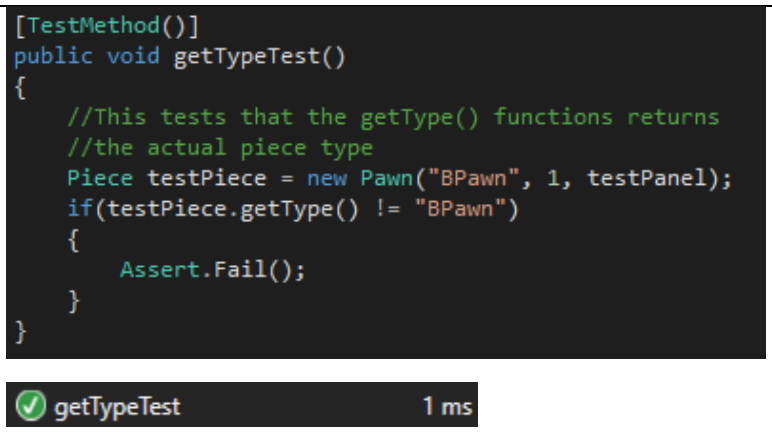
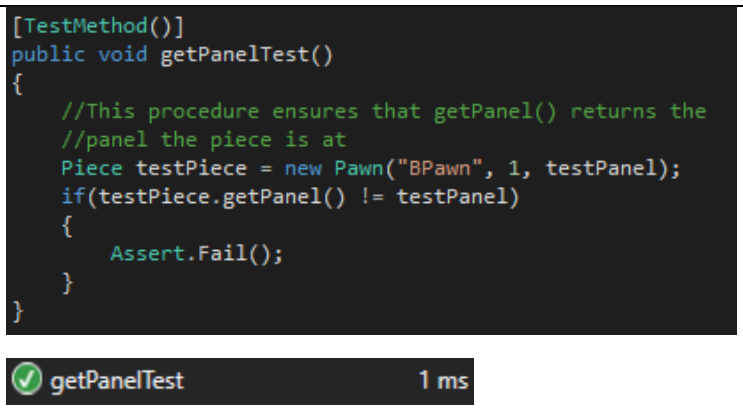
1DD1.1		GenerateBoard()	Pass	`[TestMethod()]`
				`public void TestNumberOfPieces()`
				`{`
				`  /*This procedure ensures there are 32 pieces on the board to begin with*/`
				`  Panel[,] board = gen.GenerateBoard();`
				`  if (gen.getPieces().Count != 32)`
				`  {`
				`  Assert.Fail();`
				`  }`
				`}`
				TestNumberOfPieces 49 ms

1DD1.2		GenerateBoard()	Pass	`[TestMethod()]`
				`public void TestPiecesOnBoard()`
				`{`
				`  //This ensures all pieces are of a valid type`
				`  Panel[,] board = gen.GenerateBoard();`
				`  String[] pieces = { "BPawn", "WPawn", "BRook", "WRook", "BKnight", "WKnight", "BBishop", "WBishop", "BQueen", "WQueen", "BKing", "WKing" };`
				`  foreach(Piece p in gen.getPieces())`
				`  {`
				`  if (!(pieces.Contains(p.getType()))){`
				`  Assert.Fail();`
				`  }`
				`  }`
				`}`
				TestPiecesOnBoard 65 ms

1DD1.3		GenerateBoard()	Pass	

```
[TestMethod()]
public void TestPiecesLocation()
{
    //This ensures that pieces are in the
    //correct relative positions
    Panel[,] board = gen.GenerateBoard();
    foreach (Piece p in gen.getPieces())
    {
        if ((p.getType().Contains("Pawn")) &&
            (p.getPanel().Location.Y != 40 &&
            p.getPanel().Location.Y != 240))
        {
            Assert.Fail();
        }
        if ((p.getType().Contains("Rook")) &&
            (p.getPanel().Location.Y != 0 &&
            p.getPanel().Location.Y != 280) &&
            (p.getPanel().Location.X != 0 &&
            p.getPanel().Location.X != 280))
        {
            Assert.Fail();
        }
        if ((p.getType().Contains("Knight")) &&
            (p.getPanel().Location.Y != 0 &&
            p.getPanel().Location.Y != 280) &&
            (p.getPanel().Location.X != 40 &&
            p.getPanel().Location.X != 240))
        {
            Assert.Fail();
        }
        if ((p.getType().Contains("Bishop")) &&
            (p.getPanel().Location.Y != 0 &&
            p.getPanel().Location.Y != 280) &&
            (p.getPanel().Location.X != 80 &&
            p.getPanel().Location.X != 200))
        {
            Assert.Fail();
        }
        if ((p.getType().Contains("Queen")) &&
            (p.getPanel().Location.Y != 0 &&
            p.getPanel().Location.Y != 280) &&
            (p.getPanel().Location.X != 120))
        {
            Assert.Fail();
        }
        if ((p.getType().Contains("King")) &&
            (p.getPanel().Location.Y != 0 &&
            p.getPanel().Location.Y != 280) &&
            (p.getPanel().Location.X != 160))
        {
            Assert.Fail();
        }
    }
}
```



1DD2.0	Piece	checkType()	Fail	
1DD2.1		getType()	Pass	
1DD2.2		getPanel()	Pass	

<table>
<tr>
<td>1DD2.3</td>
<td></td>
<td>getValue()</td>
<td>Pass</td>
<td>

```
[TestMethod()]
public void getValueTest()
{
    //This proceudre ensures the getValue() function
    //returns the correct value of a piece
    Piece testPiece = new Pawn("BPawn", 1, testPanel);
    if (testPiece.getValue() != 1)
    {
        Assert.Fail();
    }
}
```

</td>
</tr>
<tr>
<td>1DD2.4</td>
<td></td>
<td>checkCheck(BoardGen board)</td>
<td>Pass</td>
<td>

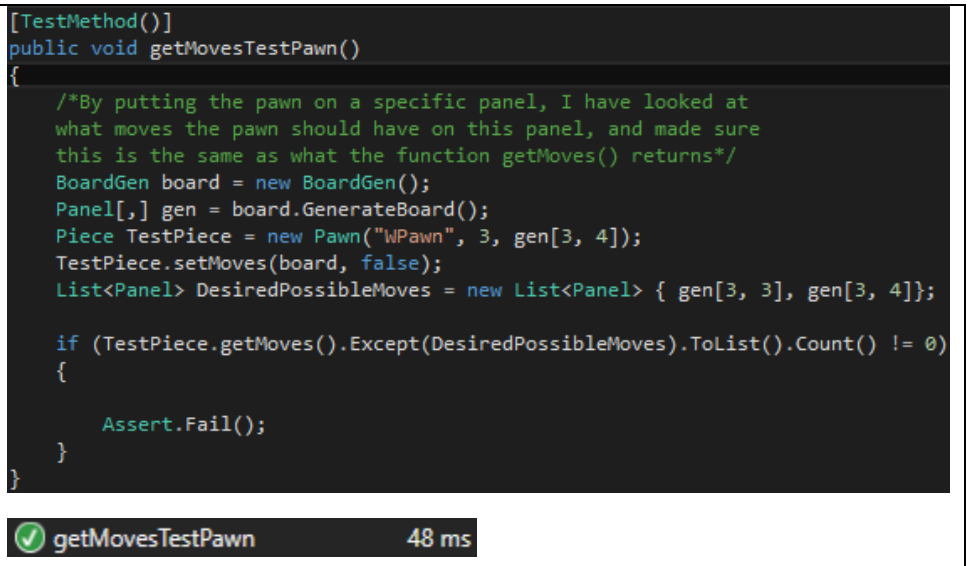
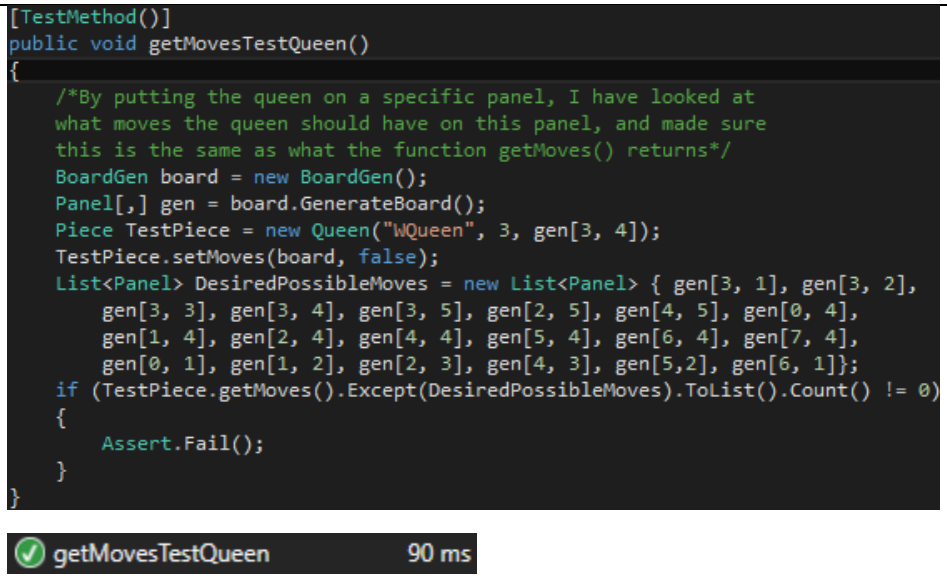
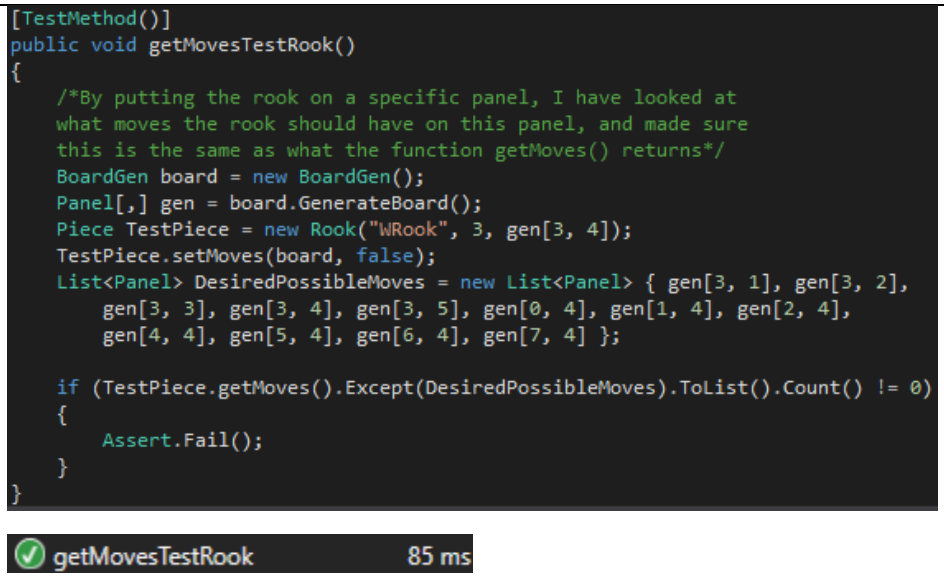
```
[TestMethod()]
public void checkCheckTest()
{
  /*By moving the white bishops to a panel where it would
    put the black king in check (and clearing the rest of
    the board, this procedure ensures check is recognised*/
    BoardGen board = new BoardGen();
    Panel[,] gen = board.GenerateBoard();
    foreach (Piece p in board.getPieces().ToList())
    {
        if(!p.getType().Contains("King") && !(p.getType() == "WBishop"))
        {
            board.RemovePiece(p);
        }
    }
    board.setPanelsInUse();

    foreach (Piece p in board.getPieces())
    {
        if (p.getType() == "WBishop")
        {
            p.setPanel(gen[0, 4]);
            p.setMoves(board, false);
            if (!(p.checkCheck(board)))
            {
                Assert.Fail();
            }
        }
    }
}
```

</td>
</tr>
</table>

1DD3.0	Bishop	getMoves()	Pass	[TestMethod()]
				public void getMovesTestBishop()
				{
				/*By putting the bishop on a specific panel, I have looked at what moves the bishop should have on this panel, and made sure this is the same as what the function getMoves() returns*/
				BoardGen board = new BoardGen();
				Panel[,] gen = board.GenerateBoard();
				Piece TestPiece = new Bishop("WBishop", 3, gen[3, 4]);
				TestPiece.setMoves(board, false);
				List<Panel> DesiredPossibleMoves = new List<Panel> {gen[3, 4], gen[4, 5], gen[2, 3], gen[1, 2], gen[0, 1], gen[2, 5], gen[4, 3], gen[5, 2], gen[6, 1] };
				if (TestPiece.getMoves().Except(DesiredPossibleMoves).ToList().Count() != 0)
				{
				Assert.Fail();
				}
				}
				getMovesTestBishop 137 ms

1DD4.0	King	getMoves()	Pass	[TestMethod()]
				public void getMovesTestKing()
				{
				/*By putting the king on a specific panel, I have looked at what moves the king should have on this panel, and made sure this is the same as what the function getMoves() returns*/
				BoardGen board = new BoardGen();
				Panel[,] gen = board.GenerateBoard();
				Piece TestPiece = new King("WKing", 3, gen[3, 4]);
				TestPiece.setMoves(board, false);
				List<Panel> DesiredPossibleMoves = new List<Panel> { gen[3, 3], gen[3, 4], gen[3, 5], gen[2, 3], gen[2, 4], gen[2, 5], gen[4, 3], gen[4, 4], gen[4, 5] };
				if (TestPiece.getMoves().Except(DesiredPossibleMoves).ToList().Count() != 0)
				{
				Assert.Fail();
				}
				}
				getMovesTestKing 48 ms

1DD5.0	Knight	getMoves()	Pass	[TestMethod()]
				public void getMovesTestKnight()
				{
				/*By putting the knight on a specific panel, I have looked at what moves the knight should have on this panel, and made sure this is the same as what the function getMoves() returns*/
				BoardGen board = new BoardGen();
				Panel[,] gen = board.GenerateBoard();
				Piece TestPiece = new Knight("WKnight", 3, gen[3, 4]);
				TestPiece.setMoves(board, false);
				List<Panel> DesiredPossibleMoves = new List<Panel> { gen[1, 3], gen[1, 5], gen[2, 2], gen[4, 2], gen[5, 3], gen[5, 5], gen[3, 4]};
				if (TestPiece.getMoves().Except(DesiredPossibleMoves).ToList().Count() != 0)
				{
				Assert.Fail();
				}
				}
				getMovesTestKnight 53 ms

1DD6.0	Pawn	getMoves()	Pass	
1DD7.0	Queen	getMoves()	Pass	
1DD8.0	Rook	getMoves()	Pass	

After the checkCheck test failed I spent some time trying to debug it. A added a break point at the return true line and realised it was never happening. I continued to move out if statements checking if they ran and the outside if statements were running. I then realised that I had written “WKing” instead of just “King” and so edited the code as follows:

```
if (piece.getType().Contains("King") &&
   (y.getType().Substring(0, 1) !=
    piece.getType().Substring(0, 1)))
{
        typeOfCheck = (piece.getType().Substring(0, 1)
        == "W") ? "W" : "B";
        return true;
}
```

This fixed the issue and the test now passed.

Post-development testing

Feature and Usability Testing

I will provide screenshots of the game being played to highlight the features included in iteration 1.

Feature ID 1.0, 2.0 and Usability ID 3.1 – Board and Piece Generation

This screenshot clearly shows that when the program is run the board is generated and the pieces are setup. All squares generated are the correct colour and all pieces are in their correct position.

When the piece was clicked, the moves it could make were showed clearly on the board, and when one of those moves was clicked the piece moved to that square.

Feature ID 4.0 – Check and Checkmate

These screenshots show white putting black in check, and when that happens, the king must move somewhere that is not check, or another piece must move in the way of check (in the above example, the last screenshot, the piece moving is the queen). And if it is checkmate, the user is notified (in screenshot 2) and no more moves can be made.

Usability ID 3.5 – Accessibility

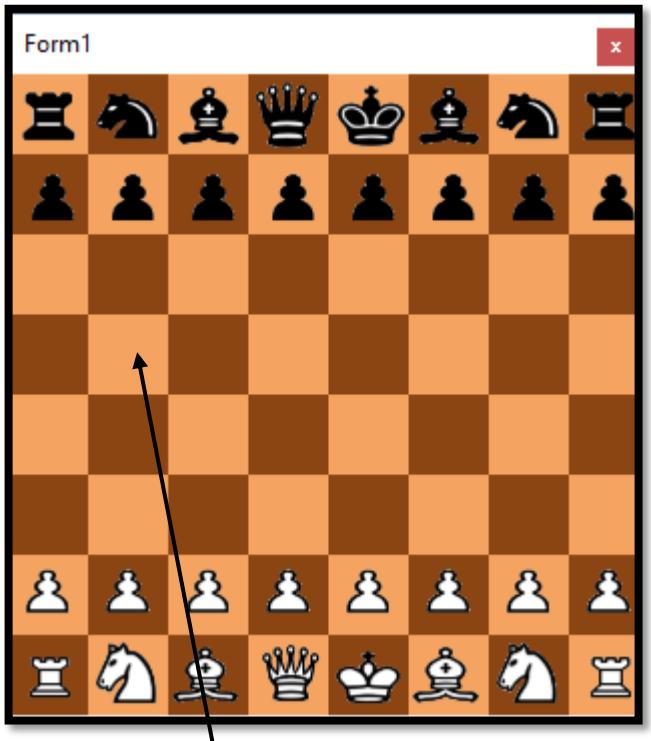
"The game should be easily played by anyone, no matter how much experience they have playing chess."

I think this has been achieved to a suitable level for this iteration. The way the moves for the piece are displayed should make it clear to anyone, even people that have never played before, what moves they can make. However, in future iterations I do hope to improve accessibility by adding some interaction with the user saying "This panel is empty" if they have clicked an empty square, "This move is invalid" for any invalid move a person tries to make and "This piece has no possible moves at the moment" for when a piece is clicked that cannot move.

As you can also see from previous screenshots, I have stuck to mainly neutral players, with more bold colours for important things. I believe this will allow colour-blind users to use the solution comfortably.

Usability ID 3.6 – Error Tolerance

The main way this program defends against errors is by not allowing erroneous events to occur. If a player tries to make an invalid move, the program rejects it. This is essentially the only way to give an invalid input.



If I click on this square after clicking on the pawn (an invalid move for the pawn that was clicked on), the moves are no longer displayed and white is free to make a different move.

	Test	Result	Comments	Pass/Fail
U3.0	Possible moves displayed correctly	Possible moves for each piece displayed as intended	Evidenced above	Pass
U3.2	The pieces move	Pieces moved as intended	Evidenced above	Pass
U3.4	Game is easily played	Game shows moves clearly to user to make it easy to use	Evidenced above	Pass
U3.5	Error Tolerance	No errors encountered		Pass

Analysis of the code

This iteration was split into eight classes: “BoardGen”, “Bishop”, “King”, “Knight”, “Piece”, “Pawn”, “Queen” and “Rook”. The class “Piece” is the super or parent class to the six classes “Bishop”, “King”, “Knight”, “King”, “Queen” and “Rook”. This is because all of these classes share a few key features which allow them to inherit from one parent, they all have a piece type, a panel (as a location on the board), a score of how much they are worth (I believe this will help when developing the gameplay AI in a later iteration), they can all put someone is check and they all need to get generate their moves. It was these properties that I put into the “Piece” class, some of which were overridden by the child class because they needed something more specific, like the actual moves a piece can make is specific to that piece’s type. The “BoardGen” class allowed me to generate the board and setup pieces efficiently, as well as being able to modify the board once it has been setup. An object-oriented approach works well with chess, as there are many unique things to be programmed, that can be abstracted relatively simply. Overall, I would say that the code for iteration 1 works quite well, there are still a few (unusually specific) errors for me to sort out in iteration 2 (for example, if there is a piece blocking the king from being in check, if any piece of the colour that would be in check can be taken, then the program thinks it is check).

Variable Identification and Justification

BoardGen

This class contains 15 (permanent) variables and 8 methods. The 15 variables are:

boardPanels is an array used to store panels in the board.

pieces is a list used to store all of the pieces on the board.

inUse is a list used to store all the panels that have a piece on them.

pawn is used to generate instances of the class “Pawn”.

rook is used to generate instances of the class “Rook”.

knight is used to generate instances of the class “Knight”.

bishop is used to generate instances of the class “Bishop”.

queen is used to generate instances of the class “Queen”.

king is used to generate instances of the class “King”.

tileSize is used to store the length of a side of a tile.

gridSize is used to store the YxY value (in this case 8x8).

colour1 is used to store the first colour of the board panels.

colour2 is used to store the second colour of the board panels.

newPanel is used to generate new panels with their size and location.

pieceType is used to store the colour of type of piece.

Form1

This class contains 9 (permanent) variables and 5 methods. The 9 variables are:

WhiteMove is used to determine whose turn it is to move.

tempBackColour is used to temporarily store the back colour of a panel whilst it is yellow.

movingPiece is used to store the piece that the user wants to move.

movingPiecePanel is used to store the panel location of the piece the user wants to move.

generation is used to generate and store the board.

board is used as an array store of the panels, so the board can be rendered.

actionPanel is used to store the panel the user wants to move to.

pieces is used to store the pieces on the board.

newQueen is used to create the new Queen when a pawn reaches the end of the board.

Piece

This class contains 6 (permanent) variables and 9 methods. The 6 variables are:

pieceType is used to store the colour of type of piece. E.g. “BPawn”.

pieceVal is used to store the value of the piece. For example, a pawn is worth 1 point.

piecePanel is used to store the panel location of the piece.

typeOfCheck is used to store which player is in check, “W” or “B”.

check is used to store whether a player is in check or not.

piece is used as a temporary store of the piece whose moves are being evaluated to see if the opposite coloured king is contained in it.

Pieces (this includes “Bishop”, “King”, “Knight”, “Pawn”, “Queen” and “Rook”)

These classes contain 6 (permanent) variables and 3 methods. The 6 variables are:

possibleMoves is used to store the moves a piece can make.

panelsRemove is used to store the moves a piece would be able to make if it weren’t for there being other being on the board, this list is removed from possibleMoves just after possibleMoves has only moves it could make if the board was empty.

piece is used to temporarily store a piece that is on a panel.

tempPan is used as a way of storing the panel’s actual panel location whilst it is moved around the board to see which (if any) of its move leave the player in check.

toRem is used as a store of a piece that is removed when seeing if any possible moves leave the player in check and is added back to the board once finished checking this move.

removed is used as a way of telling if a piece has been removed from the board or not when checking if any moves leave the player in check.

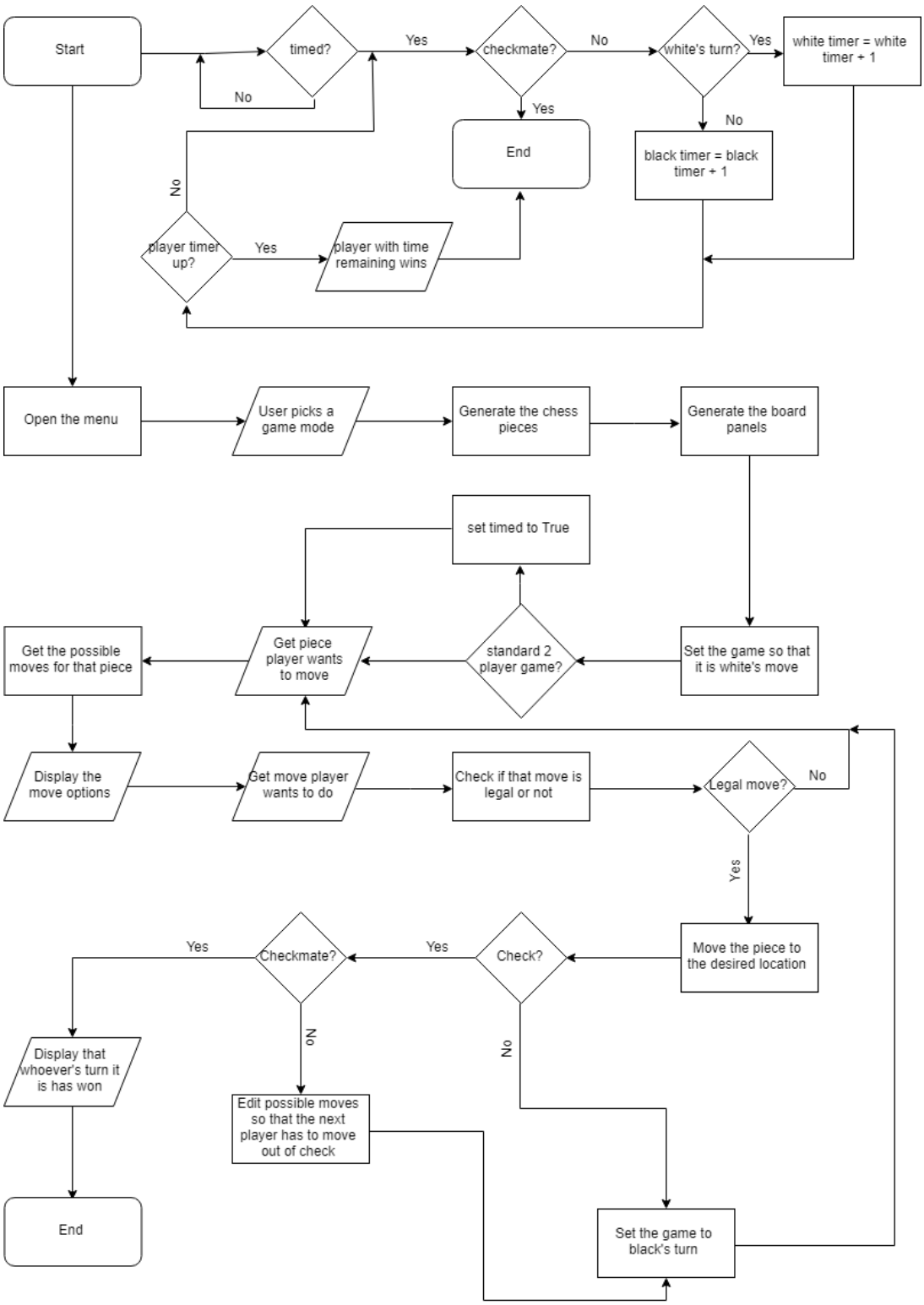
Iteration 1 Stakeholder Sign-off

After development of this iteration was complete I asked my stakeholders to test the game against the iteration requirements. After thorough testing they all agreed that the iteration requirements have been met and that no changes need to be made in this version. They have all agreed for me to move on to the next iteration, where I will ask my stakeholders what needs to be done as priority.

Iteration 2

Planning

Flowchart



Ideas for the iteration

I first thought about what would be manageable to get done in a single iteration. I decided that the AI would most likely take an iteration, and the rest of the requirements could be done in another single iteration. I thought I would get in contact with my stakeholders to see what they would like to see implemented first.

Discussion with Stakeholders

"Hello everyone,

I am currently planning iteration 2 and was wondering what you thought was a more pressing issue. I can either implement the AI in this iteration or work on implementing the game modes, changing the design to look nicer, adding the interface with the timer and pieces taken and making it more user friendly. I'd love to hear your opinions.

Many thanks,
Jordan."

The response I got was overwhelmingly in favour of the latter option. My stakeholders wanted to make sure that all the other things were implemented as they seemed more important to the project than the AI, and I could continue this design work into part of iteration 3, as again they were worried it was a bit too much for one iteration.

"Hi Jordan, I really think you should focus on making the game user friendly and adding the new game modes as this seems like it would add more to you game. The AI can come later." – Eachann

"Hey, whilst I am really looking forward to see the AI implemented, I do think that usability is much more pressing. Let me know what you decide." – Charlie

"I am really not sure. They both seem like really good things to do. I really want to see the AI, but also adding those other features seems really interesting. Sorry I can't be of more help." – Elianne

"Hello Jordan, I am kind of leaning towards the AI. That other stuff does seem really important, but I am just so excited about seeing the AI in action." – Daniel

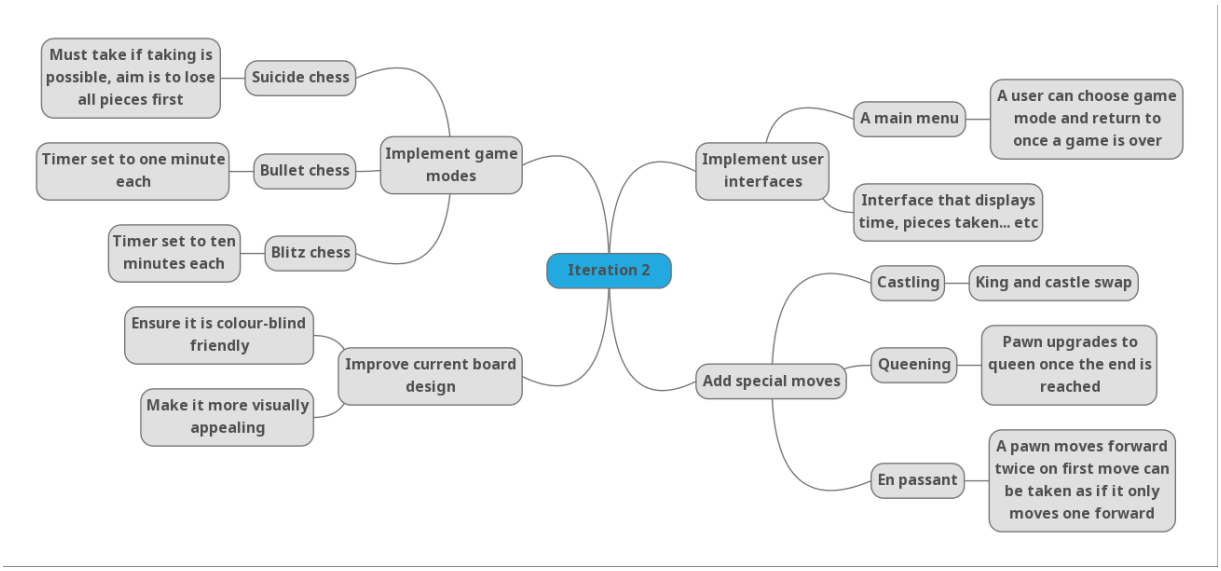
"I think you should go with the usability stuff. It will make your game so much nicer to play and use, it'll add a lot more value to the game." – Martha

"Hi, I was really excited about the new game modes and what they would bring to the game. So, I think you should go with and the things that go with it." – Lewis

Iteration requirements

ID	Description of component
1.0	Make the game easy to play and use
1.1	The user needs to be able to use the game without being confused by what they are meant to do. It should be clear what they are supposed to do in order to play how they want.
1.2	Add rules the user can take a look at if they are confused by how the game is played.
1.3	Add a description of what each game mode entails.
1.4	Make sure the buttons are easy to see and that text is easy to read. Text needs to be obvious to colour blind users.
2.0	Improve the design
2.1	A new colour scheme that appeals to users more than the current one.
2.2	A new interface that is easy to use that adds things like “quit”, “resign”, and a main interface in which you will access the different game modes.
2.3	An interface which shows which pieces have been taken during gameplay and the timers.
3.0	Game modes
3.1	Functional blitz chess (a timer is set to 10 minutes, if someone runs out of time before a checkmate or stalemate, that person loses).
3.2	Functional bullet chess (a timer is set to 1 minute, if someone runs out of time before a checkmate or stalemate, that person loses).
3.3	Functional suicide chess (if a capture is possible, then the player must take a piece. The kinds are “ordinary”, check and checkmate do not exist. Whoever runs out of pieces first wins).
4.0	Add special moves: queening, castling and en passant.

Mindmap



Further discussion with stakeholders

"In this iteration I plan to add the three game modes decided upon at the beginning of the project, change the colour scheme to be more appealing to users, add a menu where the user can access different game modes, the rules and help about what the different game modes are, add the interface which displays times, pieces taken and buttons like "quit" and "resign". Do you think this is a manageable, achievable but still challenging goal?"

In response to this email my stakeholders were very supportive of the criteria.

"Hi Jordan, if you manage to get this done then it'll improve the game so much. This is definitely a challenging goal, but I think you'll be able to do it!" – Martha

"Hey, this looks really good. Based of what you got done in iteration one, you'll be able to manage this. Good luck" – Dan

Iteration requirement 1.0 – Make the game easy to play and use

In order to make the game easier to play I am going to add help forms. These will display the rules and what each game mode actually is. This will hopefully allow the user to understand what is happening when they play the game. To do this all I have to do is create some new forms and create buttons that open these forms on the menu form.

Iteration requirement 2.0 – Improve the design

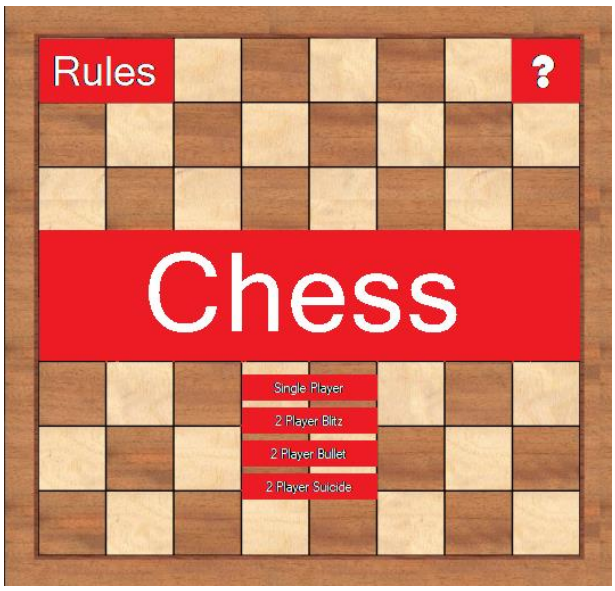
I worked with my stakeholders to choose some colour schemes for the main game play window and the menu. I presented my stake holders with the following options:

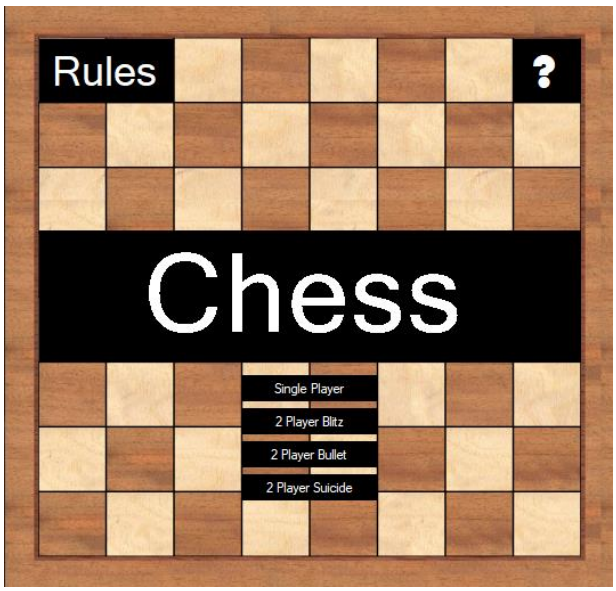
1

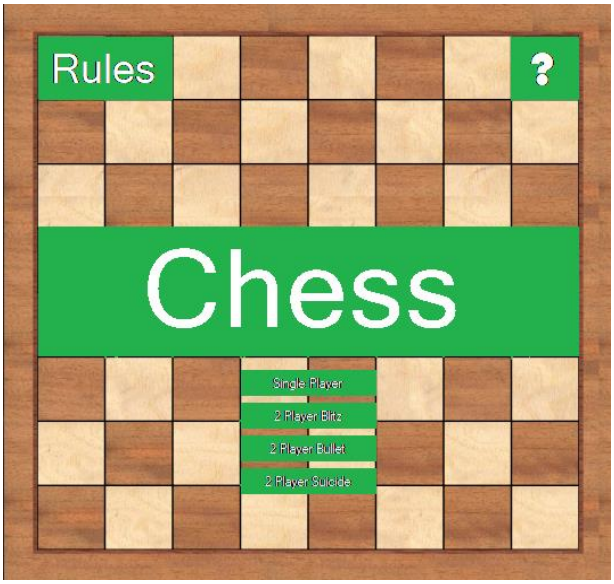
2

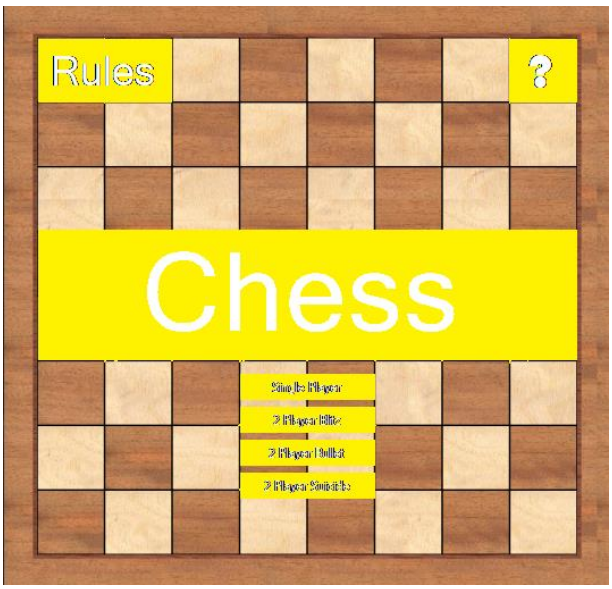
3

My stakeholders overall decided that option number 1 was the most appealing for the board. With a dark grey colour as the background of the interface. I then presented my stakeholders with a few ideas for the menu:









They decided that the black and white design was both easiest to read and looked the nicest of the ones they could read. To implement this change in design all I would have to do is create a new form and change some of the colour codes in the existing form.

Iteration requirement 3.0 – Game modes

For this I thought the best way to implement the game modes would be to add a timer that ticks every second. If the Boolean that already exists called WhiteMove is true, then every tick takes off 1 from white's time, else 1 off black's time. The time is only affected whilst the time is greater than 0, which is why for untimed games the timer value will start at -1. For bullet the timer will start at 60, and for blitz 600. (This is the time in seconds and will be formatted to minutes for a nicer display). All game modes should be played on the same form, I don't really want to have to create separate forms for each game mode as this would unnecessarily repeat a lot of code. Instead, the gameplay form should have an input of gamemode, and this will determine what happens.

Pseudocode Algorithm

```
ON bltizbutton clicked()
        Form1 form = new form1("Blitz")
        form.show()
        this.hide()
END PROCEDURE
ON bulletbutton clicked()
        Form1 form = new form1("Bullet")
        form.show()
        this.hide()
END PROCEDURE
ON suicidebutton clicked()
        Form1 form = new form1("Suicide")
        form.show()
        this.hide()
END PROCEDURE
Form 1(string gameMode)
        IF gameMode == "Blitz" OR "Bullet" THEN
                time = (gameMode == "Blitz")?600:60
        END IF
        IF gameMode == "Suicide" THEN
                IF capturePossible THEN
                        possibleMoves = possibleMoves where capturePossible
                END IF
        END IF
END PROCEDURE
```

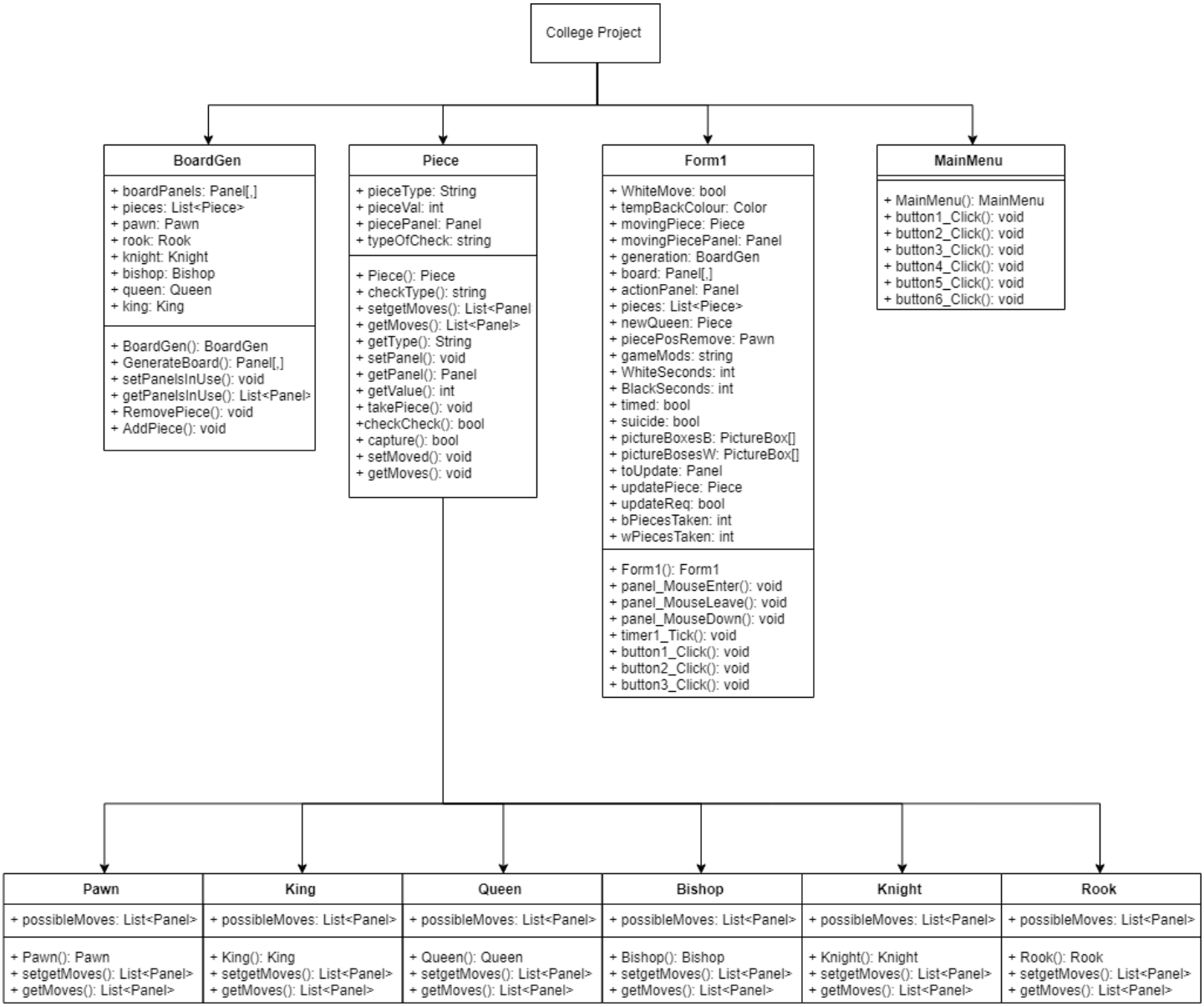
Iteration requirement 4.0 – Special moves

This will require pawns turning into queens when they reach the other side of the board. The king being able to move two squares towards the castle and the castle going on the other side of the king so long as the king hasn't already been moved and the space between the 2 is clear. And when a pawn is moved forward twice on its first move, if there is a pawn that would've been able to take it if it has only moved forward once, it still can.

Pseudocode Algorithm

```
IF attempting to castle THEN
        IF king hasn't moved THEN
                IF space between king and castle is clear THEN
                        move king two spaces towards the castle and move the castle to the other
                        side of the king
                END IF
        END IF
END IF
IF pawn at end of board THEN
        remove pawn
        add queen at pawns position
END IF
IF pawn first move moving forward 2 THEN
        pawn.setPanel(both forward 2 and 1)
END IF
```

UML Diagram


College Project
BoardGen
+ boardPanels: Panel[,]
+ pieces: List<Piece>
+ pawn: Pawn
+ rook: Rook
+ knight: Knight
+ bishop: Bishop
+ queen: Queen
+ king: King
+ BoardGen(): BoardGen
+ GenerateBoard(): Panel[,]
+ setPanelsInUse(): void
+ getPanelsInUse(): List<Panel>
+ RemovePiece(): void
+ AddPiece(): void
Piece
+ pieceType: String
+ pieceVal: int
+ piecePanel: Panel
+ typeOfCheck: string
+ Piece(): Piece
+ checkType(): string
+ setgetMoves(): List<Panel
+ getMoves(): List<Panel>
+ getType(): String
+ setPanel(): void
+ getPanel(): Panel
+ getValue(): int
+ takePiece(): void
+checkCheck(): bool
+ capture(): bool
+ setMoved(): void
+ getMoves(): void
Form1
+ WhiteMove: bool
+ tempBackColour: Color
+ movingPiece: Piece
+ movingPiecePanel: Panel
+ generation: BoardGen
+ board: Panel[,]
+ actionPanel: Panel
+ pieces: List<Piece>
+ newQueen: Piece
+ piecePosRemove: Pawn
+ gameMods: string
+ WhiteSeconds: int
+ BlackSeconds: int
+ timed: bool
+ suicide: bool
+ pictureBoxesB: PictureBox[]
+ pictureBosesW: PictureBox[]
+ toUpdate: Panel
+ updatePiece: Piece
+ updateReq: bool
+ bPiecesTaken: int
+ wPiecesTaken: int
+ Form1(): Form1
+ panel_MouseEnter(): void
+ panel_MouseLeave(): void
+ panel_MouseDown(): void
+ timer1_Tick(): void
+ button1_Click(): void
+ button2_Click(): void
+ button3_Click(): void
MainMenu
+ MainMenu(): MainMenu
+ button1_Click(): void
+ button2_Click(): void
+ button3_Click(): void
+ button4_Click(): void
+ button5_Click(): void
+ button6_Click(): void
Pawn
+ possibleMoves: List<Panel>
+ Pawn(): Pawn
+ setgetMoves(): List<Panel>
+ getMoves(): List<Panel>
King
+ possibleMoves: List<Panel>
+ King(): King
+ setgetMoves(): List<Panel>
+ getMoves(): List<Panel>
Queen
+ possibleMoves: List<Panel>
+ Queen(): Queen
+ setgetMoves(): List<Panel>
+ getMoves(): List<Panel>
Bishop
+ possibleMoves: List<Panel>
+ Bishop(): Bishop
+ setgetMoves(): List<Panel>
+ getMoves(): List<Panel>
Knight
+ possibleMoves: List<Panel>
+ Knight(): Knight
+ setgetMoves(): List<Panel>
+ getMoves(): List<Panel>
Rook
+ possibleMoves: List<Panel>
+ Rook(): Rook
+ setgetMoves(): List<Panel>
+ getMoves(): List<Panel>

Testing Plan

During development testing will need to take place, to see if the special moves work, to test the functionality of the interface and menu, to test the functionality of the game modes. Any further testing should be to highlight the resilience to erroneous inputs and make effort to find any bugs within the code.

Testing during development should show what data was inputted, the intended result, the actual result the program gave, and whether this was a match or not. This testing should also include some explanation for why it may not be working as intended, with any console output if there is a crash and any relevant variables with a content analysis.

Post-development testing will be necessary to ensure that the designs have been displayed correctly, test the functionality of the interface, objectively comment on how easy to use the game is. Also, to get some feedback on how helpful the modifications to the game have been.

Usability Testing Plan

	Test	Result	Comments
U3.1	Test if rules for each mode are available for display		
U3.4	Test to see if unexperienced players and colour-blind players could play		

During Development Testing Plan

Test ID	Class being tested	Test Aim	Expected Result	Proof
2DD1.0	Form1	Make sure the game modes are passed in correctly and causing the correct changed	Pass	
2DD1.1		Make sure the board is the correct colour	Pass	

During this iteration, not much code will be produced that is testable in this way.

Post-development Testing Plan

	Test	Data	Expected Result
F2.5	Test to see if pieces taken and timer are displayed	Start the game a click on 2 player blitz, move such that a piece is taken	The taken piece will be displayed and the timers will go down
F2.6	Test to see if special moves work	Move such that special moves are active	The special move will occur
F2.7	Test to see if different game modes work	Run game and click on each game mode	Game modes will work

Iteration 2 edits to code

BoardGen was not edited during this iteration.

GameWindow (Form1)

```
namespace CollegeProject
{
  public partial class GameWindow : Form
  {

    //declaration of variables
    Pawn piecePosRemove;
    string gameModes;
    int WhiteSeconds;
    int BlackSeconds;
    bool timed = false;
    bool suicide = false;
    private bool WhiteMove = true;
    private Color tempBackColour;
    private Piece movingPiece = null;
    private Panel movingPiecePanel;
    private BoardGen generation = new BoardGen();
    private Panel[,] board;
    private Panel actionPanel;
    private List<Piece> pieces;
    private Piece newQueen;
    PictureBox[] pictureBoxesB;
    PictureBox[] pictureBoxesW;
    Panel toUpdate = null;
    Piece updatePiece = null;
    bool updateReq = false;
    int bPiecesTaken = 0;
    int wPiecesTaken = 0;


    public Form1(String gameMode)
    {
      gameModes = gameMode;
      if (gameMode == "M3S") //if the gameMode is M3S then the user is
                             playing suicide chess
      {
        suicide = true;
      }
```

```
board = generation.GenerateBoard();
//Generate the board
pictureBoxesB = new PictureBox[15];
int z = 330;
int y = 60;
if (gameMode == "M1B" || gameMode == "M2B") //if the gameMode is
                                              M1B or M2B then the
                                              game is timed
{
  WhiteSeconds = gameMode == "M1B" ? 600 : 60; //and the time is
                                                set accordingly
  BlackSeconds = WhiteSeconds;
  timed = true;
}

foreach (PictureBox x in pictureBoxesB) //Generates the interface
                                          where taken black pieces
                                          are stored
{
  PictureBox newPictureBox = new PictureBox
  {
    Size = new Size(25, 25),
    BackgroundImageLayout = ImageLayout.Stretch,
  };
  newPictureBox.Location = new Point(z, y);
  z += 25;
  if (z > 505)
  {
    z = 330;
    y += 25;
  }
  pictureBoxesB[bPiecesTaken] = newPictureBox;
  Controls.Add(pictureBoxesB[bPiecesTaken]);
  bPiecesTaken++;

}

pictureBoxesW = new PictureBox[15];
z = 330;
y = 215;
bPiecesTaken = 0;

foreach (PictureBox x in pictureBoxesW) //Generates the interface
                                          where taken white pieces
{
  PictureBox newPictureBox = new PictureBox
  {
    Size = new Size(25, 25),
    BackgroundImageLayout = ImageLayout.Stretch,
  };
  newPictureBox.Location = new Point(z, y);
  z += 25;
  if (z > 505)
  {
    z = 330;
    y += 25;
  }
  pictureBoxesW[bPiecesTaken] = newPictureBox;
  Controls.Add(pictureBoxesW[bPiecesTaken]);
  bPiecesTaken++;

}

foreach (Panel x in board)
{
```

```
        x.MouseEnter += new EventHandler(panel_MouseEnter);
        x.MouseLeave += new EventHandler(panel_MouseLeave);
        x.MouseDown += new MouseEventHandler(panel_MouseDown);
        Controls.Add(x);
      }
      //Add the action events to the panels and then render the panels.
      bPiecesTaken = 0;
      InitializeComponent();
      if (gameMode == "M1B" || gameMode == "M2B") //formats the time to
                                                    display in minutes
      {
        label3.Text = gameMode == "M1B" ? "10:00" : "01:00";
        label4.Text = label3.Text;
      }

      timer1.Start(); //starts the timer

    }

    private void panel_MouseEnter(object sender, EventArgs e)
    {
      if (timer1.Enabled)
      {
        actionPanel = (Panel)sender;
        tempBackColour = actionPanel.BackColor;
        actionPanel.BackColor = ColorTranslator.FromHtml("#ebe867");
      }
      //When mouse is over a panel it appears yellow
    }

    private void panel_MouseLeave(object sender, EventArgs e)
    {
      if (timer1.Enabled)
      {
        actionPanel = (Panel)sender;
        actionPanel.BackColor = tempBackColour;
      }


      //Puts panel back to original colour when mouse not over it
    }

    private void panel_MouseDown(object sender, MouseEventArgs e)
    {

      Panel actionPanel = (Panel)sender;
      bool castle = false;

      //This section is for when no piece has been clicked yet
      if (timer1.Enabled) //if the game hasn't ended
      {
        if (movingPiece == null)
        {
          pieces = generation.getPieces();

          //Gets the piece that was clicked
          foreach (Piece p in pieces)
          {
            if (p.getPanel() == actionPanel)
            {
              movingPiece = p;
              movingPiecePanel = actionPanel;
              break;
            }
          }
```

```
        //If the panel that was clicked actually contains a piece
        try
        {
          //And it's that peice's colours turn
          if ((WhiteMove &&
              movingPiece.getType().Substring(0, 1).Equals("W")) ||
             (!WhiteMove &&
             movingPiece.getType().Substring(0, 1).Equals("B")))
          {
            movingPiece.setMoves(generation, true, suicide);
            //Get that piece's moves

            foreach (Panel x in movingPiece.getMoves())
            {
              x.BackColor = ColorTranslator.FromHtml("#FFFC7F");
              //And display them
            }

          }
        }
        catch (NullReferenceException)
        {
          //Could notify the player they clicked an empty sqaure here
        }
      }

      //This section is for when a piece has already been clicked
      else
      {
        movingPiece.setMoves(generation, true, suicide);
        //Update the moves of that piece for their new position

        //Get the colours of the board back to normal
        for (int k = 0; k <= 7; k++)
        {
          for (int s = 0; s <= 7; s++)
          {

            if (k % 2 == 0)
            {
              generation.getPanels()[k, s].BackColor =
                  s % 2 != 0 ? ColorTranslator.FromHtml("#80A83E") :
                               ColorTranslator.FromHtml("#D9DD76");
            }
            else
            {
              generation.getPanels()[k, s].BackColor =
                  s % 2 != 0 ? ColorTranslator.FromHtml("#D9DD76") :
                               ColorTranslator.FromHtml("#80A83E");
            }
          }
        }

        //If that panel clicked was not the same as the first panel the
          person clicked
        if (actionPanel != movingPiecePanel &&
            movingPiece.getType().Substring(0, 1) ==
            (WhiteMove ? "W" : "B"))
        {
          //And the panel is valid
          try
          {
            if (movingPiece.getMoves().Contains(actionPanel))
            {
```

```
if (generation.getPanelsInUse().Contains(actionPanel))
{
  foreach (Piece z in generation.getPieces().ToList())
  {
    if (z.getPanel() == actionPanel)
    {
      generation.RemovePiece(z);
      if (z.getType().Substring(0, 1) == "W")
      {
        pictureBoxesB[bPiecesTaken].BackgroundImage =
                            z.getPanel().BackgroundImage;
        bPiecesTaken++;

      }
      else
      {
        pictureBoxesW[wPiecesTaken].BackgroundImage =
                            z.getPanel().BackgroundImage;
        wPiecesTaken ++;
      }
    }
  }
  actionPanel.BackgroundImage = null;
  /*If the panel was a possible move and contains a
  piece, the piece that it already contains needs
  to be disposed of and stored in the interface where
  taken pieces are displayed*/

}
if (movingPiece.getType().Contains("King"))
{
  if ((actionPanel == generation.getPanels()[6,
        movingPiece.getType().Substring(0, 1) == "W" ? 7
                                                     : 0]
       || actionPanel == generation.getPanels()[2,
                       movingPiece.getType().Substring(0,
                                        1) == "W" ? 7 : 0])
       && !movingPiece.getMoved())
  {
    castle = true;
  }
}
//Recognises when the king is trying to castle


if ((WhiteMove &&
    movingPiece.getType().Substring(0, 1).Equals("W")) ||
   (!WhiteMove &&
   movingPiece.getType().Substring(0, 1).Equals("B")))
{
  actionPanel.BackgroundImage =
  movingPiece.getPanel().BackgroundImage;

  undoPiece.Add(movingPiece);
  undoPanel.Add(movingPiece.getPanel());
  noUndos++;

  movingPiece.setPanel(actionPanel);

  if (updateReq && movingPiece.getPanel() !=
      updatePiece.getPanel())
  {
    updatePiece.setPanel(toUpdate);
    generation.RemovePiece(piecePosRemove);
```

```
            generation.setPanelsInUse();

            updateReq = false;
            //if a pawn has moved forward two last move, it can no
              longer be taken as if it were on the first square,
              so don't allow that to happen.
          }
          else if (updateReq)
          {
            if (movingPiece.getType().Substring(0, 1) == "W")
            {
              pictureBoxesW[wPiecesTaken - 1].BackgroundImage =
                                                  Resources.BPawn;
            }
            else
            {
              pictureBoxesB[bPiecesTaken - 1].BackgroundImage =
                                                  Resources.WPawn;
            }
            updatePiece.getPanel().BackgroundImage =
                        movingPiece.getPanel().BackgroundImage;
            piecePosRemove.getPanel().BackgroundImage = null;
            generation.RemovePiece(piecePosRemove);
            updatePiece.setPanel(null);
            generation.RemovePiece(updatePiece);
            generation.setPanelsInUse();
            updateReq = false;

            //If a pawn has moved foward two last move, and a
              pawn has taken it as if it were on the first
              square, this allows it to be taken.
          }
          if (movingPiece.getType().Contains("Pawn"))
          {
            if (!movingPiece.getMoved() &&
                 movingPiece.getPanel().Location.Y ==
                 movingPiecePanel.Location.Y +
                 ((movingPiece.getType().Contains("W")) ? -80 :
                   80))
            {
              foreach (Panel x in generation.getPanels())
              {
                if (x.Location.Y +
                   (movingPiece.getType().Contains("W") ? -40 :
                    40) == actionPanel.Location.Y && x.Location.X
                    == actionPanel.Location.X)
                {
                  toUpdate = actionPanel;
                  updatePiece = movingPiece;
                  movingPiece.setPanel(x);
                  movingPiece.getPanel().BackgroundImage = null;
                  updateReq = true;

                  piecePosRemove = new Pawn
                                       (updatePiece.getType(),
                                        actionPanel, true);
                  generation.AddPiece(piecePosRemove);
                  break;
                }
              }
            }
          }

          generation.setPanelsInUse();
          movingPiecePanel.BackgroundImage = null;
```

```
movingPiece.setMoved(true);
movingPiece.increaseNoMoves();

//As long as the right person (whose turn it is) is
//moving then this bit above moves it

if (castle)  //if the king is attempting to castle
{
  int action = -1;
  if (actionPanel == generation.getPanels()[6,
                     movingPiece.getType().Substring(0,
                     1) == "W" ? 7 : 0])
  {
    foreach (Piece x in generation.getPieces())
    {
      if (x.getPanel() == generation.getPanels()[7,
                     movingPiece.getType().Substring(0,
                     1) == "W" ? 7 : 0])
      {
        movingPiece = x;
        action = 5;
      }
    }
  }
  if (actionPanel == generation.getPanels()[2,
                     movingPiece.getType().Substring(0,
                     1) == "W" ? 7 : 0])
  {
    foreach (Piece x in generation.getPieces())
    {
      if (x.getPanel() == generation.getPanels()[0,
          movingPiece.getType().Substring(0, 1) == "W"
          ? 7 : 0])
      {
        movingPiece = x;
        action = 3;
      }
    }
  }
  actionPanel = generation.getPanels()[action,
                     movingPiece.getType().Substring(0,
                     1) == "W" ? 7 : 0];
  movingPiecePanel = movingPiece.getPanel();
  actionPanel.BackgroundImage =
  movingPiece.getPanel().BackgroundImage;

  undoPiece.Add(movingPiece);
  undoPanel.Add(movingPiece.getPanel());
  noUndos++;


  movingPiece.setPanel(actionPanel);

  generation.setPanelsInUse();
  movingPiecePanel.BackgroundImage = null;
  movingPiece.setMoved(true);
  movingPiece.increaseNoMoves();

  //moves the castle to thing other side of the king
}

if (movingPiece.getType() == "WPawn" ^
    movingPiece.getType() == "BPawn")
{
  if ((movingPiece.getPanel().Location.Y == 0 &&
```

```
                movingPiece.getType() == "WPawn") ^
                (movingPiece.getPanel().Location.Y == 280 &&
                movingPiece.getType() == "BPawn"))
            {
              generation.RemovePiece(movingPiece);
              newQueen = new Queen
                  (movingPiece.getType().Substring(0, 1) +
                  "Queen", 9, movingPiece.getPanel(), true);
              generation.AddPiece(newQueen);
            }
          }
          //If a pawn reaches the end, it becomes a queen

          int totalPossibleMoves = 0;
          foreach (Piece y in generation.getPieces().ToList())
          {
            y.setMoves(generation, true, suicide);
            if ((WhiteMove &&
               y.getType().Substring(0, 1) == "B") ^
               (!WhiteMove &&
               y.getType().Substring(0, 1) == "W"))
            {
              totalPossibleMoves += y.getMoves().Count;
            }

          }
          //Make sure there are some possible moves

          if (!(gameModes == "M3S") &&
                movingPiece.checkCheck(generation))
          {
            MessageBox.Show("Check!");
            //Check check, if it is check, tell the players
            if (totalPossibleMoves == 0)
            {
              //There are no possible moves, it's checkmate
              timer1.Stop();
              MessageBox.Show("Check mate! " +
                              (movingPiece.checkType() == "B" ?
                               "White" : "Black") + " wins!");
              //EndGame()
            }
          }

          WhiteMove = !WhiteMove;
          movingPiece = null;
          //Next players move and a piece has not been clicked

          else
          {
            movingPiece = null;
          }


        }
      }
      else
      {
        movingPiece = null;
      }
    }
    catch (NullReferenceException)
    {
      movingPiece = null;
    }
```

```
        }
      }
    }

  }

  private void timer1_Tick(object sender, EventArgs e)
  {

    //Each time the timer ticks
    if (timed) // if this is a timed game
    {
      if (WhiteMove) //if it is white's move
      {

        if (WhiteSeconds == 1)
        {
          label4.Text = "00:00";
          MessageBox.Show("White out of time. Black wins!");
          timer1.Stop();
          //If white is out of time, end the game and announce black as
          the winner
        }
        label3.ForeColor = Color.White;
        int seconds = WhiteSeconds % 60;
        int minutes = WhiteSeconds / 60;
        string time = minutes.ToString("00") + ":" +
                      seconds.ToString("00");
        label4.Text = time;
        WhiteSeconds--;

        //decrease the time and display the new time


      }
      else
      {

        if (BlackSeconds == 1)
        {
          label3.Text = "00:00";
          MessageBox.Show("Black out of time. White wins!");
          timer1.Stop();
          //If black is out of time, end the game and announce white as
          winner.
        }
        label3.ForeColor = Color.Red;
        label4.ForeColor = Color.White;
        int seconds = BlackSeconds % 60;
        int minutes = BlackSeconds / 60;
        string time = minutes.ToString("00") + ":" +
                      seconds.ToString("00");
        label3.Text = time;
        BlackSeconds--;

        //decrease time and display new time


      }
    }
  }

  private void button1_Click(object sender, EventArgs e)
  {
```

```
      MessageBox.Show("White had resigned. Black Wins!");
      timer1.Stop();
      //If the white has resigned, end then game and display the
        information
    }

    private void button2_Click(object sender, EventArgs e)
    {
      MessageBox.Show("Black had resigned. White Wins!");
      timer1.Stop();
      //If the black has resigned, end then game and display the
        information
    }

    private void button3_Click(object sender, EventArgs e)
    {
      MainMenu menu = new MainMenu();
      menu.Show();
      this.Close();
      //If quit is pressed, go back to the main menu.
    }
}
```

Piece

A Boolean called moved was added to the parameters of Piece and all of its child classes. This was to keep track of what piece's had been moved or not primarily for the special moves, like Castling can only happen if the King hasn't moved yet. As well as the following bits of code being added or modified.

```
public override bool capture()
{
  return capturePossible;
}

public void setMoved(bool move)
{
  hasMoved = move;
}

public bool getMoved()
{
  return hasMoved;
}

public override void setMoves(BoardGen board, bool checkUp, bool suicide)
{
  ...
  if (checkUp && !suicide) //if checking for check and this is not suicide chess (as
                             chess doesn't exist in suicide chess)
      {
        ...
      }
  }
  ...

  if (capturePossible && suicide) //if it is suicide chess and there is a capture
                                    possible
  {
    List<Panel> newPossibleMoves = new List<Panel>();
    foreach (Panel x in possibleMoves)
    {
      foreach (Piece y in board.getPieces())
      {
```

```
        if (x == y.getPanel())
        {
          newPossibleMoves.Add(x);
        }
      }
    }
    possibleMoves = newPossibleMoves;
    //captures are the only possible moves if a capture is possible
  }

  else if (!capturePossible && suicide)  //if there no captures possible in suicide
                                           chess
  {
    foreach (Piece x in board.getPieces())
    {
      if (x.capture() && x.getType().Substring(0, 1) == getType().Substring(0, 1))
      {
        possibleMoves = new List<Panel>();
        break;
        //if there are no possible captures, but there are possible captures from
          other pieces, this piece cannot move
      }
    }
  }
}
```

MainMenu (Form2)

```
namespace CollegeProject
{
    public partial class MainMenu : Form
    {
        public MainMenu()
        {
            InitializeComponent();
        }

        private void button1_Click(object sender, EventArgs e)
        {
            GameWindow gameWindow = new GameWindow("S1S");
            gameWindow.Show();
            this.Hide();
            //If "Single Player" is clicked, open form1 with game code "S1S"
        }

        private void button2_Click(object sender, EventArgs e)
        {
            GameWindow gameWindow = new GameWindow("M1B");
            gameWindow.Show();
            this.Hide();
            //If "2 Player Blitz" is clicked, open form1 with game code "M1B"
        }

        private void button3_Click(object sender, EventArgs e)
        {
            GameWindow gameWindow = new GameWindow("M2B");
            gameWindow.Show();
            this.Hide();
            //If "2 Player Bullet" is clicked, open form1 with game code "M2B"
        }

        private void button4_Click(object sender, EventArgs e)
        {
            GameWindow gameWindow = new GameWindow("M3S");
            gameWindow.Show();
```

```
            this.Hide();
            //If "2 Player Suicide" is clicked, open form1 with game code "M3S"
        }

        private void button5_Click(object sender, EventArgs e)
        {
            Help help = new Help();
            help.Show();
            //If "?" is clicked, open the "help" form (form3).
        }

        private void button6_Click(object sender, EventArgs e)
        {
            Rules rules = new Rules();
            rules.Show();
            //If "Rules" is clicked, open the "rules" form (form4).
        }
    }
}
```

Help (Form3)

Automatically generated code.

Rules (Form4)

Automatically generated code.

Analysis of the code edits

The classes edited in this iteration were all to help meet the iteration requirements. There were three new forms that needed to be added: the help form, the rules form and the form for the main menu. The help and rules form only needed to contain text, so the automatically generated code from visual studio when adding a new form. The main menu needed to contain code. This would be where a user chose which game mode they wanted to play. To limit the amount of repeated code I passed in a game mode code when creating a game play form which uniquely identifies what game mode the user is trying to play. This game mode code forms the basis on which decisions are made about what happens. If the user is in single player, nothing will happen during this iteration as this is not coded yet. If the user is in blitz or bullet then a timer needed to be setup, if the user is in suicide chess then the rules of moving need to be changed, there can be no check or checkmate and there needs to be no timer. Other edits to this class include setting up Panels off of the board where taken pieces can be stored, coding the functionality of the buttons (resigning and quitting) and coding what happens when a timer ticks.

New Variable Identification and Justification

GameWindow (Form 1)

piecePosRemove exists for en passant. This variable keeps track of the abstract pawn that could be taken from an en passant, however will need to be deleted as soon as the opponent makes a move.

gameModes is to make the game mode that is being played accessible by the whole class.

WhiteSeconds keeps track of the time white has remaining.

BlackSeconds keeps track of the time black has remaining.

timed defines whether the game is timed or not.

suicide defines whether we are in a game of suicide chess or not.

pictureBoxesB is a list of the picture poxes of where black pieces will be stored when taken.

pictureBoxesW is a list of the picture boxes of where white pieces will be stored when taken.

bPiecesTaken keeps track of how many black pieces have been taken.

wPiecesTaken keeps track of how many white pieces have been taken.

action is used to determine where the king is going during a castle.

castle is used to determine whether the king is castling.

timer1 used to keep track of the time of each player.

menu is used to create and open the main menu.

updateReq

seconds keeps track of the seoncds for the time

minutes keeps track of the minutes for the time

time keeps track of time remaining

Piece

hasMoved keeps track of whether a piece has moved.

newPossibleMoves is a list that contains all the moves that would be able to take a piece if it is suicide chess.

capturePossible keeps track of whether that piece can capture a piece or not.

MainMenu

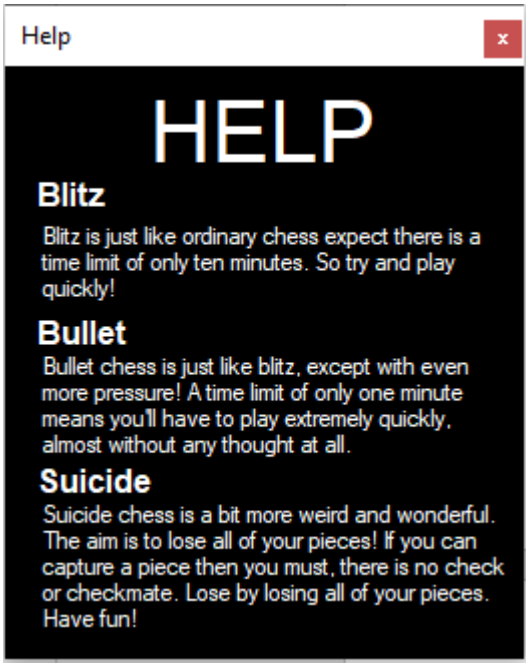
gameWindow is used to create a new game play form.

help is used to create a new help form.

rules is used to create a new rules form.

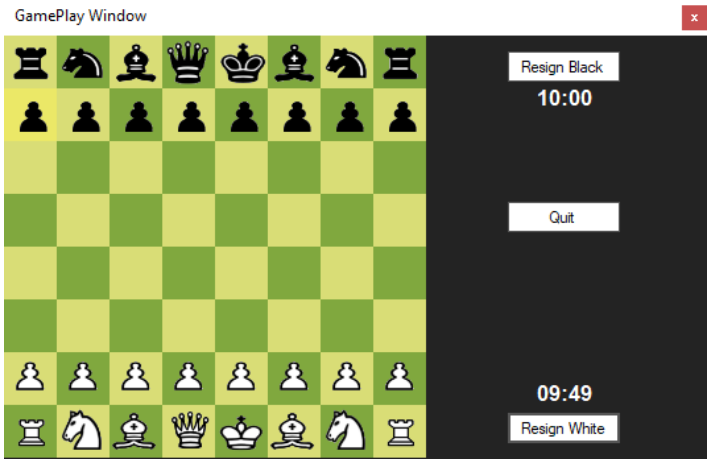
Iteration feedback and post-development testing

Usability ID - 1.0



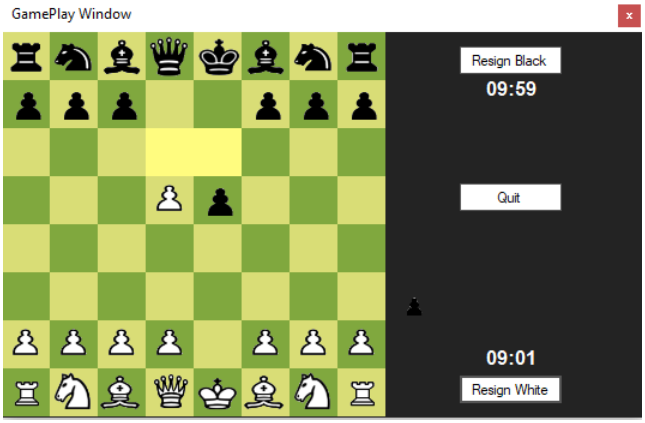
Here is the main menu, the help form and the rules form. I have tried to explain as clearly as I can the rules and chess and what the different game modes are. The design of the main menu perfectly matches that chosen by the stakeholders.

Feature ID – 3.1 Blitz chess

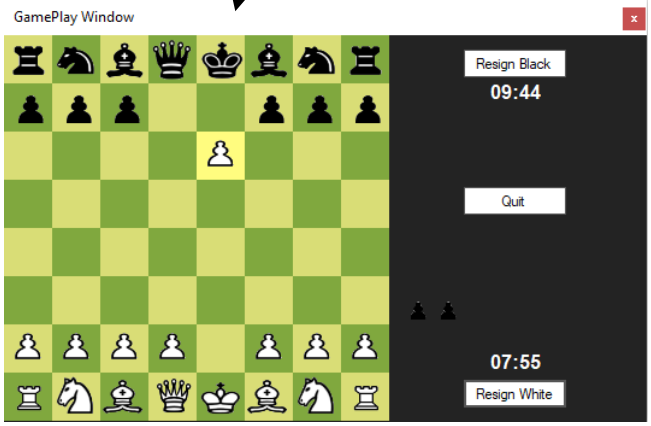


Demonstrated here is the blitz game mode. The timer is set to 10 minutes for each player and white begins.

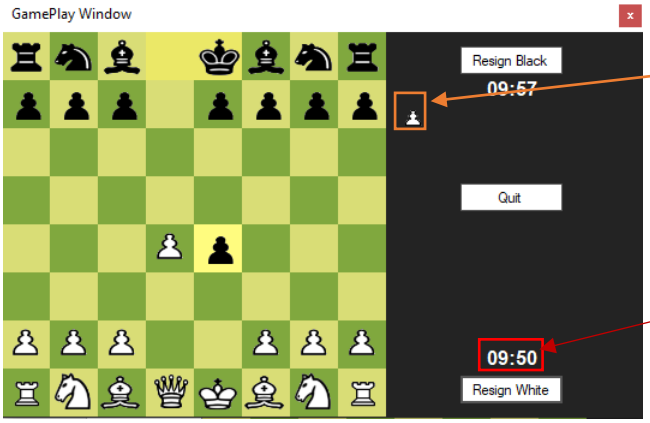
Feature ID – 4.0 Special move (en passant)



Here is the en passant special move. The black pawn has just moved forward twice, and the white pawn would've been able to take it if it has only moved forward once, so can and does take it.

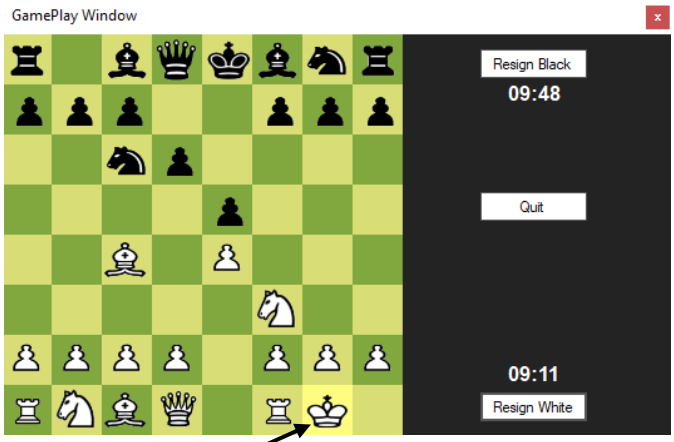


Usability ID – 2.3 interface



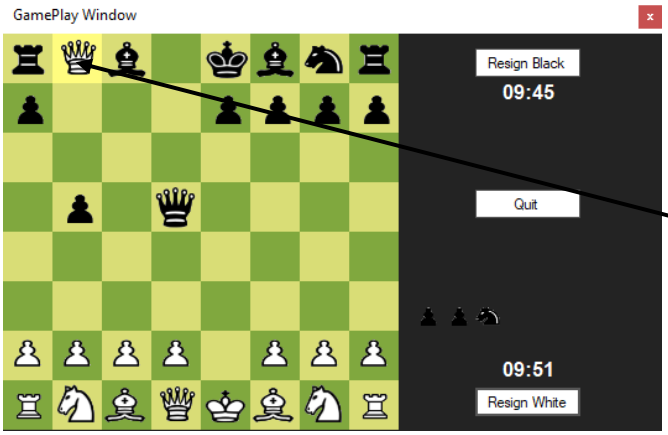
Demonstrated here is the interface which displays the pieces that have been taken. Black has taken a white piece and so the white piece is moved to the interface. The timers are also displayed.

Feature ID - 4.0 Special move (castling)



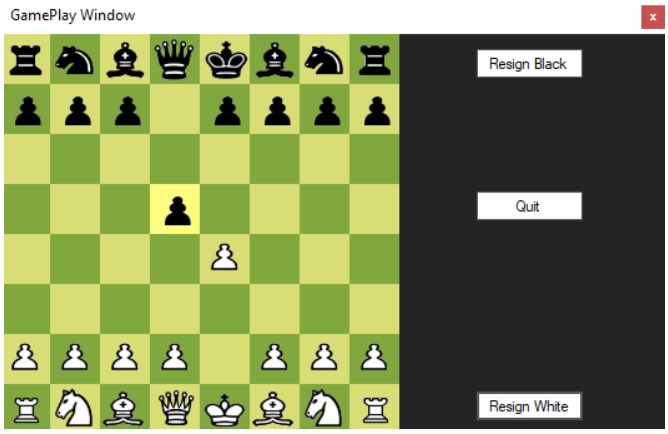
Here is castling. The king has not moved and the space between the king and castle is clear. The king has the option to castle and so does. The king moves 2 towards the castle and the castle moves to the other side of the king.

Feature ID – 4.0 Special move (Queening)



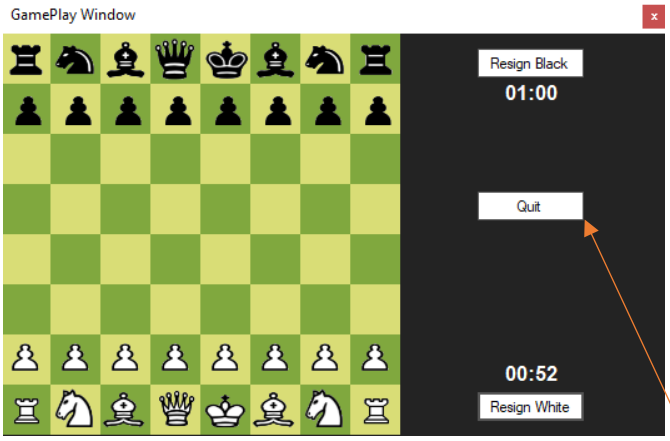
Here is an example of queeing. A pawn has reached the end of the board and so has become a queen.

Feature ID – 3.3 Suicide chess



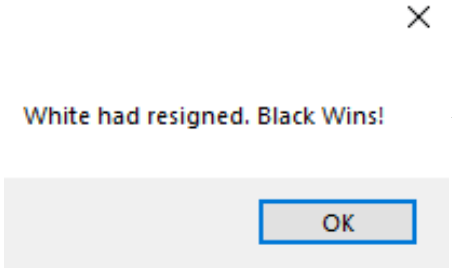
Here is suicide chess. Pieces moved normally up to this point but now white's pawn can capture black's this is the only move available.

Feature ID – 3.2 Bullet chess



This is bullet chess. The timer is originally set to a minute and white move first.

Usability ID – 2.2 Resign and Quit



If white presses the resign button this message box is shown. The same for black but white and black are swapped. If pressed the game is stopped. The quit button takes you back to the main menu. If the quit button is clicked, then the form disappears, and the main menu is shown.

Feedback

This iteration has been a success. I seem to have met all iteration requirements, my stakeholders have commented that the new design is a big improvement on the first iteration. There could be some minor improvements to this iteration. My stakeholders suggested that I add an option to restart the game once ended rather than having to quit and go back in. Perhaps this will be something I think about if I finish iteration 3 earlier than expected, however I do not feel it is completely necessary. It was also suggested to me that I make it more visible as to whose turn it is. I will think about how I might do that and try to add a way in iteration 3.

Testing

Post-development and Usability Testing

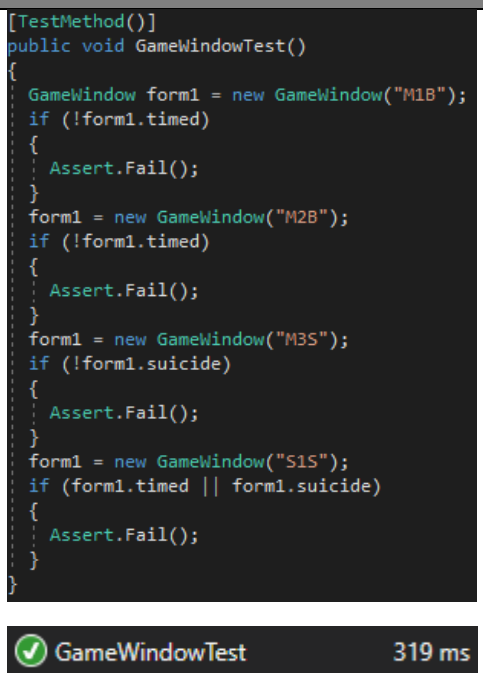
Commented evidence within iteration feedback section.

	Test	Result	Comments	Pass/Fail
U3.1	Test if rules for each mode are available for display	Rules displayed are were helpful to users according to stakeholders	Evidenced above	Pass

U3.4	Test to see if unexperienced players and colour-blind players could play	Unexperienced players found it easy to use	Colour-blind users struggled to see where the possible moves were on board	Fail

Testing during development

In addition to the tests shown, all unit tests of iteration 1 were minorly altered to fit with the edits to the code made in this iteration and all tests still passed.

Test ID	Class being tested	Test Aim	Result	Proof
2DD1.0	GameWindow	Make sure the game modes are passed in correctly and causing the correct changed	Pass	
2DD1.1		Make sure the board is the correct colour	Pass	

```
[TestMethod()]
public void GameWindowTest()
{
    GameWindow form1 = new GameWindow("M1B");
    if (!form1.timed)
    {
        Assert.Fail();
    }
    form1 = new GameWindow("M2B");
    if (!form1.timed)
    {
        Assert.Fail();
    }
    form1 = new GameWindow("M3S");
    if (!form1.suicide)
    {
        Assert.Fail();
    }
    form1 = new GameWindow("S1S");
    if (form1.timed || form1.suicide)
    {
        Assert.Fail();
    }
}
```

GameWindowTest 319 ms

```
[TestMethod()]
public void GameWindowTest2()
{
    GameWindow form = new GameWindow("M1B");
    foreach (Panel p in form.generation.getPanels())
    {
        if (!(p.BackColor ==
            ColorTranslator.FromHtml("#80A83E") ||
            p.BackColor ==
            ColorTranslator.FromHtml("#D9DD76")))
        {
            Assert.Fail();
        }
    }
}
```

GameWindowTest2 33 ms

Addressing the failed test

All that was required to fix this issue was the change the colour of the panels of the moves displayed to something more obvious that stands out. I thought an orange colour would stand out quite well whilst also keeping the design nice. I changed the colour of these panels from #FFFC7F to #FC976F which stood out a lot better. My stakeholders unanimously agreed that this was a much easier colour to see against the background.

Iteration 2 Stakeholder Sign-off

After development of this iteration was complete I asked my stakeholders to test the game against the iteration requirements. After thorough testing they all agreed that the iteration requirements have been met and that any further changes to the design could be done in iteration 3 as they were non-essential. They have all agreed for me to move on to the next iteration, where I will develop the single player game mode.

Iteration 3

As stated in the previous iteration, this iteration will focus on implementing the single player mode using artificial intelligence.

Planning

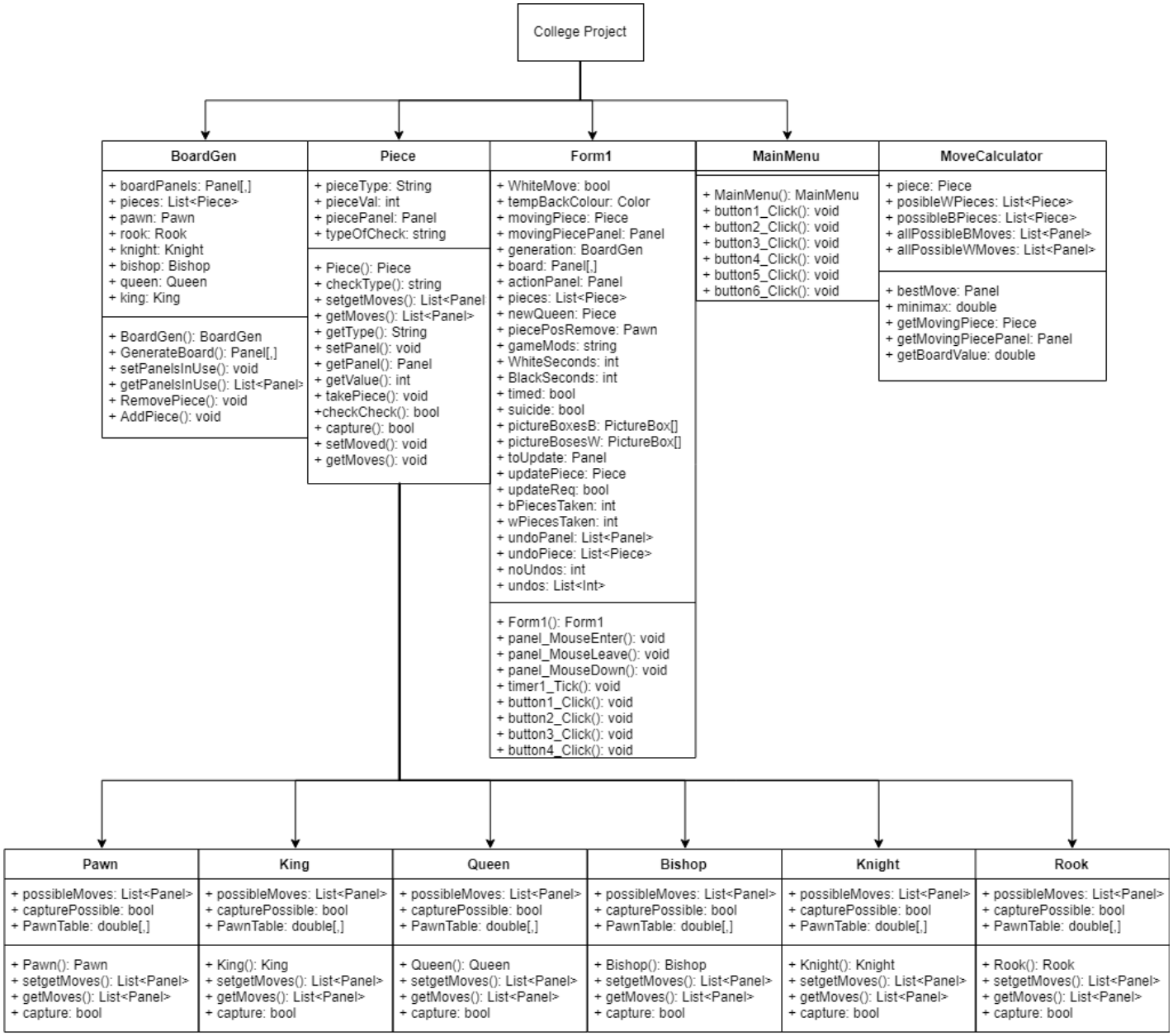
Research into AI used for chess

After deciding to use artificial intelligence in my program I started to research what kind of AI was typically used in a chess engine. After doing some research I concluded that most, if not all, good chess engines have a minimax algorithm running at the core. The main difference between today's chess AIs is how they evaluate and how they search. The first computer to beat the world's best human was Deep Blue that was developed by IBM. Deep Blue beat Garry Kasparov for the first time in 1997 3 games to 2. Deep Blue used a minimax algorithm combined with alpha-beta pruning and parallel processing and was able to evaluate 200,000,000 positions per second, getting to a search depth between 6 and 12 on average. This processing power was combined with a very hefty set of positions that were hardcoded, the evaluation function was split into 8,000 parts most of which was designed for specific positions. In the opening book there were more than 4,000 positions coded in and an endgame database that contained many 5 and 6 piece endgames. More recently chess AI's have become more efficient, in 2006 a game between an algorithm called Deep Fritz and Vladimir Kramnik the program ran on a personal computer, capable of evaluating 8,000,000 positions per second yet getting to a search depth of 17 or 18. Most recently AlphaZero was developed by DeepMind to beat the previous best chess AI stockfish. AlphaZero was given no knowledge of chess other than the basic rules (e.g. how to move). AlphaZero taught itself to play in 4 hours and became good enough to beat stockfish in that time. At its core AlphaZero also uses a however a Neural Network for the evaluation function (which is not uncommon is top chess computers now) and instead of using the normal search function of a minimax algorithm, AlphaZero uses a Monte Carlo Tree Search. This is a heuristic search algorithm, it analyses the most promising moves, expanding the search tree based on random sampling of the search space. The final result of the game is the n used to weight nodes on the tree so that better nodes are more likely to be picked in future games.

Design of the AI

For my AI I decided to try to stick with simplicity, making the AI easier to play against and making it more feasible to program with the time available. That is why I decided to use a Minimax algorithm as the core, alpha-beta pruning to make the efficient good enough so that it doesn't take too long to make moves and piece-square tables to consider for position as well as just pieces in the evaluation of a board position.

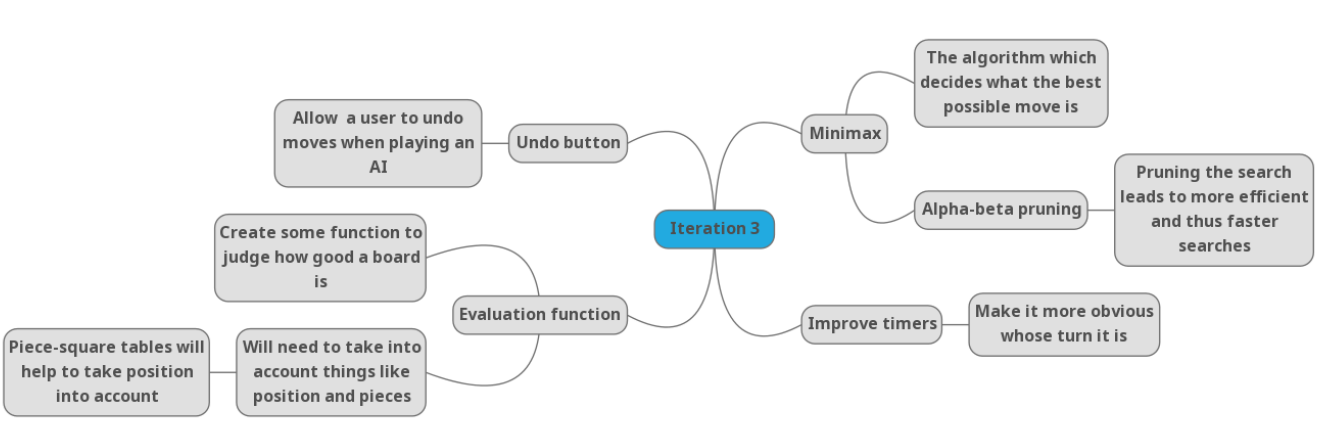
UML Diagram


College Project
BoardGen
+ boardPanels: Panel[,]
+ pieces: List<Piece>
+ pawn: Pawn
+ rook: Rook
+ knight: Knight
+ bishop: Bishop
+ queen: Queen
+ king: King
+ BoardGen(): BoardGen
+ GenerateBoard(): Panel[,]
+ setPanelsInUse(): void
+ getPanelsInUse(): List<Panel>
+ RemovePiece(): void
+ AddPiece(): void
Piece
+ pieceType: String
+ pieceVal: int
+ piecePanel: Panel
+ typeOfCheck: string
+ Piece(): Piece
+ checkType(): string
+ setgetMoves(): List<Panel
+ getMoves(): List<Panel>
+ getType(): String
+ setPanel(): void
+ getPanel(): Panel
+ getValue(): int
+ takePiece(): void
+checkCheck(): bool
+ capture(): bool
+ setMoved(): void
+ getMoves(): void
Form1
+ WhiteMove: bool
+ tempBackColour: Color
+ movingPiece: Piece
+ movingPiecePanel: Panel
+ generation: BoardGen
+ board: Panel[,]
+ actionPanel: Panel
+ pieces: List<Piece>
+ newQueen: Piece
+ piecePosRemove: Pawn
+ gameMods: string
+ WhiteSeconds: int
+ BlackSeconds: int
+ timed: bool
+ suicide: bool
+ pictureBoxesB: PictureBox[]
+ pictureBosesW: PictureBox[]
+ toUpdate: Panel
+ updatePiece: Piece
+ updateReq: bool
+ bPiecesTaken: int
+ wPiecesTaken: int
+ undoPanel: List<Panel>
+ undoPiece: List<Piece>
+ noUndos: int
+ undos: List<Int>
+ Form1(): Form1
+ panel_MouseEnter(): void
+ panel_MouseLeave(): void
+ panel_MouseDown(): void
+ timer1_Tick(): void
+ button1_Click(): void
+ button2_Click(): void
+ button3_Click(): void
+ button4_Click(): void
MainMenu
+ MainMenu(): MainMenu
+ button1_Click(): void
+ button2_Click(): void
+ button3_Click(): void
+ button4_Click(): void
+ button5_Click(): void
+ button6_Click(): void
MoveCalculator
+ piece: Piece
+ posibleWPieces: List<Piece>
+ possibleBPieces: List<Piece>
+ allPossibleBMoves: List<Panel>
+ allPossibleWMoves: List<Panel>
+ bestMove: Panel
+ minimax: double
+ getMovingPiece: Piece
+ getMovingPiecePanel: Panel
+ getBoardValue: double
Pawn
+ possibleMoves: List<Panel>
+ capturePossible: bool
+ PawnTable: double[,]
+ Pawn(): Pawn
+ setgetMoves(): List<Panel>
+ getMoves(): List<Panel>
+ capture: bool
King
+ possibleMoves: List<Panel>
+ capturePossible: bool
+ PawnTable: double[,]
+ King(): King
+ setgetMoves(): List<Panel>
+ getMoves(): List<Panel>
+ capture: bool
Queen
+ possibleMoves: List<Panel>
+ capturePossible: bool
+ PawnTable: double[,]
+ Queen(): Queen
+ setgetMoves(): List<Panel>
+ getMoves(): List<Panel>
+ capture: bool
Bishop
+ possibleMoves: List<Panel>
+ capturePossible: bool
+ PawnTable: double[,]
+ Bishop(): Bishop
+ setgetMoves(): List<Panel>
+ getMoves(): List<Panel>
+ capture: bool
Knight
+ possibleMoves: List<Panel>
+ capturePossible: bool
+ PawnTable: double[,]
+ Knight(): Knight
+ setgetMoves(): List<Panel>
+ getMoves(): List<Panel>
+ capture: bool
Rook
+ possibleMoves: List<Panel>
+ capturePossible: bool
+ PawnTable: double[,]
+ Rook(): Rook
+ setgetMoves(): List<Panel>
+ getMoves(): List<Panel>
+ capture: bool

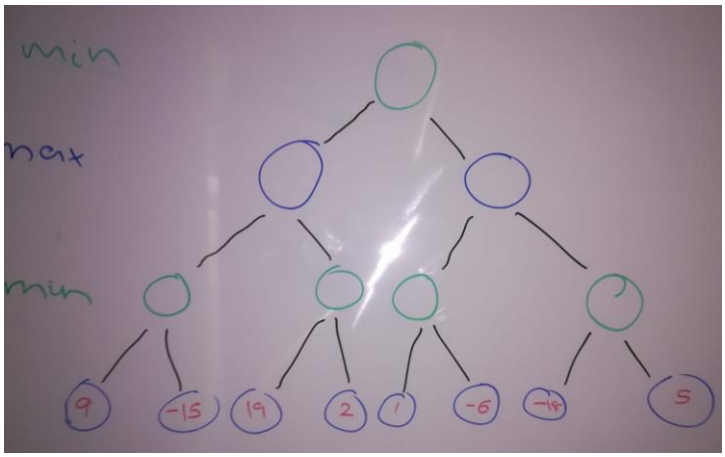
Iteration Requirements

ID	Description of Component
1.0	Implement a minimax algorithm.
2.0	Implement Alpha-Beta pruning to make the minimax algorithm more efficient.
3.0	Implement an evaluation function to judge how good a board position is.
4.0	Add piece-square tables.
5.0	Implement such that the AI makes moves during a single player game.
6.0	Implement an undo button
7.0 (OPTIONAL)	Make whose turn it is more obvious during timed games

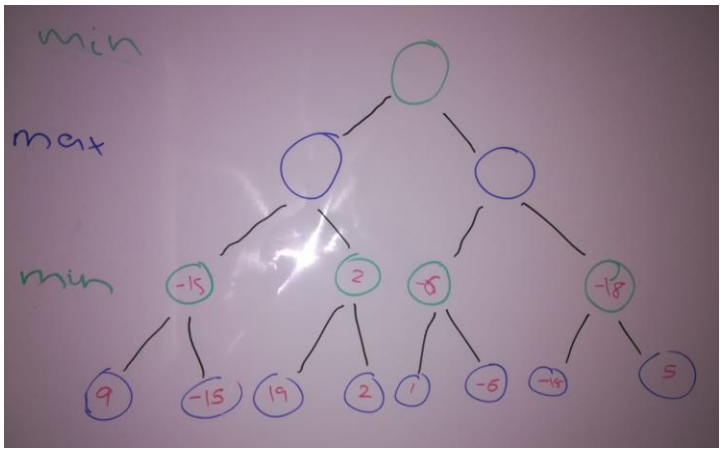
Mindmap



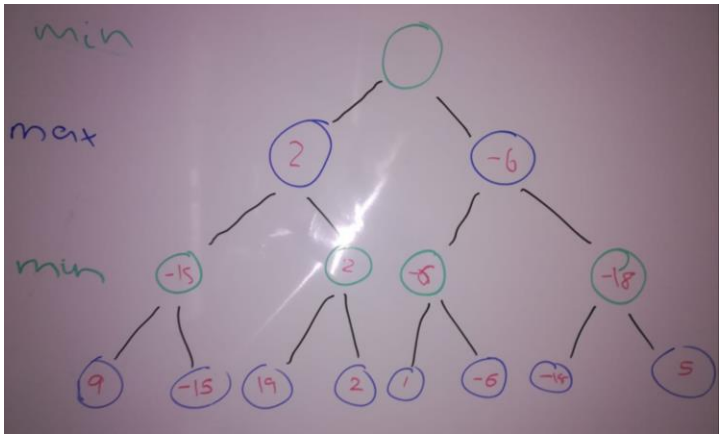
Iteration requirement 1.0 - The Minimax Algorithm



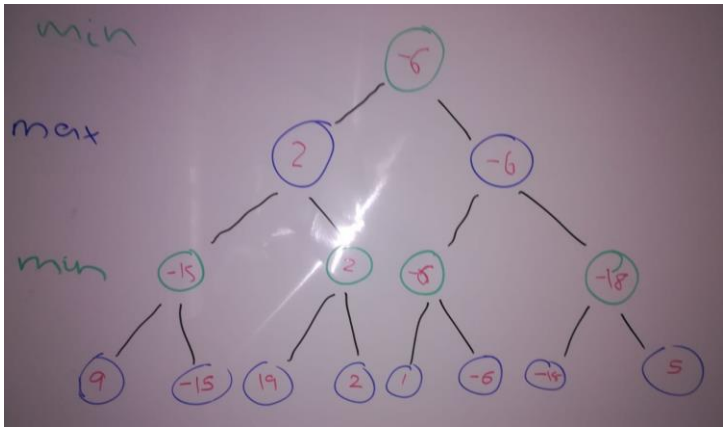
1. We start with a tree. In this example I am using a branching factor of 2 to make this easier to representing. In chess the branching factor is much higher. Each node represents a board position. The first one is the possible move we are looking at, the nodes coming out of the previous node represent a move that could be done in response to the move represented by the node above. This represents playing as black as the evaluation function is done such that a smaller value (or larger negative value) is better for black. The values at the bottom represent that value of the board if this line of moves is done.



2. The leaf node is a white move, so black must choose a move it would do to make white's move have as little impact on the score (or even make the value smaller or more negative) as possible. So black chooses it's moves it could do, making the board value as small or negative as possible as this is black's aim.



3. It is now white's opportunity to prevent black from making the board too negative or small. White will choose a move that leads to the board value being as big as possible as this indicates white is doing better.

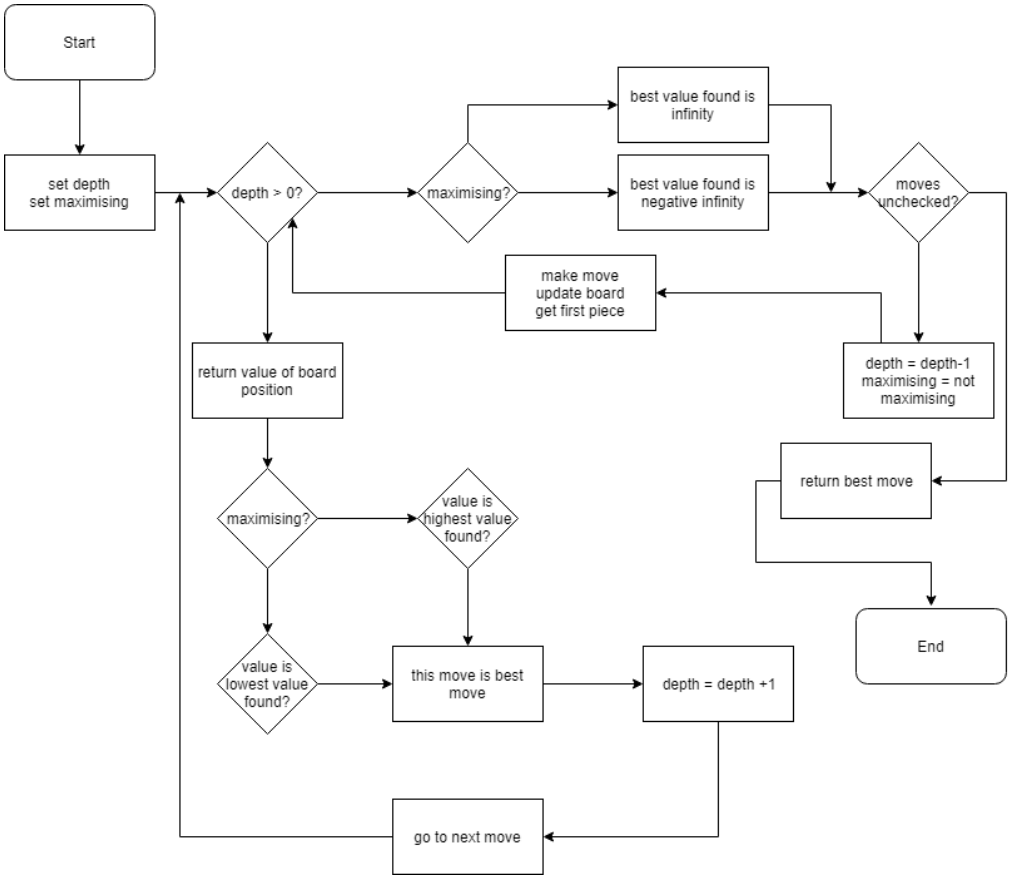


Black now knows all of the possible moves in response to it's move, all the possible moves in response to that and all the possible moves in response to that. Assuming both players have made their best possible moves white now has the board value if black chooses this move. It chooses the value that is going to lead to the smallest or most negative board value.

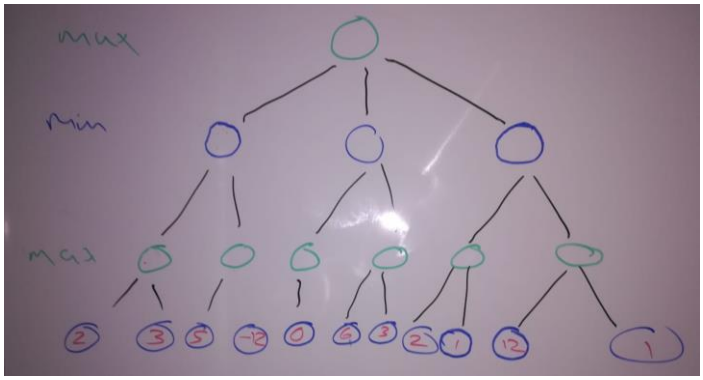
Minimax Pseudocode

```
minimax(int depth, bool maxingPlayer)
        IF(depth == 0) THEN
                RETURN boardValue
        END IF
        IF(maxingPlayer) THEN
                double bestVal = -10000
                FOR i = 0 TO numOfPossibleMoves
                        piece[i].moveTo(possibleMoves[i]
                        tempVal = minimax(!max, depth – 1)
                        IF tempVal > bestVal THEN
                                bestVal = tempVal
                        END IF
                NEXT
        END IF
        IF(!maxingPlayer) THEN
                double bestVal = 10000
                FOR i = 0 TO numOfPossibleMoves
                        piece[i].moveTo(possibleMoves[i]
                        tempVal = minimax(!max, depth – 1)
                        IF tempVal < bestVal THEN
                                bestVal = tempVal
                        END IF
                NEXT
        END IF
        RETURN bestVal
```

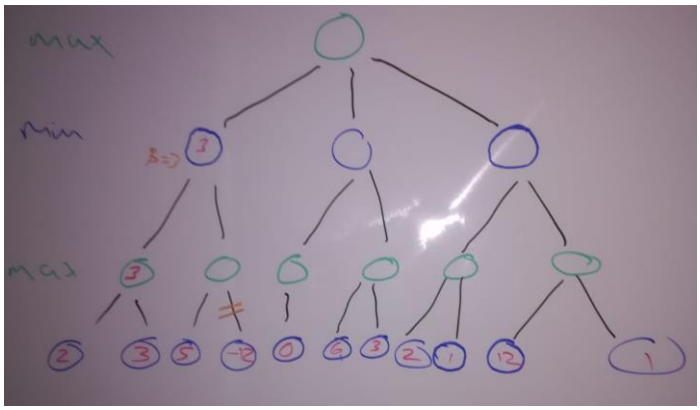
END


Start
set depth
set maximising
depth > 0?
maximising?
best value found is infinity
best value found is negative infinity
moves unchecked?
make move
update board
get first piece
depth = depth-1
maximising = not maximising
return value of board position
maximising?
value is highest value found?
value is lowest value found?
this move is best move
depth = depth +1
go to next move
return best move
End

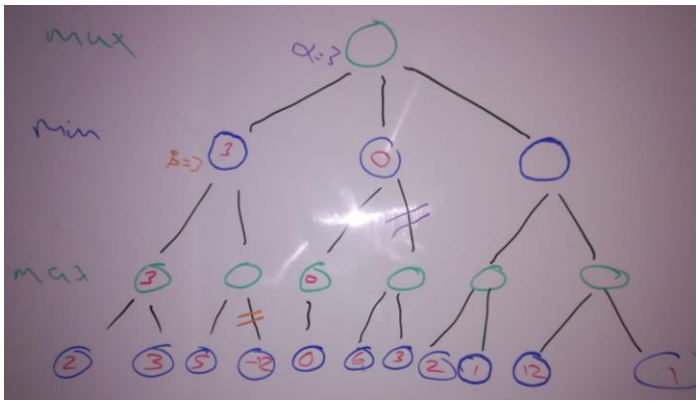
Iteration requirement 2.0 - Alpha-Beta Pruning



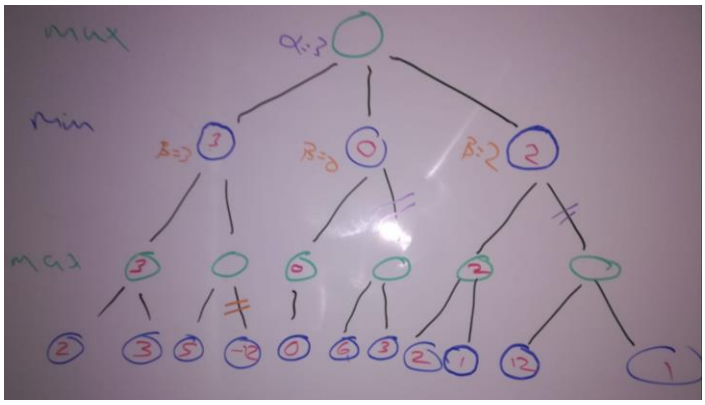
This is setup the same as minimax, except the AI is now playing as white and the branch factor is variable. These values were chosen to best represent alpha-beta pruning.



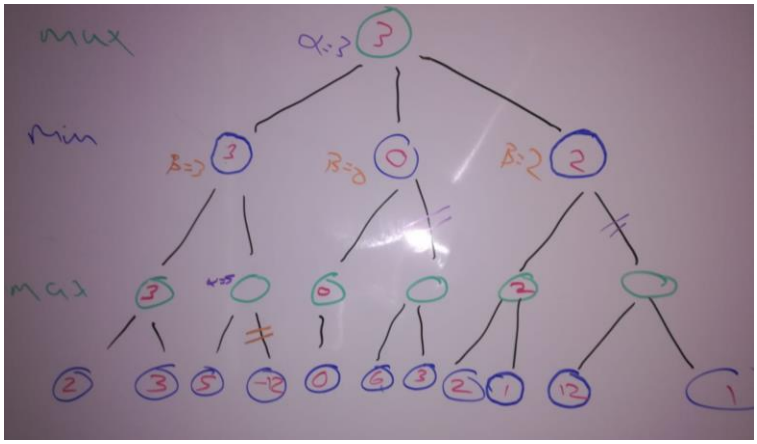
Max chooses the 3 as it is bigger. Beta at the node above is set to 3, it then looks ahead to the 5, beta is smaller than 5, and as the max function will choose the largest value then that mac function will either choose 5 or greater than 5 for that node, therefore it doesn't have to look at the other node as 5 is greater than beta, so the min function will choose beta.



Alpha for the node above is now set to 3. Looking down there is only one path after min, so Beta for that min gets set to 0, as we're on the min function, it will either choose 0 or less than 0, as alpha is greater than 0, the max function is currently going to choose alpha, so no more of this subtree needs to be looked at.



For the next subtree, as we go down the max function chooses 2, setting the value of beta to 2, as alpha is greater than 2, we don't have to look any further on this subtree.



Max then chooses the maximum value of the next nodes as it's value (which is also the value of alpha).

Alpha-Beta Pseudocode Algorithm

```
minimaxAlphaBeta(int depth, bool maxingPlayer, alpha, beta)
        IF(depth == 0) THEN
                RETURN boardValue
        END IF
        IF(maxingPlayer) THEN
                double bestVal = -10000
                FOR i = 0 TO numOfPossibleMoves
                        piece[i].moveTo(possibleMoves[i]
                        tempVal = minimaxAlphaBeta(!max, depth – 1, alpha, beta)
                        IF tempVal > bestVal THEN
                                bestVal = tempVal
                                alpha = tempVal
                        END  IF
                        IF alpha>=beta THEN
                                return bestVal
                        END IF
                NEXT
        END IF
        IF(!maxingPlayer) THEN
                double bestVal = 10000
                FOR i = 0 TO numOfPossibleMoves
                        piece[i].moveTo(possibleMoves[i]
                        tempVal = minimaxAlphaBeta(!max, depth – 1, alpha, beta)
                        IF tempVal < bestVal THEN
                                bestVal = tempVal
                                beta = tempVal
                        END IF
                        IF alpha>= beta THEN
                                return bestVal
                        END IF
                NEXT
        END IF
        RETURN bestVal
END
```

Iteration requirement 3.0 - The Evaluation Function

Psuedocode Algorithm

```
getBoardValue()
	boardValue = 0;
	FOR EACH Piece p IN board.getPieces()
		IF p.getType().Substring(0, 1) == "B" THEN
			boardValue = boardValue - p.getValue(board);
		ELSE
			boardValue = boardValue + p.getValue(board);

		END IF
	NEXT
	RETURN boardValue;
END FUNCTION
```

Iteration requirement 4.0 – Piece-square Tables

For this I found some tables typically used in AI and will make some edits to the Piece class and child classes to account for the new way of getting the value.

Iteration requirement 5.0 – The AI Makes Moves in Game

Psuedocode Algorithm

```
IF gameMode == Single Player THEN
	move = calculateBestMove
	pieceformove.setPanel(move)
END IF
```

Iteration requirement 6.0 – Undo

Psuedocode Algorithm

```
ON undobutton clicked()
	IF gameMode == Single Player THEN
		movedPiece = board.getMovedPiece()
		movedPiece.setPanel(movedPiece.previousPanel())
END PROCEDURE
```

Iteration requirement 7.0 – Turns

After thinking about how I would go about this, I thought the best way would be to make the time of whose go it is stand out more. So, all that needs to be done is set the person's whose go it is timer to a more bold colour, to show whose go it is.

Testing Plan

During development testing will need to take place, to see if the AI makes a move, see whether the undo button function, see if minimax returns the correct value for a move, see if getBoardValue returns the correct value for a given board, to see if the getMovingPiece returns the correct piece and to see if bestMove returns the best move for a position.

Testing during development should show what data was inputted, the intended result, the actual result the program gave, and whether this was a match or not. This testing should also include some explanation for why it may not be working as intended, with any console output if there is a crash and any relevant variables with a content analysis.

Post-development testing will be necessary to ensure that the changes made to the program meet the usability requirements stated from the start of the project.

Usability Testing Plan

	Test	Result	Comments
U3.3	Test to see how the different levels of AI behave		
U3.4	Test to see how experienced players find playing the AI and how beginners find playing against it		
U3.5	Test to see if the code is error tolerant		

During Development Testing Plan

Test ID	Class being tested	Test Aim	Expected Result	Proof
DD1.0	GameWindow	Test whether the AI has made a move	Pass	
DD1.1	MoveCalculator	Test to see if undo method functions	Pass	
		Test to see if minimax returns a correct board value	Pass	
		Test to see if getBoardValue returns the correct value for a given board	Pass	
		Test to see if getMovingPiece returns the correct piece	Pass	
		Test to see if bestMove returns the best move for a given board	Pass	

	Test	Data	Expected Result
F2.2	Undo button	Start the game, enter the single player game mode, move a piece and then press the undo button.	The position will revert to what it was before the move.
F2.1	The AI makes moves	Start the game and enter single player mode. Make a move and see if the AI makes a move back.	The AI will make a move.
F2.1	The AI makes an objectively good move	Start the game and enter single player mode then make a move	The AI will make a good move.

Iteration 3 code

BoardGen

```
namespace CollegeProject
{
  public class BoardGen
  {
    Panel[,] boardPanels;
    List<Piece> pieces = new List<Piece>();
    List<Panel> inUse;
    Pawn pawn;
    Rook rook;
    Knight knight;
    Bishop bishop;
    Queen queen;
    King king;
    List<Piece> possibleBPieces;
    List<Piece> possibleWPieces;

    public BoardGen()
    {

    }


    public Panel[,] GenerateBoard()
    {

      int tileSize = 40;
      int gridSize = 8;
      //Size of squares
      Color color1 = System.Drawing.ColorTranslator.FromHtml("#80A83E"); //#6
      Color color2 = System.Drawing.ColorTranslator.FromHtml("#D9DD76"); //#c
```

```
                                                                    //Colour of
sqaures

      boardPanels = new Panel[gridSize, gridSize];

      //Nested for loop to get the 8 by 8 grid set up
      for (int i = 0; i < gridSize; i++)
      {
        for (int x = 0; x < gridSize; x++)
        {

          //Set the panel details
          Panel newPanel = new Panel
          {
            Size = new Size(tileSize, tileSize),
            Location = new Point(tileSize * i, tileSize * x),
            BackgroundImageLayout = ImageLayout.Stretch,
          };

          String peiceType = null;

          //Set what colour the piece should be
          peiceType = (tileSize * x <= 40) ? "B" : "W";

          //Setting what type of pieces go where and geneerating the pieces
          if (tileSize * x == 40 || tileSize * x == 240)
          {
            peiceType += "Pawn";
            pawn = new Pawn(peiceType, newPanel, false);
            pieces.Add(pawn);
          }
          else if ((tileSize * i == 0 || tileSize * i == 280) &&
                  (tileSize * x == 0 || tileSize * x == 280))
          {
            peiceType += "Rook";
            rook = new Rook(peiceType, newPanel, false);
            pieces.Add(rook);
          }
          else if ((tileSize * i == 40 || tileSize * i == 240) &&
                  (tileSize * x == 0 || tileSize * x == 280))
          {
            peiceType += "Knight";
            knight = new Knight(peiceType, newPanel, false);
            pieces.Add(knight);
          }
          else if ((tileSize * i == 80 || tileSize * i == 200) &&
                  (tileSize * x == 0 || tileSize * x == 280))
          {
            peiceType += "Bishop";
            bishop = new Bishop(peiceType, newPanel, false);
            pieces.Add(bishop);
          }
          else if ((tileSize * i == 160) &&
                  (tileSize * x == 0 || tileSize * x == 280))
          {
            peiceType += "King";
            king = new King(peiceType, newPanel, false);
            pieces.Add(king);
          }
          else if ((tileSize * i == 120) &&
                  (tileSize * x == 0 || tileSize * x == 280))
          {
            peiceType += "Queen";
            queen = new Queen(peiceType, newPanel, false);
            pieces.Add(queen);
```

```
            }

            boardPanels[i, x] = newPanel;

            if (i % 2 == 0)
            {
              newPanel.BackColor = x % 2 != 0 ? color1 : color2;
            }
            else
            {
              newPanel.BackColor = x % 2 != 0 ? color2 : color1;
            }
          }
        }

        //update what panels are currently taken
        setPanelsInUse();
        //return the board that has been generated
        return boardPanels;
      }

      public Panel[,] getPanels()
      {
        return boardPanels;
      }

      public List<Piece> getPieces()
      {
        return pieces;
      }

      public void setPanelsInUse()
      {
        inUse = new List<Panel>();
        foreach (Panel x in boardPanels)
        {
          foreach (Piece p in pieces)
          {
            if (x == p.getPanel())
            {
              inUse.Add(x);
            }
          }
        }
        //Sets the panels that have a piece on them
      }

      public List<Panel> getPanelsInUse()
      {
        return inUse;
      }

      public void RemovePiece(Piece piece)
      {
        pieces.Remove(piece);
        inUse.Remove(piece.getPanel());
        //Removes a piece from the board
      }

      public void AddPiece(Piece piece)
      {
        pieces.Add(piece);
        inUse.Add(piece.getPanel());
        //Adds a piece to the board
      }
```

```
    public List<Panel> getAllPossibleBMoves(BoardGen board)
    {
      List<Panel> allPossibleBMoves = new List<Panel>();
      possibleBPieces = new List<Piece>();
      foreach (Piece x in getPieces().ToList())
      {
        x.setMoves(board, false, false);
        if (x.getType().Substring(0, 1) == "B")
        {
          foreach (Panel p in x.getMoves())
          {
            allPossibleBMoves.Add(p);
            possibleBPieces.Add(x);
          }
        }
      }
      //gets all the possible moves for all possible black pieces
      return allPossibleBMoves;
    }

    public List<Panel> getAllPossibleWMoves(BoardGen board)
    {
      List<Panel> allPossibleWMoves = new List<Panel>();
      possibleWPieces = new List<Piece>();
      foreach (Piece x in getPieces().ToList())
      {
        x.setMoves(board, false, false);
        if (x.getType().Substring(0, 1) == "W")
        {
          foreach (Panel p in x.getMoves())
          {
            allPossibleWMoves.Add(p);
            possibleWPieces.Add(x);
          }
        }
      }
      //gets all the possible moves for all possible white pieces
      return allPossibleWMoves;
    }

    public List<Piece> getPossibleBPieces()
    {
      return possibleBPieces;
    }

    public List<Piece> getPossibleWPieces()
    {
      return possibleWPieces;
    }

  }
}
```

Piece

```
namespace CollegeProject
{
  public abstract class Piece
  {
    String pieceType;
    Panel piecePanel;
    string typeOfCheck;
    bool hasMoved;
    int noMoves;
```

```
public Piece(String type, Panel image, bool moved)
{

  image.BackgroundImage = (Image)
      (Properties.Resources.ResourceManager.GetObject(type));

  pieceType = type;
  piecePanel = image;
  hasMoved = moved;
  noMoves = 0;
  //Sets original values of the piece
}

public string checkType()
{
  return typeOfCheck;
}

public abstract void setMoves(BoardGen board, bool checkUp, bool suicide);
//To be overriden by child classes

public abstract List<Panel> getMoves();
//To be overriden by child classes

public abstract bool capture();

public String getType()
{
  return pieceType;
}

public void setPanel(Panel newPanel)
{
  piecePanel = newPanel;
}

public Panel getPanel()
{
  return piecePanel;
}

public abstract double getValue(BoardGen board);

public bool checkCheck(BoardGen board)
{
  bool check = false;

  Piece piece = null;

  //Need to check all pieces on board because discovered check exists
  foreach (Piece y in board.getPieces())
  {

    List<Panel> possibleMoves = new List<Panel>();
    possibleMoves = y.getMoves();
    if (y.getMoves() != null)
    {
      foreach (Panel x in possibleMoves)
      {

        foreach (Piece c in board.getPieces())
        {
          if (c.getPanel() == x)
          {
```

```
                piece = c;
                if (piece.getType().Contains("King") &&
                   (y.getType().Substring(0, 1) !=
                   piece.getType().Substring(0, 1)))
                {
                  typeOfCheck = (piece.getType().Substring(0, 1)
                      == "W") ? "W" : "B";
                  return true;
                }
              }
            }
          }
        }
        //Checks if any piece has a possible move that contains "King",
        //if it does it is check

      }

      return check;

    }

    public void setMoved(bool move)
    {
      hasMoved = move;
    }

    public bool getMoved()
    {
      return hasMoved;
    }

    public int numberMoves()
    {
      return noMoves;
    }

    public void increaseNoMoves()
    {
      noMoves++;
    }

    public void decreaseNoMoves()
    {
      noMoves--;
    }
  }
}
```

GameWindow

```
private void button4_Click(object sender, EventArgs e)
    {
      bool taken = false;

      try
      {
        for (int i = 0; i < undos[undos.Count - 1]; i++)
        {
          movingPiece = undoPiece[undoPiece.Count - (i + 1)];
          if (!generation.getPieces().Contains(movingPiece))
          {
            movingPiece.getPanel().BackgroundImage =
```

```
                                                movingPiece.getType().Contains("W") ?
                                    pictureBoxesB[bPiecesTaken  - 1].BackgroundImage :
                                    pictureBoxesW[wPiecesTaken  - 1].BackgroundImage;
            generation.AddPiece(movingPiece);
            if (movingPiece.getType().Contains("W"))
            {
              pictureBoxesB[bPiecesTaken - 1].BackgroundImage = null;
              bPiecesTaken --;
            }
            else
            {
              pictureBoxesB[wPiecesTaken  - 1].BackgroundImage = null;
              wPiecesTaken --;
            }
            //If there was a piece taken before the undo, remove it from the GUI
            taken = true;
          }
          if (movingPiece == updatePiece)
          {
            piecePosRemove.getPanel().BackgroundImage = null;
            piecePosRemove.setPanel(null);
            generation.RemovePiece(piecePosRemove);
            toUpdate = null;
            updatePiece = null;
            movingPiece.getPanel().BackgroundImage =
                              movingPiece.getType().Contains("W") ? Resources.WPawn :
                                                                    Resources.BPawn;
            //Prepare the sqaure the piece needs to return to if necesary
          }

          movingPiecePanel = undoPiece[undoPiece.Count - (i + 1)].getPanel();
          actionPanel = undoPanel[undoPanel.Count - (i + 1)];
          actionPanel.BackgroundImage = movingPiece.getPanel().BackgroundImage;

          movingPiece.setPanel(actionPanel);
          generation.setPanelsInUse();
          movingPiecePanel.BackgroundImage = null;
          movingPiece.decreaseNoMoves();
          if (movingPiece.numberMoves() <= 0)
          {
            movingPiece.setMoved(false);
          }
          //return everything to the way it was before the move occured
        }
        for (int x = 0; x < undos[undos.Count - 1]; x++)
        {
          undoPiece.Remove(undoPiece[undoPiece.Count - 1]);
          undoPanel.Remove(undoPanel[undoPanel.Count - 1]);
        }

        undos.Remove(undos[undos.Count - 1]);

        if (updateReq || taken)
        {
          panel_MouseDown(null, new MouseEventArgs(MouseButtons.Left, 1, 0, 0, 0));
          updateReq = false;
        }
        //update the undos so that it wont be the same undo over and over again
      }
      catch (System.ArgumentOutOfRangeException)
      {
        MessageBox.Show("You can't go back any further.");
```

```
        //if there aren't any undos in the list, you can't go back more
      }
      noUndos = 0;
    }
}
```

Bishop

```
namespace CollegeProject
{
  public class Bishop : Piece
  {

    List<Panel> possibleMoves;
    bool capturePossible = false;
    public double[,] BishopTable;

    public Bishop(string type, Panel image, bool moved) : base(type, image, moved)
    {

      BishopTable = new double[8, 8]
          { { -2.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -2.0 },
                  { -1.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0, -1.0 },
                  { -1.0,  0.0,  0.5,  1.0,  1.0,  0.5,  0.0, -1.0 },
                  { -1.0,  0.5,  0.5,  1.0,  1.0,  0.5,  0.5, -1.0 },
                  { -1.0,  0.0,  1.0,  1.0,  1.0,  1.0,  0.0, -1.0 },
                  { -1.0,  1.0,  1.0,  1.0,  1.0,  1.0,  1.0, -1.0 },
                  { -1.0,  0.5,  0.0,  0.0,  0.0,  0.0,  0.5, -1.0 },
                  { -2.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -2.0 }
          };
    }

    public override double getValue(BoardGen board)
    {
      for (int x = 0; x < 8; x++)
      {
        for (int y = 0; y < 8; y++)
        {
          if (board.getPanels()[x, y] == getPanel())
          {
            if (board.getPanels()[x, y] == getPanel())
            {
              double val = (getType().Substring(0, 1) == "W") ? BishopTable[x, y] :
                                                                BishopTable[y, x];

              return val + 30;
              //Adds 30 (a Bishop's traditional value) to the value of the Bishop's
                position.
            }
          }
        }
      }
      return 0;
    }

    public override void setMoves(BoardGen board, bool checkUp, bool suicide)
    {
      possibleMoves = new List<Panel>();
      Piece piece = null;
      Panel tempPan = null;
      capturePossible = false;

      //A bishop may move along the diagnol it is on
      foreach (Panel x in board.getPanels())
```

```
{
  for (int i = 0; i < 8; i++)
  {
    foreach (Piece c in board.getPieces())
    {
      if (c.getPanel() == x)
      {
        piece = c;
      }
    }

    if ((getPanel().Location.X == x.Location.X + (40 * i) ||
        getPanel().Location.X == x.Location.X - (40 * i)) &&
        (getPanel().Location.Y == x.Location.Y + (40 * i) ||
        getPanel().Location.Y == x.Location.Y - (40 * i)))
    {
      if (board.getPanelsInUse().Contains(x))
      {

        foreach (Piece y in board.getPieces())
        {
          if (y.getPanel() == x)
          {
            if (getType().Substring(0, 1) !=
                y.getType().Substring(0, 1))
            {
              possibleMoves.Add(x);

            }
          }
        }
      }
      else
      {
        possibleMoves.Add(x);
      }
    }
  }

}
/* This bit just gets all of the panels in a diagonal direction from the
bishop in use */

List<Panel> panelsRemove = new List<Panel>();

foreach (Panel x in board.getPanels())
{
  for (int i = 0; i < 8; i++)
  {
    if ((getPanel().Location.X == x.Location.X + (40 * i) ||
        getPanel().Location.X == x.Location.X - (40 * i)) &&
        (getPanel().Location.Y == x.Location.Y + (40 * i) ||
        getPanel().Location.Y == x.Location.Y - (40 * i)) &&
        board.getPanelsInUse().Contains(x) && x.BackgroundImage != null)
    {
      if (x.Location.Y < getPanel().Location.Y &&
          x.Location.X < getPanel().Location.X)
      {
        foreach (Panel g in possibleMoves)
        {
          if (g.Location.Y < x.Location.Y && g.Location.X <
              x.Location.X)
          {
            panelsRemove.Add(g);
          }
```

```
            }
          }
          else if (x.Location.Y < getPanel().Location.Y &&
              x.Location.X > getPanel().Location.X)
          {
            foreach (Panel g in possibleMoves)
            {
              if (g.Location.Y < x.Location.Y && g.Location.X >
                  x.Location.X)
              {
                panelsRemove.Add(g);
              }
            }
          }
          else if (x.Location.Y > getPanel().Location.Y &&
              x.Location.X < getPanel().Location.X)
          {
            foreach (Panel g in possibleMoves)
            {
              if (g.Location.Y > x.Location.Y && g.Location.X <
                  x.Location.X)
              {
                panelsRemove.Add(g);
              }
            }
          }
          else if (x.Location.Y > getPanel().Location.Y &&
              x.Location.X > getPanel().Location.X)
          {
            foreach (Panel g in possibleMoves)
            {
              if (g.Location.Y > x.Location.Y &&
                  g.Location.X > x.Location.X)
              {
                panelsRemove.Add(g);
              }
            }
          }
        }
      }
    }
    /* This part adds all of the panels that are blocked in some way to an
       array that removes all of those panels from possible moves (the next
       part */

    foreach (Panel x in possibleMoves)
    {
      foreach (Piece p in board.getPieces())
      {
        if (x == p.getPanel())
        {
          capturePossible = true;
        }
      }
    }

    foreach (Panel x in panelsRemove)
    {
      possibleMoves.Remove(x);
    }

    if (checkUp && !suicide) //if checking for check and this is not suicide chess
                             (as chess doesn't exist in suicide chess)
    {
      Piece toRem = null;
```

```
            bool removed = false;
            tempPan = getPanel();
            foreach (Panel x in possibleMoves)
            {
              if (board.getPanelsInUse().Contains(x))
              {
                foreach (Piece z in board.getPieces().ToList())
                {
                  if (z.getPanel() == x)
                  {
                    toRem = z;
                    removed = true;
                    board.RemovePiece(z);
                  }
                }
              }
              setPanel(x);
              board.setPanelsInUse();
              foreach (Piece y in board.getPieces().ToList())
              {
                if (!y.getType().Contains(getType()))
                {
                  y.setMoves(board, false, suicide);
                }
              }
              if (checkCheck(board) && checkType() ==
                  getType().Substring(0, 1))
              {
                panelsRemove.Add(x);
              }
              setPanel(tempPan);
              if (removed)
              {
                board.AddPiece(toRem);
                removed = false;
              }
              board.setPanelsInUse();
            }
            /* This part checks all of the current possible moves for the
               the bisop to see if any would leave the player in check, if it
               would, it is not a valid move and is removed. */
            foreach (Panel x in panelsRemove)
            {
              possibleMoves.Remove(x);
            }
          }
          if (capturePossible && suicide) //if it is suicide chess and there is a capture
                                          possible
          {
            List<Panel> newPossibleMoves = new List<Panel>();
            foreach (Panel x in possibleMoves)
            {
              foreach (Piece y in board.getPieces())
              {
                if (x == y.getPanel())
                {
                  newPossibleMoves.Add(x);
                }
              }
            }
            possibleMoves = newPossibleMoves;
            //captures are the only possible moves if a capture is possible
          }
          else if (!capturePossible && suicide) //if there no captures possible in
```

```
                                                suicide chess
      {
        foreach (Piece x in board.getPieces())
        {
          if (x.capture() && x.getType().Substring(0, 1) == getType().Substring(0,
                                                                                 1))
          {
            possibleMoves = new List<Panel>();
            break;
            //if there are no possible captures, but there are possible captures from
              other pieces, this piece cannot move
          }
        }
      }
    }
    public override List<Panel> getMoves()
    {
      return possibleMoves;
    }

    public override bool capture()
    {
      return capturePossible;
    }
  }
}
```

MainMenu

```
namespace CollegeProject
{
  public partial class MainMenu : Form
  {
    public MainMenu()
    {
      InitializeComponent();
    }
    private void button1_Click(object sender, EventArgs e)
    {
      GameWindow form = new GameWindow("S1S");
      form.Show();
      this.Hide();
      //If "Single Player" is clicked, open form1 with game code "S1S"
    }
    private void button2_Click(object sender, EventArgs e)
    {
      GameWindow form = new GameWindow("M1B");
      form.Show();
      this.Hide();
      //If "2 Player Blitz" is clicked, open form1 with game code "M1B"
    }
    private void button3_Click(object sender, EventArgs e)
    {
      GameWindow form = new GameWindow("M2B");
      form.Show();
      this.Hide();
      //If "2 Player Bullet" is clicked, open form1 with game code "M2B"
    }
    private void button4_Click(object sender, EventArgs e)
    {
      GameWindow form = new GameWindow("M3S");
      form.Show();
      this.Hide();
      //If "2 Player Suicide" is clicked, open form1 with game code "M3S"
    }
```

```
    private void button5_Click(object sender, EventArgs e)
    {
      Help form = new Help();
      form.Show();
      //If "?" is clicked, open the "help" form (form3).
    }
    private void button6_Click(object sender, EventArgs e)
    {
      Rules form = new Rules();
      form.Show();
      //If "Rules" is clicked, open the "rules" form (form4).
    }
  }
}
```

MoveCalculator

```
namespace CollegeProject
{
  public class MoveCalculator
  {
    Piece piece;
    List<Piece> possibleWPieces;
    List<Piece> possibleBPieces;
    List<Panel> allPossibleBMoves;
    List<Panel> allPossibleWMoves;

    public Panel bestMove(BoardGen board, double alpha, double beta)
    {
      Panel bestMove = null;
      double bestVal = 100000;
      bool removed = false;
      Piece removedPiece = null;
      board.setPanelsInUse();
      allPossibleBMoves = board.getAllPossibleBMoves(board);
      allPossibleWMoves = board.getAllPossibleWMoves(board);
      possibleWPieces = board.getPossibleWPieces();
      possibleBPieces = board.getPossibleBPieces();

      for (var i = 0; i < allPossibleBMoves.Count(); i++)
      {
        Panel temp = possibleBPieces[i].getPanel();
        if (board.getPanelsInUse().Contains(allPossibleBMoves[i]))
        {
          foreach (Piece y in board.getPieces().ToList())
          {
            if (y.getPanel() == allPossibleBMoves[i])
            {
              removedPiece = y;
              removed = true;
              board.RemovePiece(y);
              //take piece if we're taking a piece
            }
          }
        }
        possibleBPieces[i].setPanel(allPossibleBMoves[i]);
        board.setPanelsInUse();
        board.getAllPossibleWMoves(board);
        board.getPossibleWPieces();
        double tempVal = minimax(board, true, 1, board.getAllPossibleWMoves(board),
                                 board.getPossibleWPieces(), alpha, beta);
        //start a search on this move to see where it leads
        if (tempVal < bestVal)
        {
          bestVal = tempVal;
          beta = tempVal;
          bestMove = allPossibleBMoves[i];
          piece = possibleBPieces[i];
          //if this move leads to a good board value then tempVal will be small/more
            negative. If it is then we update bestVal, update beta and update the best
            move and the piece it moves to get there
        }
        possibleBPieces[i].setPanel(temp);
        if (removed)
        {
          removed = false;
          board.AddPiece(removedPiece);
          //return the removed piece to its position
        }
        board.setPanelsInUse();
```

```
        //update board
      }
      return bestMove;
      //returns the best move we found
    }

    public double minimax(BoardGen board, bool max, int depth, List<Panel>
                          possibleMoves, List<Piece> possiblePieces, double alpha,
                          double beta)
    {
      double bestValue;
      bool removed = false;
      Piece removedPiece = null;
      board.setPanelsInUse();

      if (depth == 0)
      {
        return getBoardValue(board);
        //We have searched as far as we need to and can return what the board value
          would be in this position
      }
      if (max) //if the player is trying to maximise score
      {
        double bestVal = -100000; //set bestVal to some very large negative number so
                                  that it easily beaten

        for (var i = 0; i < possibleMoves.Count(); i++) //for every move possible from
                                                        this position
        {
          Panel temp = possiblePieces[i].getPanel();
          if (board.getPanelsInUse().Contains(possibleMoves[i]))
          {
            foreach (Piece y in board.getPieces().ToList())
            {
              if (y.getPanel() == possibleMoves[i])
              {
                removedPiece = y;
                removed = true;
                board.RemovePiece(y);
                //take piece if we're taking a piece
              }
            }
          }
          possiblePieces[i].setPanel(possibleMoves[i]);
          board.setPanelsInUse();
          board.getAllPossibleWMoves(board);
          board.getPossibleWPieces();
          //update the board for one of the possible moves
          double tempVal = minimax(board, !max, depth - 1,
                                   board.getAllPossibleBMoves(board),
                                   board.getPossibleBPieces(), alpha, beta);
          //dive deeper into a search looking ahead by 1 furhter witht tempVal
          if (tempVal > bestVal)
          {
            bestVal = tempVal;
            alpha = tempVal;

            //if tempVal is better than the current bestVal then this is the new
              tempVal and alpha is set to tempVal
          }
          possiblePieces[i].setPanel(temp);
          if (removed)
          {
            removed = false;
            board.AddPiece(removedPiece);
```

```
            //put taken piece back
          }
          board.setPanelsInUse();

          if (alpha >= beta)
          {
            return bestVal;
            //If alpha is greater or equal to beta we can prune the search
          }

        }
        bestValue = bestVal;
      }
      else
      {
        double bestVal = 100000; //We're minimising so some large value will be easily
                                 beaten here
        for (var i = 0; i < possibleMoves.Count(); i++)
        {
          Panel temp = possiblePieces[i].getPanel();
          if (board.getPanelsInUse().Contains(possibleMoves[i]))
          {
            foreach (Piece y in board.getPieces().ToList())
            {
              if (y.getPanel() == possibleMoves[i])
              {
                removedPiece = y;
                removed = true;
                board.RemovePiece(y);
                //take piece if we're taking a piece
              }
            }
          }
          possiblePieces[i].setPanel(possibleMoves[i]);
          board.setPanelsInUse();
          board.getAllPossibleBMoves(board);
          board.getPossibleBPieces();
          //update the board to do this move
          double tempVal = minimax(board, !max, depth - 1,
                                   board.getAllPossibleWMoves(board),
                                   board.getPossibleWPieces(), alpha, beta);
          //go deeper to see if this move will lead anywhere good
          beta = Math.Min(beta, tempVal);
          if (tempVal < bestVal)
          {
            bestVal = tempVal;
            beta = tempVal;
            //if this move leads somewhere good then tempVal will be less than bestVal
              as black tries to minimise, so this val is set to bestVal
          }
          possiblePieces[i].setPanel(temp);
          if (removed)
          {
            removed = false;
            board.AddPiece(removedPiece);
            //return the piece that was taken
          }
          board.setPanelsInUse();
          if (alpha >= beta)
          {
            return bestVal;
            //if alpha is greater than beta then we can prune the search a bit
          }

        }
```

```
        bestValue = bestVal;
      }
      return bestValue;
      //return the best value we have found
    }

    public Piece getMovingPiece()
    {
      return piece;
    }
    public Panel getMovingPiecePanel()
    {
      return piece.getPanel();
    }

    public double getBoardValue(BoardGen board)
    {
      double boardValue = 0; //boardValue starts as zero
      foreach (Piece p in board.getPieces().ToList())
      {
        if (p.getType().Substring(0, 1) == "B")
        {
          boardValue = boardValue - p.getValue(board);
          //when there's a black piece the board value becomes smaller/more negative
        }
        else
        {
          boardValue = boardValue + p.getValue(board);
          //otherwise the board value becomes bigger
        }
      }
      return boardValue;
    }
  }
}
```

Analysis of the code

For this iteration a new class was created called MoveCalculator. Within this class are five methods: bestMove, minimax, getMovingPiece, getMovingPiecePanel and getBoardValue. bestMove uses the minimax function to find a move that is the best move as far as it searches. Minimax uses the getBoardValue function to evaluate different positions on the board once it reaches a leaf. getMovingPiece and getMovingPiecePanel are both used to get the best move's piece and panel for the game play.

GameWindow has also been altered so that if we're in single player mode then the AI will make a move. The part of code is the same as the for when a user make's a move, except the movingPanel and movingPiece are first calculated and set to the piece/panel to be altered during the code.

BoardGen was edited to get all the moves for a given board so that the AI could have access to this information easily.

The way the pieces values were determined was also updated so that the AI could take position as a parameter when determining where to move, this made the AI slightly more capable.

Variable Identification and Justifiction

GameWindow

undoPanel is used to keep track of the panels where the pieces were.

undoPiece is used to keep ctrack of the pieces that have been moved.

noUndos is how far back you need to go for an undo.

undos is a list of how far back you need to go far each undo.

moveCalc instanciates the moveCalculator.

taken keeps track of if anything was taken when you wanted to undo.

BoardGen

allPossibleBMoves keeps a list of the possible moves for all black pieces for a given board

possibleBPieces keeps a list of the pieces for the moves black can do

allPossibleWMoves keeps a list of the possible moves for all white pieces for a given board

possibleWPieces keeps a list of the pieces for the moves white can do

Piece

hasMoved keeps track of whether a piece has moved or not

noMoves keeps track of the number of times a piece has moved

Pieces (Bishop, King, Knight, Pawn, Queen and Rook)

[PieceName]Table is a table of values gained at a certain position, corresponding to the position of the board. This 2D array is 8by8, so a position on the board directly corresponds with a value in the table. Tables differ by piece as different pieces are better in different positions.

MoveCalculator

piece keeps track of the piece that the bestMove acts upon.

possibleWPieces gets all the possible white pieces for the white moves for a given board.

possibleBPieces gets all the possible black pieces for the black moves for a given board.

allPossibleBMoves gets all the possible black moves for a given board.

allPossibleWMoves gets all the possible white moves for a given board.

bestMove keeps track of the panel on which the best move will go to.

bestVal keeps track of the best value the AI can get looking ahead.

removed keeps track of whether a piece has been removed during the search process.

removedPiece keeps track of the piece that was removed if one was removed.

temp the panel of the move that is being investigated.

tempVal is the value of the current line of moves being searched.

boardValue is the evaluation of a given board.

Iteration feedback and post-development evidence

Feature ID 2.1 – AI makes moves

Here the AI is playing as black. I make my move as white and black responds with quite a strong opening move.

I follow up my move as white moving another pawn forward into the centre. Again, black replies with quite a strong move. Although it has no built-in strategies, the position of these two pieces is greatest in these positions. More time and better hardware would give a greater searchdepth allowing the AI to develop some strategy.

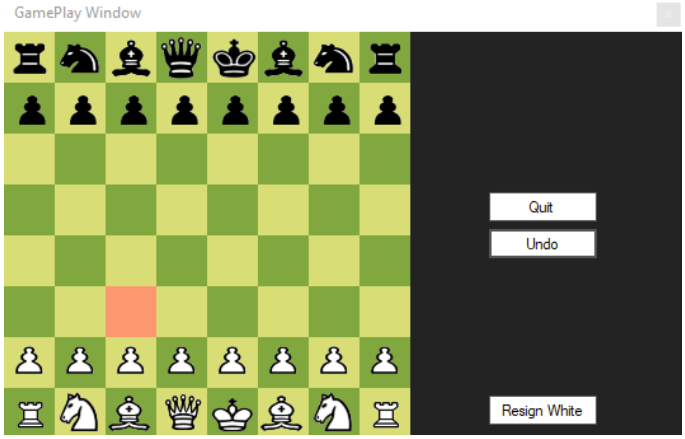
To test the AI, I ignore the threat on my pawn from the knight. In response the AI makes the obvious move and takes the pawn as expected.

Again, in an attempt to test the AI I threaten its knight. It again makes the obvious move of retreating from this threat whilst threating my pawn again. A good standard of play.

Feature ID 2.2 – Undo

A normal game up to this point.

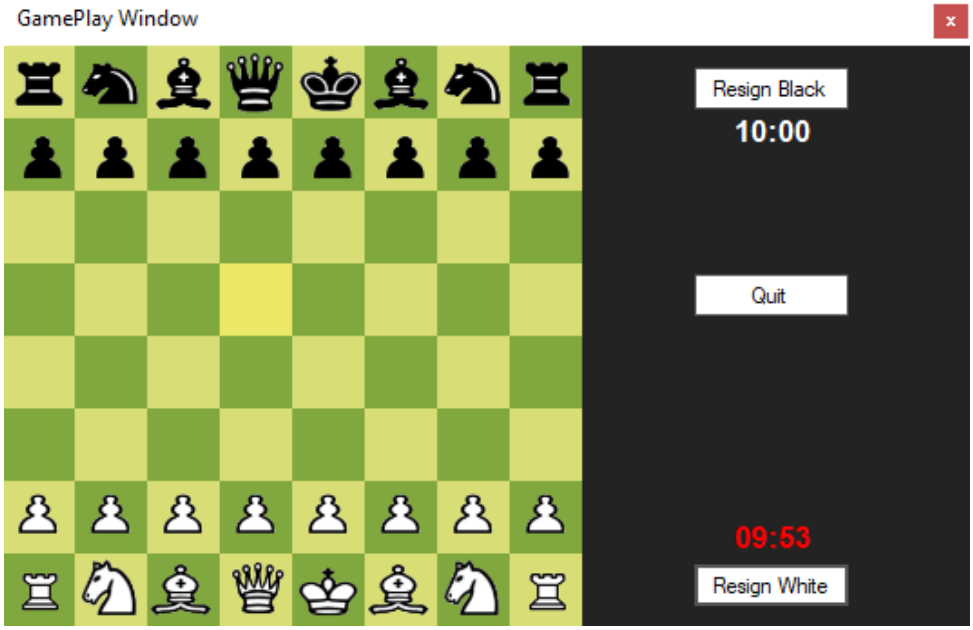
Press the undo button once and the game reverts to your last move.



Press again and the game goes back 1 more of your moves.

Press again and a messagebox pops up saying you cannot go back any further

During this iteration I was also able to add one of the features suggested in iteration 2. I thought about how I would make it more obvious whose go it was in 2 player chess and decided to make the timer a different colour to stand out when it is your go. Here is what is looked like:



Feedback

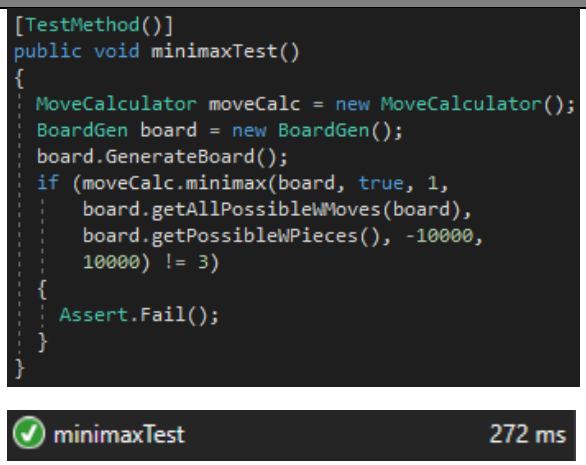
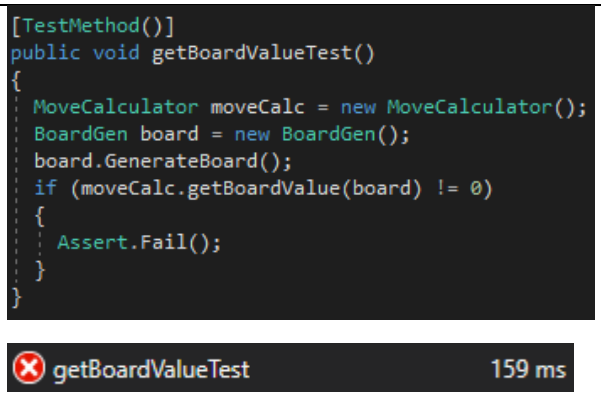
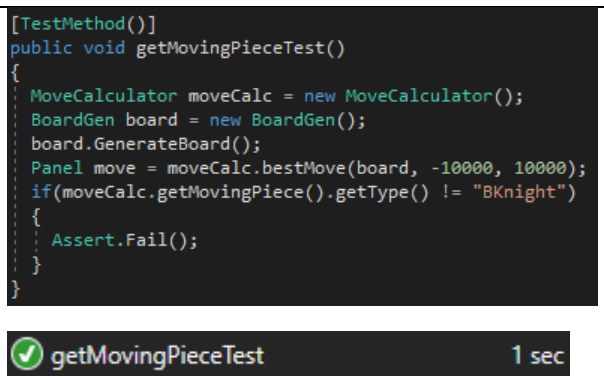
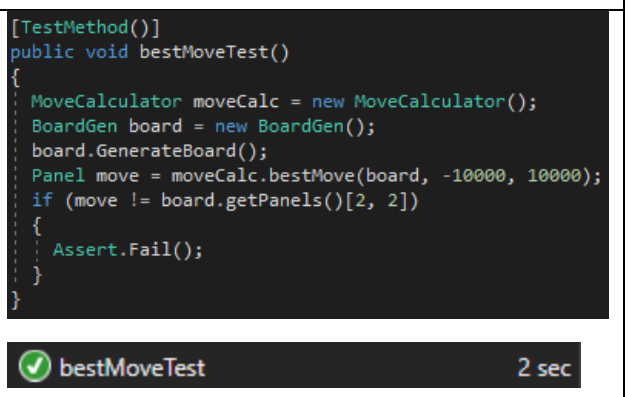
Overall, I would say this iteration was a success. Unfortunately, I was unable to add the different levels of AI due to the speed at which they would make moves. As a compromise I have developed the AI to try and be difficult enough to challenge intermediate players. Very experience players will still be able to play good games against it whilst beginner players may have a lot to learn from it. For this reason, I do not see the requirement as a necessity and as such will not be added. Aside from this, I have met all the requirements set out for this iteration and am happy that the program is finished.

Testing

Post-development testing

	Test	Result	Comments	Pass/Fail
U3.3	Test to see how the different levels of AI behave	Levels of AI were not added N/A		Fail
U3.4	Test to see how experienced players find playing the AI and how beginners find playing against it	Both beginners and experienced players enjoyed playing it	Tested by stakeholders	Pass

Testing during development

Test ID	Class being tested	Test Aim	Result	Proof
3DD1.0	MoveCalculator	Test to see if minimax returns a correct board value	Pass	
3DD1.1		Test to see if getBoardValue returns the correct value for a given board	Fail	
3DD1.2		Test to see if getMovingPiece returns the correct piece	Pass	
3DD1.3		Test to see if bestMove returns the best move for a given board	Pass	

```
[TestMethod()]
public void minimaxTest()
{
    MoveCalculator moveCalc = new MoveCalculator();
    BoardGen board = new BoardGen();
    board.GenerateBoard();
    if (moveCalc.minimax(board, true, 1,
        board.getAllPossibleWMoves(board),
        board.getPossibleWPieces(), -10000,
        10000) != 3)
    {
        Assert.Fail();
    }
}
```

minimaxTest 272 ms

```
[TestMethod()]
public void getBoardValueTest()
{
    MoveCalculator moveCalc = new MoveCalculator();
    BoardGen board = new BoardGen();
    board.GenerateBoard();
    if (moveCalc.getBoardValue(board) != 0)
    {
        Assert.Fail();
    }
}
```

getBoardValueTest 159 ms

```
[TestMethod()]
public void getMovingPieceTest()
{
    MoveCalculator moveCalc = new MoveCalculator();
    BoardGen board = new BoardGen();
    board.GenerateBoard();
    Panel move = moveCalc.bestMove(board, -10000, 10000);
    if(moveCalc.getMovingPiece().getType() != "BKnight")
    {
        Assert.Fail();
    }
}
```

getMovingPieceTest 1 sec

```
[TestMethod()]
public void bestMoveTest()
{
    MoveCalculator moveCalc = new MoveCalculator();
    BoardGen board = new BoardGen();
    board.GenerateBoard();
    Panel move = moveCalc.bestMove(board, -10000, 10000);
    if (move != board.getPanels()[2, 2])
    {
        Assert.Fail();
    }
}
```

bestMoveTest 2 sec

Addressing the failed test

After reviewing the piece tables and manually working out the initial value of the board, I realised the board started out at -2, not 0 as I had put in the test. I changed the 0 to a -2 and the test passed. Evidence below:

```
[TestMethod()]
public void getBoardValueTest()
{
    MoveCalculator moveCalc = new MoveCalculator();
    BoardGen board = new BoardGen();
    board.GenerateBoard();
    if (moveCalc.getBoardValue(board) != -2)
    {
        Assert.Fail();
    }
}
```

Iteration 3 Stakeholder Sign-off

After completing development of this iteration, I interviewed my stakeholders after asking them to use my final product and compare it to the requirements of this iteration. My stakeholders unanimously agreed that the requirements set out by this iteration had been completed and that no further changes to the product was needed.

Evaluation

Annotated Evidence of Post Development Testing

During the three iterations I designed post development tests for each iteration that I will carry out now that development has finished.

	Test	Result	Comments	Pass/Fail
F2.0	Test to see if when you click on a piece then its pieces are displayed correctly	When you click on a piece, if it is that colour's turn, the moves are displayed as intended	Evidenced within the iterations	Pass
F2.3	Test to see if the pieces move correctly	Pieces moved as intended	Evidenced within the iterations	Pass
F2.4	Test if check and checkmate behave correctly	A message box showed saying "check" for check and "checkmate" for checkmate	Evidenced within the iterations	Pass
F2.8	Test to see if an 8 by 8 grid of squares is generated	An 8 by 8 grid was generated	Evidenced within the iterations	Pass
F2.9	Test to see if the pieces are there	Pieces all showed correctly	Evidenced within the iterations	Pass
F2.5	Test to see if pieces taken and timer are displayed	Pieces taken, and timers were displayed as expected	Evidenced within the iterations	Pass
F2.6	Test to see if special moves work	Special moves worked as expected	Evidenced within the iterations	Pass
F2.7	Test to see if different game modes work	Game modes worked as expected	Evidenced within the iterations	Pass

F2.2	Undo button	Undo button behaved as expected	Evidenced within the iterations	Pass
F2.1	The AI makes moves	The AI made moves	Evidenced within the iterations	Pass
F2.1	The AI makes an objectively good move	The AI made good moves	When its piece was threatened it moved it away, when it had the opportunity to take for free it took the opportunity.	Pass

Evidence for post development testing was provided throughout the project but will further be provided during a post development video “ChessDemo.mp4”.

Annotated Evidence of Usability Testing

Tests:

	Test	**Result (Pass/OK/Fail)**	**Comments**
U3.0	Possible moves displayed correctly		
U3.1	Rules for each mode available for display		
U3.2	The pieces move		
U3.3	Different levels of AI		
U3.4	Accessible		
U3.5	Error Tolerant		

Martha Baylis:

	Test	Result (Pass/OK/Fail)	Comments
U3.0	Possible moves displayed correctly	Pass	The moves are displayed as the rules describe them.
U3.1	Rules for each mode available for display	Pass	The rules form was really helpful to read over to gain some understanding of what the rules are.
U3.2	The pieces move	Pass	The pieces move smoothly and quickly, especially in the two player modes.
U3.3	Different levels of AI	Fail	There were no different levels of AI to test because this was not a feature of the game
U3.4	Accessible	Pass	Anyone can use this product due to clear instruction and possible moves being displayed clearly for the user
U3.5	Error Tolerant	Pass	There were no errors

Lewis Clark:

	Test	Result (Pass/OK/Fail)	Comments
U3.0	Possible moves displayed correctly	Pass	The moves are displayed clearly and as the rules state is correct
U3.1	Rules for each mode available for display	Pass	They were available for each mode and helpful when it came to gaining understanding of the rules of the game
U3.2	The pieces move	Pass	They move accurately and within a reasonable time
U3.3	Different levels of AI	Fail	This product does not display different levels of AI to use
U3.4	Accessible	Pass	Rules being available for available modes and displaying the possible moves available to a player makes this product accessible to everyone
U3.5	Error Tolerant	OK	The game was in check but was not displayed as such

Charlie Knapp:

	Test	Result (Pass/OK/Fail)	Comments
U3.0	Possible moves displayed correctly	Pass	Moves were displayed according to what the rules stated to be true
U3.1	Rules for each mode available for display	Pass	Rules were easily accessible for each game mode and clear making them a very useful feature of this product
U3.2	The pieces move	Pass	The pieces move to where directed with no errors
U3.3	Different levels of AI	OK	There was a level of AI available
U3.4	Accessible	Pass	Rules clearly available for each mode and the clear display of possible moves available for each piece makes it easy to play for all abilities
U3.5	Error Tolerant	Pass	I encountered no errors when using this product

Daniel de Burgo:

	Test	Result (Pass/OK/Fail)	Comments
U3.0	Possible moves displayed correctly	OK	Possible moves were displayed as the rules stated to be correct however the moves were not displayed the first time I clicked on a piece all of the time
U3.1	Rules for each mode available for display	Pass	Rules for each game mode were clearly available and were helpful when it came to use the product
U3.2	The pieces move	Pass	Yes
U3.3	Different levels of AI	Fail	Does not exist
U3.4	Accessible	OK	I do not think this game would eb accessible for blind people
U3.5	Error Tolerant	OK	Possible move did not always display the first time I clicked a piece

Eachann Bruce:

	Test	Result (Pass/OK/Fail)	Comments
U3.0	Possible moves displayed correctly	Pass	Moves possible to the player were displayed accurate to what is stated in the rules
U3.1	Rules for each mode available for display	Pass	For each game mode, I imagine for an ill-experienced player that the rules set out in the forms would be clear, concise and helpful
U3.2	The pieces move	Pass	Pieces moved as expected
U3.3	Different levels of AI	OK	The AI provided was challenging enough, however I imagine it would not be too challenging for a beginner
U3.4	Accessible	OK	To cater for more experienced players, it would be beneficial to have more advanced tips in the help form
U3.5	Error Tolerant	OK	The game was displayed as being in check when it was not

Elianne Pekoulis David:

	Test	Result (Pass/OK/Fail)	Comments
U3.0	Possible moves displayed correctly	Pass	As far as I could tell, the possible moves were displayed as according to the rules
U3.1	Rules for each mode available for display	Pass	The rules are displayed for each game mode and are very clear and concise, making it easy to understand the game and therefore make good use of the product
U3.2	The pieces move	Pass	The movement of the pieces was as to be expected
U3.3	Different levels of AI	Fail	There was solely one level of AI available
U3.4	Accessible	Pass	Being an inexperienced player, the rules being concise and clear and the possible moves by each piece being displayed clearly made it easy to utilise this product and therefore deem it accessible
U3.5	Error Tolerant	OK	Game displayed checkmate which was false

Evaluation of Success Criteria

ID	Criteria	How to evidence	Justification
S1	An easy to use interface	Screen shots of the interface and reviews from stakeholders.	My program should be accessible, making it easy to use will allow a wider variety of users.
S2	A minimalistic interface	Screen shots and reviews from stakeholders.	A minimalistic design will keep things simple for new users
S3	Traditional design	Screen shots showing the design of the game.	A traditional design will make the came more recognisable.
S4	Click piece movement	Screen shots of before, during and after clicking on a piece and moving it.	This will provide an easy way to interact with the program
S5	AI functionality	Evidence of testing.	An AI will allow people to play by themselves, giving more people the opportunity to use the product
S6	Undo button	Screen shots of the undo button, after a move and then undoing the move.	An undo button whilst playing single player gives people an opportunity to correct their mistakes to learn better moves for a given position
S7	Display possible moves	Screen shots of a piece displaying its possible moves.	This will provide a way for new users to know what they can do before learning all the rules
S8	Check for check and checkmate	Screen shots of someone being in check and the legal moves changing. Also, a screen shot of the game ending in checkmate.	New users will not have to recognise when it is check and checkmate
S9	Display timer, taken pieces and whose turn it is	Screen shots after a piece has been taken. Screen shots showing the timer, and the game after a timer has run out.	Users will easily be able see which of their pieces have been taken and how much time they have left to play their moves
S10	Special moves	Screen shots of the result of each special move.	Users will be able to play all moves, inclusive of the moves that are quite unique and only occur in special cases
S11	Different game modes	Screen shots of the different games modes and how they differ from one another.	This will allow users to try a variety of different game modes to have some variation to standard chess
S12	Pieces move	Evidence of testing.	Pieces must move as they would during chess in a board game
S13	Visible board with recognisable pieces	Screen shots of the board and pieces.	The board should be easy to see with the pieces recognisable so that users can easily tell which piece is which
S14	Invulnerable to errors	Evidence of testing.	This is to ensure the user has a good experience, without the product doing something unexpected

S15	Accessible to a wide range of people	Stakeholder survey.	This is to make sure my product is accessible
S16	Pieces can take other pieces	Screen shots before and after taking a piece.	This is to allow standard gameplay
S17	Different levels of AI	Screen shots of parts of games from different levels of AI.	This will ensure that the single player mode is accessible to a wide range of users

S1 – Met

As shown by the comments by my stakeholders, all of my stakeholders agreed upon the face that the design was laid out in an easy to use manner.

S2 – Met

My stakeholders also commented on the simplicity of the design allowing it to be more accessible to a wider range of users.

S3 – Met

On interviewing my stakeholders, they agreed that the design of the chess board conformed with a traditional design.

S4 – Met

Although one of my stakeholders found an issue with sometimes the piece not displaying it's move when first clicked, Dan was the only person to experience this issue. This criterion was met as shown by U3.0, U3.2, F2.0 and F2.3.

S5 – Met

The AI functioned as expected, making relatively good moves as shown by evidence given for U3.3, U3.4, F2.1, 3DD1.0, 3DD1.1, 3DD1.2 and 3DD1.3.

S6 – Met

The undo button functioned as excepted, undoing the most recently done move when pressed as shown by evidence given for F2.2

S7 – Met

Although one of my stakeholders found an issue with sometimes the piece not displaying it's move when first clicked, Dan was the only person to experience this issue. This criterion was met as shown by U3.0, U3.2, F2.0, F2.3, 1DD3.0, 1DD4.0, 1DD5.0, 1DD6.0, 1DD7.0 and 1DD8.0.

S8 – Partially Met

Lots of my stakeholders commented some issues with the functionality of check. Mostly complaints about check being displayed when it was not check and one complaint about it being checkmate when it was not. In spite of this, the function still works when the board is check. There were no complaints of the function not saying check or checkmate when it was such. This is shown by the evidence given for F2.4, U3.5, 1DD2.0 and 1DD2.4. This will be something I refer to in the further development section.

S9 – Met

The timer and interface where pieces taken are shown worked as intended as shown in evidence given for F2.5

S10 – Met

Moves that occur under unique and special conditions behaved exactly as described by the rules as shown by evidence given for F2.3 and F2.6.

S11 – Met

The different game modes played and abided by the rules for the game modes as expected as shown by evidence given for F2.7 and 2DD1.0.

S12 – Met

Pieces move as expected and followed the rules set out for the game as shown by evidence given for U3.2 and F2.3

S13 – Met

My stakeholders did not have trouble recognising the pieces and commented that the board had a nice design.

S14 – Partially Met

There were some errors that minorly affected gameplay that were discovered during usability testing. A recurring error seemed to be that check was occasionally displayed when it shouldn't have been. Whilst this does give something to improve upon, there were no major errors. No errors were discovered that caused the game to crash or majorly affected gameplay, as these were the only errors to occur, the program was mostly error tolerant. This is shown by evidence given for U3.5. This will be something I refer to in the further development section.

S15 – Met

The game was accessible to as many people that I could make it accessible to. Whilst a couple of my stakeholders felt it would be difficult for blind users to play and there are things that could've been coded to make it more accessible, these would not have been realistic in the time frame (for example, something the reads out the square you are hovering over, and the opponents move). This is shown by evidence given for U3.4.

S16 – Met

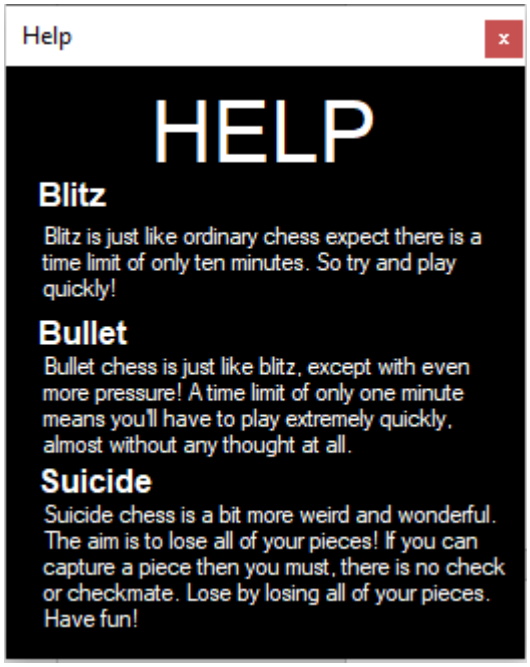
Pieces were able to take other pieces and these other pieces were displayed correctly in the interface. This is shown by evidence given for F2.3 and F2.5.

S17 – Partially Met

Although there were no different levels of AI, this success criteria have not been failed. The aim of this success criterion was to provide a range of challenged for a range of ability of players. My stakeholders and I believe that this AI is challenging enough for an experienced player to enjoy a game against or for a complete beginner to be challenged by and perhaps learn from it. The other aim of this criterion was to ensure that there was a functioning AI that any person could play against (rather than just experienced players or novices), and this was met. This is evidenced by U3.3 and F2.1. This will be something I refer to in the further development section.

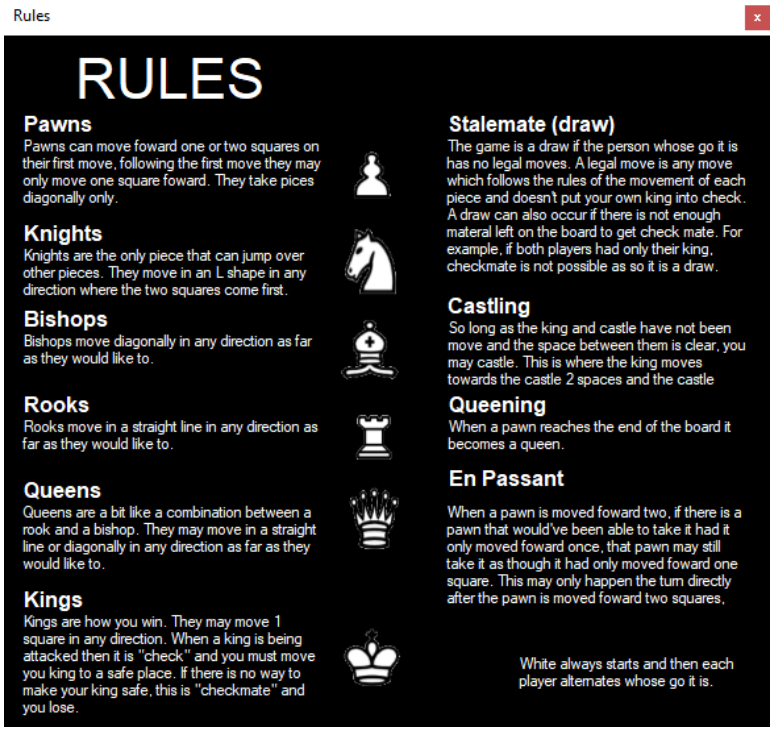
Usability Features

Help



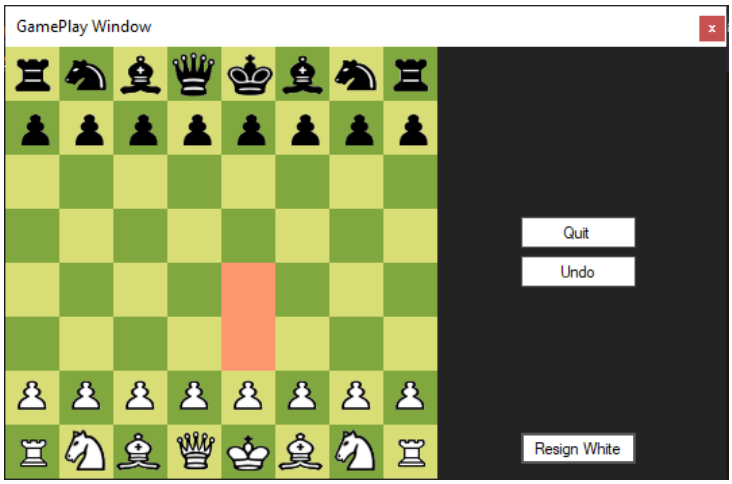
This usability feature gives the user some understanding of what the game modes are. I explain what each game mode means, what happens and the rules. This was a met usability feature and my stakeholders said they found it to be quite helpful, especially as suicide chess is quite an obscure game mode. This was a **met** usability criterion.

Rules



This usability feature is where I explain the rules to standard chess. My stakeholders (especially the more inexperienced ones) found these to be very helpful when trying to understand how to play. This was a **met** usability criterion.

Possible moves displayed

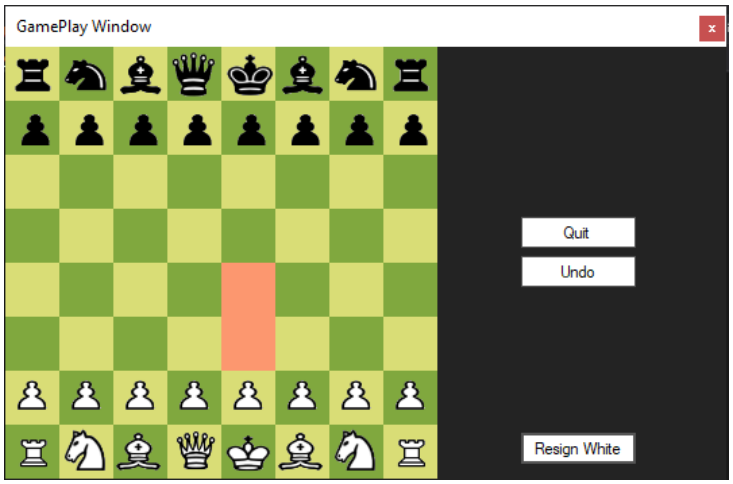


This usability feature allows players who are still unsure of the rules to moving pieces to know what they can do. My stakeholders found this feature extremely helpful and was possibly the most successful usability feature and a **met** criterion.

Click piece movemen

Above is how the pieces move from iteration 1. This didn't change throughout the iterations because this way of moving was simple, intuitive and allowed users to start playing more quickly. My stakeholders said that they had no trouble using this type of movement and that this was an excellent way of moving pieces. This was also a **met** usability criterion.

Colour-blind friendly



Whilst changing the design in iteration 2 I did briefly come across some trouble with colour-blind users not being able to see the possible moves as the colour was too similar to one of the background colours. This was quickly changed to a much more obvious colour and once again the colours were easy to see. This was a **met** usability criterion.

Using some online software which simulates how people with specific types of colour-blindness see images, I have successfully shown that anyone would be able to distinguish between the different colours of things except for monochromacy which is extremely difficult to account for as people with this kind of colour-blindness see everything in black and white, making the colours essentially irrelevant. It is also difficult for people who are green blind to distinguish between the orange of the possible moves and some of the board panels, however it is possible to distinguish.

Red-Weak/Protanomaly

Green-Weak/Deuteranomaly

Blue-Weak/Tritanomaly

Red-Blind/Protanopia

Green-Blind/Deuteranopia

Blue-Blind/Tritanopia

Monochromacy/Achromatopsia

Blue Cone Monochromacy

Accessible AI

This was the only **partially met** usability criterion. Originally, I would have liked to add different levels of AI so that a wide range of users could play against the AI, however due to restrictions on time and processing power this was not possible. There is an AI available that is adequate to challenge experienced players and help beginners to learn, however this was not what I originally set out to do and why this is only **partially met**. This will be something I refer to in the further development section.

Maintenance Issues

Visual Studio

Updates to visual studio could impact the development of this project. One of the reasons Visual Studio was picked for this project was because of its renowned backwards compatibility, which means almost all code written on an old version of visual studio will still be able to be modified and developed in the most recent version of visual studio. Because of this backwards compatibility the project should be maintainable from this perspective.

Architecture

The code was written on a 64x bit architecture PC. This means that the code will work on any 64-bit machine. Whilst this is much more portable than 32-bit code, it means that the code will be able to use on a wider range of devices for more time. This is because as almost all modern computers that are produced are 64 bits, at some point in the future machines are likely to become 128-bit, which will cause some issues with keeping the code relevant, however the 64-bit code should last quite a long time.

Comments

I have written comments on various lines of code so that when I return to the project, I will be able to realise the function of different bits of code that I have already written much more quickly, allowing maintenance to the code to be much easier. It also means that if I ever decided to get another developer to help me with the project, it would be much easier for them to understand the purpose and meaning behind what I was doing when I originally wrote the code. This means that maintenance both on a possible future developers part and my part will be made slightly easier. However, I could have added more commenting to the code, explaining what the code was doing and why it was doing it in much more than I did, which would've made maintenance easier, which I will keep in mind if I decided to develop this project further.

Windows

One real maintenance issue is windows updates. The code is compatible with Windows 10 but will not necessarily be compatible with future windows updates, this could be an issue if windows drastically changes the way it operates, however because it is an executable file, should run on windows for the foreseeable future.

Variable Names

The variables have been named so that it is obvious what their purpose is. For example, the variable named “boardPanels” stores the panels on the board. This will make maintenance easier.

Resources

External files used within the project have all been added to the resources of the project. This means that the code is much more portable, as you do not have to have a specific file in a specific location that is called a specific thing for the program to work. It can run completely independently, allowing future files that need to be used to be added to resources also will be much easier than using external files.

Limitations of the solution and potential improvements

Limited to local play

You may only play against real people if you play locally. This limits the number of people that will use the project as many people would want to play over the internet against friends. In future development there will be the opportunity to add a multiplayer option.

Chess AI is not that strong

Also limiting the number of users of my project is the strength of the AI. Due to the type of algorithm chosen at the core of the AI, the processing power to gain any advanced insight into the game is much more than the average desktop PC can manage. To improve the strength of the AI it is likely a different algorithm would need to be implemented (e.g. a Monte Carlo search) and combined with a more in depth evaluation function (perhaps using a neural network), the other option for improving the AI would be to parallelise it, this would mean the computer could be searching multiple branches of the tree at one, dramatically reducing search time and allowing a greater search depth.

Levels of AI not available

Currently there is no way to select how good a player you are and play an AI that will roughly match your skill level for the most optimal game. In any further development done to the project it would be relatively simple to add different levels of AI as long as the core of the AI was edited and improved appropriately, currently all that would be required would be selecting a higher search depth for a more challenging AI and a lower one for an easier AI, however as the AI is currently programmed, it cannot have a higher search depth and there is not much room for a lower one.

Hints

Hints that show players the best possible move when playing against a computer would also be a good idea to add in future development as it would allow players to see what moves to make in what positions and learn from that tactically. To do this all that would need be done is to have the AI work out the best possible move and highlight the piece and possible move that this is.

Final code

Bishop

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Windows.Forms;

namespace CollegeProject
{
  [Serializable]
  public class Bishop : Piece
  {

    List<Panel> possibleMoves;
    bool capturePossible = false;
    public double[,] BishopTable;

    public Bishop(string type, Panel image, bool moved) : base(type, image, moved)
    {

      BishopTable = new double[8, 8]
          { { -2.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -2.0 },
                  { -1.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0, -1.0 },
                  { -1.0,  0.0,  0.5,  1.0,  1.0,  0.5,  0.0, -1.0 },
                  { -1.0,  0.5,  0.5,  1.0,  1.0,  0.5,  0.5, -1.0 },
                  { -1.0,  0.0,  1.0,  1.0,  1.0,  1.0,  0.0, -1.0 },
                  { -1.0,  1.0,  1.0,  1.0,  1.0,  1.0,  1.0, -1.0 },
                  { -1.0,  0.5,  0.0,  0.0,  0.0,  0.0,  0.5, -1.0 },
                  { -2.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -2.0 }
          };
    }

    public override double getValue(BoardGen board)
    {
      for (int x = 0; x < 8; x++)
      {
        for (int y = 0; y < 8; y++)
        {
          if (board.getPanels()[x, y] == getPanel())
          {
            if (board.getPanels()[x, y] == getPanel())
            {
              double val = (getType().Substring(0, 1) == "W") ? BishopTable[x, y] :
                                                                BishopTable[y, x];

              return val + 30;
            }
          }
        }
      }
      return 0;
    }

    public override void setMoves(BoardGen board, bool checkUp, bool suicide)
    {
      possibleMoves = new List<Panel>();
      Piece piece = null;
      Panel tempPan = null;
      capturePossible = false;
```

```
//A bishop may move along the diagnol it is on

foreach (Panel x in board.getPanels())
{
  for (int i = 0; i < 8; i++)
  {
    foreach (Piece c in board.getPieces())
    {
      if (c.getPanel() == x)
      {
        piece = c;
      }
    }

    if ((getPanel().Location.X == x.Location.X + (40 * i) ||
        getPanel().Location.X == x.Location.X - (40 * i)) &&
        (getPanel().Location.Y == x.Location.Y + (40 * i) ||
        getPanel().Location.Y == x.Location.Y - (40 * i)))
    {
      if (board.getPanelsInUse().Contains(x))
      {

        foreach (Piece y in board.getPieces())
        {
          if (y.getPanel() == x)
          {
            if (getType().Substring(0, 1) !=
                y.getType().Substring(0, 1))
            {
              possibleMoves.Add(x);

            }
          }
        }
      }
      else
      {
        possibleMoves.Add(x);
      }
    }
  }

}
/* This bit just gets all of the panels in a diagonal direction from the
   bishop in use */

List<Panel> panelsRemove = new List<Panel>();

foreach (Panel x in board.getPanels())
{
  for (int i = 0; i < 8; i++)
  {
    if ((getPanel().Location.X == x.Location.X + (40 * i) ||
        getPanel().Location.X == x.Location.X - (40 * i)) &&
        (getPanel().Location.Y == x.Location.Y + (40 * i) ||
        getPanel().Location.Y == x.Location.Y - (40 * i)) &&
        board.getPanelsInUse().Contains(x))
    {
      if (x.Location.Y < getPanel().Location.Y &&
          x.Location.X < getPanel().Location.X)
      {
```

```
            foreach (Panel g in possibleMoves)
            {
              if (g.Location.Y < x.Location.Y && g.Location.X <
                  x.Location.X)
              {
                panelsRemove.Add(g);
              }
            }
          }
          else if (x.Location.Y < getPanel().Location.Y &&
              x.Location.X > getPanel().Location.X)
          {
            foreach (Panel g in possibleMoves)
            {
              if (g.Location.Y < x.Location.Y && g.Location.X >
                  x.Location.X)
              {
                panelsRemove.Add(g);
              }
            }
          }
          else if (x.Location.Y > getPanel().Location.Y &&
              x.Location.X < getPanel().Location.X)
          {
            foreach (Panel g in possibleMoves)
            {
              if (g.Location.Y > x.Location.Y && g.Location.X <
                  x.Location.X)
              {
                panelsRemove.Add(g);
              }
            }
          }
          else if (x.Location.Y > getPanel().Location.Y &&
              x.Location.X > getPanel().Location.X)
          {
            foreach (Panel g in possibleMoves)
            {
              if (g.Location.Y > x.Location.Y &&
                  g.Location.X > x.Location.X)
              {
                panelsRemove.Add(g);
              }
            }
          }
        }
      }
    }
    /* This part adds all of the panels that are blocked in some way to an
       array that removes all of those panels from possible moves (the next
       part */

    foreach (Panel x in possibleMoves)
    {
      foreach (Piece p in board.getPieces())
      {
        if (x == p.getPanel())
        {
          capturePossible = true;
        }
      }
    }
```

```
foreach (Panel x in panelsRemove)
{
  possibleMoves.Remove(x);
}

if (checkUp && !suicide) //if checking for check and this is not suicide chess
                          (as chess doesn't exist in suicide chess)
{
  Piece toRem = null;
  bool removed = false;
  tempPan = getPanel();
  foreach (Panel x in possibleMoves)
  {
    if (board.getPanelsInUse().Contains(x))
    {
      foreach (Piece z in board.getPieces().ToList())
      {
        if (z.getPanel() == x)
        {
          toRem = z;
          removed = true;
          board.RemovePiece(z);
        }
      }
    }

    setPanel(x);

    board.setPanelsInUse();

    foreach (Piece y in board.getPieces().ToList())
    {
      if (!y.getType().Contains(getType()))
      {
        y.setMoves(board, false, suicide);
      }
    }

    if (checkCheck(board) && checkType() ==
        getType().Substring(0, 1))
    {
      panelsRemove.Add(x);
    }

    setPanel(tempPan);

    if (removed)
    {
      board.AddPiece(toRem);
      removed = false;
    }

    board.setPanelsInUse();
  }
  /* This part checks all of the current possible moves for the
     the bisop to see if any would leave the player in check, if it
     would, it is not a valid move and is removed. */

  foreach (Panel x in panelsRemove)
  {
```

```
          possibleMoves.Remove(x);
        }
      }

      if (capturePossible && suicide) //if it is suicide chess and there is a capture
                                      possible
      {
        List<Panel> newPossibleMoves = new List<Panel>();
        foreach (Panel x in possibleMoves)
        {
          foreach (Piece y in board.getPieces())
          {
            if (x == y.getPanel())
            {
              newPossibleMoves.Add(x);
            }
          }
        }
        possibleMoves = newPossibleMoves;
        //captures are the only possible moves if a capture is possible
      }
      else if (!capturePossible && suicide) //if there no captures possible in suicide
                                            chess
      {
        foreach (Piece x in board.getPieces())
        {
          if (x.capture() && x.getType().Substring(0, 1) == getType().Substring(0, 1))
          {
            possibleMoves = new List<Panel>();
            break;
            //if there are no possible captures, but there are possible captures from
               other pieces, this piece cannot move
          }
        }
      }
    }

    public override List<Panel> getMoves()
    {
      return possibleMoves;
    }

    public override bool capture()
    {
      return capturePossible;
    }
  }
}
```

Boardgen

```
using System;
using System.Collections.Generic;
using System.Drawing;
using System.Linq;
using System.Windows.Forms;

namespace CollegeProject
{
  public class BoardGen
  {
```

```
    Panel[,] boardPanels;
    List<Piece> pieces = new List<Piece>();
    List<Panel> inUse;
    Pawn pawn;
    Rook rook;
    Knight knight;
    Bishop bishop;
    Queen queen;
    King king;
    MoveCalculator MoveCalc = new MoveCalculator();
    List<Piece> possibleBPieces;
    List<Piece> possibleWPieces;

    public BoardGen()
    {

    }


    public Panel[,] GenerateBoard()
    {

      int tileSize = 40;
      int gridSize = 8;
      //Size of squares
      Color color1 = System.Drawing.ColorTranslator.FromHtml("#80A83E");
      Color color2 = System.Drawing.ColorTranslator.FromHtml("#D9DD76");
                                                                    //Colour of
                                                                      sqaures


      boardPanels = new Panel[gridSize, gridSize];

      //Nested for loop to get the 8 by 8 grid set up
      for (int i = 0; i < gridSize; i++)
      {
        for (int x = 0; x < gridSize; x++)
        {

          //Set the panel details
          Panel newPanel = new Panel
          {
            Size = new Size(tileSize, tileSize),
            Location = new Point(tileSize * i, tileSize * x),
            BackgroundImageLayout = ImageLayout.Stretch,
          };

          String peiceType = null;

          //Set what colour the piece should be
          peiceType = (tileSize * x <= 40) ? "B" : "W";

          //Setting what type of pieces go where and geneerating the pieces
          if (tileSize * x == 40 || tileSize * x == 240)
          {
            peiceType += "Pawn";
            pawn = new Pawn(peiceType, newPanel, false);
            pieces.Add(pawn);
          }
          else if ((tileSize * i == 0 || tileSize * i == 280) &&
                  (tileSize * x == 0 || tileSize * x == 280))
          {
            peiceType += "Rook";
```

```
            rook = new Rook(peiceType, newPanel, false);
            pieces.Add(rook);
          }
          else if ((tileSize * i == 40 || tileSize * i == 240) &&
                  (tileSize * x == 0 || tileSize * x == 280))
          {
            peiceType += "Knight";
            knight = new Knight(peiceType, newPanel, false);
            pieces.Add(knight);
          }
          else if ((tileSize * i == 80 || tileSize * i == 200) &&
                  (tileSize * x == 0 || tileSize * x == 280))
          {
            peiceType += "Bishop";
            bishop = new Bishop(peiceType, newPanel, false);
            pieces.Add(bishop);
          }
          else if ((tileSize * i == 160) &&
                  (tileSize * x == 0 || tileSize * x == 280))
          {
            peiceType += "King";
            king = new King(peiceType, newPanel, false);
            pieces.Add(king);
          }
          else if ((tileSize * i == 120) &&
                  (tileSize * x == 0 || tileSize * x == 280))
          {
            peiceType += "Queen";
            queen = new Queen(peiceType, newPanel, false);
            pieces.Add(queen);
          }

          boardPanels[i, x] = newPanel;

          if (i % 2 == 0)
          {
            newPanel.BackColor = x % 2 != 0 ? color1 : color2;
          }
          else
          {
            newPanel.BackColor = x % 2 != 0 ? color2 : color1;
          }
        }
      }

      //update what panels are currently taken
      setPanelsInUse();
      //return the board that has been generated
      return boardPanels;
    }

public Panel[,] getPanels()
    {
      return boardPanels;
    }

    public List<Piece> getPieces()
    {
      return pieces;
    }

    public void setPanelsInUse()
    {
```

```
      inUse = new List<Panel>();
      foreach (Panel x in boardPanels)
      {
        foreach (Piece p in pieces)
        {
          if (x == p.getPanel())
          {
            inUse.Add(x);
          }
        }
      }
      //Sets the panels that have a piece on them
    }

    public List<Panel> getPanelsInUse()
    {
      return inUse;
    }

    public void RemovePiece(Piece piece)
    {
      pieces.Remove(piece);
      inUse.Remove(piece.getPanel());
      //Removes a piece from the board
    }

    public void AddPiece(Piece piece)
    {
      pieces.Add(piece);
      inUse.Add(piece.getPanel());
      //Adds a piece to the board
    }

    public List<Panel> getAllPossibleBMoves(BoardGen board)
    {
      List<Panel> allPossibleBMoves = new List<Panel>();
      possibleBPieces = new List<Piece>();
      foreach (Piece x in getPieces().ToList())
      {
        x.setMoves(board, false, false);
        if (x.getType().Substring(0, 1) == "B")
        {
          foreach (Panel p in x.getMoves())
          {
            allPossibleBMoves.Add(p);
            possibleBPieces.Add(x);
          }
        }
      }
      //gets all the possible moves for all possible black pieces
      return allPossibleBMoves;
    }

    public List<Panel> getAllPossibleWMoves(BoardGen board)
    {
      List<Panel> allPossibleWMoves = new List<Panel>();
      possibleWPieces = new List<Piece>();
      foreach (Piece x in getPieces().ToList())
      {
        x.setMoves(board, false, false);
        if (x.getType().Substring(0, 1) == "W")
        {
          foreach (Panel p in x.getMoves())
          {
            allPossibleWMoves.Add(p);
```

```
            possibleWPieces.Add(x);
          }
        }
      }
      //gets all the possible moves for all possible white pieces
      return allPossibleWMoves;
    }

    public List<Piece> getPossibleBPieces()
    {
      return possibleBPieces;
    }

    public List<Piece> getPossibleWPieces()
    {
      return possibleWPieces;
    }
  }
}
```

GameWindow

```
using CollegeProject.Properties;
using System;
using System.Collections.Generic;
using System.Drawing;
using System.Linq;
using System.Windows.Forms;

namespace CollegeProject
{
  public partial class Form1 : Form
  {

    //declaration of variables
    List<Panel> undoPanel = new List<Panel>();
    List<Piece> undoPiece = new List<Piece>();
    Pawn piecePosRemove;
    int noUndos = 0;
    List<int> undos = new List<int>();
    string gameModes;
    int WhiteSeconds;
    int BlackSeconds;
    public bool timed = false;
    public bool suicide = false;
    private bool WhiteMove = true;
    private Color tempBackColour;
    private Piece movingPiece = null;
    private Panel movingPiecePanel;
    public BoardGen generation = new BoardGen();
    private MoveCalculator moveCalc = new MoveCalculator();
    private Panel[,] board;
    private Panel actionPanel;
    private List<Piece> pieces;
    private Piece newQueen;
    PictureBox[] pictureBoxesB;
    PictureBox[] pictureBoxesW;
    Panel toUpdate = null;
    Piece updatePiece = null;
    bool updateReq = false;
    int bPiecesTaken = 0;
    int wPiecesTaken = 0;
```

```
public Form1(String gameMode)
{
  gameModes = gameMode;
  if (gameMode == "M3S") //if the gameMode is M3S then the user is playing suicide
                         chess
  {
    suicide = true;
  }
  board = generation.GenerateBoard();
  //Generate the board
  pictureBoxesB = new PictureBox[15];
  int z = 330;
  int y = 60;
  if (gameMode == "M1B" || gameMode == "M2B") //if the gameMode is M1B or M2B then
                                                the game is timed
  {
    WhiteSeconds = gameMode == "M1B" ? 600 : 60; //and the time is set accordingly
    BlackSeconds = WhiteSeconds;
    timed = true;
  }

  foreach (PictureBox x in pictureBoxesB) //Generates the interface where taken
                                            black pieces are stored
  {
    PictureBox newPictureBox = new PictureBox
    {
      Size = new Size(25, 25),
      BackgroundImageLayout = ImageLayout.Stretch,
    };
    newPictureBox.Location = new Point(z, y);
    z += 25;
    if (z > 505)
    {
      z = 330;
      y += 25;
    }
    pictureBoxesB[bPiecesTaken] = newPictureBox;
    Controls.Add(pictureBoxesB[bPiecesTaken]);
    bPiecesTaken ++;

  }

  pictureBoxesW = new PictureBox[15];
  z = 330;
  y = 215;
  bPiecesTaken = 0;

  foreach (PictureBox x in pictureBoxesW) //Generates the interface where taken
                                            white pieces
  {
    PictureBox newPictureBox = new PictureBox
    {
      Size = new Size(25, 25),
      BackgroundImageLayout = ImageLayout.Stretch,
    };
    newPictureBox.Location = new Point(z, y);
    z += 25;
    if (z > 505)
    {
      z = 330;
      y += 25;
```

```
        }
        pictureBoxesW[bPiecesTaken] = newPictureBox;
        Controls.Add(pictureBoxesW[bPiecesTaken]);
        bPiecesTaken ++;

      }

      foreach (Panel x in board)
      {
        x.MouseEnter += new EventHandler(panel_MouseEnter);
        x.MouseLeave += new EventHandler(panel_MouseLeave);
        x.MouseDown += new MouseEventHandler(panel_MouseDown);
        Controls.Add(x);
      }
      //Add the action events to the panels and then render the panels.
      bPiecesTaken = 0;
      InitializeComponent();
      if (gameMode == "M1B" || gameMode == "M2B") //formats the time to display in
                                                    minutes
      {
        label4.ForeColor = Color.Red;
        label3.Text = gameMode == "M1B" ? "10:00" : "01:00";
        label4.Text = label3.Text;
      }

      timer1.Start(); //starts the timer

    }

    private void panel_MouseEnter(object sender, EventArgs e)
    {
      if (timer1.Enabled)
      {
        actionPanel = (Panel)sender;
        tempBackColour = actionPanel.BackColor;
        actionPanel.BackColor = ColorTranslator.FromHtml("#ebe867");
      }


      //When mouse is over a panel it appears yellow
    }

    private void panel_MouseLeave(object sender, EventArgs e)
    {
      if (timer1.Enabled)
      {
        actionPanel = (Panel)sender;
        actionPanel.BackColor = tempBackColour;
      }


      //Puts panel back to original colour when mouse not over it
    }
    private void panel_MouseDown(object sender, MouseEventArgs e)
    {

      Panel actionPanel = (Panel)sender;
      bool castle = false;

      //This section is for when no piece has been clicked yet
      if (timer1.Enabled) //if the game hasn't ended
      {
```

```
        if (movingPiece == null)
        {
          pieces = generation.getPieces();

          //Gets the piece that was clicked
          foreach (Piece p in pieces)
          {
            if (p.getPanel() == actionPanel)
            {
              movingPiece = p;
              movingPiecePanel = actionPanel;
              break;
            }
          }

          //If the panel that was clicked actually contains a piece
          try
          {
            //And it's that peice's colours turn
            if ((WhiteMove &&
                movingPiece.getType().Substring(0, 1).Equals("W")) ||
               (!WhiteMove &&
               movingPiece.getType().Substring(0, 1).Equals("B")))
            {
              movingPiece.setMoves(generation, true, suicide);
              //Get that piece's moves

              foreach (Panel x in movingPiece.getMoves())
              {
                x.BackColor = ColorTranslator.FromHtml("#FFFC7F");
                //And display them
              }

            }
          }
          catch (NullReferenceException)
          {
            //Could notify the player they clicked an empty sqaure here
          }
        }

        //This section is for when a piece has already been clicked
        else
        {
          movingPiece.setMoves(generation, true, suicide);
          //Update the moves of that piece for their new position

          //Get the colours of the board back to normal
          for (int k = 0; k <= 7; k++)
          {
            for (int s = 0; s <= 7; s++)
            {

              if (k % 2 == 0)
              {
                generation.getPanels()[k, s].BackColor =
                    s % 2 != 0 ? ColorTranslator.FromHtml("#80A83E") :
                                                ColorTranslator.FromHtml("#D9DD76");
              }
              else
              {
                generation.getPanels()[k, s].BackColor =
```

```
                s % 2 != 0 ? ColorTranslator.FromHtml("#D9DD76"):
                                        ColorTranslator.FromHtml("#80A83E");
            }
          }
        }

        //If that panel clicked was not the same as the first panel the person
          clicked
        if (actionPanel != movingPiecePanel && movingPiece.getType().Substring(0, 1)
                                                == (WhiteMove ? "W" : "B"))
        {
          //And the panel is valid
          try
          {
            if (movingPiece.getMoves().Contains(actionPanel))
            {

              if (generation.getPanelsInUse().Contains(actionPanel))
              {
                foreach (Piece z in generation.getPieces().ToList())
                {
                  if (z.getPanel() == actionPanel)
                  {
                    generation.RemovePiece(z);
                    if (z.getType().Substring(0, 1) == "W")
                    {
                      pictureBoxesB[bPiecesTaken].BackgroundImage =
                                                z.getPanel().BackgroundImage;
                      bPiecesTaken ++;

                    }
                    else
                    {
                      pictureBoxesW[wPiecesTaken].BackgroundImage =
                                                z.getPanel().BackgroundImage;
                      wPiecesTaken++;
                    }
                  }
                }
                actionPanel.BackgroundImage = null;
                /*If the panel was a possible move and contains a
                piece, the piece that it already contains needs
                to be disposed of and stored in the interface where
                taken pieces are displayed*/

              }
              if (movingPiece.getType().Contains("King"))
              {
                if ((actionPanel == generation.getPanels()[6,
                     movingPiece.getType().Substring(0, 1) == "W" ? 7 : 0] ||
                     actionPanel == generation.getPanels()[2,
                     movingPiece.getType().Substring(0, 1) == "W" ? 7 : 0]) &&
                     !movingPiece.getMoved())
                {
                  castle = true;
                }
              }
              //Recognises when the king is trying to castle


              if ((WhiteMove &&
                  movingPiece.getType().Substring(0, 1).Equals("W")) ||
```

```
            (!WhiteMove &&
            movingPiece.getType().Substring(0, 1).Equals("B")))
        {
          actionPanel.BackgroundImage =
          movingPiece.getPanel().BackgroundImage;

          undoPiece.Add(movingPiece);
          undoPanel.Add(movingPiece.getPanel());
          noUndos++;

          movingPiece.setPanel(actionPanel);

          if (updateReq && movingPiece.getPanel() != updatePiece.getPanel())
          {
            updatePiece.setPanel(toUpdate);
            generation.RemovePiece(piecePosRemove);
            updateReq = false;
            //if a pawn has moved foward two last move, it can no longer be
              taken as if it were on the first sqaure, so don't allow that to
              happen.
          }
          else if (updateReq)
          {
            if (movingPiece.getType().Substring(0, 1) == "W")
            {
              pictureBoxesW[wPiecesTaken - 1].BackgroundImage =
                                                          Resources.BPawn;
            }
            else
            {
              pictureBoxesB[bPiecesTaken - 1].BackgroundImage =
                                                          Resources.WPawn;
            }
            updatePiece.getPanel().BackgroundImage =
                                      movingPiece.getPanel().BackgroundImage;
            piecePosRemove.getPanel().BackgroundImage = null;
            generation.RemovePiece(piecePosRemove);
            updatePiece.setPanel(null);

            updateReq = false;

            //If a pawn has moved foward two last move, and a pawn has taken
              it as if it were on the first square, this allows it to be
              taken.
          }

          if (movingPiece.getType().Contains("Pawn"))
          {
            if (!movingPiece.getMoved() && movingPiece.getPanel().Location.Y
                == movingPiecePanel.Location.Y +
                ((movingPiece.getType().Contains("W")) ? -80 : 80))
            {
              foreach (Panel x in generation.getPanels())
              {
                if (x.Location.Y + (movingPiece.getType().Contains("W") ?
                    -40 : 40) == actionPanel.Location.Y && x.Location.X ==
                    actionPanel.Location.X)
                {
                  toUpdate = actionPanel;
                  updatePiece = movingPiece;
                  movingPiece.setPanel(x);
                  updateReq = true;
```

```
                piecePosRemove = new Pawn(updatePiece.getType(),
                                          actionPanel, true);
                generation.AddPiece(piecePosRemove);
                break;
              }
            }
          }
        }

        generation.setPanelsInUse();
        movingPiecePanel.BackgroundImage = null;
        movingPiece.setMoved(true);
        movingPiece.increaseNoMoves();

        //As long as the right person (whose turn it is) is
        //moving then this bit above moves it

        if (castle)  //if the king is attempting to castle
        {
          int action = -1;
          if (actionPanel == generation.getPanels()[6,
                                  movingPiece.getType().Substring(0, 1) ==
                                  "W" ? 7 : 0])
          {
            foreach (Piece x in generation.getPieces())
            {
              if (x.getPanel() == generation.getPanels()[7,
                                  movingPiece.getType().Substring(0, 1) ==
                                  "W" ? 7 : 0])
              {
                movingPiece = x;
                action = 5;
              }
            }
          }
          if (actionPanel == generation.getPanels()[2,
                                movingPiece.getType().Substring(0, 1) ==
                                "W" ? 7 : 0])
          {
            foreach (Piece x in generation.getPieces())
            {
              if (x.getPanel() == generation.getPanels()[0,
                                    movingPiece.getType().Substring(0, 1)
                                    == "W" ? 7 : 0])
              {
                movingPiece = x;
                action = 3;
              }
            }
          }
          actionPanel = generation.getPanels()[action,
                              movingPiece.getType().Substring(0, 1) == "W" ?
                              7 : 0];
          movingPiecePanel = movingPiece.getPanel();
          actionPanel.BackgroundImage =
          movingPiece.getPanel().BackgroundImage;

          undoPiece.Add(movingPiece);
          undoPanel.Add(movingPiece.getPanel());
          noUndos++;
```

```
            movingPiece.setPanel(actionPanel);

            generation.setPanelsInUse();
            movingPiecePanel.BackgroundImage = null;
            movingPiece.setMoved(true);
            movingPiece.increaseNoMoves();

            //moves the castle to thing other side of the king
          }

          if (movingPiece.getType() == "WPawn" ^
              movingPiece.getType() == "BPawn")
          {
            if ((movingPiece.getPanel().Location.Y == 0 &&
                 movingPiece.getType() == "WPawn") ^
                 (movingPiece.getPanel().Location.Y == 280 &&
                 movingPiece.getType() == "BPawn"))
            {
              generation.RemovePiece(movingPiece);
              newQueen = new Queen
                  (movingPiece.getType().Substring(0, 1) +
                  "Queen", movingPiece.getPanel(), true);
              generation.AddPiece(newQueen);
            }
          }
          //If a pawn reaches the end, it becoms a queen

          int totalPossibleMoves = 0;
          foreach (Piece y in generation.getPieces().ToList())
          {
            y.setMoves(generation, true, suicide);
            if ((WhiteMove &&
               y.getType().Substring(0, 1) == "B") ^
               (!WhiteMove &&
               y.getType().Substring(0, 1) == "W"))
            {
              totalPossibleMoves += y.getMoves().Count;
            }

          }
          //Make sure there are some possible moves

          if (!(gameModes == "M3S") && movingPiece.checkCheck(generation))
          {
            MessageBox.Show("Check!");
            //Check check, if it is check, tell the players
            if (totalPossibleMoves == 0)
            {
              //There are no possible moves, it's checkemate
              timer1.Stop();
              MessageBox.Show("Check mate! " + (movingPiece.checkType() == "B"
                                              ? "White" : "Black") + " wins!");
              //EndGame()
            }
          }

          double val = moveCalc.getBoardValue(generation);
          WhiteMove = !WhiteMove;
          movingPiece = null;
          //Next players move and a piece has not been clicked
          if (gameModes == "S1S")
          {
```

```
generation.setPanelsInUse();
actionPanel = moveCalc.bestMove(generation, -10000, 10000);
movingPiece = moveCalc.getMovingPiece();
movingPiecePanel = moveCalc.getMovingPiecePanel();

if (generation.getPanelsInUse().Contains(actionPanel))
{
  foreach (Piece z in generation.getPieces().ToList())
  {
    if (z.getPanel() == actionPanel)
    {
      generation.RemovePiece(z);
      if (z.getType().Substring(0, 1) == "W")
      {
        pictureBoxesB[bPiecesTaken].BackgroundImage =
                                        z.getPanel().BackgroundImage;
        bPiecesTaken++;

      }
      else
      {
        pictureBoxesW[s].BackgroundImage =
                                        z.getPanel().BackgroundImage;
        s++;
      }
    }
  }
  actionPanel.BackgroundImage = null;
  /*If the panel was a possible move and contains a
  piece, the piece that it already contains needs
  to be disposed of and stored in the interface where
  taken pieces are displayed*/

}
if (movingPiece.getType().Contains("King"))
{
  if ((actionPanel == generation.getPanels()[6,
       movingPiece.getType().Substring(0, 1) == "W" ? 7 : 0] ||
       actionPanel == generation.getPanels()[2,
       movingPiece.getType().Substring(0, 1) == "W" ? 7 : 0]) &&
       !movingPiece.getMoved())
  {
    castle = true;
  }
}
//Recognises when the king is trying to castle



if ((WhiteMove &&
    movingPiece.getType().Substring(0, 1).Equals("W")) ||
   (!WhiteMove &&
   movingPiece.getType().Substring(0, 1).Equals("B")))
{
  actionPanel.BackgroundImage =
  movingPiece.getPanel().BackgroundImage;

  undoPiece.Add(movingPiece);
  undoPanel.Add(movingPiece.getPanel());
  noUndos++;
```

```
movingPiece.setPanel(actionPanel);

if (updateReq && movingPiece.getPanel() !=
    updatePiece.getPanel())
{
  updatePiece.setPanel(toUpdate);
  generation.RemovePiece(piecePosRemove);
  updateReq = false;

  //if a pawn has moved foward two last move, it can no longer
    be taken as if it were on the first sqaure, so don't allow
    that to happen.
}
else if (updateReq)
{
  if (movingPiece.getType().Substring(0, 1) == "W")
  {
    pictureBoxesW[wPiecesTaken - 1].BackgroundImage =
                                                Resources.BPawn;
  }
  else
  {
    pictureBoxesB[bPiecesTaken - 1].BackgroundImage =
                                                Resources.WPawn;
  }
  updatePiece.getPanel().BackgroundImage =
                          movingPiece.getPanel().BackgroundImage;
  piecePosRemove.getPanel().BackgroundImage = null;
  generation.RemovePiece(piecePosRemove);
  updatePiece.setPanel(null);

  updateReq = false;

  //If a pawn has moved foward two last move, and a pawn has
    taken it as if it were on the first square, this allows it
    to be taken.
}
if (movingPiece.getType().Contains("Pawn"))
{
  if (!movingPiece.getMoved() &&
      movingPiece.getPanel().Location.Y ==
      movingPiecePanel.Location.Y +
      ((movingPiece.getType().Contains("W")) ? -80 : 80))
  {
    foreach (Panel x in generation.getPanels())
    {
      if (x.Location.Y + (movingPiece.getType().Contains("W") ?
                          -40 : 40) == actionPanel.Location.Y &&
                          x.Location.X == actionPanel.Location.X)
      {
        toUpdate = actionPanel;
        updatePiece = movingPiece;
        movingPiece.setPanel(x);
        updateReq = true;
        piecePosRemove = new Pawn(updatePiece.getType(),
                                  actionPanel, true);
        generation.AddPiece(piecePosRemove);
        break;
      }
    }
  }
}
```

```
generation.setPanelsInUse();
movingPiecePanel.BackgroundImage = null;
movingPiece.setMoved(true);
movingPiece.increaseNoMoves();

//As long as the right person (whose turn it is) is
//moving then this bit above moves it

if (castle)  //if the king is attempting to castle
{
  int action = -1;
  if (actionPanel == generation.getPanels()[6,
                              movingPiece.getType().Substring(0,
                                          1) == "W" ? 7 : 0])
  {
    foreach (Piece x in generation.getPieces())
    {
      if (x.getPanel() == generation.getPanels()[7,
                              movingPiece.getType().Substring(0,
                                          1) == "W" ? 7 : 0])
      {
        movingPiece = x;
        action = 5;
      }
    }
  }
  if (actionPanel == generation.getPanels()[2,
         movingPiece.getType().Substring(0, 1) == "W" ? 7 : 0])
  {
    foreach (Piece x in generation.getPieces())
    {
      if (x.getPanel() == generation.getPanels()[0,
                              movingPiece.getType().Substring(0,
                                          1) == "W" ? 7 : 0])
      {
        movingPiece = x;
        action = 3;
      }
    }
  }
  actionPanel = generation.getPanels()[action,
         movingPiece.getType().Substring(0, 1) == "W" ? 7 : 0];
  movingPiecePanel = movingPiece.getPanel();
  actionPanel.BackgroundImage =
  movingPiece.getPanel().BackgroundImage;

  undoPiece.Add(movingPiece);
  undoPanel.Add(movingPiece.getPanel());
  noUndos++;

  movingPiece.setPanel(actionPanel);

  generation.setPanelsInUse();
  movingPiecePanel.BackgroundImage = null;
  movingPiece.setMoved(true);
  movingPiece.increaseNoMoves();

  //moves the castle to thing other side of the king
}

if (movingPiece.getType() == "WPawn" ^
```

```
                    movingPiece.getType() == "BPawn")
                {
                  if ((movingPiece.getPanel().Location.Y == 0 &&
                      movingPiece.getType() == "WPawn") ^
                      (movingPiece.getPanel().Location.Y == 280 &&
                      movingPiece.getType() == "BPawn"))
                  {
                    generation.RemovePiece(movingPiece);
                    newQueen = new Queen
                        (movingPiece.getType().Substring(0, 1) +
                        "Queen", movingPiece.getPanel(), true);
                    generation.AddPiece(newQueen);
                  }
                }
                //If a pawn reaches the end, it becoms a queen

                totalPossibleMoves = 0;
                foreach (Piece y in generation.getPieces().ToList())
                {
                  y.setMoves(generation, true, suicide);
                  if ((WhiteMove &&
                     y.getType().Substring(0, 1) == "B") ^
                     (!WhiteMove &&
                     y.getType().Substring(0, 1) == "W"))
                  {
                    totalPossibleMoves += y.getMoves().Count;
                  }

                }
                //Make sure there are some possible moves

                if (!(gameModes == "M3S") && movingPiece.checkCheck(generation))
                {
                  MessageBox.Show("Check!");
                  //Check check, if it is check, tell the players
                  if (totalPossibleMoves == 0)
                  {
                    //There are no possible moves, it's checkemate
                    timer1.Stop();
                    MessageBox.Show("Check mate! " + (movingPiece.checkType() ==
                                    "B" ? "White" : "Black") + " wins!");
                    //EndGame()
                  }
                }

                WhiteMove = !WhiteMove;
                movingPiece = null;
              }
              else
              {
                movingPiece = null;
              }
            }

          }
        }
        else
        {
          movingPiece = null;
        }
      }
      catch (NullReferenceException)
```

```
            {
              movingPiece = null;
            }
            undos.Add(noUndos);
            noUndos = 0;

          }
        }
      }
    }


    private void timer1_Tick(object sender, EventArgs e)
    {

      //Each time the timer ticks
      if (timed) // if this is a timed game
      {
        if (WhiteMove) //if it is white's move
        {

          if (WhiteSeconds == 1)
          {
            label4.Text = "00:00";
            MessageBox.Show("White out of time. Black wins!");
            timer1.Stop();
            //If white is out of time, end the game and announce black as the winner
          }
          label4.ForeColor = Color.Red;
          label3.ForeColor = Color.White;
          int seconds = WhiteSeconds % 60;
          int minutes = WhiteSeconds / 60;
          string time = minutes.ToString("00") + ":" + seconds.ToString("00");
          label4.Text = time;
          WhiteSeconds--;

          //decrease the time and display the new time


        }
        else
        {

          if (BlackSeconds == 1)
          {
            label3.Text = "00:00";
            MessageBox.Show("Black out of time. White wins!");
            timer1.Stop();
            //If black is out of time, end the game and announce white as winner.
          }
          label3.ForeColor = Color.Red;
          label4.ForeColor = Color.White;
          int seconds = BlackSeconds % 60;
          int minutes = BlackSeconds / 60;
          string time = minutes.ToString("00") + ":" + seconds.ToString("00");
          label3.Text = time;
          BlackSeconds--;

          //decrease time and display new time


        }
```

```
    }
  }

  private void button1_Click(object sender, EventArgs e)
  {
    MessageBox.Show("White had resigned. Black Wins!");
    timer1.Stop();
    //If the white has resigned, end then game and display the information
  }

  private void button2_Click(object sender, EventArgs e)
  {
    MessageBox.Show("Black had resigned. White Wins!");
    timer1.Stop();
    //If the black has resigned, end then game and display the information
  }

  private void button3_Click(object sender, EventArgs e)
  {
    if (timed)
    {
      timer1.Stop();
    }
    MainMenu menu = new MainMenu();
    menu.Show();
    this.Hide();
    //If quit is pressed, go back to the main menu.
  }

  private void button4_Click(object sender, EventArgs e)
  {
    bool taken = false;

    try
    {
      for (int i = 0; i < undos[undos.Count - 1]; i++)
      {
        movingPiece = undoPiece[undoPiece.Count - (i + 1)];
        if (!generation.getPieces().Contains(movingPiece))
        {
          movingPiece.getPanel().BackgroundImage =
                                              movingPiece.getType().Contains("W") ?
                                  pictureBoxesB[bPiecesTaken  - 1].BackgroundImage :
                                  pictureBoxesW[wPiecesTaken  - 1].BackgroundImage;
          generation.AddPiece(movingPiece);
          if (movingPiece.getType().Contains("W"))
          {
            pictureBoxesB[bPiecesTaken - 1].BackgroundImage = null;
            bPiecesTaken --;
          }
          else
          {
            pictureBoxesB[wPiecesTaken  - 1].BackgroundImage = null;
            wPiecesTaken --;
          }
          //If there was a piece taken before the undo, remove it from the GUI
          taken = true;
        }
        if (movingPiece == updatePiece)
        {
          piecePosRemove.getPanel().BackgroundImage = null;
          piecePosRemove.setPanel(null);
```

```
            generation.RemovePiece(piecePosRemove);
            toUpdate = null;
            updatePiece = null;
            movingPiece.getPanel().BackgroundImage =
                              movingPiece.getType().Contains("W") ? Resources.WPawn :
                                                                    Resources.BPawn;
            //Prepare the sqaure the piece needs to return to if necesary
          }

          movingPiecePanel = undoPiece[undoPiece.Count - (i + 1)].getPanel();
          actionPanel = undoPanel[undoPanel.Count - (i + 1)];
          actionPanel.BackgroundImage = movingPiece.getPanel().BackgroundImage;

          movingPiece.setPanel(actionPanel);
          generation.setPanelsInUse();
          movingPiecePanel.BackgroundImage = null;
          movingPiece.decreaseNoMoves();
          if (movingPiece.numberMoves() <= 0)
          {
            movingPiece.setMoved(false);
          }
          //return everything to the way it was before the move occured
        }
        for (int x = 0; x < undos[undos.Count - 1]; x++)
        {
          undoPiece.Remove(undoPiece[undoPiece.Count - 1]);
          undoPanel.Remove(undoPanel[undoPanel.Count - 1]);
        }

        undos.Remove(undos[undos.Count - 1]);

        if (updateReq || taken)
        {
          panel_MouseDown(null, new MouseEventArgs(MouseButtons.Left, 1, 0, 0, 0));
          updateReq = false;
        }
        //update the undos so that it wont be the same undo over and over again
      }
      catch (System.ArgumentOutOfRangeException)
      {
        MessageBox.Show("You can't go back any further.");
        //if there aren't any undos in the list, you can't go back more
      }
      noUndos = 0;
    }
  }
}
```

Help

Automatically generated coe

King

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Windows.Forms;

namespace CollegeProject
{
```

```
[Serializable]
public class King : Piece
{
  List<Panel> possibleMoves;
  double[,] KingTable;
  bool capturePossible = false;

  public King(string type, Panel image, bool moved) : base(type, image, moved)
  {
    KingTable = new double[8, 8]
        { { -3.0, -4.0, -4.0, -5.0, -5.0, -4.0, -4.0, -3.0 },
                { -3.0, -4.0, -4.0, -5.0, -5.0, -4.0, -4.0, -3.0 },
                { -3.0, -4.0, -4.0, -5.0, -5.0, -4.0, -4.0, -3.0 },
                { -3.0, -4.0, -4.0, -5.0, -5.0, -4.0, -4.0, -3.0 },
                { -2.0, -3.0, -3.0, -4.0, -4.0, -3.0, -3.0, -2.0 },
                { -1.0, -2.0, -2.0, -2.0, -2.0, -2.0, -2.0, -1.0 },
                {  2.0,  2.0,  0.0,  0.0,  0.0,  0.0,  2.0,  2.0 },
                {  2.0,  3.0,  1.0,  0.0,  0.0,  1.0,  3.0,  2.0 }
        };
    if (type.Substring(0, 1) == "B")
    {
      KingTable = new double[8, 8]
      { {  2.0,  3.0,  1.0,  0.0,  0.0,  1.0,  3.0,  2.0 },
                {  2.0,  2.0,  0.0,  0.0,  0.0,  0.0,  2.0,  2.0 },
                { -1.0, -2.0, -2.0, -2.0, -2.0, -2.0, -2.0, -1.0 },
                { -2.0, -3.0, -3.0, -4.0, -4.0, -3.0, -3.0, -2.0 },
                { -3.0, -4.0, -4.0, -5.0, -5.0, -4.0, -4.0, -3.0 },
                { -3.0, -4.0, -4.0, -5.0, -5.0, -4.0, -4.0, -3.0 },
                { -3.0, -4.0, -4.0, -5.0, -5.0, -4.0, -4.0, -3.0 },
                { -3.0, -4.0, -4.0, -5.0, -5.0, -4.0, -4.0, -3.0 }
      };
    }
  }

  public override double getValue(BoardGen board)
  {
    for (int x = 0; x < 8; x++)
    {
      for (int y = 0; y < 8; y++)
      {
        if (board.getPanels()[x, y] == getPanel())
        {
          if (board.getPanels()[x, y] == getPanel())
          {
            double val = (getType().Substring(0, 1) == "W") ? KingTable[x, y] :
                                                              KingTable[y, x];
            return val + 10000;
          }
        }
      }
    }

    return 0;
  }

  public override void setMoves(BoardGen board, bool checkUp, bool suicide)
  {
    possibleMoves = new List<Panel>();
    List<Panel> panelsRemove = new List<Panel>();
    Piece piece = null;
    Panel tempPan = null;
    capturePossible = false;
```

```
//A king can move 1 sqaure in any direction.


foreach (Panel x in board.getPanels())
{
  foreach (Piece c in board.getPieces())
  {
    if (c.getPanel() == x)
    {
      piece = c;
    }
  }
  if (x.Location.X == getPanel().Location.X)
  {
    if (x.Location.Y == getPanel().Location.Y + 40 ||
        x.Location.Y == getPanel().Location.Y - 40)
    {
      possibleMoves.Add(x);
    }
  }
  if (x.Location.X == getPanel().Location.X + 40)
  {
    if (x.Location.Y == getPanel().Location.Y + 40 ||
        x.Location.Y == getPanel().Location.Y - 40 ||
        x.Location.Y == getPanel().Location.Y)
    {
      possibleMoves.Add(x);
    }
  }
  if (x.Location.X == getPanel().Location.X - 40)
  {
    if (x.Location.Y == getPanel().Location.Y + 40 ||
        x.Location.Y == getPanel().Location.Y - 40 ||
        x.Location.Y == getPanel().Location.Y)
    {
      possibleMoves.Add(x);
    }
  }
  /*Similarly to other pieces this section gets all moves that would
    be possible for the king on an empty board*/


  if (piece != null)
  {
    if (piece.getType().Substring(0, 1) == getType().Substring(0, 1) &&
    board.getPanelsInUse().Contains(x))
    {
      panelsRemove.Add(x);
    }
    else
    {

    }
  }
  /*And this just adds all the blocked sqaures to a list of not
    possible moves*/

}
```

```
      foreach (Panel x in panelsRemove)
      {
        possibleMoves.Remove(x);
      }
      bool castle1 = false;
      bool castle2 = false;
      if (!getMoved())
      {
        if (!board.getPanelsInUse().Contains(board.getPanels()[5,
            getType().Substring(0, 1) == "W" ? 7 : 0]) &&
            !board.getPanelsInUse().Contains(board.getPanels()[6,
            getType().Substring(0, 1) == "W" ? 7 : 0]))
        {
          possibleMoves.Add(board.getPanels()[6, getType().Substring(0, 1) == "W" ? 7
                                                                              : 0]);

          castle1 = true;
        }

        if (!board.getPanelsInUse().Contains(board.getPanels()[1,
            getType().Substring(0, 1) == "W" ? 7 : 0]) &&
            !board.getPanelsInUse().Contains(board.getPanels()[2,
            getType().Substring(0, 1) == "W" ? 7 : 0]) &&
            !board.getPanelsInUse().Contains(board.getPanels()[3,
            getType().Substring(0, 1) == "W" ? 7 : 0]))
        {
          possibleMoves.Add(board.getPanels()[2, getType().Substring(0, 1) == "W" ? 7
                                                                              : 0]);

          castle2 = true;
        }
      }

      if (checkUp && !suicide) //if checking for check and this is not suicide chess
                               (as chess doesn't exist in suicide chess)
      {

        Piece toRem = null;
        bool removed = false;
        tempPan = getPanel();
        foreach (Panel x in possibleMoves)
        {
          if (board.getPanelsInUse().Contains(x))
          {
            foreach (Piece z in board.getPieces().ToList())
            {
              if (z.getPanel() == x)
              {
                toRem = z;
                removed = true;
                board.RemovePiece(z);
              }
            }
          }
          setPanel(x);
          board.setPanelsInUse();

          foreach (Piece y in board.getPieces().ToList())
          {
            if (!y.getType().Contains(getType()))
            {
              y.setMoves(board, false, suicide);
```

```
            }
          }

          if (checkCheck(board))
          {
            if (castle1 && x == board.getPanels()[5, getType().Substring(0, 1) == "W"
                                                                              ? 7 : 0])
            {
              panelsRemove.Add(board.getPanels()[6, getType().Substring(0, 1) == "W" ?
                                                                               7 : 0]);
            }
            if (castle2 && x == board.getPanels()[3, getType().Substring(0, 1) == "W"
                                                                              ? 7 : 0])
            {
              panelsRemove.Add(board.getPanels()[2, getType().Substring(0, 1) == "W" ?
                                                                               7 : 0]);
            }
            panelsRemove.Add(x);
          }

          setPanel(tempPan);

          if (removed)
          {
            board.AddPiece(toRem);
            removed = false;
          }

          board.setPanelsInUse();
        }

        foreach (Panel x in panelsRemove)
        {
          possibleMoves.Remove(x);
        }

        foreach (Panel x in possibleMoves)
        {
          foreach (Piece p in board.getPieces())
          {
            if (x == p.getPanel())
            {
              capturePossible = true;
            }
          }
        }



      }
      /* Exactly like all other pieces, this part checks all of the current
         possible moves for the queen to see if any would leave the player
         in check, if it would, it is not a valid move and is removed. */

      if (capturePossible && suicide)  //if it is suicide chess and there is a capture
                                         possible
      {
        List<Panel> newPossibleMoves = new List<Panel>();
        foreach (Panel x in possibleMoves)
        {
          foreach (Piece y in board.getPieces())
          {
```

```
            if (x == y.getPanel())
            {
              newPossibleMoves.Add(x);
            }
          }
        }
        possibleMoves = newPossibleMoves;
        //captures are the only possible moves if a capture is possible
      }
      else if (!capturePossible && suicide) //if there no captures possible in suicide
                                              chess
      {
        foreach (Piece x in board.getPieces())
        {
          if (x.capture() && x.getType().Substring(0, 1) == getType().Substring(0, 1))
          {
            possibleMoves = new List<Panel>();
            break;
            //if there are no possible captures, but there are possible captures from
              other pieces, this piece cannot move
          }
        }
      }
    }

    public override List<Panel> getMoves()
    {
      return possibleMoves;
    }

    public override bool capture()
    {
      return capturePossible;
    }
  }
}
```

Knight

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Windows.Forms;

namespace CollegeProject
{
  [Serializable]
  public class Knight : Piece
  {
    List<Panel> possibleMoves;
    bool capturePossible = false;
    double[,] KnightTable;

    public Knight(string type, Panel image, bool moved) : base(type, image, moved)
    {
      KnightTable = new double[8, 8]
          { { -5.0, -4.0, -3.0, -3.0, -3.0, -3.0, -4.0, -5.0 },
                  { -4.0, -2.0,  0.0,  0.0,  0.0,  0.0, -2.0, -4.0 },
                  { -3.0,  0.0,  1.0,  1.5,  1.5,  1.0,  0.0, -3.0 },
                  { -3.0,  0.5,  1.5,  2.0,  2.0,  1.5,  0.5, -3.0 },
                  { -3.0,  0.0,  1.5,  2.0,  2.0,  1.5,  0.0, -3.0 },
```

```
                { -3.0,  0.5,  1.0,  1.5,  1.5,  1.0,  0.5, -3.0 },
                { -4.0, -2.0,  0.0,  0.5,  0.5,  0.0, -2.0, -4.0 },
                { -5.0, -4.0, -3.0, -3.0, -3.0, -3.0, -4.0, -5.0 }
            };
        }

        public override double getValue(BoardGen board)
        {
          for (int x = 0; x < 8; x++)
          {
            for (int y = 0; y < 8; y++)
            {
              if (board.getPanels()[x, y] == getPanel())
              {
                if (board.getPanels()[x, y] == getPanel())
                {
                  double val = (getType().Substring(0, 1) == "W") ? KnightTable[x, y] :
                                                                    KnightTable[x, y];
                  return val + 30;
                }
              }
            }
          }
          return 0;
        }

        public override void setMoves(BoardGen board, bool checkUp, bool suicide)
        {
          possibleMoves = new List<Panel>();
          Panel tempPan = null;
          capturePossible = false;
          List<Panel> panelsRemove = new List<Panel>();

          /*A knight moves in a L shape pattern, any "L" shape (3 squares in a
          straight line and then one to the left or right).*/

          foreach (Panel x in board.getPanels())
          {
            if ((getPanel().Location.X == x.Location.X + 40 ||
                getPanel().Location.X == x.Location.X - 40) &&
                (getPanel().Location.Y == x.Location.Y + 80 ||
                getPanel().Location.Y == x.Location.Y - 80))
            {
              if (board.getPanelsInUse().Contains(x))
              {

                foreach (Piece y in board.getPieces())
                {
                  if (y.getPanel() == x)
                  {
                    if (getType().Substring(0, 1) !=
                        y.getType().Substring(0, 1))
                    {
                      possibleMoves.Add(x);

                    }
                  }
                }
              }
              else
              {
                possibleMoves.Add(x);
```

```
        }
      }
      if ((getPanel().Location.X == x.Location.X + 80 ||
          getPanel().Location.X == x.Location.X - 80) &&
          (getPanel().Location.Y == x.Location.Y + 40 ||
          getPanel().Location.Y == x.Location.Y - 40))
      {
        if (board.getPanelsInUse().Contains(x))
        {

          foreach (Piece y in board.getPieces())
          {
            if (y.getPanel() == x)
            {
              if (getType().Substring(0, 1) !=
                  y.getType().Substring(0, 1))
              {
                possibleMoves.Add(x);

              }
            }
          }
        }
        else
        {
          possibleMoves.Add(x);
        }
      }
    }

    foreach (Panel x in possibleMoves)
    {
      foreach (Piece p in board.getPieces())
      {
        if (x == p.getPanel())
        {
          capturePossible = true;
        }
      }
    }


    /*This part gets all the possible moves, no need to remove panels being
      blocked as a knight can jump over pieces. The only thing that would
      make it not possible is if a same coloured piece is on the panel, this
      was easy enough to add to the if statements*/

    if (checkUp && !suicide) //if checking for check and this is not suicide chess
                             (as chess doesn't exist in suicide chess)
    {
      Piece toRem = null;
      bool removed = false;
      tempPan = getPanel();
      foreach (Panel x in possibleMoves)
      {
        if (board.getPanelsInUse().Contains(x))
        {
          foreach (Piece z in board.getPieces().ToList())
          {
            if (z.getPanel() == x)
            {
              toRem = z;
```

```
                removed = true;
                board.RemovePiece(z);
              }
            }
          }

          setPanel(x);

          board.setPanelsInUse();

          foreach (Piece y in board.getPieces().ToList())
          {
            if (!y.getType().Contains(getType()))
            {
              y.setMoves(board, false, suicide);
            }
          }

          if (checkCheck(board) && checkType() ==
              getType().Substring(0, 1))
          {
            panelsRemove.Add(x);
          }

          setPanel(tempPan);

          if (removed)
          {
            board.AddPiece(toRem);
            removed = false;
          }

          board.setPanelsInUse();
        }

        foreach (Panel x in panelsRemove)
        {
          possibleMoves.Remove(x);
        }

      }

      /* Exactly like all other pieces, this part checks all of the current
         possible moves for the the bisop to see if any would leave the player
         in check, if it would, it is not a valid move and is removed. */

      if (capturePossible && suicide) //if it is suicide chess and there is a capture
                                      possible
      {
        List<Panel> newPossibleMoves = new List<Panel>();
        foreach (Panel x in possibleMoves)
        {
          foreach (Piece y in board.getPieces())
          {
            if (x == y.getPanel())
            {
              newPossibleMoves.Add(x);
            }
          }
        }
        possibleMoves = newPossibleMoves;
        //captures are the only possible moves if a capture is possible
```

```
      }
      else if (!capturePossible && suicide)  //if there no captures possible in
                                               suicide chess
      {
        foreach (Piece x in board.getPieces())
        {
          if (x.capture() && x.getType().Substring(0, 1) == getType().Substring(0, 1))
          {
            possibleMoves = new List<Panel>();
            break;
            //if there are no possible captures, but there are possible captures from
              other pieces, this piece cannot move
          }
        }
      }

    }
    public override List<Panel> getMoves()
    {
      return possibleMoves;
    }

    public override bool capture()
    {
      return capturePossible;
    }
  }
}
```

Move Calculator

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace CollegeProject
{
  public class MoveCalculator
  {
    Piece piece;
    List<Piece> possibleWPieces;
    List<Piece> possibleBPieces;
    List<Panel> allPossibleBMoves;
    List<Panel> allPossibleWMoves;

    public Panel bestMove(BoardGen board, double alpha, double beta)
    {
      Panel bestMove = null;
      double bestVal = 100000;
      bool removed = false;
      Piece removedPiece = null;
      board.setPanelsInUse();
      allPossibleBMoves = board.getAllPossibleBMoves(board);
      allPossibleWMoves = board.getAllPossibleWMoves(board);
      possibleWPieces = board.getPossibleWPieces();
      possibleBPieces = board.getPossibleBPieces();

      for (var i = 0; i < allPossibleBMoves.Count(); i++)
```

```
      {
        Panel temp = possibleBPieces[i].getPanel();
        if (board.getPanelsInUse().Contains(allPossibleBMoves[i]))
        {
          foreach (Piece y in board.getPieces().ToList())
          {
            if (y.getPanel() == allPossibleBMoves[i])
            {
              removedPiece = y;
              removed = true;
              board.RemovePiece(y);
              //take piece if we're taking a piece
            }
          }
        }
        possibleBPieces[i].setPanel(allPossibleBMoves[i]);
        board.setPanelsInUse();
        board.getAllPossibleWMoves(board);
        board.getPossibleWPieces();
        double tempVal = minimax(board, true, 1, board.getAllPossibleWMoves(board),
                                 board.getPossibleWPieces(), alpha, beta);
        //start a search on this move to see where it leads
        if (tempVal < bestVal)
        {
          bestVal = tempVal;
          beta = tempVal;
          bestMove = allPossibleBMoves[i];
          piece = possibleBPieces[i];
          //if this move leads to a good board value then tempVal will be small/more
          //negative.If it is then we update bestVal, update beta and update the best
          //move and the piece it moves to get there
        }
        possibleBPieces[i].setPanel(temp);
        if (removed)
        {
          removed = false;
          board.AddPiece(removedPiece);
          //return the removed piece to its position
        }
        board.setPanelsInUse();
        //update board
      }
      return bestMove;
      //returns the best move we found
    }

    public double minimax(BoardGen board, bool max, int depth, List<Panel>
                          possibleMoves, List<Piece> possiblePieces, double alpha,
                          double beta)
    {
      double bestValue;
      bool removed = false;
      Piece removedPiece = null;
      board.setPanelsInUse();

      if (depth == 0)
      {
        return getBoardValue(board);
        //We have searched as far as we need to and can return what the board value
        //would be in this position
      }
      if (max) //if the player is trying to maximise score
```

```
        {
          double bestVal = -100000; //set bestVal to some very large negative number so
                                    //that it easily beaten

          for (var i = 0; i < possibleMoves.Count(); i++) //for every move possible from
                                                          //this position
          {
            Panel temp = possiblePieces[i].getPanel();
            if (board.getPanelsInUse().Contains(possibleMoves[i]))
            {
              foreach (Piece y in board.getPieces().ToList())
              {
                if (y.getPanel() == possibleMoves[i])
                {
                  removedPiece = y;
                  removed = true;
                  board.RemovePiece(y);
                  //take piece if we're taking a piece
                }
              }
            }
            possiblePieces[i].setPanel(possibleMoves[i]);
            board.setPanelsInUse();
            board.getAllPossibleWMoves(board);
            board.getPossibleWPieces();
            //update the board for one of the possible moves
            double tempVal = minimax(board, !max, depth - 1,
                                     board.getAllPossibleBMoves(board),
                                     board.getPossibleBPieces(), alpha, beta);
            //dive deeper into a search looking ahead by 1 furhter witht tempVal
            if (tempVal > bestVal)
            {
              bestVal = tempVal;
              alpha = tempVal;

              //if tempVal is better than the current bestVal then this is the new
              //tempVal and alpha is set to tempVal
            }
            possiblePieces[i].setPanel(temp);
            if (removed)
            {
              removed = false;
              board.AddPiece(removedPiece);
              //put taken piece back
            }
            board.setPanelsInUse();

            if (alpha >= beta)
            {
              return bestVal;
              //If alpha is greater or equal to beta we can prune the search
            }

          }
          bestValue = bestVal;
        }
        else
        {
          double bestVal = 100000; //We're minimising so some large value will be easily
                                   //beaten here
          for (var i = 0; i < possibleMoves.Count(); i++)
          {
```

```
            Panel temp = possiblePieces[i].getPanel();
            if (board.getPanelsInUse().Contains(possibleMoves[i]))
            {
              foreach (Piece y in board.getPieces().ToList())
              {
                if (y.getPanel() == possibleMoves[i])
                {
                  removedPiece = y;
                  removed = true;
                  board.RemovePiece(y);
                  //take piece if we're taking a piece
                }
              }
            }
            possiblePieces[i].setPanel(possibleMoves[i]);
            board.setPanelsInUse();
            board.getAllPossibleBMoves(board);
            board.getPossibleBPieces();
            //update the board to do this move
            double tempVal = minimax(board, !max, depth - 1,
                                     board.getAllPossibleWMoves(board),
                                     board.getPossibleWPieces(), alpha, beta);
            //go deeper to see if this move will lead anywhere good
            beta = Math.Min(beta, tempVal);
            if (tempVal < bestVal)
            {
              bestVal = tempVal;
              beta = tempVal;
              //if this move leads somewhere good then tempVal will be less than bestVal
              //as black tries to minimise, so this val is set to bestVal
            }
            possiblePieces[i].setPanel(temp);
            if (removed)
            {
              removed = false;
              board.AddPiece(removedPiece);
              //return the piece that was taken
            }
            board.setPanelsInUse();
            if (alpha >= beta)
            {
              return bestVal;
              //if alpha is greater than beta then we can prune the search a bit
            }

          }
          bestValue = bestVal;
        }
        return bestValue;
        //return the best value we have found
      }

      public Piece getMovingPiece()
      {
        return piece;
      }
      public Panel getMovingPiecePanel()
      {
        return piece.getPanel();
      }

      public double getBoardValue(BoardGen board)
```

```
    {
      double boardValue = 0; //boardValue starts as zero
      foreach (Piece p in board.getPieces().ToList())
      {
        if (p.getType().Substring(0, 1) == "B")
        {
          boardValue = boardValue - p.getValue(board);
          //when there's a black piece the board value becomes smaller/more negative
        }
        else
        {
          boardValue = boardValue + p.getValue(board);
          //otherwise the board value becomes bigger
        }
      }
      return boardValue;
    }
  }
}
```

Pawn

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Windows.Forms;

namespace CollegeProject
{
  [Serializable]
  public class Pawn : Piece
  {
    public List<Panel> possibleMoves;
    bool capturePossible = false;
    double[,] PawnTable;

    public Pawn(string type, Panel image, bool moved) : base(type, image, moved)
    {
      if (type.Substring(0, 1) == "B")
      {
        PawnTable = new double[8, 8]
            { { 0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0 },
                      { 0.5,  1.0,  1.0, -2.0, -2.0,  1.0,  1.0,  0.5 },
                      { 0.5, -0.5, -1.0,  0.0,  0.0,  2.0,  1.0,  0.5 },
                      { 0.0,  0.0,  0.0,  2.0,  2.0,  1.0,  0.5,  0.0 },
                      { 0.5,  0.5,  1.0,  2.5,  2.5,  1.0,  0.5,  0.5 },
                      { 1.0,  1.0,  2.0,  3.0,  3.0,  2.0,  1.0,  1.0 },
                      { 5.0,  5.0,  5.0,  5.0,  5.0,  5.0,  5.0,  5.0 },
                      { 0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0 }
            };
      }
      else
      {
        PawnTable = new double[8, 8]
            { { 0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0 },
                      { 5.0,  5.0,  5.0,  5.0,  5.0,  5.0,  5.0,  5.0 },
                      { 1.0,  1.0,  2.0,  3.0,  3.0,  2.0,  1.0,  1.0 },
                      { 0.5,  0.5,  1.0,  2.5,  2.5,  1.0,  0.5,  0.5 },
                      { 0.0,  0.0,  0.0,  2.0,  2.0,  0.0,  0.0,  0.0 },
                      { 0.5, -0.5, -1.0,  0.0,  0.0, -1.0, -0.5,  0.5 },
                      { 0.5,  1.0,  1.0, -2.0, -2.0,  1.0,  1.0,  0.5 },
```

```
                    { 0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0 }
            };
        }
    }

    public override double getValue(BoardGen board)
    {
        for (int x = 0; x < 8; x++)
        {
            for (int y = 0; y < 8; y++)
            {
                if (board.getPanels()[x, y] == getPanel())
                {
                    double val = (getType().Substring(0, 1) == "W") ? PawnTable[y, x] :
                                                                      PawnTable[y, x];
                    return val + 10;
                }
            }
        }
        return 0;
    }

    public override void setMoves(BoardGen board, bool checkUp, bool suicide)
    {
        Piece piece = null;
        Panel tempPan = null;
        possibleMoves = new List<Panel>();
        List<Panel> panelsRemove = new List<Panel>();
        capturePossible = false;
        bool noMoves = false;

        /*White's pawns can move up the board, Black's move down. If it is
          the first move then a pawn can move two squares, otherwise only
          one. Pawns also take other pieces diagonaly.*/

        foreach (Panel x in board.getPanels())
        {
            foreach (Piece c in board.getPieces())
            {
                if (c.getPanel() == x)
                {
                    piece = c;
                }
            }
            if (piece != null)
            {
                if (getType().Substring(0, 1).Equals("W"))
                {
                    if (board.getPanelsInUse().Contains(x) &&
                        (x.Location.Y == getPanel().Location.Y - 40) &&
                        (x.Location.X == getPanel().Location.X))
                    {
                        noMoves = true;
                        foreach (Panel z in board.getPanels())
                        {
                            if ((((getPanel().Location.Y == z.Location.Y + 40) &&
                            (z.Location.X == getPanel().Location.X - 40)) ||
                            (z.Location.X == getPanel().Location.X + 40)))
                            {
                                noMoves = false;
                            }
                        }
```

```
            }
            else if (!(board.getPanelsInUse().Contains(x)) &&
                    (getPanel().Location.Y == 240) &&
                    ((x.Location.Y == getPanel().Location.Y - 40) ||
                    (x.Location.Y == getPanel().Location.Y - 80)) &&
                    (x.Location.X == getPanel().Location.X))
            {
              possibleMoves.Add(x);
            }
            else if (!(board.getPanelsInUse().Contains(x)) &&
                    (x.Location.Y == getPanel().Location.Y - 40) &&
                    (x.Location.X == getPanel().Location.X))
            {
              possibleMoves.Add(x);
            }
            else if (board.getPanelsInUse().Contains(x) &&
                    (getType().Substring(0, 1) !=
                    piece.getType().Substring(0, 1)))
            {
              if ((getPanel().Location.Y == x.Location.Y + 40 &&
                  (x.Location.X == getPanel().Location.X - 40 ||
                  x.Location.X == getPanel().Location.X + 40)))
              {
                possibleMoves.Add(x);

              }
            }
          }
          else if (getType().Substring(0, 1).Equals("B"))
          {
            if (board.getPanelsInUse().Contains(x) &&
                (x.Location.Y == getPanel().Location.Y + 40) &&
                (x.Location.X == getPanel().Location.X))
            {
              noMoves = true;
              foreach (Panel z in board.getPanels())
              {
                if ((((getPanel().Location.Y == z.Location.Y - 40) &&
                (z.Location.X == getPanel().Location.X - 40)) ||
                (z.Location.X == getPanel().Location.X + 40)))
                {
                  noMoves = false;
                }
              }

            }
            else if (!(board.getPanelsInUse().Contains(x)) &&
                    (getPanel().Location.Y == 40) &&
                    ((x.Location.Y == getPanel().Location.Y + 40) ||
                    (x.Location.Y == getPanel().Location.Y + 80)) &&
                    (x.Location.X == getPanel().Location.X))
            {
              possibleMoves.Add(x);
            }
            else if (!(board.getPanelsInUse().Contains(x)) &&
                    (x.Location.Y == getPanel().Location.Y + 40) &&
                    (x.Location.X == getPanel().Location.X))
            {
              possibleMoves.Add(x);
            }
            else if (board.getPanelsInUse().Contains(x) &&
                    (getType().Substring(0, 1) !=
```

```
                    piece.getType().Substring(0, 1)))
          {
            if ((getPanel().Location.Y == x.Location.Y - 40 &&
                (x.Location.X == getPanel().Location.X - 40 ||
                x.Location.X == getPanel().Location.X + 40)))
            {
              possibleMoves.Add(x);

            }
          }
        }
      }

    }

    /*This section adds all the moves the pawn could make if the board were
      empty to a list called possibleMoves and then all the ones that are
      made invalid due to other pieces are added to the list called
      panelsRemove*/

    foreach (Panel x in panelsRemove)
    {
      possibleMoves.Remove(x);
    }

    foreach (Panel x in possibleMoves)
    {
      foreach (Piece p in board.getPieces())
      {
        if (x == p.getPanel())
        {
          capturePossible = true;
        }
      }
    }


    if (checkUp && !suicide) //if checking for check and this is not suicide chess
                             (as chess doesn't exist in suicide chess)
    {
      Piece toRem = null;
      bool removed = false;
      tempPan = getPanel();
      foreach (Panel x in possibleMoves)
      {
        if (board.getPanelsInUse().Contains(x))
        {
          foreach (Piece z in board.getPieces().ToList())
          {
            if (z.getPanel() == x)
            {
              toRem = z;
              removed = true;
              board.RemovePiece(z);
            }
          }
        }

        setPanel(x);

        board.setPanelsInUse();
```

```
        foreach (Piece y in board.getPieces().ToList())
        {
          if (!y.getType().Contains(getType()))
          {
            y.setMoves(board, false, suicide);
          }
        }

        if (checkCheck(board) && checkType() == getType().Substring(0, 1))
        {
          panelsRemove.Add(x);
        }

        setPanel(tempPan);

        if (removed)
        {
          board.AddPiece(toRem);
          removed = false;
        }

        board.setPanelsInUse();
      }

      foreach (Panel x in panelsRemove)
      {
        possibleMoves.Remove(x);
      }

      if (noMoves)
      {
        possibleMoves = new List<Panel>();
      }
    }

    /* Exactly like all other pieces, this part checks all of the current
       possible moves for the pawn to see if any would leave the player
       in check, if it would, it is not a valid move and is removed. */

    if (capturePossible && suicide)  //if it is suicide chess and there is a capture
                                       possible
    {
      List<Panel> newPossibleMoves = new List<Panel>();
      foreach (Panel x in possibleMoves)
      {
        foreach (Piece y in board.getPieces())
        {
          if (x == y.getPanel())
          {
            newPossibleMoves.Add(x);
          }
        }
      }
      possibleMoves = newPossibleMoves;
      //captures are the only possible moves if a capture is possible
    }
    else if (!capturePossible && suicide) //if there no captures possible in suicide
                                            chess
    {
      foreach (Piece x in board.getPieces())
      {
        if (x.capture() && x.getType().Substring(0, 1) == getType().Substring(0, 1))
```

```
          {
            possibleMoves = new List<Panel>();
            break;
            //if there are no possible captures, but there are possible captures from
              other pieces, this piece cannot move
          }
        }
      }
    }

    public override List<Panel> getMoves()
    {
      return possibleMoves;
    }

    public override bool capture()
    {
      return capturePossible;
    }
  }
}
```

Piece

```
using System;
using System.Collections.Generic;
using System.Drawing;
using System.Windows.Forms;

namespace CollegeProject
{
  public abstract class Piece
  {
    String pieceType;
    Panel piecePanel;
    string typeOfCheck;
    bool hasMoved;
    int noMoves;

    public Piece(String type, Panel image, bool moved)
    {

      image.BackgroundImage = (Image)
          (Properties.Resources.ResourceManager.GetObject(type));

      pieceType = type;
      piecePanel = image;
      hasMoved = moved;
      noMoves = 0;
      //Sets original values of the piece
    }

    public string checkType()
    {
      return typeOfCheck;
    }

    public abstract void setMoves(BoardGen board, bool checkUp, bool suicide);
    //To be overriden by child classes

    public abstract List<Panel> getMoves();
    //To be overriden by child classes
```

```
    public abstract bool capture();

    public String getType()
    {
      return pieceType;
    }

    public void setPanel(Panel newPanel)
    {
      piecePanel = newPanel;
    }

    public Panel getPanel()
    {
      return piecePanel;
    }

    public abstract double getValue(BoardGen board);

    public bool checkCheck(BoardGen board)
    {
      bool check = false;

      Piece piece = null;

      //Need to check all pieces on board because discovered check exists
      foreach (Piece y in board.getPieces())
      {

        List<Panel> possibleMoves = new List<Panel>();
        possibleMoves = y.getMoves();
        if (y.getMoves() != null)
        {
          foreach (Panel x in possibleMoves)
          {

            foreach (Piece c in board.getPieces())
            {
              if (c.getPanel() == x)
              {
                piece = c;
                if (piece.getType().Contains("King") &&
                   (y.getType().Substring(0, 1) !=
                   piece.getType().Substring(0, 1)))
                {
                  typeOfCheck = (piece.getType().Substring(0, 1)
                      == "W") ? "W" : "B";
                  return true;
                }
              }
            }
          }
        }
        //Checks if any piece has a possible move that contains "King",
        //if it does it is check

      }

      return check;

    }
```

```
    public void setMoved(bool move)
    {
      hasMoved = move;
    }

    public bool getMoved()
    {
      return hasMoved;
    }

    public int numberMoves()
    {
      return noMoves;
    }

    public void increaseNoMoves()
    {
      noMoves++;
    }

    public void decreaseNoMoves()
    {
      noMoves--;
    }
  }
}
```

Queen

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Windows.Forms;

namespace CollegeProject
{
  [Serializable]
  public class Queen : Piece
  {
    List<Panel> possibleMoves;
    bool capturePossible = false;
    double[,] QueenTable;

    public Queen(string type, Panel image, bool moved) : base(type, image, moved)
    {
      QueenTable = new double[8, 8]
          { { -2.0, -1.0, -1.0, -0.5, -0.5, -1.0, -1.0, -2.0 },
                  { -1.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0, -1.0 },
                  { -1.0,  0.0,  0.5,  0.5,  0.5,  0.5,  0.0, -1.0 },
                  { -0.5,  0.0,  0.5,  0.5,  0.5,  0.5,  0.0, -0.5 },
                  {  0.0,  0.0,  0.5,  0.5,  0.5,  0.5,  0.0, -0.5 },
                  { -1.0,  0.5,  0.5,  0.5,  0.5,  0.5,  0.0, -1.0 },
                  { -1.0,  0.0,  0.5,  0.0,  0.0,  0.0,  0.0, -1.0 },
                  { -2.0, -1.0, -1.0, -0.5, -0.5, -1.0, -1.0, -2.0 }
          };
    }

    public override double getValue(BoardGen board)
    {
      for (int x = 0; x < 8; x++)
```

```
      {
        for (int y = 0; y < 8; y++)
        {
          if (board.getPanels()[x, y] == getPanel())
          {
            if (board.getPanels()[x, y] == getPanel())
            {
              double val = (getType().Substring(0, 1) == "W") ? QueenTable[x, y] :
                                                                QueenTable[y, x];
              return val + 90;
            }
          }
        }
      }
      return 0;
    }

    public override void setMoves(BoardGen board, bool checkUp, bool suicide)
    {
      possibleMoves = new List<Panel>();
      Panel tempPan = null;
      capturePossible = false;

      /*Queens move like a combination of a rook and bishop. Along a diagnol or
        in a straight line*/

      foreach (Panel x in board.getPanels())
      {
        if ((getPanel().Location.Y == x.Location.Y ||
            getPanel().Location.X == x.Location.X))
        {
          if (board.getPanelsInUse().Contains(x))
          {

            foreach (Piece y in board.getPieces())
            {
              if (y.getPanel() == x)
              {
                if (getType().Substring(0, 1) !=
                    y.getType().Substring(0, 1))
                {
                  possibleMoves.Add(x);
                }
              }
            }
          }
          else
          {
            possibleMoves.Add(x);
          }
        }

      }

      //Gets all the moves a queen could make if the board were empty

      List<Panel> panelsRemove = new List<Panel>();

      foreach (Panel x in board.getPanels())
      {
        if (getPanel().Location.X == x.Location.X &&
            board.getPanelsInUse().Contains(x))
```

```
        {
          if (x.Location.Y > getPanel().Location.Y)
          {
            foreach (Panel g in possibleMoves)
            {
              if (g.Location.Y > x.Location.Y)
              {
                panelsRemove.Add(g);
              }
            }
          }
          else if (x.Location.Y < getPanel().Location.Y)
          {
            foreach (Panel g in possibleMoves)
            {
              if (g.Location.Y < x.Location.Y)
              {
                panelsRemove.Add(g);
              }
            }
          }
        }
        if (getPanel().Location.Y == x.Location.Y &&
            board.getPanelsInUse().Contains(x))
        {
          if (x.Location.X > getPanel().Location.X)
          {
            foreach (Panel g in possibleMoves)
            {
              if (g.Location.X > x.Location.X)
              {
                panelsRemove.Add(g);
              }
            }

          }
          else if (x.Location.X < getPanel().Location.X)
          {
            foreach (Panel g in possibleMoves)
            {
              if (g.Location.X < x.Location.X)
              {
                panelsRemove.Add(g);
              }
            }
          }
        }
      }
      Piece piece = null;

      foreach (Panel x in board.getPanels())
      {
        for (int i = 0; i < 8; i++)
        {
          foreach (Piece c in board.getPieces())
          {
            if (c.getPanel() == x)
            {
              piece = c;
            }
          }
```

```
        if ((getPanel().Location.X == x.Location.X + (40 * i) ||
            getPanel().Location.X == x.Location.X - (40 * i)) &&
            (getPanel().Location.Y == x.Location.Y + (40 * i) ||
            getPanel().Location.Y == x.Location.Y - (40 * i)))
        {
          if (board.getPanelsInUse().Contains(x))
          {

            foreach (Piece y in board.getPieces())
            {
              if (y.getPanel() == x)
              {
                if (getType().Substring(0, 1) !=
                    y.getType().Substring(0, 1))
                {
                  possibleMoves.Add(x);
                }
              }
            }
          }
          else
          {
            possibleMoves.Add(x);
          }
        }
      }

    }
    foreach (Panel x in board.getPanels())
    {
      for (int i = 0; i < 8; i++)
      {
        if ((getPanel().Location.X == x.Location.X + (40 * i) ||
            getPanel().Location.X == x.Location.X - (40 * i)) &&
            (getPanel().Location.Y == x.Location.Y + (40 * i) ||
            getPanel().Location.Y == x.Location.Y - (40 * i)) &&
            board.getPanelsInUse().Contains(x))
        {
          if (x.Location.Y < getPanel().Location.Y &&
              x.Location.X < getPanel().Location.X)
          {
            foreach (Panel g in possibleMoves)
            {
              if (g.Location.Y < x.Location.Y &&
                  g.Location.X < x.Location.X)
              {
                panelsRemove.Add(g);

              }
            }
          }
          else if (x.Location.Y < getPanel().Location.Y &&
              x.Location.X > getPanel().Location.X)
          {
            foreach (Panel g in possibleMoves)
            {
              if (g.Location.Y < x.Location.Y &&
                  g.Location.X > x.Location.X)
              {
                panelsRemove.Add(g);

              }
```

```
            }
          }
          else if (x.Location.Y > getPanel().Location.Y &&
                   x.Location.X < getPanel().Location.X)
          {
            foreach (Panel g in possibleMoves)
            {
              if (g.Location.Y > x.Location.Y &&
                  g.Location.X < x.Location.X)
              {
                panelsRemove.Add(g);
              }
            }
          }
          else if (x.Location.Y > getPanel().Location.Y &&
              x.Location.X > getPanel().Location.X)
          {
            foreach (Panel g in possibleMoves)
            {
              if (g.Location.Y > x.Location.Y &&
                  g.Location.X > x.Location.X)
              {
                panelsRemove.Add(g);
              }
            }
          }
        }
      }
    }

    foreach (Panel x in panelsRemove)
    {
      possibleMoves.Remove(x);
    }

    foreach (Panel x in possibleMoves)
    {
      foreach (Piece p in board.getPieces())
      {
        if (x == p.getPanel())
        {
          capturePossible = true;
        }
      }
    }


    //Removes any moves blocked by another piece.

    if (checkUp && !suicide) //if checking for check and this is not suicide chess
                             (as chess doesn't exist in suicide chess)
    {
      Piece toRem = null;
      bool removed = false;
      tempPan = getPanel();
      foreach (Panel x in possibleMoves)
      {
        if (board.getPanelsInUse().Contains(x))
        {
          foreach (Piece z in board.getPieces().ToList())
          {
            if (z.getPanel() == x)
```

```
            {
              toRem = z;
              removed = true;
              board.RemovePiece(z);
            }
          }
        }

        setPanel(x);

        board.setPanelsInUse();

        foreach (Piece y in board.getPieces().ToList())
        {
          if (!y.getType().Contains(getType()))
          {
            y.setMoves(board, false, suicide);
          }

        }

        if (checkCheck(board) && checkType() == getType().Substring(0, 1))
        {
          panelsRemove.Add(x);
        }

        setPanel(tempPan);

        if (removed)
        {
          board.AddPiece(toRem);
          removed = false;
        }

        board.setPanelsInUse();
      }

      foreach (Panel x in panelsRemove)
      {
        possibleMoves.Remove(x);
      }
    }

    /* This part checks all of the current possible moves for the
       the bisop to see if any would leave the player in check, if it
       would, it is not a valid move and is removed. */

    if (capturePossible && suicide) //if it is suicide chess and there is a capture
                                    possible
    {
      List<Panel> newPossibleMoves = new List<Panel>();
      foreach (Panel x in possibleMoves)
      {
        foreach (Piece y in board.getPieces())
        {
          if (x == y.getPanel())
          {
            newPossibleMoves.Add(x);
          }
        }
      }
      possibleMoves = newPossibleMoves;
```

```
        //captures are the only possible moves if a capture is possible
      }
      else if (!capturePossible && suicide) //if there no captures possible in suicide
                                              chess
      {
        foreach (Piece x in board.getPieces())
        {
          if (x.capture() && x.getType().Substring(0, 1) == getType().Substring(0, 1))
          {
            possibleMoves = new List<Panel>();
            break;
            //if there are no possible captures, but there are possible captures from
              other pieces, this piece cannot move
          }
        }
      }
    }
    public override List<Panel> getMoves()
    {
      return possibleMoves;
    }

    public override bool capture()
    {
      return capturePossible;
    }
  }
}
```

Rook

```
using System;
using System.Collections.Generic;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace CollegeProject
{
    public class Rook : Piece
    {

        public List<Panel> possibleMoves;
        Panel tempPan = null;
        bool capturePossible = false;
        double[,] RookTable;

        public Rook(string type, Panel image, bool moved) : base(type, image, moved)
        {
            RookTable = new double[8, 8]
                { {  0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0 },
                  {  0.5,  1.0,  1.0,  1.0,  1.0,  1.0,  1.0,  0.5 },
                  { -0.5,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0, -0.5 },
                  { -0.5,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0, -0.5 },
                  { -0.5,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0, -0.5 },
                  { -0.5,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0, -0.5 },
                  { -0.5,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0, -0.5 },
                  {  0.0,  0.0,  0.0,  0.5,  0.5,  0.0,  0.0,  0.0 }
                };
```

```
        }

        public override double getValue(BoardGen board)
        {
            for (int x = 0; x < 8; x++)
            {
                for (int y = 0; y < 8; y++)
                {
                    if (board.getPanels()[x, y] == getPanel())
                    {
                        if (board.getPanels()[x, y] == getPanel())
                        {
                            double val = (getType().Substring(0, 1) == "W") ?
                                          RookTable[x, y] : RookTable[y, x];
                            return val + 50;
                        }
                    }
                }
            }
            return 0;
        }

        public override void setMoves(BoardGen board, bool checkUp, bool suicide)
        {
            possibleMoves = new List<Panel>();
            capturePossible = false;
            //Rooks can move in a straigh line, up, down, left and right.
            foreach(Panel x in board.getPanels()) {
                if ((getPanel().Location.Y == x.Location.Y ||
                    getPanel().Location.X == x.Location.X))
                {
                    if (board.getPanelsInUse().Contains(x))
                    {

                        foreach(Piece y in board.getPieces())
                        {
                            if(y.getPanel() == x)
                            {
                                if(getType().Substring(0, 1) !=
                                   y.getType().Substring(0 , 1))
                                {
                                    possibleMoves.Add(x);
                                }
                            }
                        }
                    }
                    else
                    {
                        possibleMoves.Add(x);
                    }
                }
            }

            /*This part just gets all of the panels with the same x or y value as
              the rook in use*/

            List<Panel> panelsRemove = new List<Panel>();

            foreach(Panel x in board.getPanels())
            {
                if (getPanel().Location.X == x.Location.X &&
                    board.getPanelsInUse().Contains(x))
```

```
            {
                if (x.Location.Y > getPanel().Location.Y)
                {
                    foreach (Panel g in possibleMoves)
                    {
                        if (g.Location.Y > x.Location.Y)
                        {
                            panelsRemove.Add(g);
                        }
                    }
                }
                else if (x.Location.Y < getPanel().Location.Y)
                {
                    foreach (Panel g in possibleMoves)
                    {
                        if (g.Location.Y < x.Location.Y)
                        {
                            panelsRemove.Add(g);
                        }
                    }
                }
            }
            if (getPanel().Location.Y == x.Location.Y &&
                board.getPanelsInUse().Contains(x))
            {
                if (x.Location.X > getPanel().Location.X)
                {
                    foreach (Panel g in possibleMoves)
                    {
                        if (g.Location.X > x.Location.X)
                        {
                            panelsRemove.Add(g);
                        }
                    }

                }
                else if (x.Location.X < getPanel().Location.X)
                {
                    foreach (Panel g in possibleMoves)
                    {
                        if (g.Location.X < x.Location.X)
                        {
                            panelsRemove.Add(g);
                        }
                    }
                }
            }
        }
        /*This part adds any moves that aren't possible due to a piece blocking
          the move to an array that is removed from possible moves*/


        foreach(Panel x in panelsRemove)
        {
            possibleMoves.Remove(x);
        }

        foreach(Panel x in possibleMoves)
        {
            foreach(Piece p in board.getPieces())
            {
                if (x == p.getPanel())
```

```
                {
                    capturePossible = true;
                }
            }
        }

        if (checkUp && !suicide) //if checking for check and this is not suicide
                                 chess (as chess doesn't exist in suicide chess)
        {
            Piece toRem = null;
            bool removed = false;
            tempPan = getPanel();
            foreach (Panel x in possibleMoves)
            {
                if (board.getPanelsInUse().Contains(x))
                {
                    foreach (Piece z in board.getPieces().ToList())
                    {
                        if (z.getPanel() == x)
                        {
                            toRem = z;
                            removed = true;
                            board.RemovePiece(z);
                        }
                    }
                }

                setPanel(x);

                board.setPanelsInUse();

                foreach (Piece y in board.getPieces().ToList())
                {
                    if (!y.getType().Contains(getType()))
                        y.setMoves(board, false, suicide);
                }

                if (checkCheck(board) && checkType() ==
                    getType().Substring(0, 1))
                {
                    panelsRemove.Add(x);
                }

                setPanel(tempPan);

                if (removed)
                {
                    board.AddPiece(toRem);
                    removed = false;
                }

                board.setPanelsInUse();
            }

            foreach (Panel x in panelsRemove)
            {
                possibleMoves.Remove(x);
            }
        }

        /* Exactly like all other pieces, this part checks all of the current
           possible moves for the the bisop to see if any would leave the player
```

```
                in check, if it would, it is not a valid move and is removed. */

            if (capturePossible && suicide) //if it is suicide chess and there is a
                                            capture possible
            {
                List<Panel> newPossibleMoves = new List<Panel>();
                foreach (Panel x in possibleMoves)
                {
                    foreach (Piece y in board.getPieces())
                    {
                        if (x == y.getPanel())
                        {
                            newPossibleMoves.Add(x);
                        }
                    }
                }
                possibleMoves = newPossibleMoves;
                //captures are the only possible moves if a capture is possible
            }
            else if (!capturePossible && suicide) //if there no captures possible in
                                                  suicide chess
            {
                foreach (Piece x in board.getPieces())
                {
                    if (x.capture() && x.getType().Substring(0, 1) ==
                        getType().Substring(0, 1))
                    {
                        possibleMoves = new List<Panel>();
                        break;
                        //if there are no possible captures, but there are possible
                          captures from other pieces, this piece cannot move
                    }
                }
            }

        }

        public override List<Panel> getMoves()
        {
            return possibleMoves;
        }

        public override bool capture()
        {
            return capturePossible;
        }

    }
}
```

Rules

Automatically generate code