

Computer Science NEA

Table of Contents

Analysis	8
Features that make the problem solvable by computational methods	8
Identifying suitable stakeholders.....	9
Research into problem	21
Identifying essential features of solution	26
Identifying limitations of solution	28
Hardware and software requirements	28
Success criteria	29
Design	33
Decomposition	33
Structure of solution	37
Algorithms	39
Show board.....	39
Select move.....	41
Make move	42
Undo move	43
Update GUI	44
Show moves	45
Reset buttons.....	46
Show winner	47
End game	48
Move validation	49
Determining check	55
Determining checkmate.....	56
Computer to move.....	57
Usability features	66
Key variables, data structures, classes & validation.....	68
Testing	70
Implementation	77
Stages of development	77
Creating main data structures	78

Development	78
Testing	81
Remedial	83
Review	83
Creating initial GUI	84
Development	84
Testing	86
Remedial	86
Review	86
Allowing user to select pieces	87
Development	87
.....	87
.....	87
.....	87
.....	87
Testing	87
.....	87
.....	87
Review	87
Allowing user to make a move	88
Development	88
Testing	89
Review	89
Turn JLabel	90
Development	90
Testing	90
.....	90
Review	90
Showing possible moves	91
Development	91
Testing	92
Review	92
Knight move validation	93
Development	93
Testing	93
King move validation	94

Development	94
Testing	94
Rook move validation	95
Development	95
Testing	96
Bishop move validation.....	97
Development	97
Testing	98
Queen move validation.....	99
Development	99
Testing	100
Pawn move validation.....	101
Development	101
Testing	102
Review	102
.....	102
Implementing turns.....	103
Development	103
Testing	103
.....	103
Review	104
Showing opponents previous move	105
Development	105
Testing	105
Remedial.....	105
.....	105
Review	106
Determining check	107
Development	107
Review	111
Preventing user making moves that put / keep them in check.....	112
Development	112
Testing	113
Remedial	113
Review	115
Telling the user if they are in check	116

Development	116
Testing	116
Review	117
Checkmate and ending the game	118
Development	118
Testing	119
Remedial	120
Review	121
Allowing user to end the game	122
Development	122
Testing	122
Review	122
Selecting opponent	123
Development	123
Testing	124
Review	125
.....	125
Allowing computer to move	126
Development	126
Remedial	128
Testing	133
Remedial	134
Review	138
Allowing user to undo moves	139
Development	139
Testing	139
Review	140
.....	140
Pawn promotion	141
Development	141
Testing	142
Remedial	143
Testing	144
Remedial	144
Testing	146
Remedial	146

Review	146
.....	146
Showing user instructions.....	147
Development	147
Testing	147
Review	148
.....	148
Changing colours of GUI.....	149
Development	149
Testing	152
Review	155
Saving games.....	156
Development	156
Testing	157
Loading previously saved games.....	158
Development	158
Testing	159
Review	160
Changing look of GUI.....	161
Development	161
Testing	163
Remedial	163
Testing	164
Development	164
Testing	165
Review	166
Evaluation.....	167
Post-development testing.....	167
Black box testing	167
Beta Testing	208
.....	212
Acceptance testing.....	213
Evaluation of success criteria and usability features.....	216
Future maintenance and limitations	255
Full code	256
Bishop.java.....	256

.....	256
.....	256
Chess.java	257
ChessBoard.java.....	272
Help.java	290
King.java.....	291
Knight.java	292
MoveHistory.java.....	293
Pawn.java.....	294
Piece.java	295
Queen.java.....	296
Rook.java	298
SelectPlayer.java	299
ShowWinner.java.....	301
Images.....	302

Analysis

Features that make the problem solvable by computational methods

I am going to produce a program that will allow a user to play a game of chess on a computer, against another person, or against an AI opponent.

Chess is a two-player board game in which players take it in turns to move pieces (8x pawns, 2x rooks, 2x knights, 2x bishops, 1x queen, 1x king, at the beginning of the game) around on an 8 x 8 board, according to specific rules (depending on the type of piece, and whether you are in check), in order to try and put the opponent's king into checkmate (which is when the king is directly under attack, and cannot make a move to escape). If a player's opponent is in checkmate, that player wins the game. One player uses black pieces, whilst the other uses white. The player using the white pieces takes the first move, and it alternates from there. Players are not allowed to not move a piece on their turn, even if making a move would be harmful to their strategy. If a player is in check (and not checkmate), their king is under attack / could be captured on the opponents next turn, but it is possible for the player to make a move that brings them out of check. If in check, a move is only legal if it brings you out of check. A player cannot make a move that puts themselves into check. Players can capture any of their opponent's pieces, except their opponent's king. There is no ambiguity around whether a move is legal or not, or whether a player is in check or checkmate, and so a computer could determine these things, and much faster / more accurately than a human would be able to, making it amenable to a computational approach.

One feature that my solution will need to include is a graphical user interface, so that the user can easily provide input to the program, and visualise the chess board. This is suited to a computational approach, since it would allow the user to play a game of chess without having to use a physical board / pieces, which they may not have (making the game accessible to more people, and cheaper), may be easily lost or damaged (which cannot happen if played on a computer), and may take a long time to set up at the beginning of each game (which could happen automatically / much more quickly on a computer). In addition, carrying a board & pieces around is not very practical – a computational approach would allow the user to play anywhere, as long as they have access to a computer. Furthermore, a GUI could be easily customised to suit the preferences of the client, for example a specific colour scheme could be used, which would be much more difficult to do with an actual board & pieces. The user could make a move by pressing a button to select which piece to move, and another button to choose where to move it to. Their input would need to be validated – for example to check if they have selected one of their own pieces, and that the move they are making is legal.

Another feature that my solution will need is the ability to determine whose turn it is. This is suited to a computational approach, since a variable could be used to keep track of whose turn it is currently, and logic implemented to not allow the wrong player to make a move / for a player to move pieces other than their own. If chess were played with a physical board, there is the potential for the players to forget whose turn it is / for a player to make a move when they shouldn't – a computational approach would prevent this from happening.

An additional feature that my solution will need is the ability to play against an AI opponent. This is inherently suited to a computational approach, since it involves playing 'against the computer'. It allows the user to play a game of chess, even if they have no one else to play against. The user needs to decide at the beginning of the game whether they would like to play against another human, or

the AI. If against another human, the program will allow each player to make a move, alternating between white and black, until the game ends. If against the AI, the AI needs to make a move after each move the player makes, unless the game ends.

A further feature that my solution will require is the ability to determine whether a move is legal or not. This is suited to a computational approach, as algorithms could be implemented to check this when a user clicks a button to make a move. If a move is not legal, it should not happen and the user could be told of this, and allowed to make a different move, preventing them from accidentally making a move that is not legal, preventing cheating, and helping users who may not know the rules of chess learn how to play the game. When a user clicks the piece that they would like to move, the GUI could show the user all possible (legal) places to move that piece to, for example by highlighting them in a different colour, helping the user to visualise which moves are legal and which are not. On the other hand, when playing with a physical board, a player could accidentally perform a move that is not legal, and they would not be shown which moves they could make, which could make it more difficult to play, especially for a beginner.

Another feature that my solution will need is the ability to determine whether someone is in check or checkmate. This is suited to a computational approach, since algorithms could be implemented in different functions to quickly determine whether someone is in check / checkmate, and the GUI updated to notify the user of this, determining this much quicker than a player may be able to using a physical board, and preventing the user from accidentally not noticing that they are in check / checkmate, which could potentially happen if playing chess on a physical board, meaning that the player might not move out of check, or might not notice that the game has finished.

Due to these features described above, I believe that the problem is amenable to a computational approach.

I will represent the board using a class called `Board`, which has a 2D array of pieces that consists of 8 rows and 8 columns, and methods to move pieces, determine whether someone is in check / checkmate, switch turns, get current state of the board, etc. A value of null in the array would indicate that the space is empty. Since the user will interact with the board through an 8x8 grid of buttons, they will not be able to go out of bounds of the array. Each different type of piece could have its own class, each of which inherit from a class called `Piece`. Each would have a colour ('w' or 'b') and a type ('p', 'r', 'n', 'b', 'q', 'k'), and methods to determine whether a particular move is valid, and to get/set the colour/type.

Identifying suitable stakeholders

One of my stakeholders is my client – my brother Ben, who is 17 years old. He enjoys playing chess in his free time, and would like to be able to quickly play a game of chess on his own, in order to improve his skills, and to play against friends at college, in order to pass time during breaks. The suggested solution to this is a digital version of chess – this would allow him to play against an AI (without the requirement of an internet connection) and improve his skills, and to play against his friends at college on his laptop or on a computer.

Here is a log of the conversation I had with my client, so that I could learn more about what they expect from the solution, and to ensure that the solution made will meet their requirements.

Q: Why would you like a digital version of chess?

“Because it would allow me to quickly play a game of chess against my friends at college, without having to carry a chess board and pieces around with me, since I have a lot of books to carry with me”

each day, and a chess board would take up too much room in my bag. Also, I don't want to risk losing any pieces of the game. A digital version would mean I wouldn't have to worry about losing anything, and it would make it a lot quicker for me to start playing, as I don't need to spend any time setting up the board at the start of each game, and putting it away at the end, which wastes time when my breaks at college are quite short. Also, if I have to finish a game early, for example if I need to go to a lesson, with an actual board, I can't come back to it later to finish it, and so a digital version that enables this would be very helpful."

Q: Would you like the program to include a graphical user interface?

"Yes, because I think that interacting with the program via the console would make the game more difficult to play and less appealing."

Q: What basic functionality do you expect the program to have?

"I expect it to let me play a game of chess, either as a two-player or one-player mode, and to make sure that any moves made are legal & that the rules are followed, and to know when someone is in check or checkmate, or if the game has ended. It should also tell you who the winner is."

Q: Are there any specific features you would like the solution to include?

"I would like the solution to include the ability to play against the computer, so I can play even if I am not with my friends, and to be able to save a game so that I could come back to it again later."

Q: What features do you not like about existing chess programs?

"I don't like how many existing programs are online and so require an internet connection, since the Wi-Fi at my college is very slow."

Q: Is there a specific type of design or colour scheme you would like the program to include?

"I want the design of the program to be relatively simple / not too complex, so that it does not distract from the focus of the game, and so that it can be easily seen, even if on a smaller laptop screen."

Q: Do you think that an explanation of how to play the game is necessary to be included?

"I think it should be included just in case others don't know how to play the game, or if they are slightly unsure of the rules."

Q: Would you like to be able to play over the internet with others?

"No – I would primarily like to play this game with my friends in person on a computer, as that would be more social, or to play a game quickly on my own, so I don't have to wait for someone else to take their move."

Q: Would you like to be able to "undo" moves?

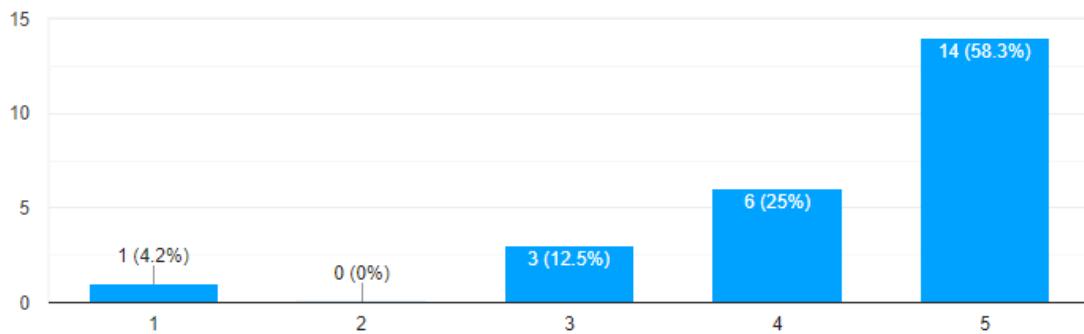
"Yes, just in case the user accidentally presses the wrong thing and accidentally make a move they didn't mean to, or if they make a move that doesn't turn out to be very good, so that they can go back and change their mind."

The target audience for this program will be 16-19 year old college students. In order to find out what features my end-users would want, and what their opinions are on the design of the program, I created a survey and asked them to fill it out.

Firstly, I asked them how important a graphical user interface is. This is to help me determine how much time needs to be spent on the design of the GUI.

How important is a graphical user interface?

24 responses

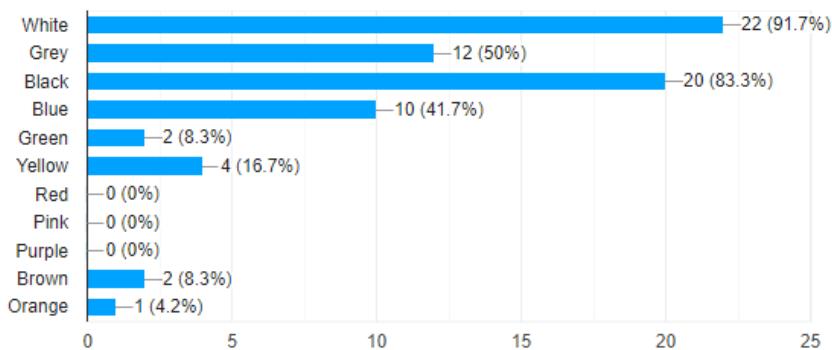


From the results, I can see that the end-users do feel that a GUI is important; 58.3% feel that it is “very important”. Therefore, I will need to make sure that a lot of time is spent ensuring that the GUI is user-friendly, and meets the needs of the end users. I could ask for their feedback on the GUI when I begin creating it, to ensure that it will appeal to them.

Secondly, I asked them what colours they would like to be used in the program. This is to help me determine which should be used in the program, and which should be avoided.

What colour scheme would you like the program to use? (can select multiple colours)

24 responses



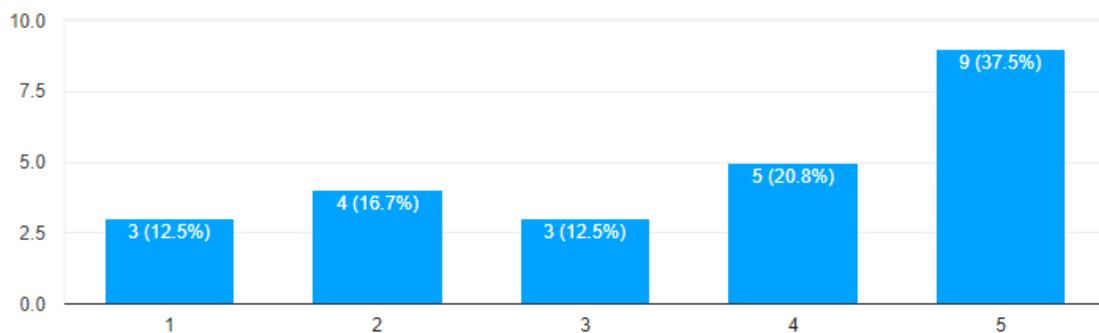
From the results, I can see that the neutral colours white, grey and black are the most popular, and that blue and yellow are also fairly popular. Therefore, I will try and aim to use these within the

program. In addition, red, pink and purple all had 0 votes, so I should try and avoid using these colours.

Next, I asked how confident they are with the rules of chess. This is to help me determine whether features such as highlighting possible legal moves on the board when a piece is selected, and telling the user for example when they are in check would be necessary.

How confident are you with the rules of chess?

24 responses

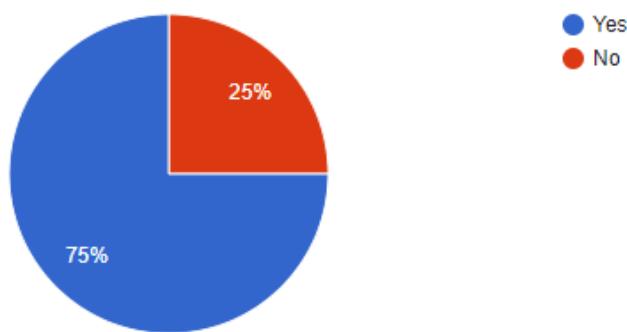


Whilst there are 9 people who are very confident with the rules, there were a lot of responses that indicate they are not entirely sure, so I think that the features mentioned above should be included, as they will help those who do not know the rules, or aren't 100% sure of them, to be able to play the game.

Then, I asked whether a written explanation of the rules would be useful to them, in order to determine whether this should be included in the program or not.

Would a written explanation of the rules of chess be useful to you?

24 responses

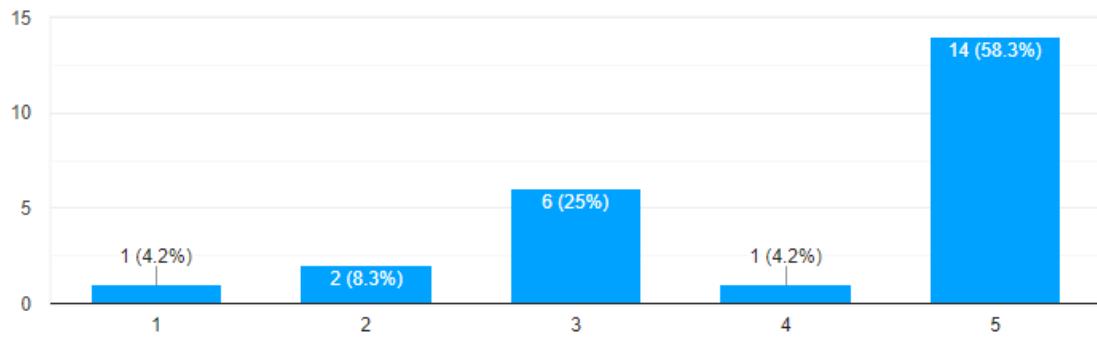


From the results, I can see that 75% would find this useful, and so is something that should be included in the program. Interestingly, whilst 9 people in the previous question said they were "very confident" with the rules of chess, only 6 people said that a written explanation of the rules would not be useful to them, showing that an explanation of the rules would be useful to even those who are very confident with them.

Then, I asked the end-users whether the ability to save a game to an external file, and to load a game in would be useful to them, to determine whether this functionality should be included.

How useful would it be to be able to save a game to an external file & to load previous games in from a file?

24 responses

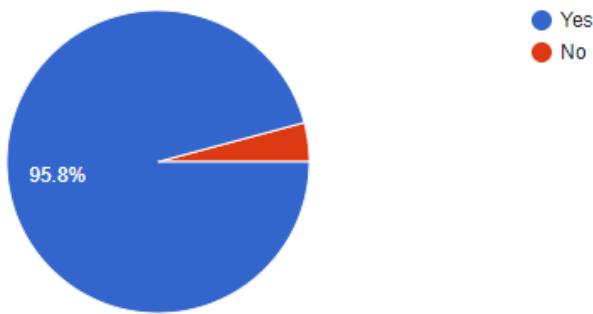


From the results, I can see that 58.3% of respondents would find this “very useful”, and so this is something that I should aim to include in the program.

After that, I asked them whether they would like to be able to resign a game, so that I can determine whether this needs to be included in the program or not.

Would you like to be able to resign / surrender a game?

24 responses



From the results, I can see that the overwhelming majority of them would like to be able to do this, so this is definitely something that should be included in the program.

Next, I decided to ask for their opinions on the design of existing chess programs, so that I could get a better idea of the features they would like to see, and which they dislike, so that I can more easily design a GUI that is suited to their needs.

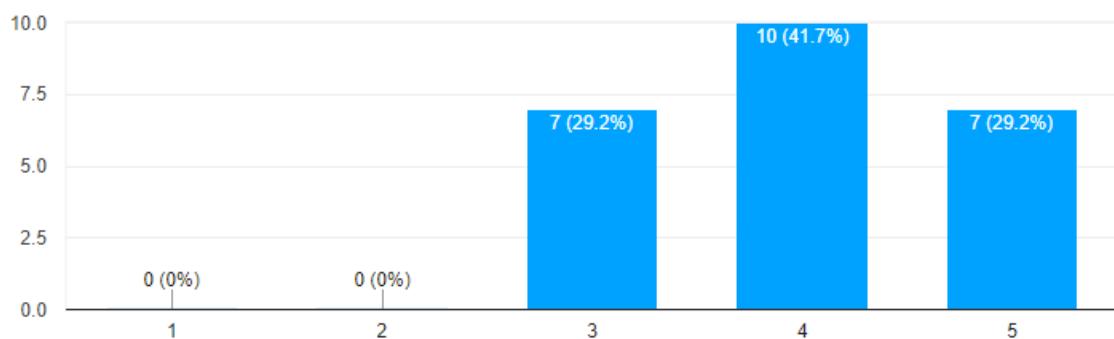
For each, I asked them to rate on a scale from 1 to 5 how much they like the design of the program, and then gave them the option to state any features they like, and any they don't.

Firstly, I asked about the following program, from the website www.chess.com :



How much do you like the design of this program?

24 responses



As you can see from the results, the design of this program was very well liked by the end-users, and so is something which I could take inspiration from.

The following are comments made on what they like about the design:

"the colours of the pieces stand out on the background"

"Colourful chess board and clear pieces"

"The icons"

"white and green is cool, my dude"

"quite simple and clear board"
"it's easy to distinguish between pieces"
"Clear, concise"
"simple, clean, contrasting colours, intuitive icons for controls"
"Graphics are clear in representing each piece, interface fills the screen suitably and looks large enough to be accessible to those with poor eyesight."
"It looks clean and simple, easy to interact with and navigate."
"The simplicity and the clarity"
"Pieces simple, big board, clarity"
"Clean, easy to tell what's going on, nice pretty colours."
"Minimalistic but still intuitive and full of features"
"neat and very simple to use"
"I like the design and colour scheme of the chess board. Very nice simple layout"
"It shows what is important clearly: the pieces on the board"

From this, I can see that they would like a program that is simple / minimalist, where it is easy & clear to see what is going on, intuitive to use, and for the colours used to contrast well, and so I should ensure that my program contains these features.

The following are comments made on what they dislike about the design:

"The colour scheme"
"is not white and blue"
"unclear icons and low contrast"
"the side part seems unnecessary/ the game should take up the centre space"
"text too small"
"Difficult to tell exactly what to do upon first inspection, and no obvious help or instructions menu (to me anyway)"
"It looks a bit generic; there're lots of programs that look exactly like this."
"The green foes not work well with the cream"
"Green & white"
"I don't understand what the buttons do e.g back and forth button, do they allow you to go back on a position or do they move to the next or previous opponent"
"I'm not very sure what to click"

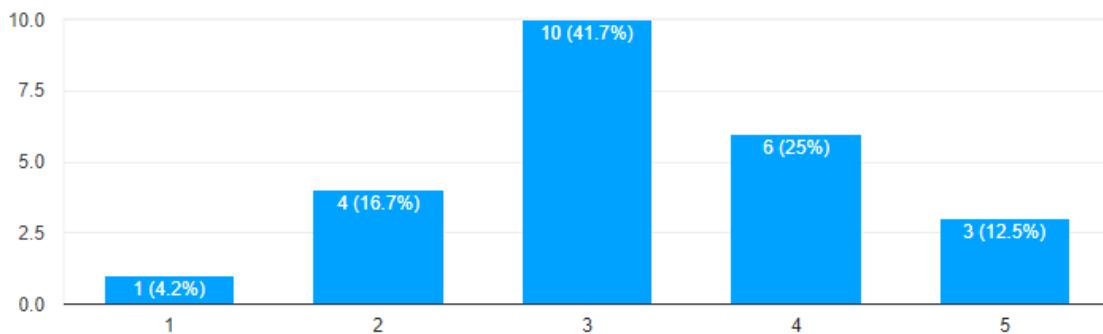
From this, I can see that a few of the users dislike the white & green colour scheme. Therefore, in my program, I will avoid this by using the colours that they have previously stated they would like to be used. Furthermore, some of them think that the design of the program is not intuitive enough, or that the function of some buttons is confusing, so I should try and ensure that my program is as intuitive and easy to use as possible. Also, I should ensure that the text / icons used in my program are large enough to be easily seen by all, and that it takes up the centre of the screen.

Secondly, I asked for their opinions on the following program, from www.mathsisfun.com/games/chess.html :



How much do you like the design of this program?

24 responses



From the results, I can see that there is a much more mixed opinion on this one, and that 41.7% neither like nor dislike it.

Here are some of the features that they like about the design:

<i>"the colours behind the piece that you have just played"</i>
<i>"Clear characters"</i>
<i>"The icons"</i>
<i>"is white and brown"</i>
<i>"very simple and good contract board. nice buttons"</i>
<i>"the chessboard is centre of the screen"</i>
<i>"Clear"</i>
<i>"highlights last move clearly"</i>
<i>"Makes moves clear and obvious, available menu options are intuitive and obvious"</i>
<i>"It looks simply."</i>
<i>"The pieces are very well designed and the board colours are complementary to each other"</i>
<i>"Piece clarity, background colours, simplicity"</i>
<i>"Clean, easy to tell what's going on, nice pretty colours."</i>
<i>"I like the description box showing you where u moved at the bottom"</i>

From this, I can see that again they would like a simplistic design that is easy to see / clear & contrasts well, with the board in the centre of the screen. Furthermore, they seem to like the highlighting of the opponent's previous move, and the box showing all previous moves, and so I should try and include these in my program.

Here are some features that they dislike about the program:

"the outline on the white pieces is too harsh for the background colour"
"Not very colourful"
"The colour scheme"
"is not white and blue"
"the designs for the pieces don't look as good"
"Too basic,"
"highlights colour fading just seems off"
"Move tracker contains no indication of which pieces were moved and from where, no obvious instructions / help menu"
"It looks a bit clunky; like the ui of an iphone pre the iPhone 5c era."
"The outline of the pieces are a bit too thick, especially on the white"
"I dislike the colours of green and yellow showing how you moved"

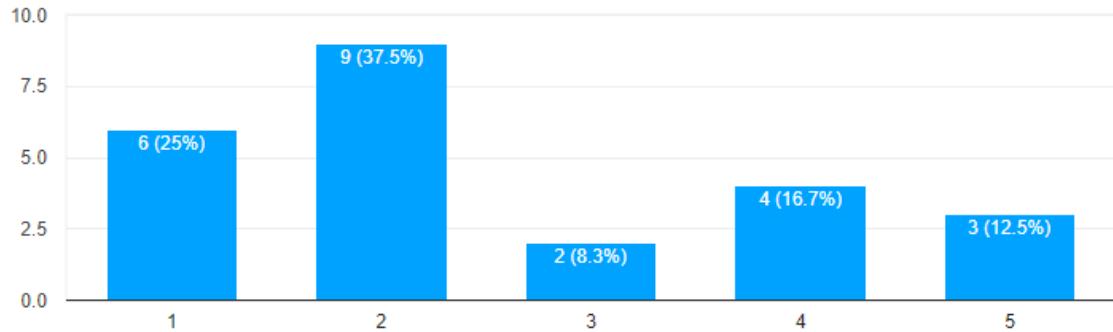
From this, I can see that they do not like the designs of the pieces very much, and so I should try and use designs that look different to these in my program. They also dislike the use of the green / yellow on the board, and so I should try and use more complimentary colours in my program. They would also like clearer instructions / a help menu, and so I should include this in my program.

Lastly, I asked for their opinion on the following program, from <https://plainchess.timwoelfle.de>



How much do you like the design of this program?

24 responses



From the results, I can see that this program is not as well liked as the previous two, and so could act more as an example of what not to include in mine.

Here are some features that they like about the program:

“the background colour”
“The pieces are clear and the board has border”
“The wood effect”
“colour scheme”
“chessboard is centre stage”
“Timer”
“compact”
“Clearly keeps track of turn, timer, moves, interface is uncluttered and polished”
“It gives a cute kind of nostalgic vibe.”
“There is a nice boarder”
“Clean, easy to tell what's going on, nice pretty colours.”
“You know this one looks sophisticated :)”
“I love the colours, design and the the features at the bottom showing important details like the time”

From this, I can see that they like how information such as whose turn it is, what the last move was, and the time elapsed is displayed in a clear, understandable and easy to see way, and so I could try and implement something like this in my program. Furthermore, they also like the border, and some like the colour scheme used, which is similar to that you would expect to find on a physical chess board.

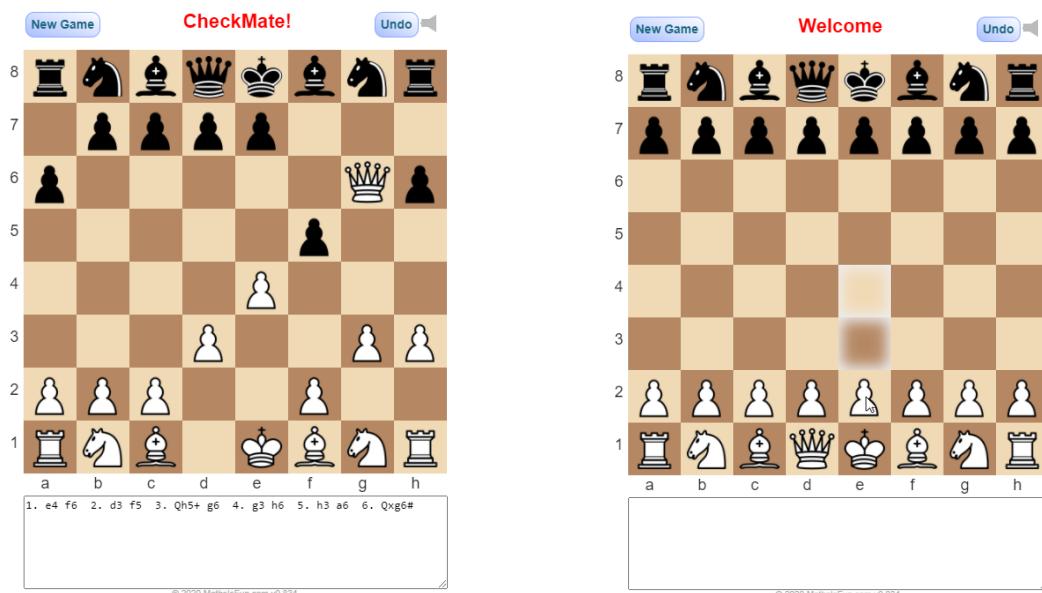
And here are some features that they dislike about the program:

<i>"the pieces blend in with the background because they aren't solid colours"</i>
<i>"It looks a little boring and colour could make it more interesting"</i>
<i>"The curved board squares and the boldness of the icons"</i>
<i>"ugly"</i>
<i>"far too much detail in pieces. don't like outlines"</i>
<i>"Colour scheme"</i>
<i>"feels cluttered due to the colours and there is no highlighting of last square; not very user friendly"</i>
<i>"Not personally a fan of the more detailed aesthetics, preferred simple designs, and the rounded corners on each square don't work for me"</i>
<i>"I don't like the brown and it looks a bit muddy / dull."</i>
<i>"The board squares are strange and the pieces are too detailed (distracting)"</i>
<i>"Pieces are not clear/simple enough and the colour scheme is too dramatic"</i>
<i>"Icons look ugly. Can't really explain why idk. Also each square is rounded which looks messy"</i>
<i>"symbols are too complex... might mismove incorrect piece"</i>
<i>"More clickable features? To click on? Like next opponent or 'give up'"</i>
<i>"looks old fashioned"</i>

From this, I can see that they do not like how detailed the icons are, and how they blend into the board, and so in my program I should aim to use simpler icons that stand out more. They also do not like how there is no option for example to undo a move, or to resign, and so I should aim to include these in my program. In addition, many do not like how the squares are rounded, and so I should avoid using rounded squares. Also, they would prefer a more modern design, rather than old-fashioned. Whilst some people liked the brown colours used in this, as seen from the previous question, the results from this question show that some people do not like it. Perhaps to overcome this, I could include the ability to choose a colour scheme, so that more users will like it.

Research into problem

The screenshot shows the MathsIsFun.com chess game interface. At the top, there are buttons for "New Game", "Check" (which is highlighted in red), and "Undo". The chessboard is shown with black pieces on light squares and white pieces on dark squares. The board is numbered 1 to 8 on the vertical axis and a-h on the horizontal axis. A legend at the bottom left indicates piece types: King (red crown), Queen (blue crown), Rook (red castle), Bishop (blue diamond), Knight (red horse), and Pawn (red or blue triangle). The board shows a standard starting setup. On the right, a text box displays a sequence of moves: 1. e4 f6 2. d3 f5 3. Qh5+. Below the board, a note says "Play against the computer or a friend. Highlights possible moves for each piece." On the left, player settings are shown: Black player is set to "Computer (Beginner)" and "Endless"; White player is set to "Human" and "Endless"; and Pieces are set to "Standard". There are also "New Game" and "Check" buttons.



Link to website: <https://www.mathsisfun.com/games/chess.html>

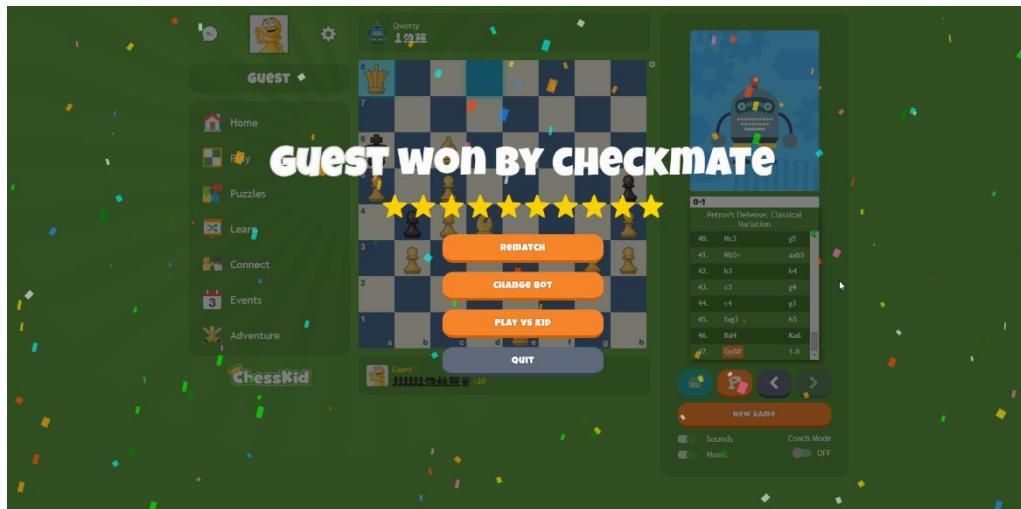
When you first visit the page, you are asked who you would like to play as each colour (from the options: human, computer (beginner), computer (novice), computer (skilled), computer (hard), computer (ruthless)), whether you would like there to be a time limit for either player, and what style of pieces you would like to use. This allows you to play against another person, to play against the computer, or even have two ‘computer’ players play against each other. The fact that you can play against different skill levels is good, since it allows a wide range of people to enjoy the game – from beginner to advanced, meaning it appeals to more people. The option to change the type of pieces used also offers some customisability to the user, which again could make it appeal to more people. When you start the game, white moves first, however one negative is that this game does not explicitly tell you whose turn it is, which could be confusing, especially if you are playing against another human, and you forgot whose turn it is. To make a move, you drag and drop the piece you would like to move. Whilst you are holding a piece with the mouse, it highlights in white the possible places you can move it to. If you are unable to move the piece, nothing shows. If you attempt to

move a piece to a place it is not allowed, it will simply return back to its original place, and you can make a different move. After each turn, a textbox underneath the board is updated to show all of the moves that have been made so far, written in algebraic chess notation. At the top of the board, text is used to notify the user if they are in check or checkmate. At any time, you can press the “New Game” button to finish the current game, and begin a new one. Also, there is an “Undo” button, which allows you to take back a move you have just made, which is useful if you have made a mistake, however could make the game too easy. It has a simple layout, and does not overwhelm the user with lots of information, and I would like to include a similarly minimalistic layout in my solution. It is very straightforward to play if you are familiar with chess, however for beginners, it may be slightly confusing, as it offers no written explanation of the rules, and no guidance on whose turn it is. I would like to take inspiration from the relatively simplistic layout of this game, however I would also like to make it more user-friendly, especially for new players, for example by offering an explanation of the rules, and providing slightly more information to the user, on things such as whose turn it is, without compromising the simplistic design.

Advantages: Very simple design – not cluttered, purpose of each button is easy to understand, shows previous moves taken in the game, shows possible moves a piece can take when it is selected, clearly tells the user when they are in check or checkmate, pieces contrast well with board / stand out, can play against both human and AI.

Disadvantages: Colour scheme may be seen as boring to some users, no help or explanation of how to play for beginners.





Link to website: <https://www.chesskid.com/play/computer>

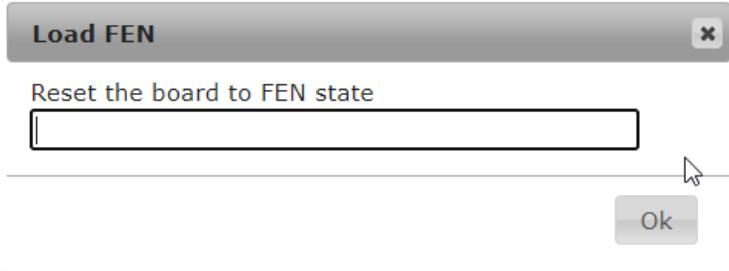
As with the last game, when you first visit the website, you are asked what difficulty of bot you would like to play against, and whether you would like to play as white or black. This has similar features to the previous game; however, it has a very different design and user interface. However here, you can make a move both by clicking the piece to move, and the location to move it to, and by dragging and dropping the piece. One downside of this game is that it does not show you which moves are legal and which are not, which may make the game more difficult for new users, however this website does give an explanation of how to play the game, which is something that I could include in my solution. This makes use of brighter colours – e.g. the green background, and orange buttons, which may make it appeal more to younger users, however it may not make the game appeal more to older users. Whilst I prefer this design to the previous, I would like to make sure that my solution appeals to a wider range of ages than children. In addition, if you win the game, it shows animated confetti, which again may make the game appeal more to younger users, but would be difficult to incorporate into my solution in the given time restraints. One feature of this program that I like, and would like to include in my design, is a way of clearly showing to the user the move the computer has made, by highlighting it in a different colour (here, red), since if this was not included, the user may not be sure what move the computer has made. This game also features a “coach

mode”, which would help players who are new to the game learn how to play & improve, which would make the game appeal more to those who would like to get better at playing chess, but again, this would also be very difficult to incorporate into my solution, in the given time frame.

Advantages: Can choose to play against either someone else or the computer, shows all previous moves taken, has a colourful yet fairly minimalistic design, pieces are easy to see on the board, has many features intended to help those new to the game (explanations of rules / tactics, a “coach mode”), shows opponents previous move in a different colour on the board.

Disadvantages: Seems to be designed for younger users – may not appeal to the target audience of my solution, 16-19 year olds, doesn’t show the user which moves would be possible for a particular piece.

The image consists of three screenshots of the quickchess.net website. The top screenshot shows a standard 8x8 chessboard with black pieces on the top row and white pieces on the bottom row. A white pawn is highlighted with a red dashed square on its square. The middle screenshot shows a dropdown menu under the 'Game' button with options: New Game, Difficulty (selected), Undo Move, Show FEN, and Load FEN. The 'Difficulty' option has a hand cursor icon over it. The bottom screenshot shows a modal window titled 'Current Position - FEN' containing the FEN string: rnbqkbnr/pppp1ppp/8/4p3/4P3/8/PPPP1PPP/RNBQKBNR w KQkq e6 3 2. Below the FEN string is a note: 'Use this URL to return to the game later: http://quickchess.net/?fen=rnbqkbnr%2Fppp1ppp%2F8%2F4p3%2F4P3%2F8%2FPPPP1PPP%2FRNBQKBNR%20w%20KQkq%20e6%203%202'.



Link to website: <https://quickchess.net/>

When you first visit this website, you are immediately greeted with a chess board in the default state. There is no need to press a button or anything similar in order to start the game, you just start moving pieces. This has a very minimalistic user interface – with just the board, and a table on the right showing previous moves. This program uses a greyscale colour scheme, which whilst means that the pieces contrast well with the board and can be seen clearly, could potentially be seen as boring. The end-users I surveyed did say that they would like to see these colours in the final solution, however they also commented that the reason they did not like some of the other programs was because of the lack of colour. In order to make a move, you drag and drop the pieces. One downside is that it does not show you which moves would be allowed for that piece, however it does not allow you to make an illegal move, which is good, as it forces the user to play the game properly. Another downside is that there is no clear way to see whose turn it is, and it does not show you the previous move of the opponent. Whilst you can undo a move, the option to do so is hidden in a menu, and so may not be immediately obvious to the user, which is not very user friendly. One good feature of this program is that it allows you to save the current position of the board so that you can come back to it later, and to load in previous board states, which is something that my end users have said they would find useful, and is something that I would like to include in my solution as well. This program achieves this functionality through the use of Forsyth-Edwards Notation (FEN), which provides the all information necessary on things such as the state of the board, whose turn it is to move, the ability for castling / en passant, and the number of moves taken, in order to start a game. Another downside of this program is that it only allows you to play against the computer, and not against another human. In my program, I must include the functionality to do both, as this is something the client has requested.

Advantages: Contains a variety of different features (e.g. to undo a move, to load a previous board), very simple & minimalistic design, design of pieces is simple and contrasts well with the board.

Disadvantages: Only allows you to play against the computer, and not against another human, doesn't have any explanation of the rules, doesn't show possible moves, functionality of buttons may not be clear to some users (e.g. they may not know what "FEN" means).

As a summary, here is a list of features seen in the other products I have researched that I would like to include in my solution:

A way of letting the user know whether they are in check or checkmate, making the game easier for the user, as they do not have to determine whether or not they are in check themselves. Furthermore, this must be presented in a clear and easy to see way, to ensure that the user is able to notice this.

An undo button to take back a move just in case a bad move was made, or the user accidentally moves the wrong piece.

The ability to play both against another human, and against the computer, so that the user can play both against someone else (e.g. with friends at college), or on their own.

Checking whether a move is legal, and ensuring that illegal moves cannot be made, preventing cheating.

Showing the user possible legal moves, by highlighting in a different colour, helping particularly those who may not be sure how each piece moves.

Showing the user the previous move of their opponent, by highlighting this in a different colour.

A simple & minimalistic / uncluttered design, where the purpose of any buttons is intuitive and easy to understand for all users.

A way to save the current state of the game / load in a previous game, so that a user can come back to a previous game at another time if they would like to.

Pieces should not blend in / contrast well with the board, and the different types of pieces should be easy to distinguish from each other, so that the user does not accidentally select the wrong piece, and so that those using devices with a small or dim screen, or those with poor eyesight, can use the program.

Identifying essential features of solution

My solution will be a program which will firstly allow the user to select whether they would like to play against another human or an AI, and show them the board. It will then allow them to play a game of chess, with each side taking turns to make a move, by selecting the piece they would like to move, and where they would like to move it to. This will continue until either one player resigns, or the game comes to an end (i.e. one player is in checkmate). They will then be given the option of again choosing who they would like to play against, and then starting a new game, with the board reset.

The following is a list of features that are essential to my solution:

A graphical user interface (GUI). This is essential, since my client has specifically requested a GUI, and the survey for the end-users shows that they think this is a very important part of the program. Alternatively, the user could interact with the program through a console, however this would likely be much more difficult for the user, and more time consuming. They could, for example, enter the location of the piece they would like to move, and where they wanted to move it to, or they could enter the move they would like to make in a particular format (e.g. algebraic chess notation). Each of these would be much more prone to error and less user friendly than interacting with the program via a GUI. Additionally, the GUI must have a minimalistic design, which is easy to understand, and use colours that allow everything to be easily seen / contrast well, as this is also something that both my client and the end users have requested. The end-users have also said that they would like the board to take up the centre of the screen, for any buttons to be intuitive and unambiguous, and for there to be a clear way of seeing the previous moves taken, and so my GUI must include this. The GUI should also make use of the colours that the end-users have specified they would like to see in the program.

The ability for the user to input their move. This is essential, as it is the main way in which the user will need to interact with the program. The board will be represented by an 8x8 grid of buttons, each representing a space on the board. The user should be able to click a button, to select the piece that

they would like to move, and then select another button to choose where they would like to move that piece to.

The ability to determine whether a move is legal or not. This is essential, since it is one of the basic features that my client expects the program to include. If there is no way to validate moves, then users would be able to cheat, or accidentally make a wrong move without realising; the fact that there is no way to automatically validate moves when using a physical chess board is one of the reasons a computational solution was being made in the first place. This is also important for the AI – it must only be allowed to make legal moves.

When making a move, as the user selects the piece they would like to move, the program should find all the possible legal moves for that piece, and show all the possible locations for that piece to the user by highlighting these in a different colour on the board. If there are no available legal moves for the selected piece, this could also be shown to the user, for example by highlighting the selected piece in another different colour. This is essential, since many of the end-users who were surveyed were not completely sure of the rules of chess, and so this would help them be able to play the game correctly, and could help them to learn the rules of the game.

The ability to determine whether a player is in check or not, and to notify the user of this in a clear way. Part of determining whether a move is legal is determining whether the player is in check (in which case they would need to make a move that brings them out of check), or whether making a particular move would put them directly in check (in which case, the move is not legal). This is essential, as detecting this and notifying the user prevents them from accidentally not noticing that they are in check, which was another reason a computational solution was wanted, and is a feature that the client has stated they are expecting. Also, the program would be able to determine whether a player is in check potentially much quicker than a human could. This would especially help those who are new to chess, who may not be sure whether they are in check or not.

The ability to determine whether a player is in checkmate. This is essential, since if a player is in checkmate, the game should end, and so the program needs a way to determine whether the game has finished or not, otherwise it would continue on forever. After each move, the program must check whether either side is in checkmate, and if so, tell the user this & tell them who has won (the player not in checkmate), and stop the game.

The ability to reset the board and start a new game after the current game has finished. This is essential, as otherwise the user would have to close and re-open the program each time they wanted to play a new game, which is not very practical, and would take much longer for the user than just resetting the board.

The ability to resign a game. This would allow the user to stop the game before it would otherwise come to an end, for example if they believe they are going to lose, or if they have run out of time. This is essential, as it is something that the end-users have specified they would like to be included in the program from the survey.

The ability to undo a move. This would allow the user to take back a move, for example if they accidentally move the wrong piece or move a piece to the wrong location, or if they make a move that turns out to not be very good. Again, this is essential because it is a feature that the end-users have requested, and also something which my client has requested.

Showing the opponent's previous move in a different colour on the board. This is essential, as it is a feature that the end-users seemed to like in other examples of chess programs, and so is something which should be included in my solution.

The ability to save the current state of the board to an external file, and to load one in. This could be achieved through the use of FEN notation, which provides all the information required to start a game. This is essential, as it is something that the end-users have said would be very useful, and is something that my client has said they would like to be included.

A way of keeping track of whose turn it is, and the ability to switch turns. This is essential, since keeping track of whose turn it is means that this can be shown to the user, so that they can avoid the problem of forgetting who should move next, which was a reason a computational solution was needed. Only allowing the player whose turn it is to move prevents a player cheating by missing out their own move, or from taking multiple moves in a row.

The ability to play against an AI opponent. This is essential, as it is one of the main features that my client has requested, and is one of the main reasons a computational approach was needed.

Identifying limitations of solution

This project is limited in time. Therefore, there are some restrictions on what will be included in the final solution.

For example, the graphics will not be in 3D, since this would be very difficult to implement in the given time, and is not something that the client or end users have specifically requested, and so would not be worth spending additional time on.

Another limitation is that the game will not be playable over a network / be connected to a server. This is because it would increase the complexity of the program by a lot, and so the time needed to create it by a lot also, and internet connectivity is something that the client does not want, since he would prefer to play in-person against others, rather than over the internet, as this is more social, and doesn't require both players to have access to a device at once. However, if more time were available, this would be a good feature to implement, as it would give the end users the opportunity to play against each other on different devices, if this is something they wanted to do.

Another limitation is that the pieces will be selected by pressing a button, rather than by dragging and dropping them. This is because creating the functionality to drag and drop pieces could be very time consuming, and would leave less time to focus on the features that the client would find more important, which are also time consuming, namely the AI.

Hardware and software requirements

In order to be able to play the game, for the hardware, the user would require a PC or laptop to be able to run the game, a mouse or other similar device (e.g. trackpad) to be able to input data to the program (since the GUI will consist mostly of buttons), and some sort of screen or monitor, to be able to actually see the user interface.

To create this game, I will be using Java. Therefore, in order to be able to run the program, the user must have Java, and meet the following system requirements (which are the requirements in order to run the Java Virtual Machine, taken from the following website:

java.com/en/download/help/sysreq.html):

Windows:

- Windows 10 (8u51 and above)
- Windows 8.x (Desktop)
- Windows 7 SP1
- Windows Vista SP2
- Windows Server 2008 R2 SP1 (64-bit)
- Windows Server 2012 and 2012 R2 (64-bit)
- RAM: 128 MB
- Disk space: 124 MB for JRE; 2 MB for Java Update
- Processor: Minimum Pentium 2 266 MHz processor

Mac OS X:

- Intel-based Mac running Mac OS X 10.8.3+, 10.9+
- Administrator privileges for installation

Linux:

- Oracle Linux 5.5+1
- Oracle Linux 6.x (32-bit), 6.x (64-bit)2
- Oracle Linux 7.x (64-bit)2 (8u20 and above)
- Red Hat Enterprise Linux 5.5+1 6.x (32-bit), 6.x (64-bit)2
- Red Hat Enterprise Linux 7.x (64-bit)2 (8u20 and above)
- Suse Linux Enterprise Server 10 SP2+, 11.x
- Suse Linux Enterprise Server 12.x (64-bit)2 (8u31 and above)
- Ubuntu Linux 12.04 LTS, 13.x
- Ubuntu Linux 14.x (8u25 and above)
- Ubuntu Linux 15.04 (8u45 and above)
- Ubuntu Linux 15.10 (8u65 and above)

In order for me to create the program, I will be using NetBeans, and so I will need to make sure that I have the NetBeans IDE and the correct Java Development Kit version.

Success criteria

Essential

Usability

Desirable

AI

#	Criteria	Description	Why is it required?
1	Graphical User Interface (GUI) shows board	GUI contains an 8x8 grid of buttons representing the board, in the centre of the screen.	Since it is something that my client has requested, and is something that the end-users have said they feel is very important. Using for example a command line interface would be much

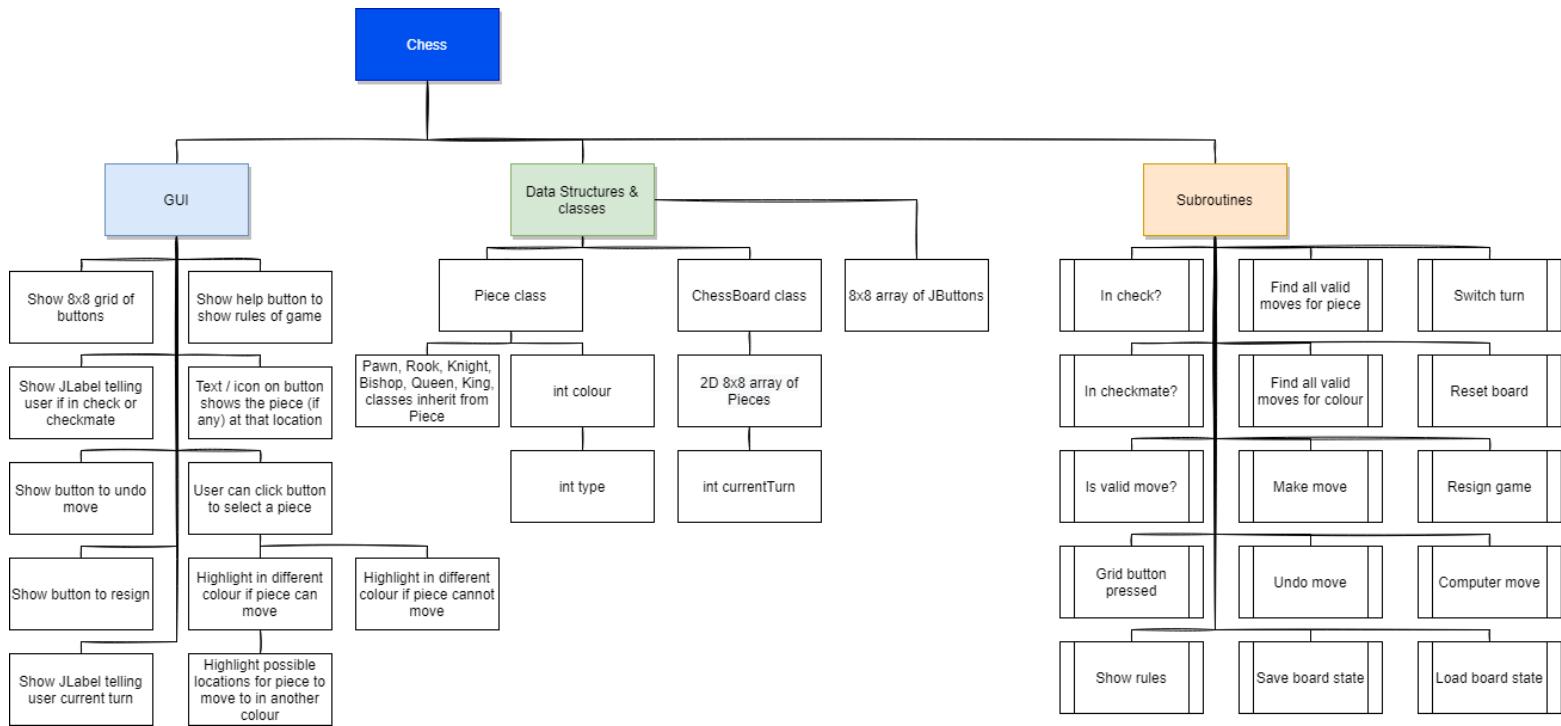
			more time consuming and wouldn't be user-friendly.
2	GUI has a minimalistic design	GUI contains only features required, and is uncluttered. Could ask the client on their opinions of the design, and whether it is simple enough.	Because both my client and end-users would like the GUI to have a simple design, and so including this ensures that the program will appeal to the end-users.
3	GUI uses the colours the end-users have specified	GUI uses colours from the following: white, grey, black, blue, green, yellow, brown and orange.	So that the design of the program will appeal to the end-users, and so they would want to play the game.
4	GUI uses distinctive images to represent pieces	Icons must contrast well with the colour of the board, and not have a thick outline.	So that the users will be able to easily see their pieces on the board, avoiding them accidentally selecting the wrong piece, and ensuring those using a smaller screen or those with poor eyesight can see them easily.
5	User can choose who to play against	At the beginning of the game, the user can select whether they would like to play against an AI or another human.	So that the users can either play against another person or the computer, depending on whether they would like to play in-person against someone else, or on their own – not restricted to just one of the two options.
6	User can input their move	User can select the piece they would like to move, and the location they would like to move the piece to.	So that the user can actually interact with the program and is able to input their move, in order to play the game.
7	Program can switch turns	Starting with white, after each move is taken, a variable keeping track of whose turn it is must switch to the other player. Only the player's pieces whose turn it currently is should be able to move.	So that the user doesn't have to keep track of whose turn it is themselves, and prevents cheating by stopping one player from taking multiple moves at once, or from not taking a move.
8	Show previous move of opponent	After opponent has made a move, the starting and ending location of the piece they moved should be shown in a different colour on the board.	Because it is something that the end-users have said that they liked in other programs, and so should be included in my solution.
9	AI can play	AI can determine which move would it should take out of all	Because the client has requested that they can play both against another human and against an AI, depending on

		possible legal moves that they could possibly take.	whether they want to play against someone else or play on their own.
10	Can find all possible legal moves for a particular piece	Function created that returns all of the possible legal moves for a certain piece, based on the current state of the game.	So that the possible legal moves for a particular piece when selected could be shown to the user on the board by highlighting them in a different colour, helping those who may not know how each piece moves. Furthermore, this is essential in determining whether someone is in check or checkmate, and for the AI to be able to select a move.
11	Only makes a move if it is legal	If a move is made that is legal, it should be allowed. Otherwise, the user should be able to input a different move.	Because it would help those users who may not know the rules be able to learn the game, as they would be prevented from making illegal moves accidentally, and would also prevent users from knowingly taking illegal moves and cheating.
12	Can determine whether either side is in check	A function created returning a boolean that returns true if the king is directly under attack by the other side, and false otherwise.	So that it can be determined whether a move is legal or not, since if not in check, a move is only legal if it does not put the player in check, and if in check, a move is only legal if it brings you out of check
13	Can tell the user if they are in check	A JLabel should be used which informs the player when they are in check.	So that the user can quickly and easily tell whether or not they are in check, without having to determine this for themselves (which may be particularly difficult for those who are very new to the game)
14	Can determine whether either side is in checkmate	Uses the previous functions of determining whether a player is in check, and to determine the possible legal moves to determine whether they are in check, and cannot move out of check.	So that the program is able to determine whether the game should finish or not – otherwise it could just carry on
15	Can inform the user who has won	A JLabel should be used to inform the user when either they or their opponent are in checkmate, informing them who has won the game.	So that the user is able to quickly and easily see who is in checkmate, and so who has lost, without having to determine whether or not it is actually checkmate for themselves

16	Allows user to resign	A button should be in the program, which when pressed stops the current game, resets the board, and allows the user to start a new game.	Because this is a feature that my end-users have said they would like to be included in the program
17	Allows user to undo a move	A button should be in the program, which when pressed, undo's the previous move of the player.	Since this is something which both my end users and client have requested
18	Allows user to save current board state to a file	The user should be able to press a button / menu item to export the game to an external text file, using FEN notation.	Since this is something that many of the end-users who were surveyed said they would find useful
19	Allows user to load previous board state from a file	The user should be able to load a previous game into the program, again using FEN notation.	This is again something which many of the end-users said they would find helpful
20	A way of showing the user a written explanation of the rules	The user should be able to press a help button, which will then display a written set of rules of chess.	As this is something that both my client and end-user would like to be included, and so that the program is accessible to more people – not just those who are already very confident with how to play chess

Design

Decomposition



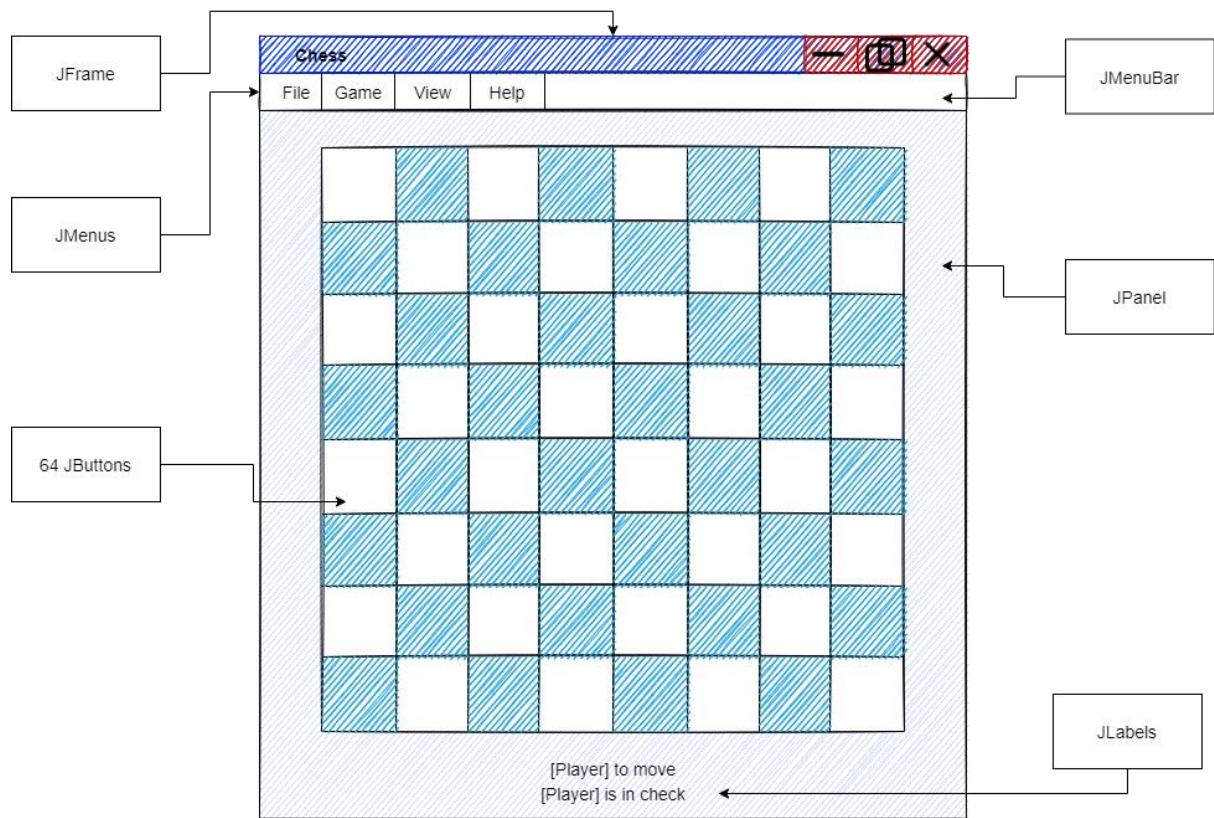
1. Program needs to show GUI so that the user can see the board and input their moves. Must be user-friendly and intuitive to use. Should use the colours requested by the end-users from the survey: blue, white, grey, black, yellow, green.
 - a. JFrame form.
 - i. Contains all other GUI components.
 - b. 8x8 grid of JButtons. Must be in the centre of the screen, as this is something that the end-users have requested.
 - i. JButtons stored in 8x8 2D array, so that they can correspond with the 8x8 2D array of pieces representing the board, and can be updated to show the user the layout of the pieces on the board.
 1. JButtons should have a border that is either light or dark in colour, to represent the chequered board. Should use a blue colour scheme, as this is a colour the end-users have said they would like in the program.
 - ii. If there is a piece in the array representing the board at a particular position, the button at that same position should have an icon showing that piece to the user, so that they can see their pieces on the screen. Each different type of Piece should have a variable storing the location of an image to represent it.
 - c. JLabels, so that the user can be informed of the status of the game, e.g. being told if they are in check.
 - i. JLabel telling user if they are in check, so that they can easily determine whether or not they are in check, without having to work this out

- themselves. Text must be large enough for the user to read easily, and contrast well. Should only be set to visible when they are in check.
- ii. JLabel telling user if they are in checkmate. Should tell the user when someone is in checkmate, and who has won the game. Should only be visible when someone is in checkmate.
 - iii. JLabel telling user whose turn it is, so that the players can easily keep track of who should move next. Should be updated after each turn with the new player to move.
- d. JMenuItems to allow the user to perform other useful actions in the program that the client and end users have requested.
- i. JMenuItem with text “Undo”. When pressed, it should undo the previous two moves taken, if any have been taken.
 - ii. JMenuItem with text “Surrender”. When pressed, it should end the game.
 - iii. JMenuItem with text “Save game”. When pressed, a string should be generated representing the current state of the game, and written to an external text file.
 - iv. JMenuItem with text “Open game”. When pressed, the user should be able to select a file containing a string representing the layout of the board. It should be checked whether the string is valid, and if it is, be loaded into the ChessBoard. Otherwise, should inform the user that it is not valid.
 - v. “Help” JMenuItem. When pressed, a JDialogBox should be shown to the user, containing text explaining the rules of chess to the user.
2. User can input move using JButtons representing the board.
- a. If user has not selected valid piece to move, allow them to select a piece.
 - i. If user selects piece that is their own (colour of piece equals colour of current side to move), and is not empty, check if there are any valid moves for that piece.
 - 1. If there are valid moves, highlight piece in green, and store this location in an int called startPos. startPos / 8 should give the row, and startPos % 8 should give the column of the piece.
 - a. Use function to generate all possible valid moves for a piece. Change the background colour of the corresponding JButtons to yellow, so that the user can see which moves they would be allowed to make.
 - 2. If there are no valid moves, change background colour of JButton to red, to indicate to the user that this piece cannot move.
 - b. If user has selected a piece to move (i.e. startPos contains a valid location), allow them to select a location to move the piece to.
 - i. If valid location, store this location in an int called endPos, and move the piece at the location stored in startPos to the location stored in endPos. Set the piece at location startPos in the 2D array representing the board to null. Then, switch turns and update GUI to show the new state of the board. Also, reset startPos and endPos to -1.
 - ii. Otherwise, reset startPos and endPos to -1, and allow user to select a new piece to move.
3. User can resign the game. This is needed as it is a feature the end-users have requested.
- a. When user presses “Surrender” JMenuItem, reset board and allow user to choose who they would like to play against again.

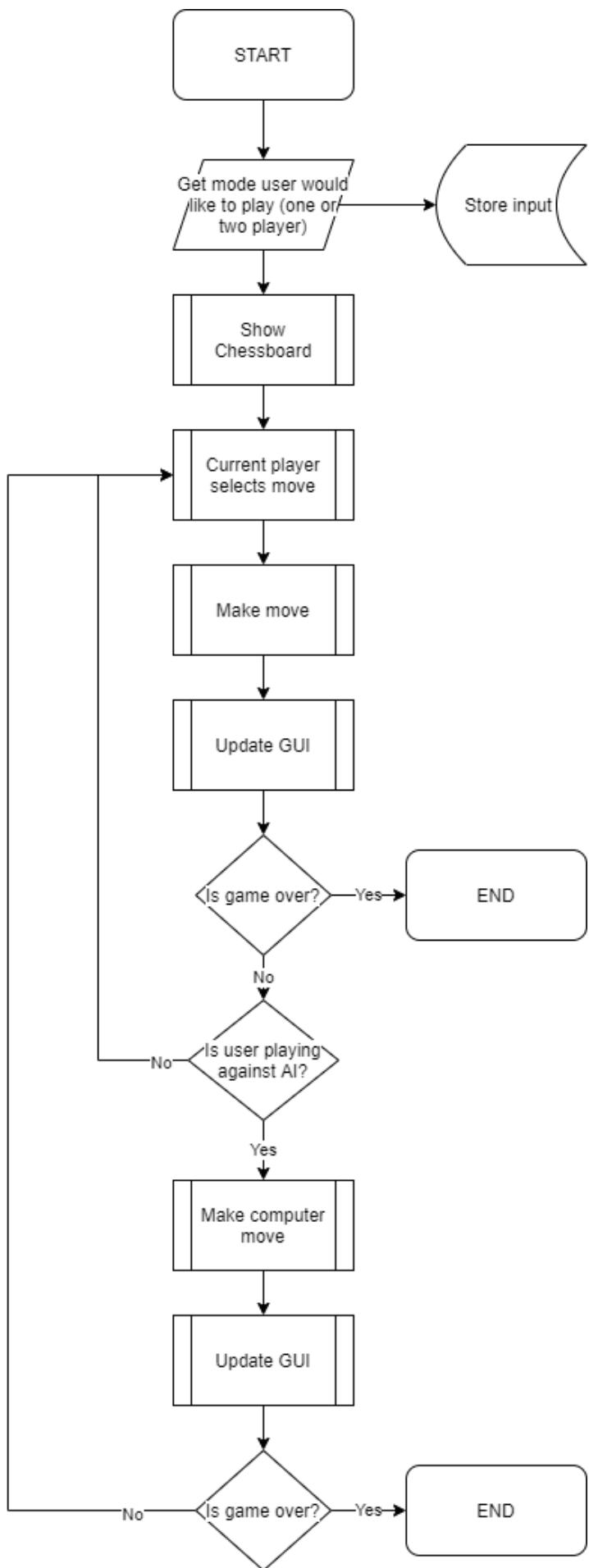
4. User can undo move. This is also a feature that the end-users have requested.
 - a. When user presses “Undo” JMenuItem, undo the opponents last move and the user’s last move.
5. Computer can make move, so that the user is able to use the program as a single player in addition to playing against someone else, as this is something the client is expecting.
 - a. Determine all possible valid moves for AI side for all pieces.
 - i. Iterate through the 2D array of pieces representing the board, if colour of piece is equal to colour of AI, find all valid moves for that piece, and add to ArrayList storing all moves for all AI pieces.
 - b. From these valid moves, select the move to make.
 - c. Make the move. In the 2D array of pieces representing the board, set the piece at the location to move the piece to equal to the piece, and set the piece at the original location to null.
 - d. Update GUI so that new status of the board can be shown to the user after this move has been made, allowing them to see the move that has been made.
 - e. Switch turns, so that the next user can make their next move.
6. User can save game to external file. This is needed, as it is something that the end-users have said they would find useful to have in the program.
 - a. Calculate string for current game state, using function getGameState() of the ChessBoard class, to get a way of representing the board in a way that can be written to the file.
 - b. Write game string to external text file using FileWriter class.
7. User can load game from a string. This is needed as it is again something that the end-users would like to be included in the program.
 - a. When user presses “Load” JMenuItem, allow them to select a file containing a previously saved string representing the game.
 - b. If valid string, load this into program, and update the GUI, using function loadGameState() of the ChessBoard class.
 - c. Otherwise, if it is not valid string or file, the user should be informed of this, so that they will know why the saved game was not loaded into the program.
8. User can see rules of game. This is required so that the program will be able to be used by as many users as possible, regardless of their knowledge of chess, and how confident they are with the rules. Many end-users also said that they were not confident with the rules, and so this would help them.
 - a. When user presses “Help” JButton, a JDialogForm should be shown, showing the user a short, written explanation of the rules of chess / how to play, that is easy for them to understand, so that they are able to use the program.
9. Moves can be validated. This is essential, so that the AI can determine which moves it could possibly take, and decide between them, and so that no player can take an illegal move, helping new players to learn the rules, and preventing players from cheating.
 - a. Each piece has a function to check whether a particular move is pseudo-legal (i.e. legal ignoring check) or not, returning a boolean.
 - i. Pawn: If piece hasn’t previously moved, can move either one or two squares forwards, in the same column. Otherwise, can move just one square forwards, in the same column. For both, this is providing that the square to move to is empty. If capturing a piece of the opposite colour, the pawn can only move one square diagonally forwards and to the left or right.
 - ii. Rook: Can move either horizontally or vertically. Cannot jump over pieces.

- iii. Knight: Can move in an L shape, and can jump over pieces
 - iv. Bishop: Can only move diagonally, cannot jump over pieces
 - v. Queen: Can move in any direction, cannot jump over pieces
 - vi. King: Can move one square in any direction
 - b. Returns true if move is legal, and false if it is not legal.
10. Can determine when the game should finish, so that the program does not continue on forever, even if a player is in checkmate. Otherwise, it would come to a point where the user would have no possible moves to make, but the game would still carry on.
- a. When game ends, the board should be reset to its initial state, and GUI updated to show this.
 - b. Should ask user who they would like to play against again – another human or the AI.
11. Can switch turns, so that both players can take moves, and only the side whose turn it is can move their pieces.
- a. After each turn, increment currentTurn by 1.
 - b. If $\text{currentTurn} \% 2 == 0$ (i.e. currentTurn is even), then the current side to move is white. Otherwise, if currentTurn is odd, the current side to move is black.
12. Can find the position of a particular king on the board, which is needed to determine whether a king is under attack by opponent.
13. Can determine whether a player is in check. This is needed in order to determine whether a move is valid, since a move is only legal if it does not put your own king into check.
- a. Should find the position of king of a certain colour, and determine whether this piece is under attack by opponent.
 - i. To do this, keep checking in each direction from the piece (i.e. up, diagonally up and right, right, etc.) until either a piece is reached, or the edge of the board is reached.
 - ii. If a piece is reached, and it is the opposite colour to the king, get the type of the piece, and determine whether it can attack in this direction.
14. Can determine whether a player is in checkmate. This is needed in order to determine when the game should finish, and who the winner is.
- a. Should use previous function to determine whether a player is in check.
 - b. If they are in check, and there are no places for the king to move to that would bring them out of check, player is in checkmate, so return true. Otherwise, return false.
15. Pawn can be promoted. This is needed as it is a key part of the rules of chess.
- a. When a pawn reaches the opposite side of the board, if it is a human who is playing, the user should be given the option between rook, knight, bishop and queen to promote the piece to.
 - b. If the AI is playing, then the chosen type of piece to promote to should automatically be a Queen.
 - c. Then, the piece should be replaced in the 2D array representing the board by a new piece of the selected type.
 - d. Then, the GUI should be updated to show the new state of the board.

Structure of solution



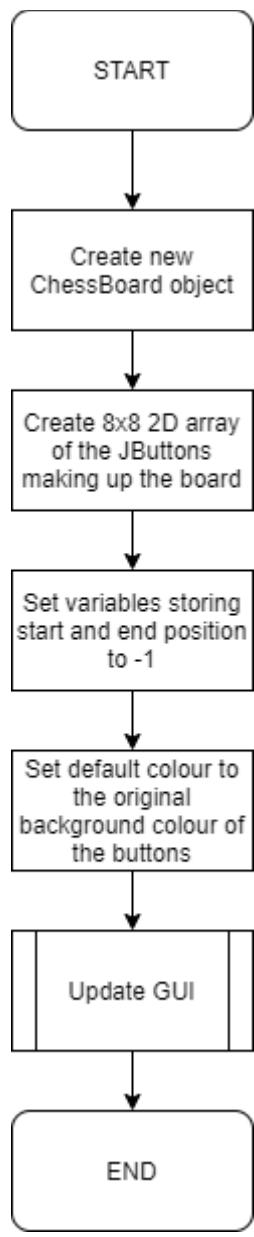
This diagram shows the rough layout of the design of the GUI. I have decided to have the chessboard in the centre of the window, as this is something which the end users had requested. The chessboard will consist of 64 JButtons, allowing the user to press a button to select which piece they would like to move, and then another button to select where they would like to move it to. Then there are two JLabels at the bottom of the window, informing the user of whose turn it is, and who is in check. This is to allow the user to easily keep track of who should move next, and whether or not they are in check. Everything else the user may interact with, I have decided to put in a menu bar, so that the main window is very simple and not too cluttered, which is again something the end users had requested. Putting these things (such as undo move, surrender, help, etc) in a menu bar also allows them to have descriptive labels on the buttons, making the GUI more intuitive / easy to use, as their functions will be clearer.



This is the main flowchart for this program. When the program is first run, the user is asked whether they would like to play against the computer or not. Then, they are shown the main window, and can begin to play the game. They select a piece to move and a place to move it to, and then the GUI is updated to show the result of this move. Then, assuming the game is not over (i.e. no one is in check), if the user selected to play against the computer, the computer makes a move. Otherwise, the program allows the other player to make their move. This then carries on until the game comes to an end.

Algorithms

Show board



This is the flowchart showing the “Show chessboard” process. This is used right at the beginning of the program – a new ChessBoard object is created, and an 8x8 2D array of JButtons is created, storing the 64 JButtons that make up the graphical representation of the chessboard.

The two variables startPosition and endPosition, which store the location of the piece the user selected to move, and the location of the piece the user selected to move that piece to respectively, are initially set to -1, indicating a location has not yet been chosen by the user.

Then, the original background colour of the buttons is stored in the variable defaultColour, so that the background colour can be reset if it later is changed.

Then, the GUI is updated, which will show the chessboard to the user with all of the pieces in their starting positions.

This is needed so that all the data structures that will be used within the program can be initialised, and the GUI shown to the user, allowing the user to see the board and play the game.



I need a way to represent a move, so that the moves that have been taken in the game can be stored, and to pass the move between different subroutines. To do this, I will store the move in a string. The first two characters will represent the location of the piece that is moving (if this is below 10, it will start with a 0, e.g. 06), the next two characters will represent the location that the piece is moving to, and the last character will represent the type of piece that was captured ('0' for none, '1' for pawn, '2' for rook, '3' for knight, '4' for bishop, '5' for queen).

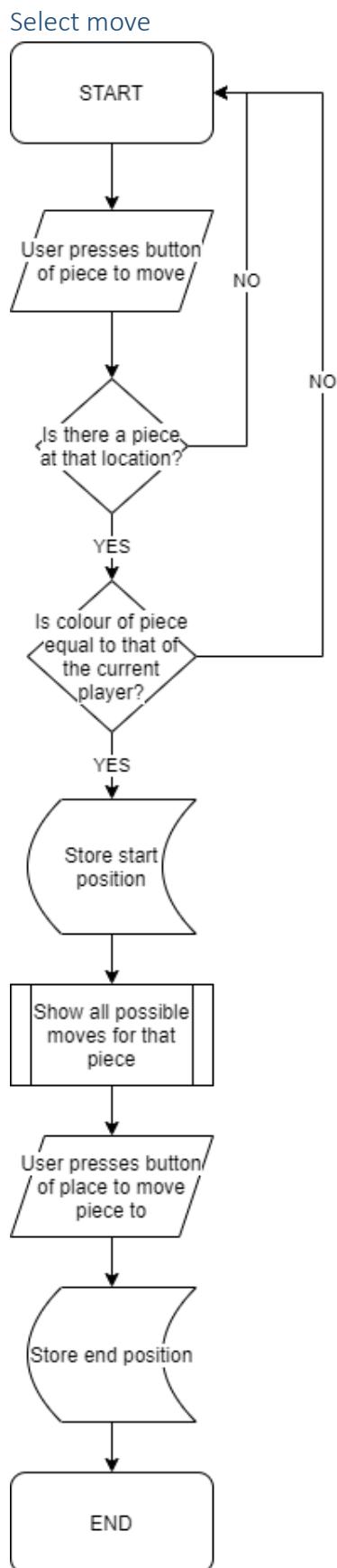
Positions on the chessboard are stored as an integer between 0 and 63 inclusive, which correspond to the following positions on the board:

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

This is so that the position can be stored in a single int, rather than for example an array of two integers, with one value representing the row, and the other representing the column, as using just one int is easier and uses less memory.

To get the row from a position, you can do "position / 8".

To get the column from a position, you can do "position % 8".



This is the flowchart showing the “Current player selects move” process.

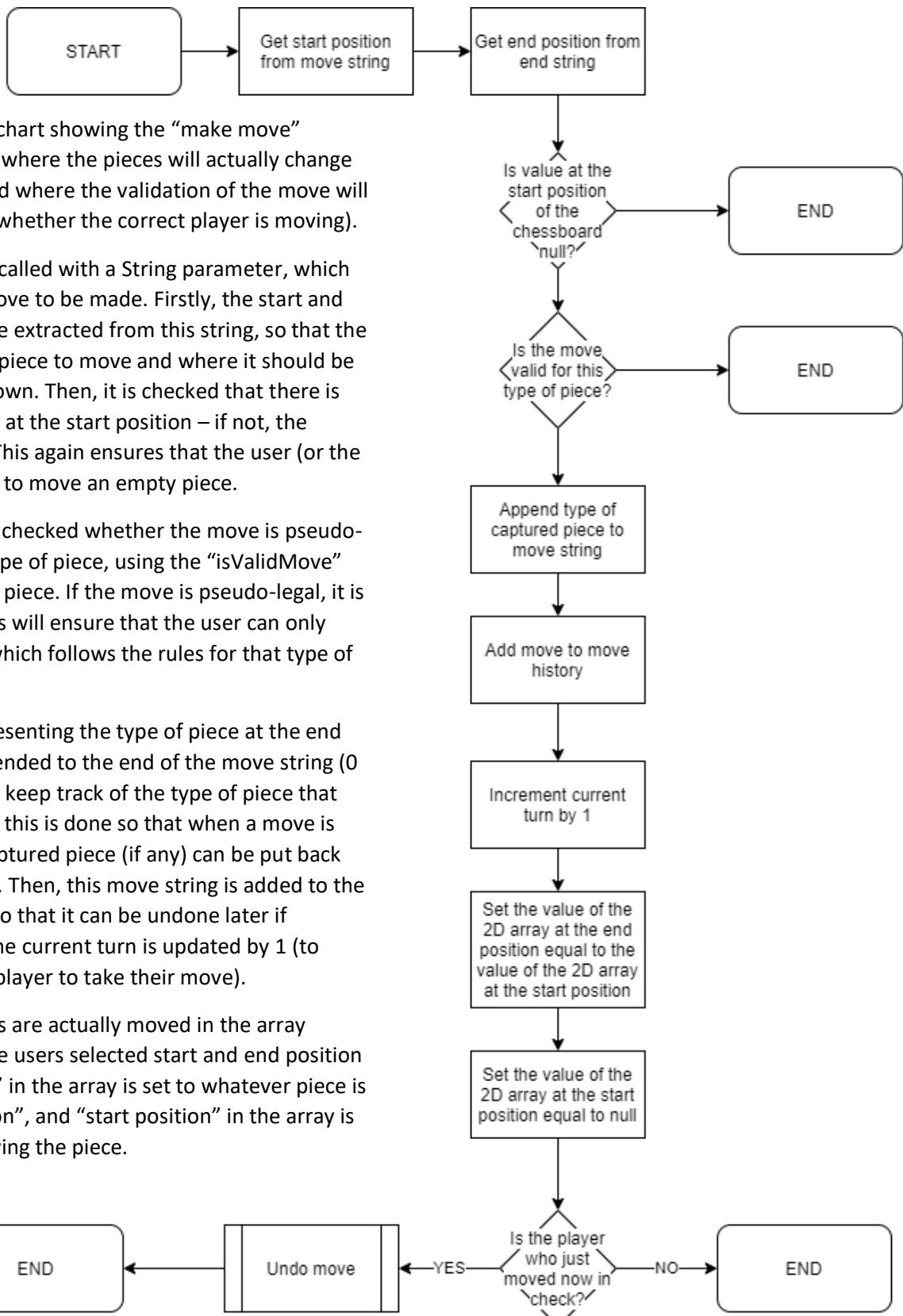
When the user selects the piece they would like to move, it then checks whether there is a piece at that location, and then if there is, whether the colour of that piece is the same as whose turn it currently is. If there is no piece there, or if the piece belongs to the wrong player, then the program waits for the user to select a different piece. This is where validation occurs to check that the correct player is moving – without this, the wrong player may be able to move, or a user may be able to move an empty square as a piece.

Otherwise, it sets the variable startPosition equal to the corresponding position, and then highlights the possible places that the selected piece is allowed to move to in another colour to the user. This makes the program more user-friendly to those who may not know how the different pieces move.

Then, it allows the user to select a location to move the selected piece to, and stores this position.

Storing the selected start and end locations will allow these to be passed to the make move process, where the correct pieces will be moved within the array (if the move is legal).

Make move



This is the flowchart showing the “make move” process. This is where the pieces will actually change in the array, and where the validation of the move will occur (beyond whether the correct player is moving).

This method is called with a String parameter, which contains the move to be made. Firstly, the start and end position are extracted from this string, so that the position of the piece to move and where it should be moved to is known. Then, it is checked that there is actually a piece at the start position – if not, the method ends. This again ensures that the user (or the AI) is not trying to move an empty piece.

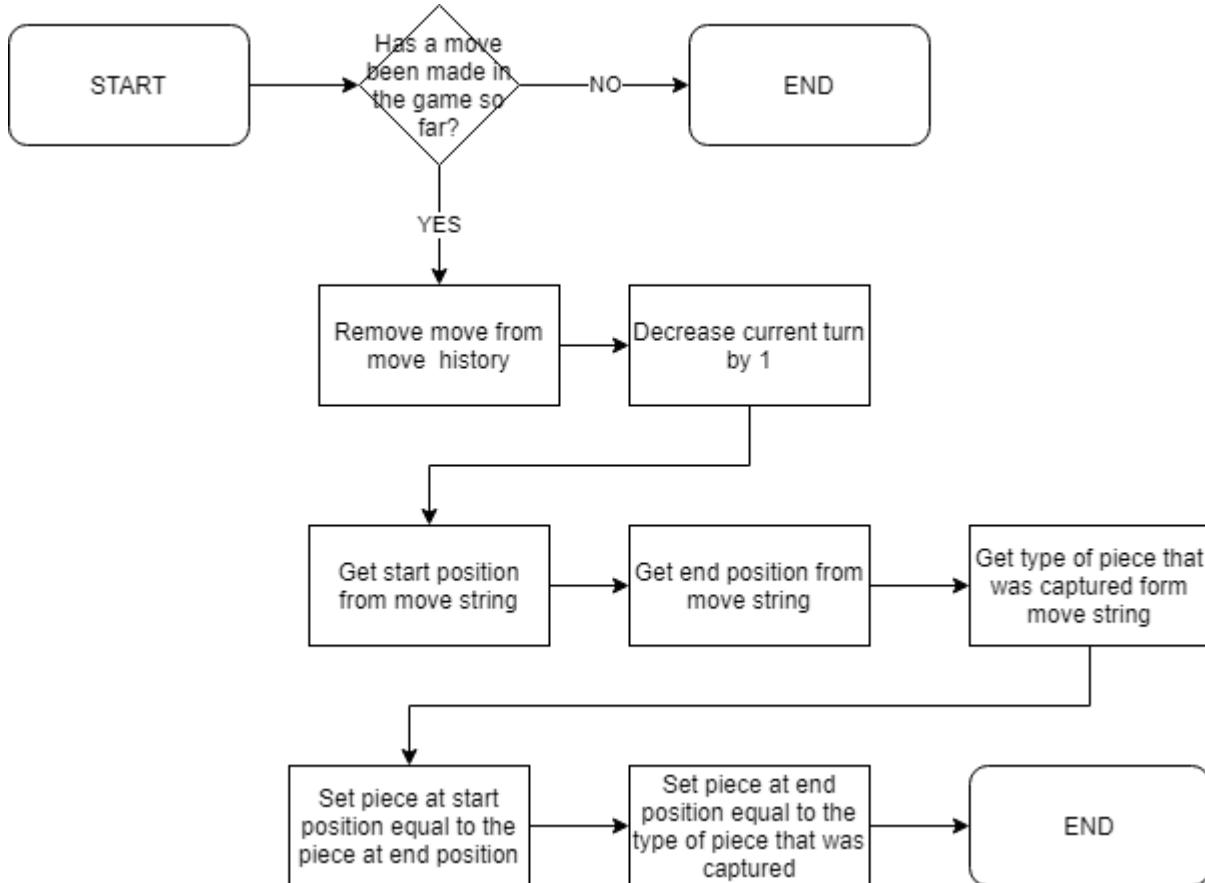
Otherwise, it is checked whether the move is pseudo-legal for that type of piece, using the “isValidMove” method for the piece. If the move is pseudo-legal, it is then made. This will ensure that the user can only make a move which follows the rules for that type of piece.

A number representing the type of piece at the end position is appended to the end of the move string (0 for no piece) to keep track of the type of piece that was captured – this is done so that when a move is undone, the captured piece (if any) can be put back onto the board. Then, this move string is added to the move history (so that it can be undone later if needed), and the current turn is updated by 1 (to allow the next player to take their move).

Next, the pieces are actually moved in the array according to the users selected start and end position – “end positon” in the array is set to whatever piece is at “start position”, and “start position” in the array is set to null, moving the piece.

Then, it checks whether the player who just moved is now in check. If they are, the move shouldn’t be made, as a move that puts or keeps yourself in check is not legal, so it is undone, resulting in no overall change in the array. As the GUI is not updated between the pieces being swapped and the move being undone, it would appear to the user that no move had been made / in the same way that a move that is not pseudo-legal would be shown – it is not made and currentTurn is not incremented, so the same player can select a different move. The current turn will end up staying the same since it will be decremented by 1 in the “undo move” process, after being previously incremented by 1 in this process.

Undo move



This is the flowchart showing the “undo move” process. This will be used to allow the user to undo a move if they would like to, when determining check, and to allow the AI to look ahead at moves to make.

Firstly, it checks whether a move has been made previously or not. If no move has been made, this process ends. This is needed as a move cannot be undone if none have been made so far in the game, as this would lead to an error.

Then, if a move has been made, firstly the move that was most recently taken is removed from the move history, and the current turn is decreased by one (so that the correct player will be allowed to move).

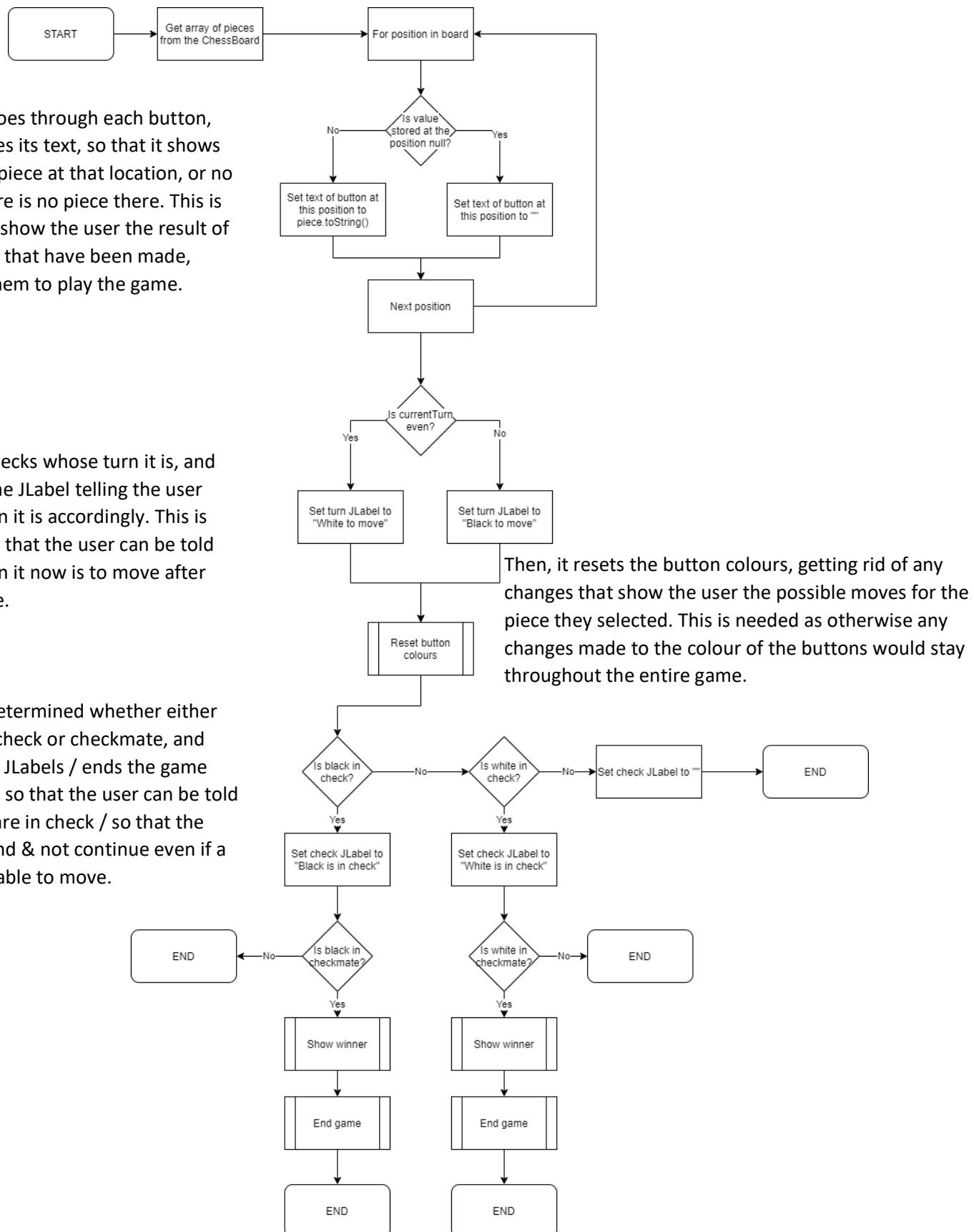
Then, the starting position of the piece that was moved, the position that piece was moved to, and the type of piece that was captured in that move is taken from this move string that was removed from the move history. This is needed so that it can be determined which positions in the array of Pieces need to be changed / what they need to be changed to.

Then, the piece at start position (the location at which the piece originally was) is set to the piece at end position (the location to which this piece was moved to). This moves the piece back to where it was before the move was made.

However, if a piece was captured, this will also need to be placed back onto the board. So, the piece at end position in the array is set to a new piece which is of the type that was captured (or null if no piece was captured).

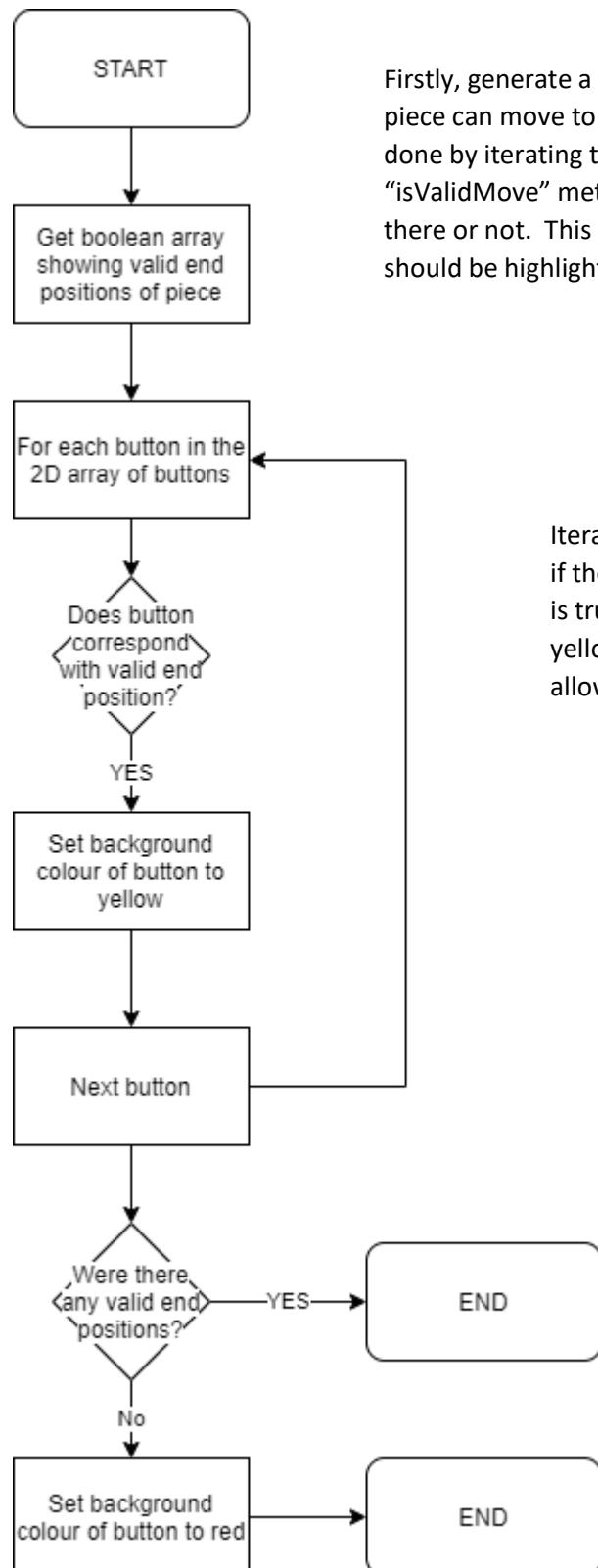
Update GUI

This is the flowchart showing the “Update GUI” process. This ensures that the user can see the layout of the pieces on the board, updates the JLabels relating to whose turn it is / whether they are in check, and checks for checkmate.



Show moves

This is the flowchart showing the “Show all possible moves for that piece” process. This highlights the possible positions (if any) that a piece the user has selected can move to in a different colour. This will make the program more usable as it helps those who may not know how each piece moves to play the game.



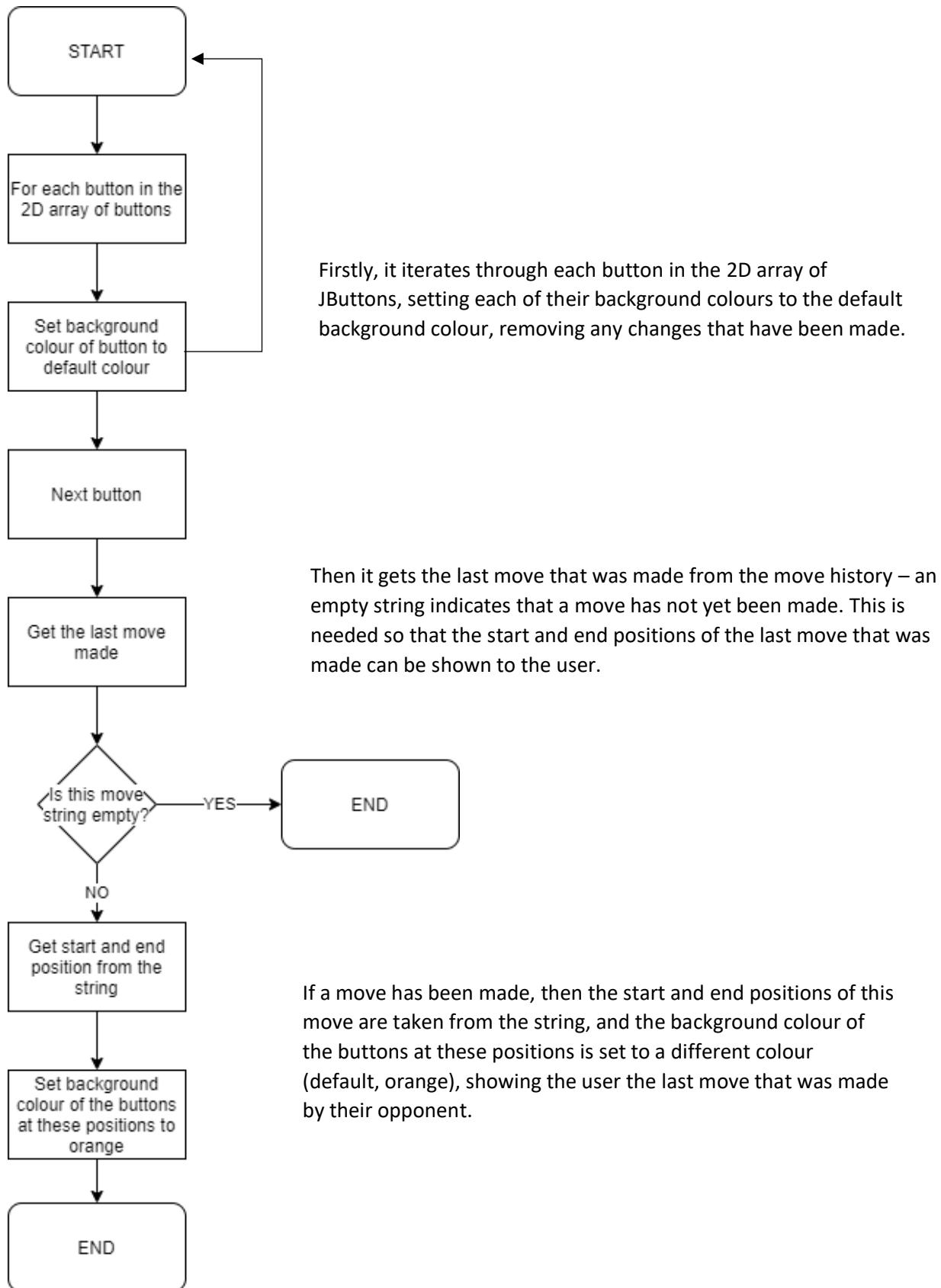
Firstly, generate a 2D array of booleans, with true indicating that the piece can move to that location, and false indicating that it cannot. This is done by iterating through each square on the board, and using the “isValidMove” method for the piece to determine whether it can move there or not. This is needed so that it can be determined which buttons should be highlighted to the user, and which shouldn’t.

Iterate through each button the 2D array of buttons, and if the value at the same position in the array of booleans is true, set the background colour of that button to yellow. This is what shows the user the places the piece is allowed to move to.

If there were no valid places for the piece to move to (i.e. all values in the array were false), set the background colour of the button representing the piece the user selected to red, which will show them that they cannot move that piece, letting them know that they should select another piece.

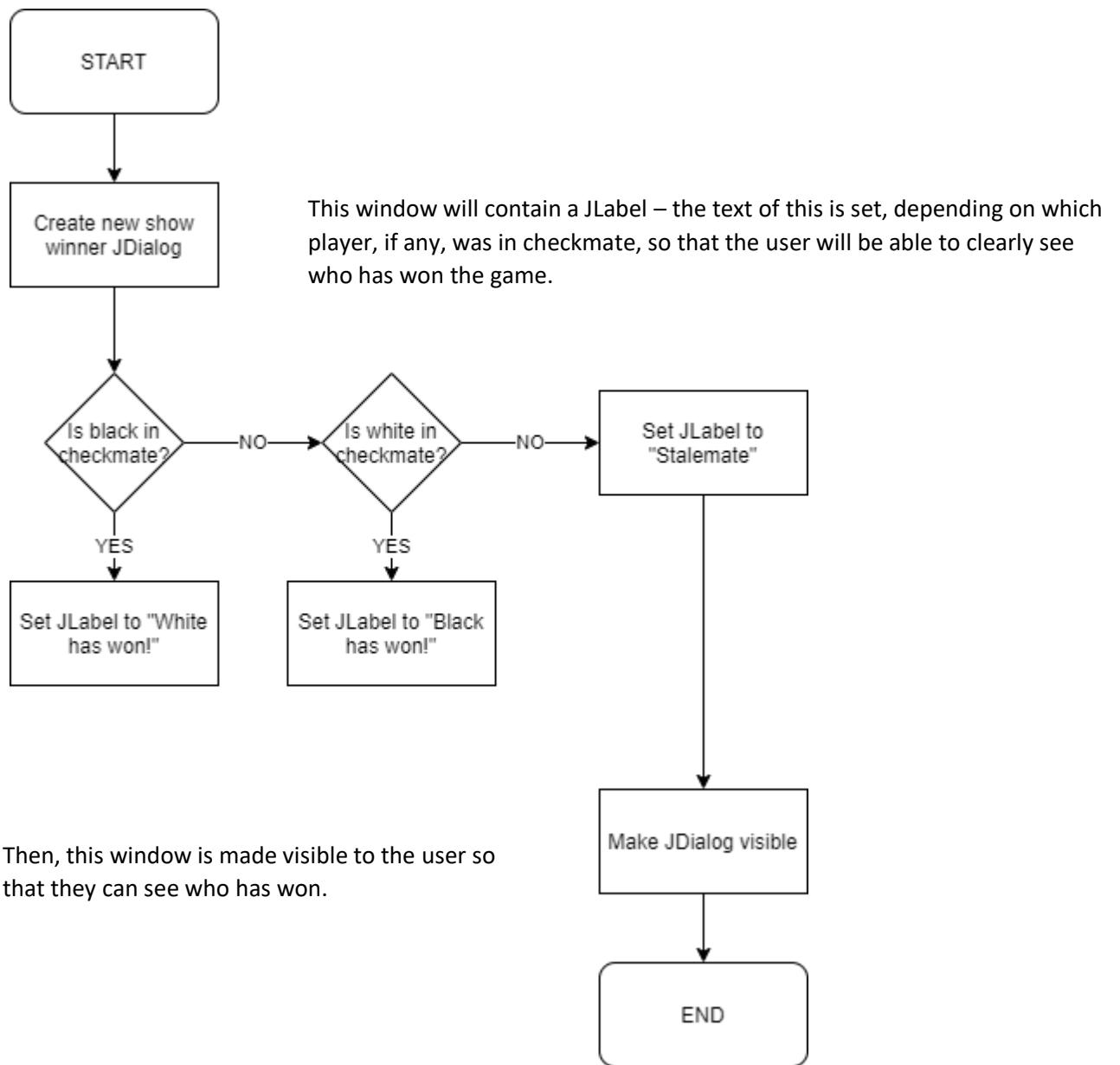
Reset buttons

This flowchart shows the “Reset button colours” process. This will reset any changes to the background colours of the buttons e.g. after the user is shown where they can move a piece to, and also shows the user the previous move that was made. Resetting the colours is needed as otherwise buttons would be shown in different colours when they shouldn’t, which could confuse the user / make the program harder to use. Showing the user the previous move is needed as it is something which the end users have requested.



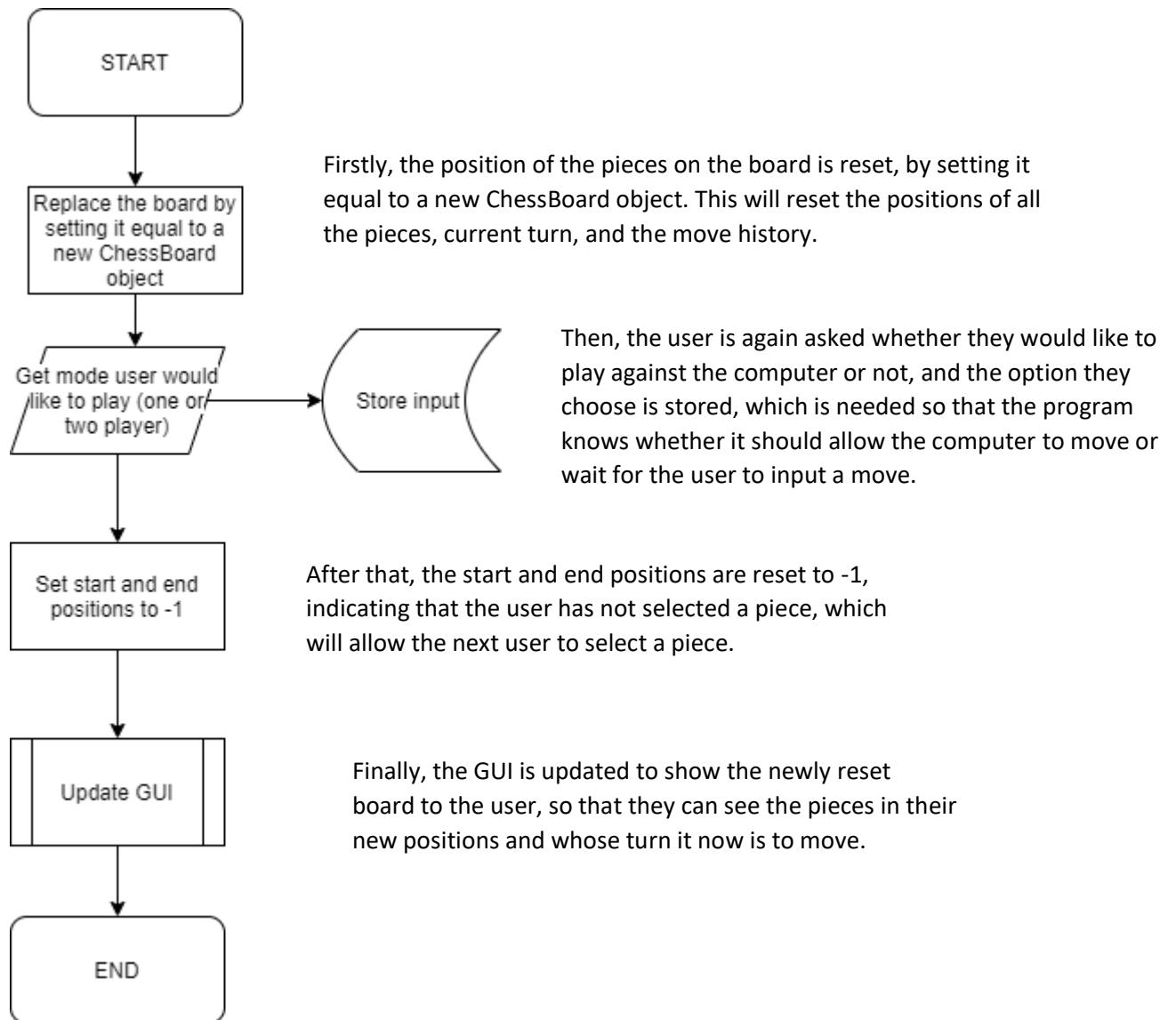
Show winner

This flowchart shows the “Show winner” process. If the game has ended, this process will show a window to the user, telling them who has won. This is needed to make sure that when the game should end, the user can easily see both that the game has ended and who has won, without them needing to determine themselves whether they are in checkmate or not.



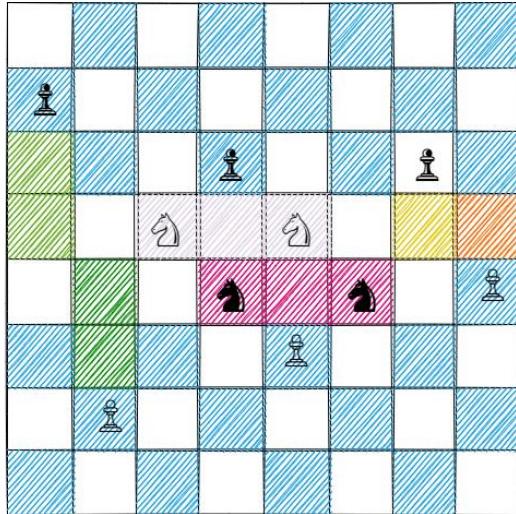
End game

This flowchart shows the “End game” process. This resets the board, allowing the user to select a new opponent and play a new game after the previous game has finished. This is needed so that they can play another game without having to restart the program.



Move validation

The following pages show the flowcharts for the validation of the moves of different kinds of pieces. These determine whether or not the move is pseudo-legal. This allows the game to determine which moves should be allowed to be made, and which shouldn't. This is needed so that the computer can generate a set of moves to pick from, and to prevent the user from making an illegal move. Whether a move is legal (i.e. pseudo legal *and* doesn't put them into check) or not will be determined in the "make move process".

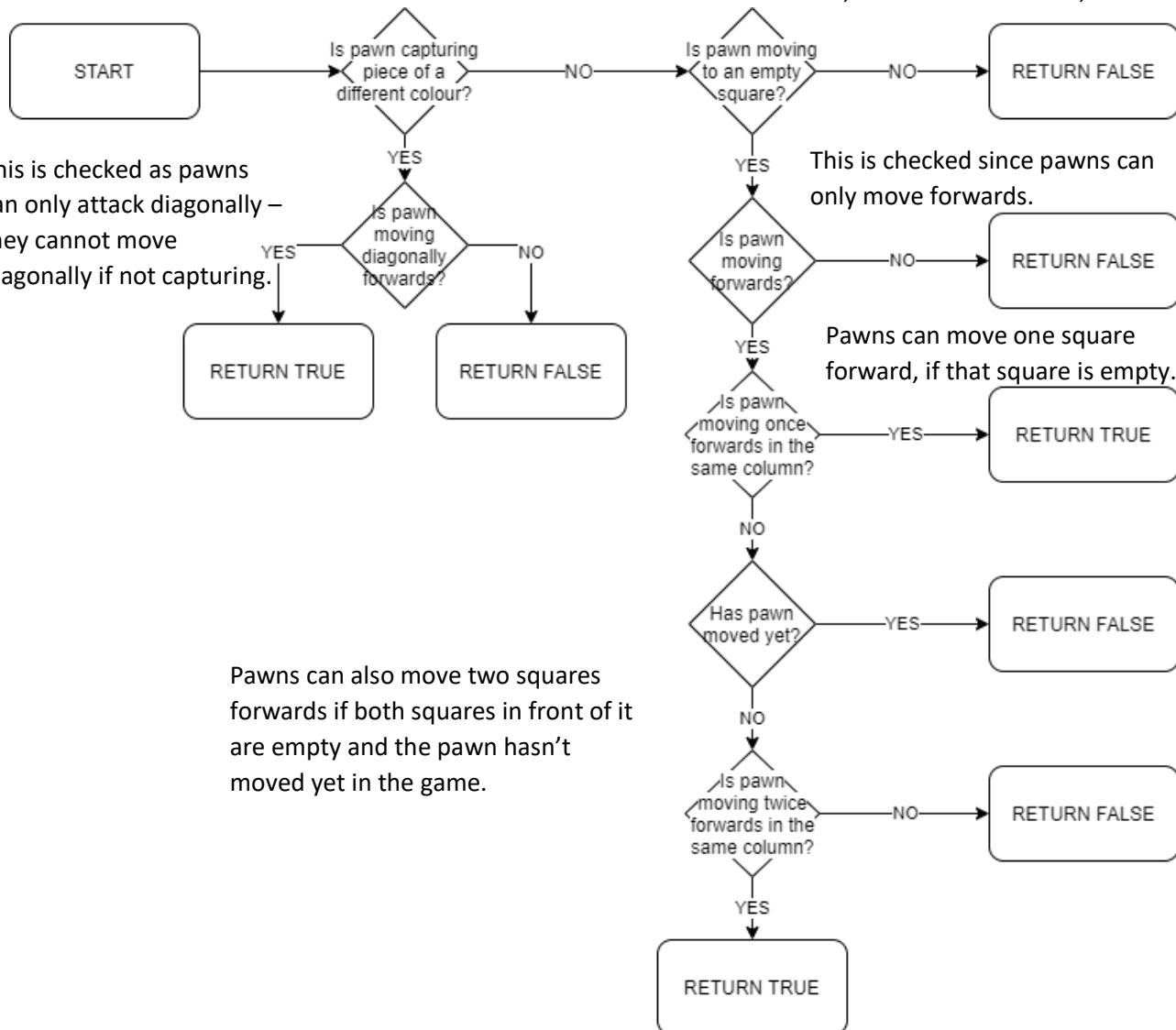


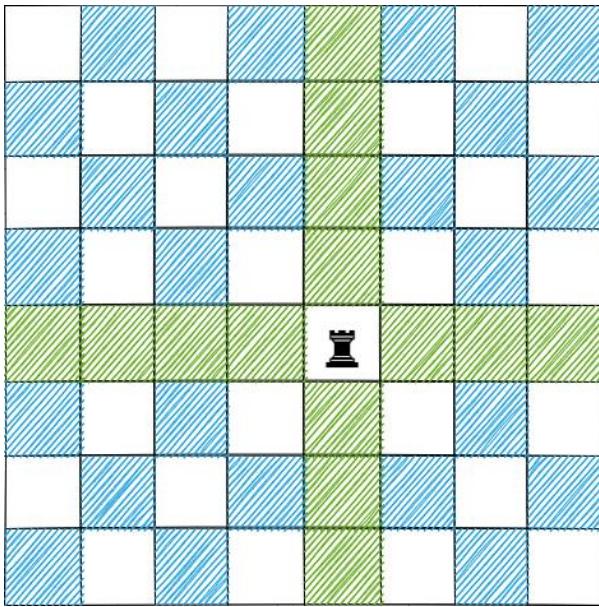
How pawns can move:

- They can only move forwards
- They can move one square forward if that square is not occupied by another piece
- They can move two squares forwards if they have not yet moved, and both the square in front, and the square two in front of them is not occupied.
- They can move one square diagonally forwards left or right, if that square is occupied by a piece of the opposite colour.

If not capturing piece of different colour or moving to empty square, it must be attacking itself, which is not allowed, so return false.

This is checked as pawns can only attack diagonally – they cannot move diagonally if not capturing.

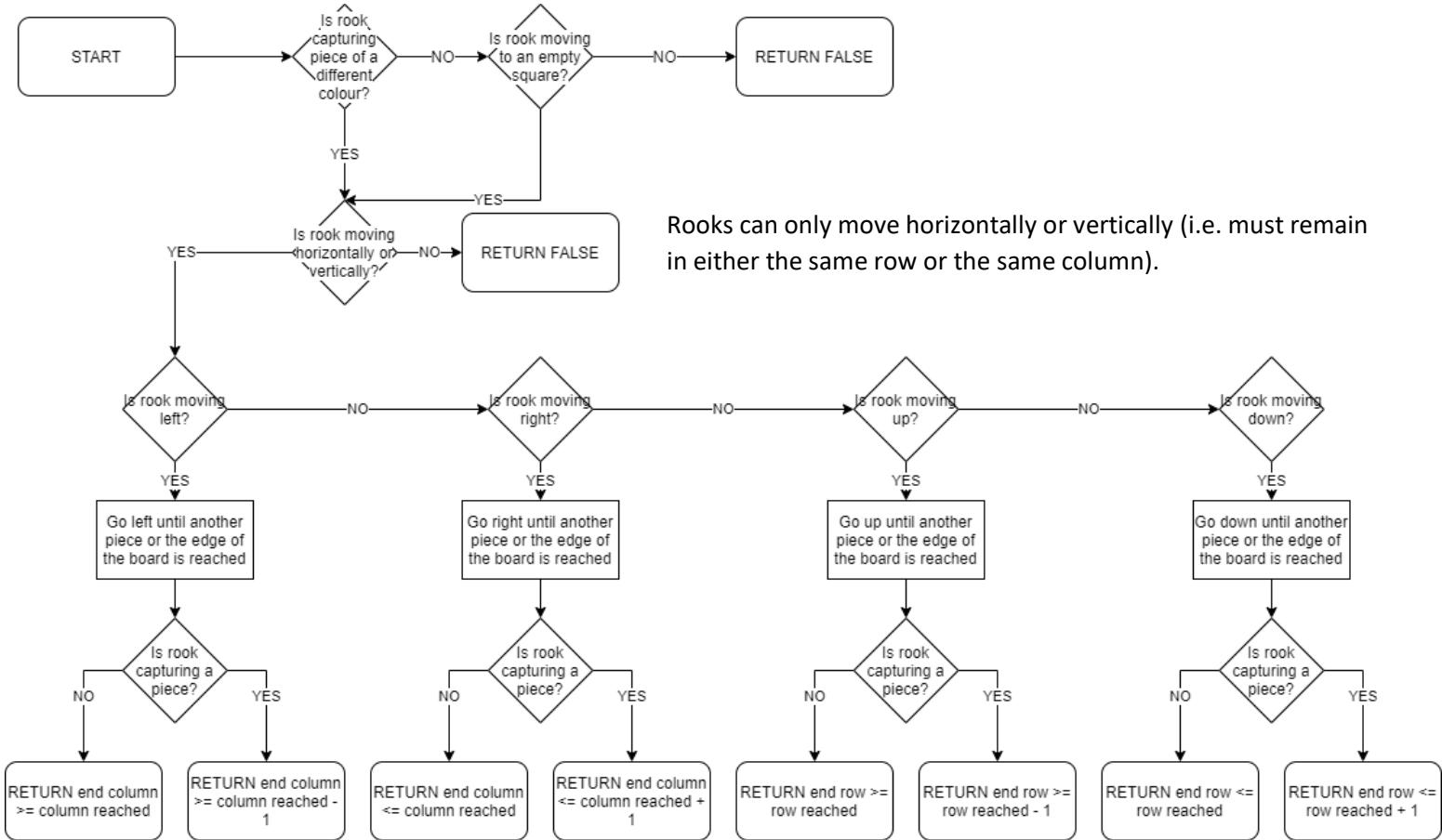




A rook can move any number of squares vertically or horizontally, but it cannot jump over any pieces.

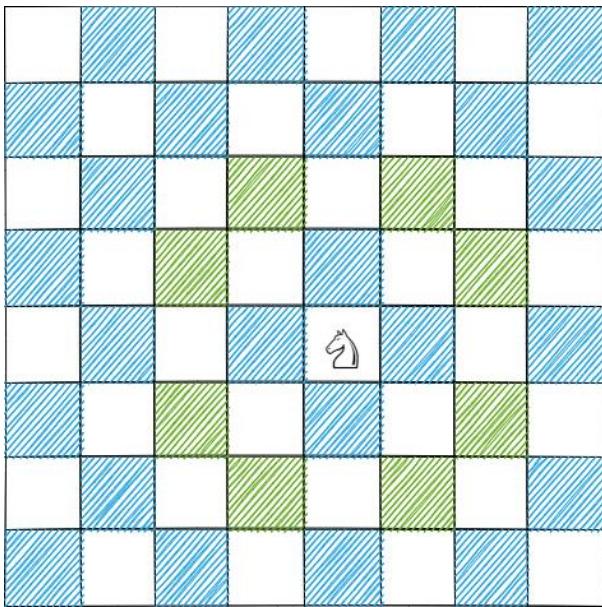
This is the flowchart showing move validation for rooks:

If not capturing piece of different colour or moving to empty square, must be attacking itself, which is not allowed, so return false.



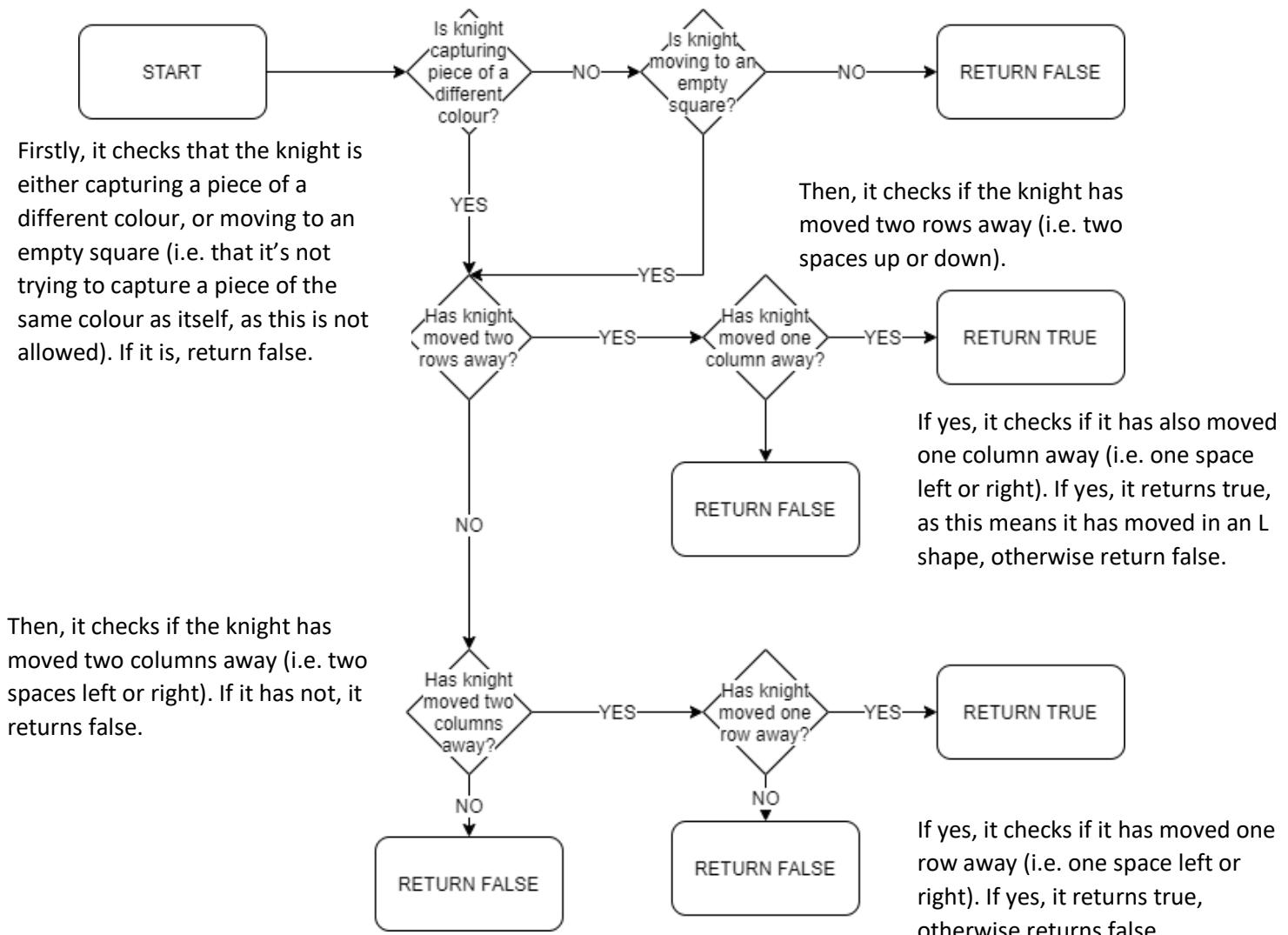
Rooks can only move horizontally or vertically (i.e. must remain in either the same row or the same column).

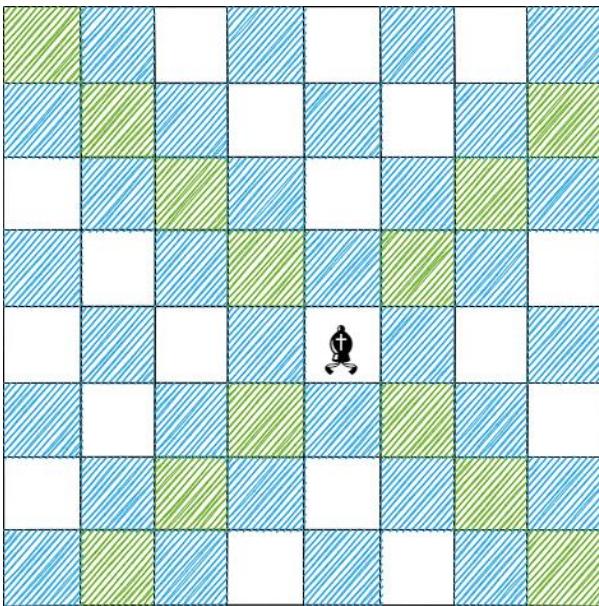
These find the furthest possible distance the rook can go in a particular direction until you reach another piece, or the edge of the board. This is because rooks cannot jump over other pieces, and so cannot go any further than this. If the rook is capturing a piece, the piece they are attacking is also a valid move (i.e. go as far as you can until you hit the piece, and then go one further in that direction, so that the piece being captured is included in the range).



A knight can move two pieces horizontally / vertically, and then one piece in a direction perpendicular to that, as shown in this diagram (i.e. in an L shape).

This is the flowchart showing move validation for knights:

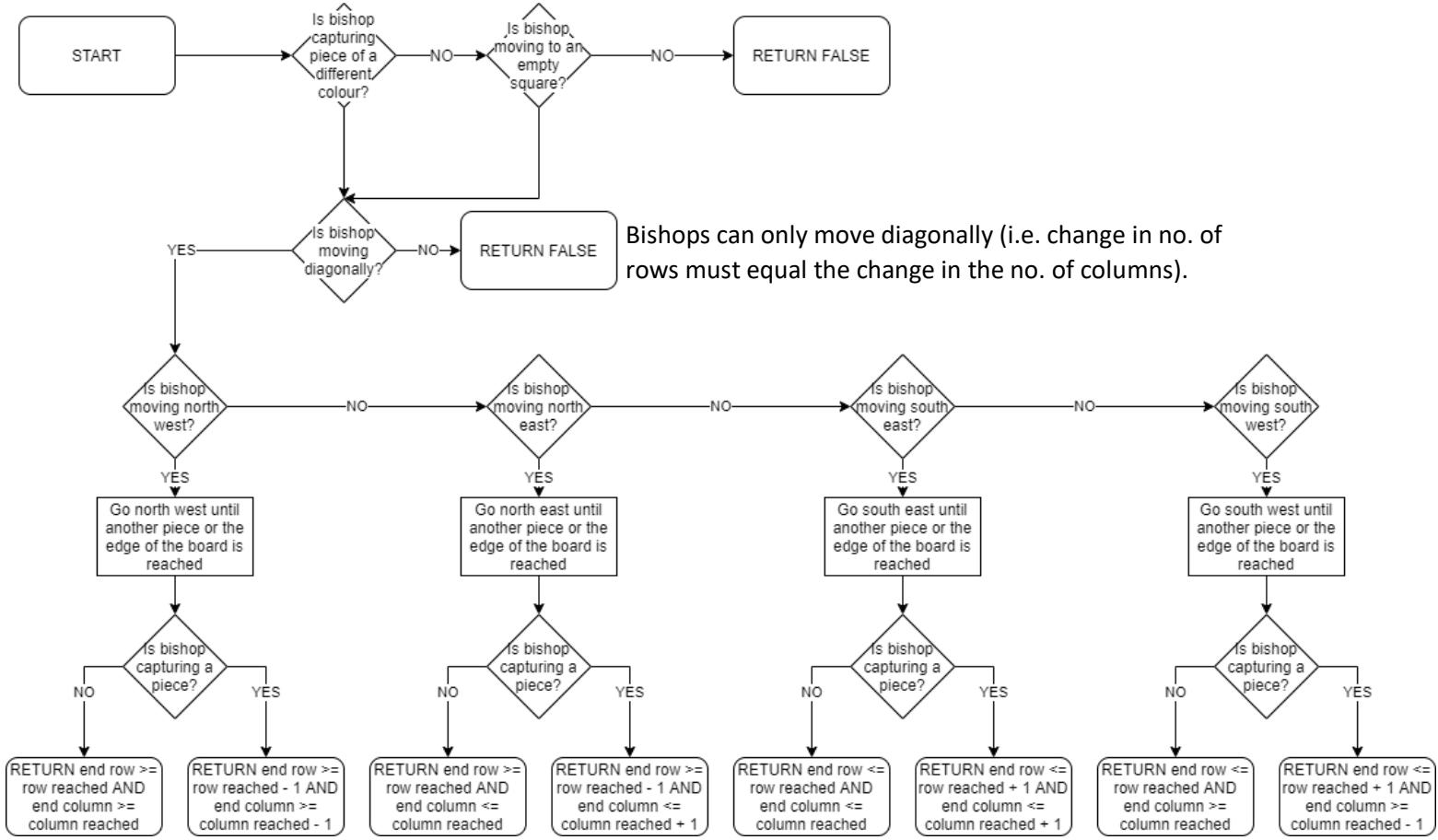




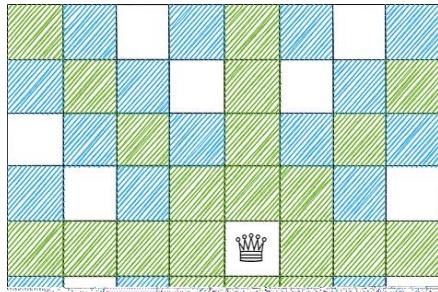
A bishop can move any number of squares diagonally, but cannot jump over any pieces.

This is the flowchart showing move validation for bishops:

If not capturing piece of different colour or moving to empty square, must be attacking itself, which is not allowed.



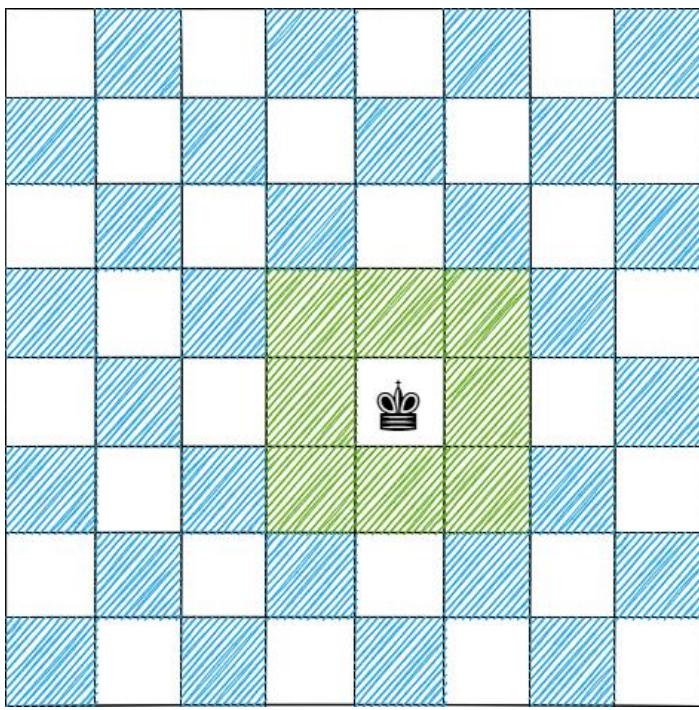
These find the furthest possible distance you can go in a particular direction until you reach another piece or the edge of the board. This is because bishops are unable to jump over other pieces, and so cannot move past this point. If the bishop is capturing a piece, the piece they are attacking is also a valid move (i.e. go as far as you can until you hit a piece, and then go one further in that direction if attacking, so that the piece being attacked is included in the range).



A queen can move any number of squares diagonally or horizontally, but cannot jump over any pieces. This is essentially a combination of a bishop and a rook.

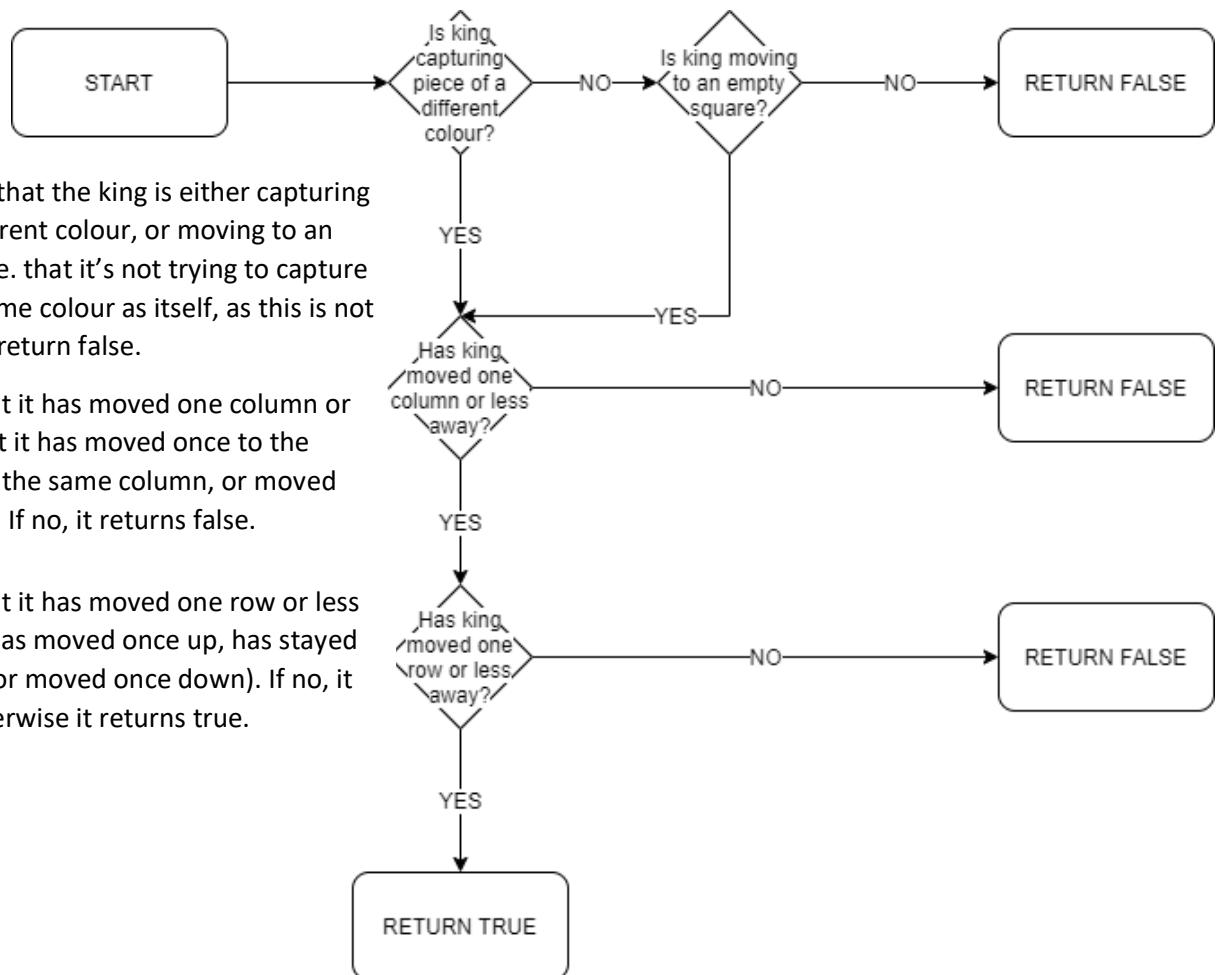
This is the flowchart showing move validation for queens, which is the combination of those for the rook and the bishop:





A king can move one square in any direction.

This is the flowchart showing move validation for kings:



Firstly, it checks that the king is either capturing a piece of a different colour, or moving to an empty square (i.e. that it's not trying to capture a piece of the same colour as itself, as this is not allowed). If it is, return false.

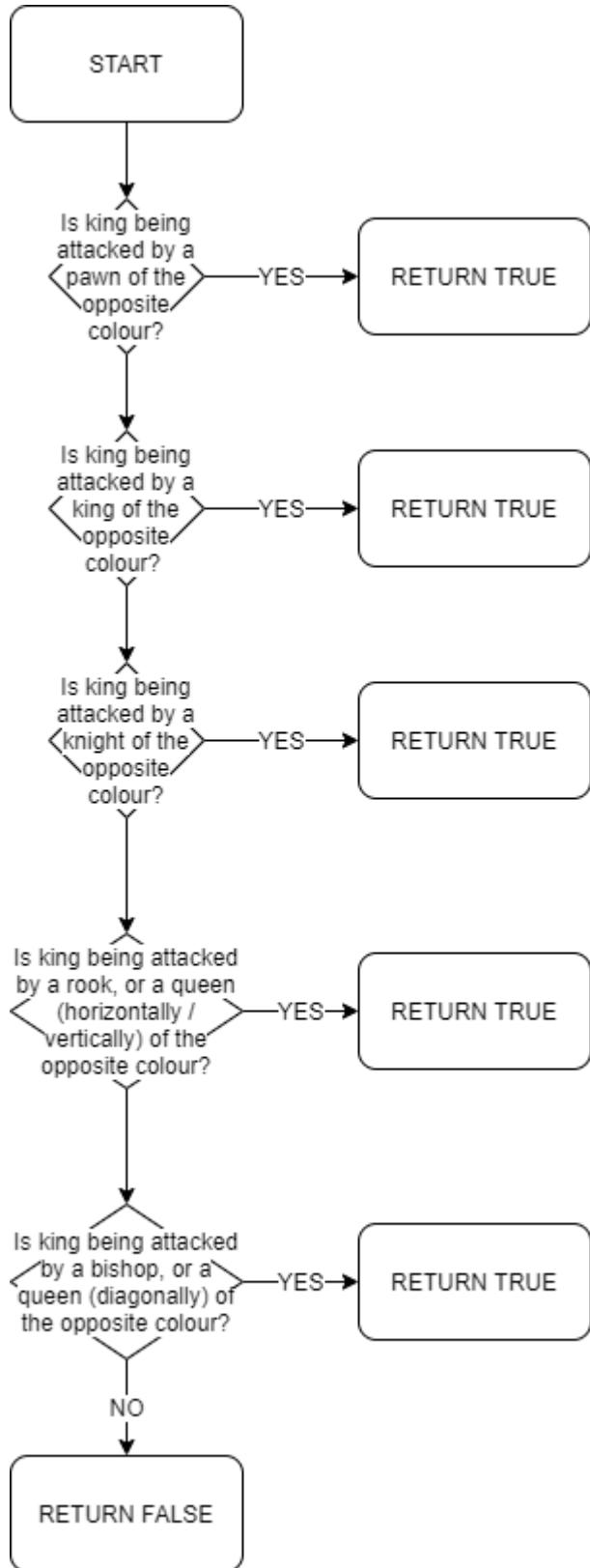
Then it checks that it has moved one column or less away (i.e. that it has moved once to the left, has stayed in the same column, or moved once to the right). If no, it returns false.

Then it checks that it has moved one row or less away (i.e. that it has moved once up, has stayed in the same row, or moved once down). If no, it returns false, otherwise it returns true.

Determining check

An important part of determining whether a move is valid is not only determining whether it follows the rules explained previously for its particular type, but also determining whether or not this move results in a player being in check. In chess, a move is not legal if it puts yourself into check, or, if you are already in check, does not bring you out of check. So, the program needs a way to determine whether a particular player is in check, so that a move can not be allowed if it results in a player putting themselves in check. Furthermore, the ability to determine whether a player is in check is needed to work out whether someone is in checkmate / whether the game should end.

This is the flowchart for the method that will determine whether or not a player is in check:

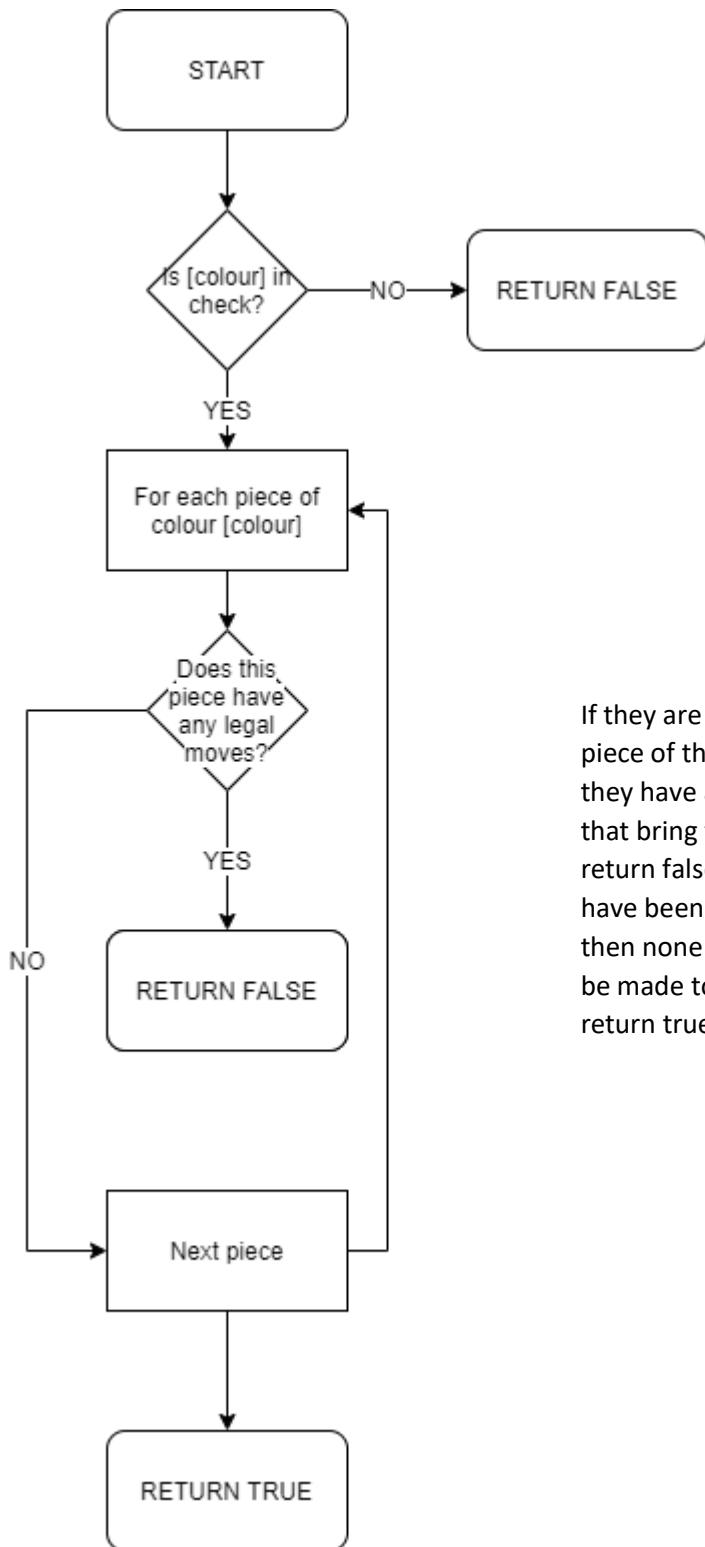


It goes through each type of piece, and checks whether there is a piece of that type attacking the king. If there is, it returns true. Otherwise, it checks the next type of piece, until all have types been checked. If all pieces have been checked and it has not returned true, then it is not under attack, so return false.

After the user makes a move, it will be checked whether that player is now in check. If they are, the move will be undone, and the user allowed to make a different move. They will not see this move being undone – it will appear to them as if the move was not made.

Determining checkmate

Another important part is determining whether or not a player is in checkmate. If someone is in checkmate, their king is under attack, and they have no moves they can make to bring them out of check. If this occurs, the attacking player is the winner of the game, and the game should end.



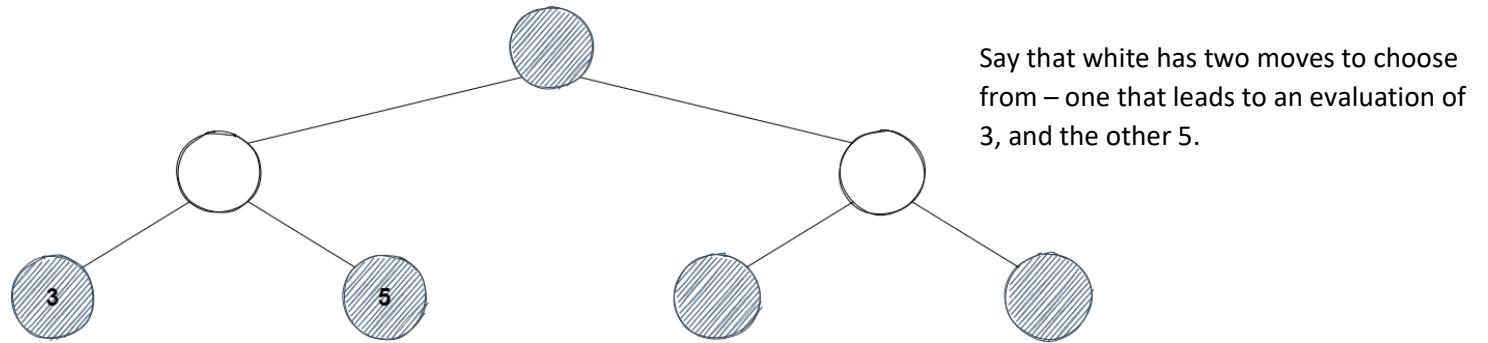
Firstly, it determines if the player is in check – a player cannot be in checkmate if they are not in check.

If they are in check, it goes through each piece of that colour, and checks whether they have any legal moves (i.e. any moves that bring them out of check) – if they do, return false. If after all pieces of that colour have been checked, false was not returned, then none of them have any moves that can be made to bring the player out of check, so return true – they are in checkmate.

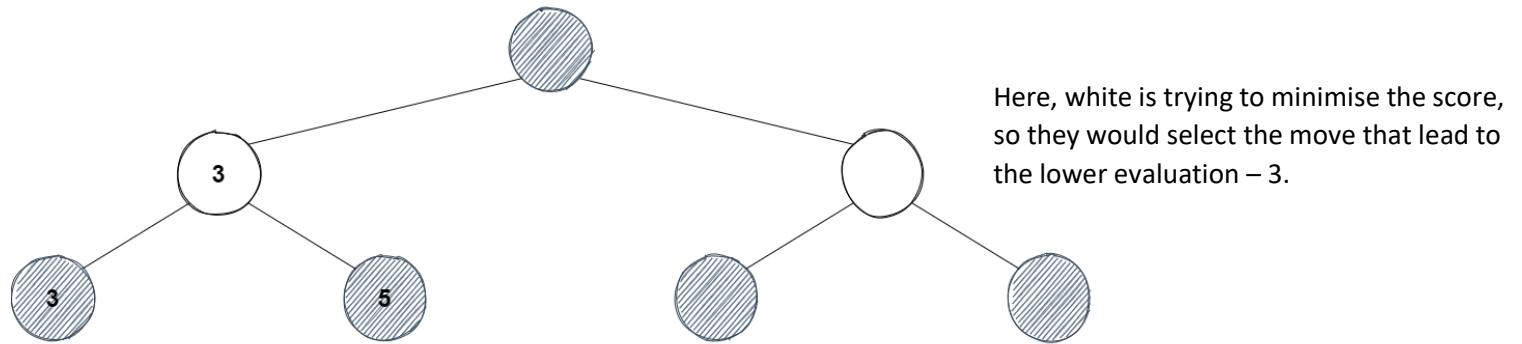
Computer to move

In order for the user to be able to play against the AI, the computer needs a way to determine which move to make. In order to do this, I will use the minimax algorithm. This will need to determine a way to evaluate the board, and come up with a “score” for it, depending on which pieces are on the board. One player (black / the computer) will be trying to maximise this score, whereas the other player (white / the human) will be trying to minimise this score.

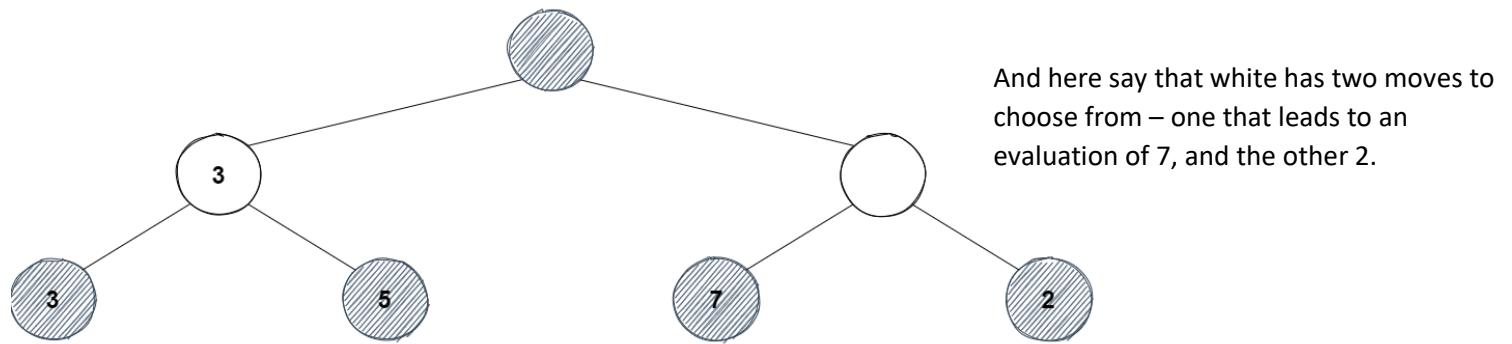
As an example, here each node represents a particular layout of the board (and the number within each node represents the board’s evaluation), and each branch represents a particular move that leads to the next layout. The colour of each node represents whose turn it is to move – white or black. Whilst an actual game of chess would have more than just two branches from each node, it still works in the same way.



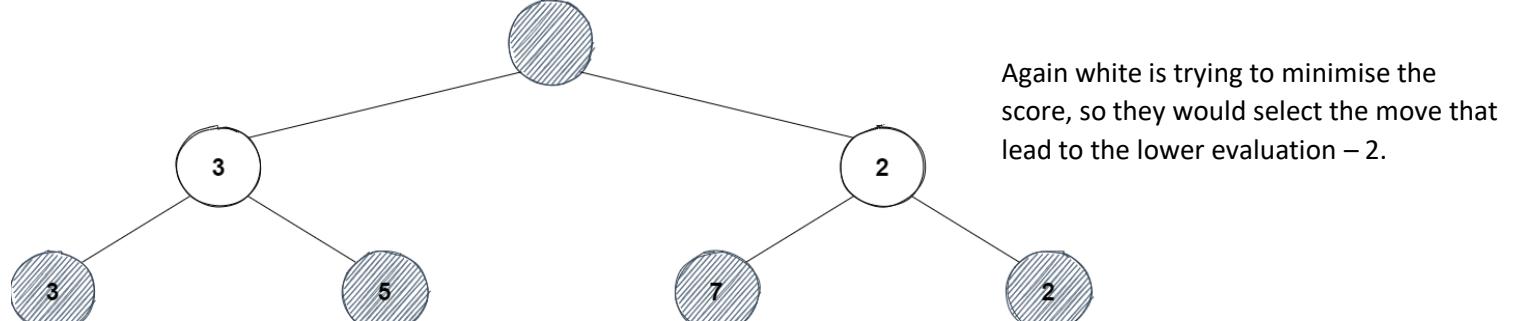
Say that white has two moves to choose from – one that leads to an evaluation of 3, and the other 5.



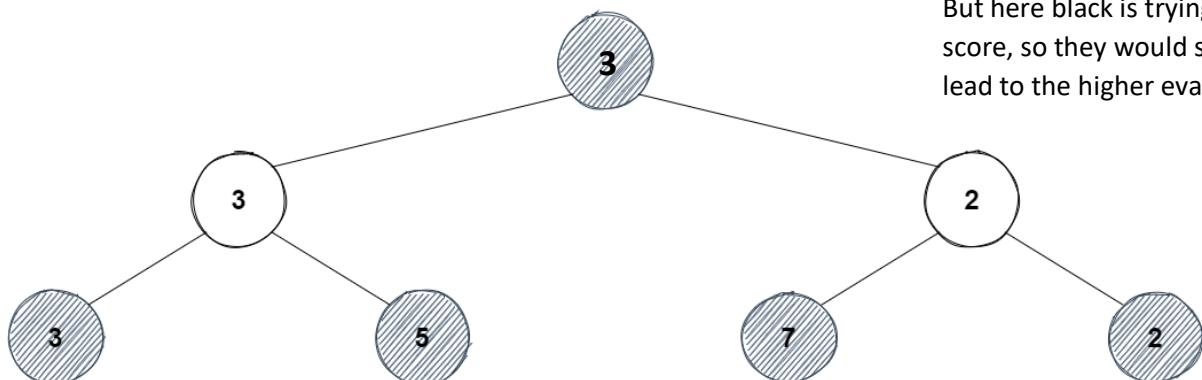
Here, white is trying to minimise the score, so they would select the move that lead to the lower evaluation – 3.



And here say that white has two moves to choose from – one that leads to an evaluation of 7, and the other 2.



Again white is trying to minimise the score, so they would select the move that lead to the lower evaluation – 2.



3

But here black is trying to maximise the score, so they would select the move that lead to the higher evaluation – 3.

This allows black to select the move that leads to the highest evaluation if white makes all the optimal moves to try and stop this.

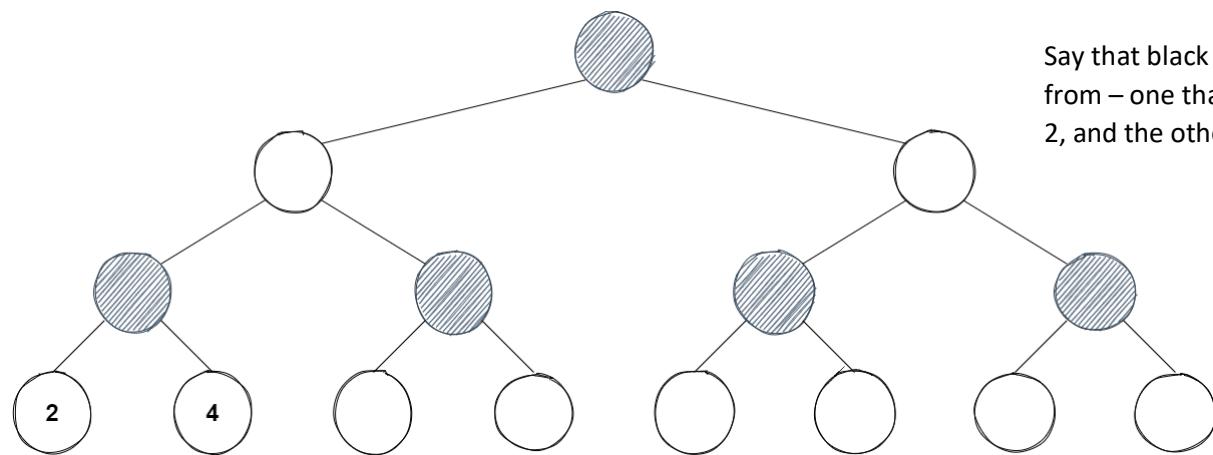
Instead of just finding the maximum or minimum value of two values however, in my program it will need to consider all possible legal moves of a particular player, so it will need to find the maximum / minimum of many more values.

Looking ahead at more moves would improve the performance of the AI, however it would also increase the time taken to determine which move to make, as adding more layers can very quickly increase the size of the tree. When I program this, I will need to balance the performance of the AI with the speed of the AI, by making sure I look far enough ahead to get a good move, without looking too far ahead, since if it takes too long to decide which move to make, the user will notice the delay, making the program less user friendly.

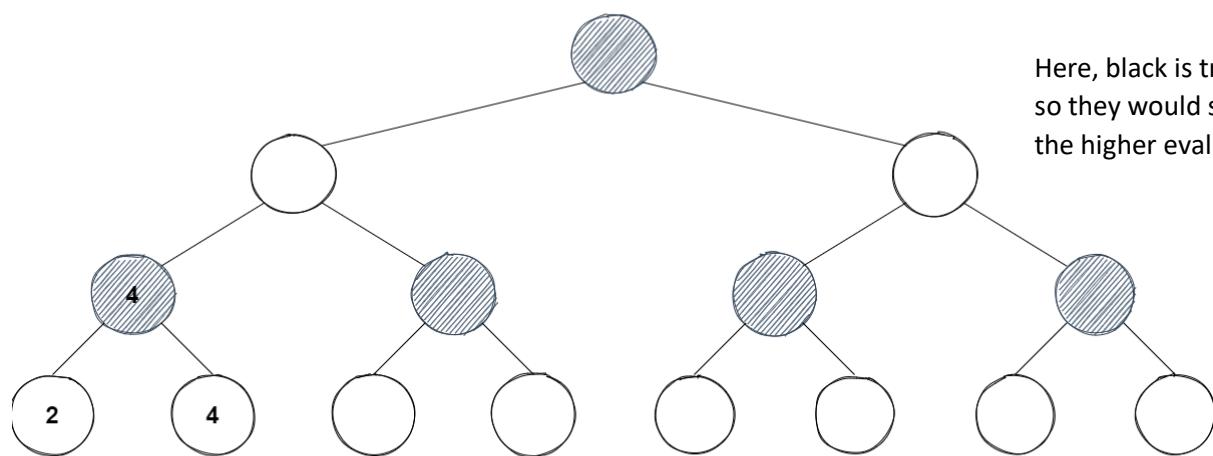
```
FUNCTION minimax(board, depth, maximisingPlayer)
    IF depth == 0 OR current player has no legal moves on board THEN
        RETURN evaluate(board)
    ENDIF

    IF maximisingPlayer THEN
        maxEvaluation = -infinity
        FOR each move that can be made by current player on board
            evaluation = minimax(board.makeMove(move), depth - 1, false)
            IF evaluation > maxEvaluation THEN
                maxEvaluation = evaluation
            ENDIF
        NEXT move
        RETURN maxEvaluation
    ELSE
        minEvaluation = infinity
        FOR each move that can be made by current player on board
            evaluation = minimax(board.makeMove(move), depth - 1, true)
            IF evaluation < minEvaluation THEN
                minEvaluation = evaluation
            ENDIF
        NEXT move
        RETURN minEvaluation
    ENDIF
ENDFUNCTION
```

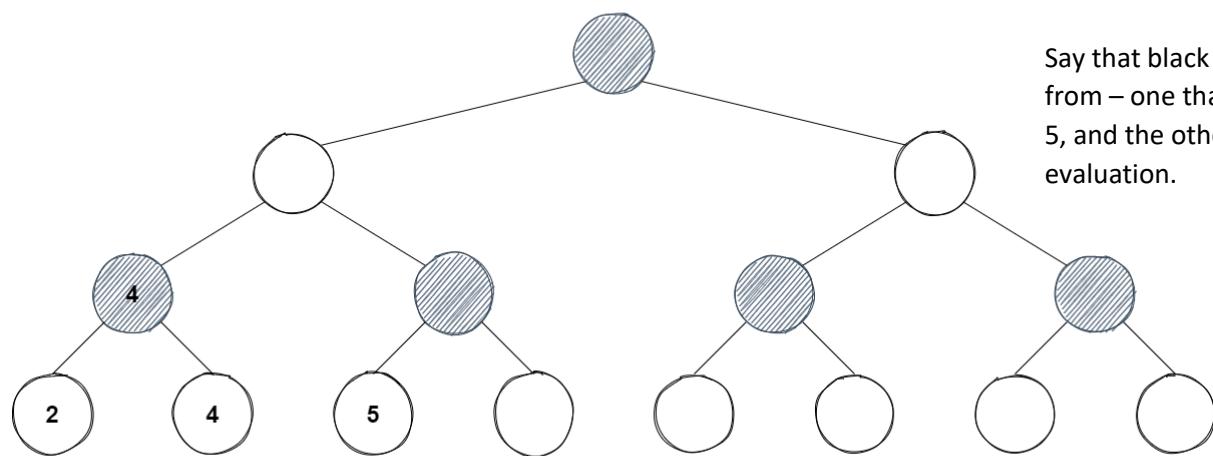
This minimax algorithm can be sped up slightly through the use of alpha-beta pruning. This speeds the algorithm up, as it avoids finding the evaluation of a board if it is not necessary.



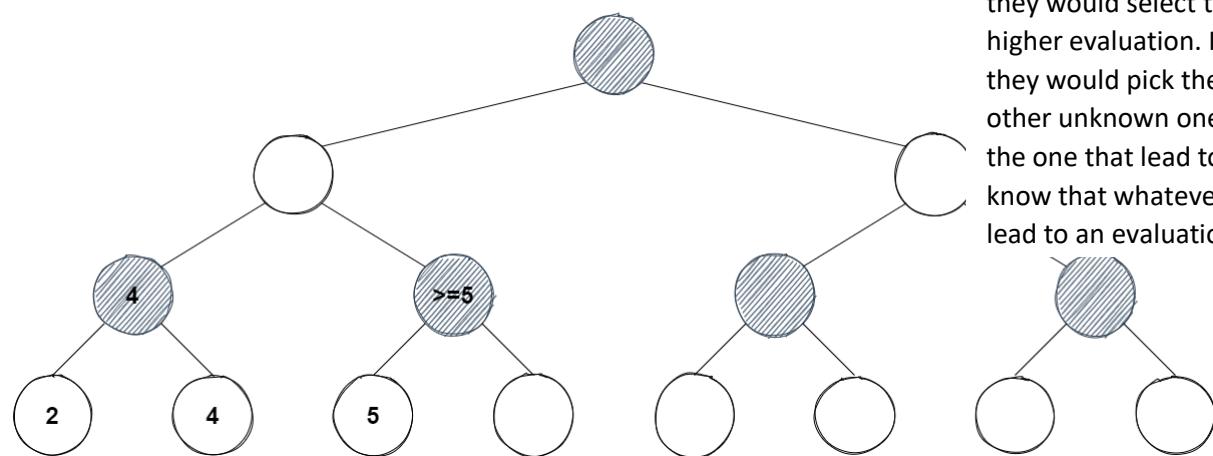
Say that black has two moves to choose from – one that leads to an evaluation of 2, and the other 4.



Here, black is trying to maximise the score, so they would select the move that lead to the higher evaluation, 4.

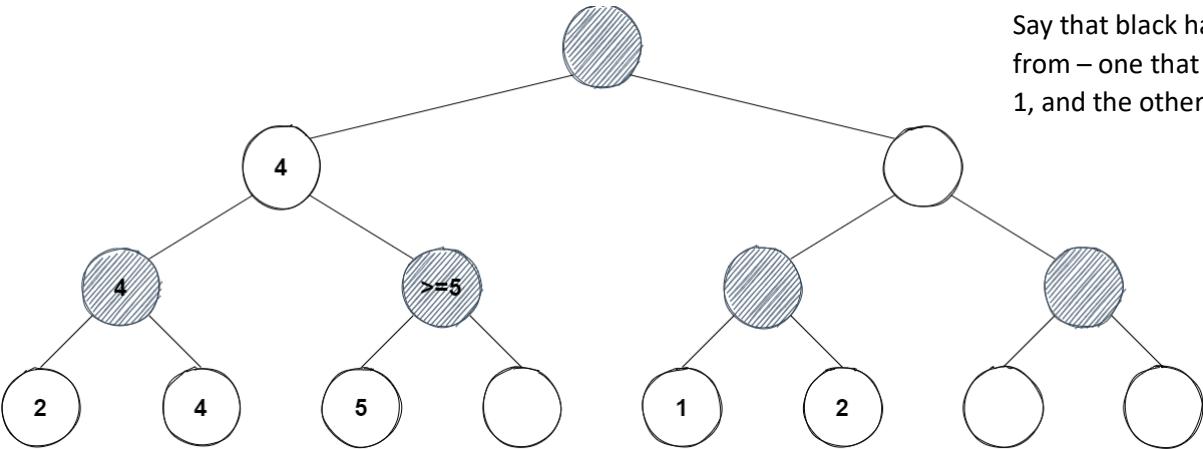


Say that black has two moves to choose from – one that leads to an evaluation of 5, and the other that leads to an unknown evaluation.

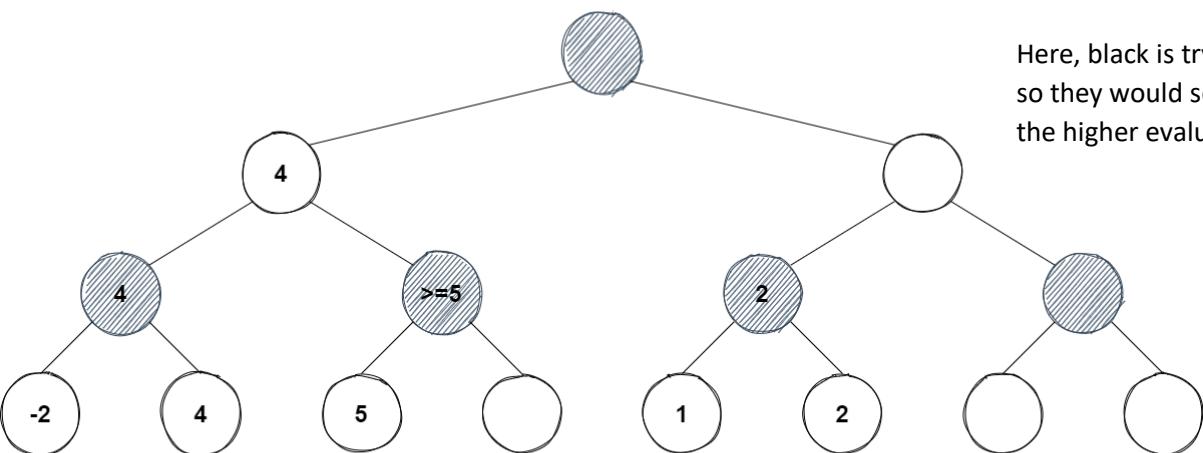


Here, black is trying to maximise the score, so they would select the move that lead to the higher evaluation. If 5 is the larger of the two, they would pick the one that lead to 5. If the other unknown one is larger, they would pick the one that lead to that evaluation. So, we know that whatever move black picks, it will lead to an evaluation greater than or equal to 5.

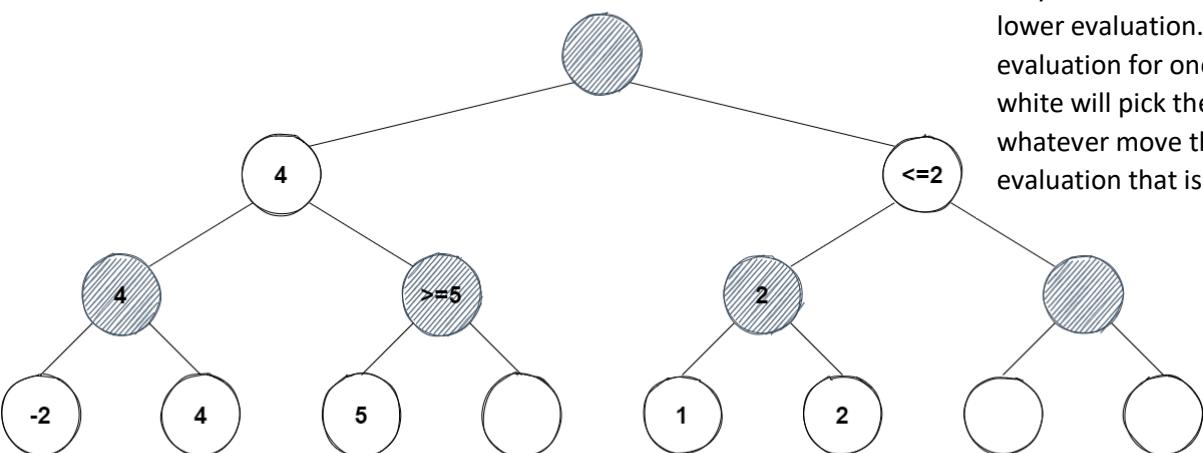
Say that black has two moves to choose from – one that leads to an evaluation of 1, and the other 2.



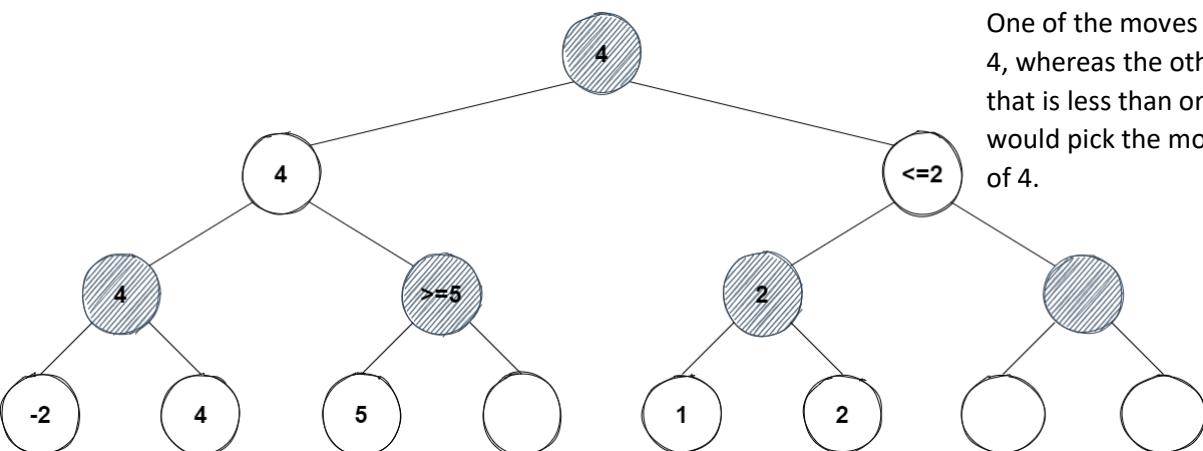
Here, black is trying to maximise the score, so they would select the move that lead to the higher evaluation, 2.



Here, white is trying to minimise the score, so they would select the move that lead to the lower evaluation. However we only know the evaluation for one of the two child nodes. Since white will pick the lower of the two values, whatever move they pick, it will lead to an evaluation that is less than or equal to 2.



Here, black is trying to maximise the score. One of the moves will lead to an evaluation of 4, whereas the other will lead to an evaluation that is less than or equal to 2. Therefore, black would pick the move that lead to an evaluation of 4.



So, using alpha-beta pruning, you do not need to evaluate as many boards, meaning that the minimax algorithm would run quicker than without using this. This could allow the computer to select a move quicker, meaning that the user would not need to wait as long between moves, or could allow the computer to search deeper in the tree in the same amount of time, making it more likely for them to come up with a better move, making the AI harder to play against, improving the experience for the user.

```

FUNCTION minimax(board, depth, maximisingPlayer, alpha, beta)
    IF depth == 0 OR current player has no legal moves on board THEN
        RETURN evaluate(board)
    ENDIF

    IF maximisingPlayer THEN
        maxEvaluation = -infinity
        FOR each move that can be made by current player on board
            evaluation = minimax(board.makeMove(move), depth - 1, false, alpha, beta)
            IF evaluation > maxEvaluation THEN
                maxEvaluation = evaluation
            ENDIF
            IF evaluation > alpha THEN
                alpha = evaluation
            ENDIF
            IF beta <= alpha THEN
                BREAK
            ENDIF
        NEXT move
        RETURN maxEvaluation
    ELSE
        minEvaluation = infinity
        FOR each move that can be made by current player on board
            evaluation = minimax(board.makeMove(move), depth - 1, true, alpha, beta)
            IF evaluation < minEvaluation THEN
                minEvaluation = evaluation
            ENDIF
            IF evaluation < beta THEN
                beta = evaluation
            ENDIF
            IF beta <= alpha THEN
                BREAK
            ENDIF
        NEXT move
        RETURN minEvaluation
    ENDIF
ENDFUNCTION

```

To evaluate the board, I will determine a score based on the pieces that are on the board, and where these pieces are.

Piece	Value
Pawn	100
Rook	500
Knight	300
Bishop	300
Queen	900
King	20000

Firstly, I will calculate the overall “value” of the pieces on the board – find the total value of all of the white pieces, and subtract this from the total value of all of the black pieces. Then, I will add a bonus to the overall score, depending on the type, colour and position of each piece (the values of these are taken from https://www.chessprogramming.org/Simplified_Evaluation_Function).

0	0	0	0	0	0	0	0
-50	-50	-50	-50	-50	-50	-50	-50
-10	-10	-20	-30	-30	-20	-10	-10
-5	-5	-10	-25	-25	-10	-5	-5
0	0	0	-20	-20	0	0	0
-5	5	10	0	0	10	5	-5
-5	-10	-10	20	20	-10	-10	-5
0	0	0	0	0	0	0	0

This is the value added for a particular position for the white pawns (for black pawns the board would be flipped, and the values multiplied by -1).

0	0	0	0	0	0	0	0
-5	-10	-10	-10	-10	-10	-10	-5
5	0	0	0	0	0	0	5
5	0	0	0	0	0	0	5
5	0	0	0	0	0	0	5
5	0	0	0	0	0	0	5
5	0	0	0	0	0	0	5
0	0	0	-5	-5	0	0	0

This is the value added for a particular position for the white rooks (for black rooks the board would be mirrored horizontally, and the values multiplied by -1).

50	40	30	30	30	30	40	50
40	20	0	0	0	0	20	40
30	0	-10	-15	-15	-10	0	30
30	-5	-15	-20	-20	-15	-5	30
30	0	-15	-20	-20	-15	0	30
30	-5	-10	-15	-15	-10	-5	30
40	20	0	-5	-5	0	20	40
50	40	30	30	30	30	40	50

This is the value added for a particular position for the white knights (for black knights the board would be mirrored horizontally, and the values multiplied by -1).

20	10	10	10	10	10	10	20
10	0	0	0	0	0	0	10
10	0	-5	-10	-10	-5	0	10
10	-5	-5	-10	-10	-5	-5	10
10	0	-10	-10	-10	-10	0	10
10	-10	-10	-10	-10	-10	-10	10
10	-5	0	0	0	0	-5	10
20	10	10	10	10	10	10	20

This is the value added for a particular position for the white bishops (for black bishops the board would be mirrored horizontally, and the values multiplied by -1).

20	10	10	5	5	10	10	20
10	0	0	0	0	0	0	10
10	0	-5	-5	-5	-5	0	10
5	0	-5	-5	-5	-5	0	5
0	0	-5	-5	-5	-5	0	5
10	-5	-5	-5	-5	-5	0	10
10	0	-5	0	0	0	0	10
20	10	10	5	5	10	10	20

This is the value added for a particular position for the white queens (for black queens the board would be mirrored horizontally, and the values multiplied by -1).

30	40	40	50	50	40	40	30
30	40	40	50	50	40	40	30
30	40	40	50	50	40	40	30
30	40	40	50	50	40	40	30
20	30	30	40	40	30	30	20
10	20	20	20	20	20	20	10
-20	-20	0	0	0	0	-20	-20
-20	-30	-10	0	0	-10	-30	-20

This is the value added for a particular position for the white kings (for black kings the board would be mirrored horizontally, and the values multiplied by -1).

Usability features

When user selects a piece, it will be highlighted in another colour (default, blue), to show them that they have selected that piece. This ensures that it is clear to the user whether or not they have already selected a piece – if not it, may result in them accidentally making a move they did not mean to, as they didn't realise they had already selected a piece to move, making the program more difficult to use.

Also, if the piece has any legal moves, those will be shown to the user in another colour (default, yellow). This will especially help those who are new to the game, or are unsure of the rules of chess, as it will explicitly show them which moves they are allowed to make with that piece, and which moves they are not allowed to make. This could also help them to learn how to play the game over time, as it will help them become familiar with how each piece is allowed move.

If a piece is selected and it has no legal moves, it will be highlighted in a different colour (default, red), to show that it cannot be moved. This will make it immediately obvious to the user that the piece cannot be moved, and that they should instead select another piece, making the program easier to use for those who are not sure of the rules, as they won't keep trying to move a piece that actually cannot be moved.

The user will not be allowed to make any moves that are not legal. If a move is selected which is not pseudo-legal, it will not be made. If a move is made by a player which puts / keeps themselves in check, it will be made, but then immediately undone (and will appear to the user that it has not been made, as the GUI is only updated after the move has been undone, not in between it being done and undone). If either of these occur, the user will be allowed to select another move. This helps those who are new to the game from accidentally making an illegal move, and prevents users from cheating by making moves that are not allowed, making the program easier to use.

If there are any previous moves, the most recent move taken will be highlighted to the user in another colour (default, orange). If the user is playing against the computer, this will help them to easily see what move the computer made, as it may occur quickly or the user may not notice at first which piece was moved. If they are playing against another human, this will help them to keep track of what their opponent is doing. Also, this is something that the end users have said they would like to be included in the program in order to make it easier to use.

A JLabel will be used to tell the user whose turn it is to move. This will help the users to keep track of whose turn it is to move, meaning that they will not have to remember this themselves, making it easier to play. Whilst this is not technically needed, as only pieces of the colour of the current player can be selected, this will make it clear to the user at a glance whose turn it is, without them needing to see if it is possible or not to select a particular coloured piece.

Only the side whose turn it is to move will be allowed to move. This will prevent the wrong user from accidentally making a move, or from a player accidentally moving their opponent's pieces instead of their own, or from a player cheating by taking multiple moves at once / by not making a move and skipping their go. This makes the program easier to use as users who are not sure of the rules will not be able to accidentally break them.

A JLabel will be used to tell the user whether they are in check. This will help those who are new to the game who may not be sure whether or not they are in check to know if they are, and will allow the user to very quickly know whether or not they are in check, without having to work this out

themselves. This is needed as whether or not the player is in check will determine the moves they are allowed to make.

When the game ends, a window will pop up, informing the user of the winner of the game. This will allow the user to immediately know that the game has ended as a window has opened, and will make it clear to them which player won the game, making it easier to use as there will be no confusion about which side has won and which hasn't or whether the game has ended or not.

In the menu bar of the program, there will be a help button, which the user can press to read how to play the game. This will help those who may be new to the game or who are not sure of the rules learn how to play it, by reading a set of instructions, explain the rules of chess to them, and how to use this particular program.

The buttons in the menu bar will have descriptive names / text (e.g. "help", "undo move", "surrender"), so that their function is clear to the user. This will ensure that the user is clear what each button does, helping to avoid them accidentally pressing the wrong thing, or not being clear what a particular button does / its function.

This is a summary of the usability features that will be included:

#	Usability feature
1	Highlighting selected piece in another colour
2	Highlighting the places the piece can move to in another colour
3	Highlighting previous move made in another colour
4	Highlighting piece in different colour if it has no legal moves
5	JLabel telling user whose should move next
6	JLabel telling the user who (if anyone) is in check.
7	Not allowing users to accidentally make illegal moves
8	Not allowing the wrong side to accidentally move
9	User is able to open a window containing a written explanation of the rules & how to use the program.
10	When the game ends, a window opens telling the user who won the game
11	Button allowing the user to undo a move.

Key variables, data structures, classes & validation

I will create a class called “Piece” which will have two attributes of type int to keep track of the type and colour of the piece, which is needed so that the program can keep track of the different types and colours of the pieces on the board for use in determining legal moves / check, to show the user the correct icon / text for the piece, and so the correct piece can be placed back on the board when a move is undone. This class will have method to get / set the colour & type, so that these attributes can be set and accessed by the ChessBoard, and to determine whether a particular move is valid, so the user / AI can be prevented from making illegal moves. I will then have 6 classes called “Pawn”, “Rook”, “Knight”, “Bishop”, “Queen” and “King” which will inherit from this “Piece” class, and each will have their own implementation of the method to determine whether or not a move is valid.

These will all inherit from the “Piece” class since they are all more specific types of piece, and will share some common functionality (i.e. all pieces have a colour and a type, and methods to get / set these, and the ability to check whether a move is valid), however different classes are needed, as they all require different implementations of the method to check whether a move is valid or not.

I will also create a class called “ChessBoard” which will hold an array of pieces representing the board, and deal with the functionality of making moves and keeping track of whose turn it is. It will contain a 2D 8x8 array of type “Piece”, where null represents that there is no piece at that location. This is used as it mimics the layout of an actual chessboard, an 8x8 board. It will also have a variable of type int to keep track of the current turn (so that only the correct player will be allowed to move), and a variable of type boolean, which will be true if the user selected to play against the computer, and false otherwise (so that it can determine whether the computer needs to make a move, or if it should wait for the user to input a move). It will also have a variable of type “MoveHistory”, which is needed to keep track of the moves taken, so that moves can be undone, and so that the previous move can be shown to the user. This “ChessBoard” class will contain methods to make / undo moves (allowing the user to actually play the game), to determine whether a player is in check (to determine whether or not a move is legal) and to make AI moves (to allow the user to play against the computer, as this is a key feature the client has requested). An important part of making the AI moves will be the evaluation function, which will assign a numerical value, indicating the value of the board, depending on the pieces on it, allowing it to select the move that will lead to the best board evaluation. And also, another important part is the implementation of the minimax algorithm to look ahead at possible moves, in order to determine the best possible one to take / the move that will lead to the optimum evaluation, so that the AI is able to play against the user.

The class “MoveHistory” will have an ArrayList which will hold strings representing the moves that have been made in the game. This is used as it will allow the moves to be stored in the order that they have been made, allowing the correct move to be undone, and is not of a fixed size, so can grow as more moves are made. It will have methods to get the size of the list of moves (so it can be determined how many moves have been made so far in the game, which is needed to determine whether a move can be undone or not), to add a move to the list (so that the move can be undone later if needed & so it can be shown to the user on the board), to remove a move from the end of the list (for undoing a move), and to look at the move at the end of the list (so that the user can be shown the previous move made in another colour). This functions like a stack.

In the JFrame I will store the 64 JButtons used in the GUI to represent the board in a 2D array. This is so that the positions of buttons in this array will correspond with the same position in the 2D array of Pieces in the “ChessBoard” class. This will allow me to easily access the buttons and iterate through them to update their text / icon to show the user correct piece or to change their background colours to show the selected piece / previous move / possible legal moves.

In the main JFrame I will also have methods to update the GUI (so that the text / icons of the buttons will change as the locations of the Pieces in the ChessBoard change), to show the user possible moves for the piece they selected (allowing the user to know which moves the piece can and cannot make), to reset the button colours (to get rid of any changes made to the colours e.g. when showing possible moves / which piece they selected), to end the game (so it does not continue even if a user is unable to make a valid move), and to allow the user to press buttons to select their move (allowing them to make a move and interact with the program). It will have a variable of type ChessBoard, which will contain the board and the functionality associated with it and two ints to store the start position (the location of the piece the user selects to move) and the end position (the place the user selects to move the piece to), so that the program knows which pieces to move within the array. If these variables store -1, then this indicates a position is yet to be selected, so that the program can differentiate between whether a user is trying to select a piece to move, or a location to move the piece to. This is needed so that it knows whether to make the move the user has selected, or wait for the user to input where to move it to. It will also have a variable of type Color, which will store the default background colour of the buttons, so that they can be reset if they are changed in the reset button colours process.

Validation will need to be used throughout the program in order to ensure that the user cannot enter invalid data, which could potentially cause an error to occur, or for the user to be able to perform actions they shouldn't (e.g. making an illegal move or the wrong player moving).

For example, each different type of piece will have their own implementation of a "isValidMove" function, which will be used to determine whether a particular move is pseudo-legal or not. One way in which this will be used will be to not allow the user to make a move if this function returns false, and instead have to make a different move. This will prevent the user from cheating by making an illegal move or a user who does not know the rules from accidentally moving a piece incorrectly. Another way in which this will be used will be to allow the AI to generate a list of possible legal moves to pick from. This will prevent the AI from making illegal moves.

Another example of validation in this program will be a function to determine whether or not a player is in check. In chess, a move is not legal if it puts / keeps yourself in check, and so this function will be used to prevent the user from making a move like this that is not legal. After the player makes their move, if they are then in check, the move is not legal, and so would be undone, and the user allowed to pick another move.

The user selects their moves by pressing buttons, and so there is no possibility of them entering their move in the wrong way (e.g. if they chose a move by typing some text instead, they may enter something invalid) - there is no way for them to choose a value that might be out of bounds of the board or to enter a move in an invalid format, only an incorrect combination of locations. This means that validation to check that the move the user has entered is within the bounds of the board is not needed – only whether the move is legal or not.

Another example of validation in this program is that only the pieces of the player whose turn it currently is will be able to move – places on the board that are empty, or do not belong to the current player will not be able to be selected as a piece to move, preventing the user from moving the wrong coloured piece / from a user trying to move an empty piece / from a user trying to take multiple moves / skip their move, all of which are not allowed.

Testing

Throughout the development of this program, I will need to test it, in order to avoid bugs that may prevent the program from working properly, ensure that it is suitable for the end users, and to make sure that it meets my client's expectations.

One way in which I will do this is through black box testing. This involves looking at the functionality of the program, and determining whether it is working as expected. If errors are identified through black box testing, I can use white box testing to help fix these if needed. To help with this, I will use the following table, which outlines what the tests are, why that test needs to be performed, and the expected result of the test.

#	Test	Reason	Expected Result
1	Is user able to select who to play against at the beginning of a game?	Client has stated that they would like the option to play both against an AI and another human	Valid: program is started by user - User is shown dialog asking them whether they would like to play against the computer or not Valid: Game is ended due to user surrendering - User is shown dialog asking them whether they would like to play against the computer or not Valid: game is ended due to checkmate / stalemate - User is shown dialog asking them whether they would like to play against the computer or not Invalid: new game is not about to start (i.e. program has not just started and game has not just finished) – user should not be shown window asking them to select an opponent Borderline: N/A
2	Is user shown the main window containing the 64 buttons representing the GUI?	Allows the user to see the board and input their moves	Valid: After user has selected who to play against, window is shown, containing 64 JButtons and the JMenuBar. Invalid:
3	Is the user prevented from starting until who they are playing against has been picked?	Program needs to know whether to wait for user input or to allow the computer to play	Valid: Dialog asking user who they would like to play against is open – User is not allowed to select pieces on the board to move Invalid: user has selected opponent and dialog asking user who they would like to play against is not currently open – user should be allowed to input moves. Borderline: N/A
4	Is user stopped from making a move that is not	One of the reasons a computational solution was needed was to ensure	Invalid: illegal move inputted - move should not be made

	pseudo-legal for a pawn?	that players are not able to make illegal moves	Valid: legal move inputted – move should be made. Borderline: N/A
5	Is user stopped from making a move that is not pseudo-legal for a knight?	One of the reasons a computational solution was needed was to ensure that players are not able to make illegal moves	Invalid: illegal move inputted - move should not be made Valid: legal move inputted – move should be made. Borderline: N/A
6	Is user stopped from making a move that is not pseudo-legal for a rook?	One of the reasons a computational solution was needed was to ensure that players are not able to make illegal moves	Invalid: illegal move inputted - move should not be made Valid: legal move inputted – move should be made. Borderline: N/A
7	Is user stopped from making a move that is not pseudo-legal for a bishop?	One of the reasons a computational solution was needed was to ensure that players are not able to make illegal moves	Invalid: illegal move inputted - move should not be made Valid: legal move inputted – move should be made. Borderline: N/A
8	Is user stopped from making a move that is not pseudo-legal for a queen?	One of the reasons a computational solution was needed was to ensure that players are not able to make illegal moves	Invalid: illegal move inputted - move should not be made Valid: legal move inputted – move should be made. Borderline: N/A
9	Is user stopped from making a move that is not pseudo-legal for a king?	One of the reasons a computational solution was needed was to ensure that players are not able to make illegal moves	Invalid: illegal move inputted - move should not be made Valid: legal move inputted – move should be made. Borderline: N/A
10	Is user stopped from making a move that puts / keeps themselves into check?	A move that puts themselves into check is not legal and so should not be allowed to be made	Invalid: move that puts / keeps themselves in check is inputted - move should not be made Valid: move inputted that does not leave / put themselves in check – move should be made. Borderline: N/A
11	Is player allowed to make another move if the one they select is illegal?	User should be allowed to make a valid move and their turn should not be skipped because they did not	Invalid: user selects illegal move - Should not go onto the next turn and the same payer should be able to input a new move Valid: user selects legal move – should move onto the next turn and not allow the same player to move again until the other player has moved. Borderline: N/A

12	Does the JLabel showing who (if anyone) is in check update correctly?	So that a player will know straight away whether they are in check or not and will not need to work this out for themselves	<p>Valid: white is in check – text of JLabel should be set to “White is in check”, black is in check – text of JLabel should be set to “Black is in check”.</p> <p>Borderline: no one is in check – text of JLabel should be set to “”.</p> <p>Invalid: N/A</p>
13	Is piece the user selects shown in a different colour?	So that the user can easily see which piece they have selected, helping them to avoid accidentally making a move they didn't mean to	<p>Valid: user selects a piece of the colour of the current turn - the background colour of this button should change (default to cyan)</p> <p>Invalid: user selects piece of the wrong colour or user selects an empty square – background colour of the button should not be changed</p> <p>Borderline: N/A</p>
14	Is the user shown in another colour the possible places they can move their selected piece to?	So that they can quickly see where they can and cannot move the piece to, especially helping those who are unsure of how each piece moves.	<p>Valid: user selects a piece with valid moves – <i>all</i> of the legal moves should be shown to the user by changing the background colour of the button (default to yellow), and none that are illegal should be shown.</p> <p>Borderline: user selects piece with no valid moves – no possible moves should be shown to the user in yellow.</p> <p>Invalid: piece of wrong colour is selected – no possible moves should be shown to the user.</p>
15	Is the previous move made shown in a different colour?	Helps the user to know what move the computer made, and is a feature the end users said they would like to have	<p>Valid: move has been made - the background colours of the buttons at the start and end location of the piece that was last moved should be changed (Default to orange)</p> <p>Invalid: no moves have been made – background of buttons should not be attempted to be changed.</p> <p>Borderline: N/A</p>
16	Does the game end when someone is in checkmate?	Game should end if someone is in checkmate as the player in checkmate can no longer move	<p>Valid: Player is in checkmate - user should no longer be able to input moves / the game should not continue, and the board should be reset.</p> <p>Borderline: Player is in stalemate (no legal moves, not in check) - user should no longer be able to input moves / the game should not continue, and the board should be reset.</p>

			Invalid: No players are in checkmate or stalemate – game should continue and not end.
17	When the game ends is the user shown who won?	Helps those who may not be familiar with the game know who has won	Valid: white is in checkmate - dialog should be shown to user telling them that black has won, user should no longer be able to input moves, when window is closed, board should be reset. Valid: black is in checkmate – dialog should be shown to the user telling them that white has won, user should no longer be able to input moves, when window is closed by the user, the board should be reset. Borderline: player in stalemate - dialog should be shown to the user telling them that the game was a stalemate, user should no longer be able to input moves, when window is closed by the user, the board should be reset. Invalid: N/A
18	Does the game end when the user presses surrender?	Something which the end users have requested	Valid: user presses surrender button - game should end, board should be reset, and user should be again asked who they would like to play against. Borderline: N/A Invalid: N/A
19	Can the computer make a move?	Allows the user to play a game against the AI	Valid: User selected to play against computer - after the user has moved, the computer should determine what move to make, and then make that move. Invalid: User chose to play against another person – after the user has made a move, the computer should not make a move, and should wait for the other player to input a move. Borderline: N/A
20	Is user shown an explanation of the rules when they press help?	Helps those who may not know how to play the game learn how it works, allowing it to be played by more people	Valid: user presses help button - dialog should be displayed showing written explanation of how to play the game Borderline: N/A Invalid: N/A

21	Can the user save the game to a file?	Feature end users requested, allows user to save and come back to a game at another time.	Valid: user presses save button - user should be asked to select a file, and string representing the board should be written to this file Invalid: User selects an invalid file – user should be informed that the file they selected is not valid and nothing should be written. Borderline: N/A
22	Can the user open a previous game from a file?	Feature end users requested, allows them to open games they have previously saved	Valid: User selects file containing a string in the correct format - the string representing the board should be read from this file and loaded into the board Borderline: User does not select a file – nothing should happen to the board Invalid: User selects a file that does not contain a string in the correct format - user should be told that the file they selected is not valid, and nothing should happen to the board.

Another way I will do this is through white-box testing. I will use this to help me fix any errors that are found through black-box testing, by looking at the code itself, and going through it (e.g. by using trace tables) to find any bugs, so that these can be fixed.

Another way I will do this is through beta testing. This involves allowing the end users to use the product, and report any errors found and to share their opinions on the program through the use of a survey. This will help me to know if there are any bugs that hadn't been identified previously, what they think of the user interface, and whether they would like the program to include any additional features. This will be especially useful in identifying errors relating to the user attempting to use the program in an unexpected way, and in determining how successful the usability features are.

These are the questions which I will ask the beta testers after they have used the program:

- 1) How easy did you find the program to use? (1-5)

I will ask this so that I can get an overall view of how effective the usability features were for the program, to determine whether this is an area of the program that needs to be worked on more. This is because how usable the program is will be difficult for me to judge, as I am familiar with the program, and so I need to ask the beta testers for their experiences with the program.

- 2) Are there any features which could be added that you think would make the program easier to use?

I will ask this so that I can find out what could be implemented next in the program in order to improve its usability.

- 3) Was the piece you selected being highlighted in blue a useful feature? (Y/N/Don't know)
- 4) Was the piece you selected being highlighted in red if it had no legal moves a useful feature? (Y/N/Don't know)

- 5) Were the places the selected piece can move to being highlighted in yellow a useful feature? (Y/N/Don't know)
- 6) Was the previous move made being highlighted in orange a useful feature? (Y/N/Don't know)
- 7) Was the text telling you whose turn it was to move a useful feature? (Y/N/Don't know)
- 8) Was the text telling you who was in check a useful feature? (Y/N/Don't know)
- 9) Was the help window containing instruction on how to use the program a useful feature? (Y/N/Don't know)

I will ask questions 3-9 so that I can see specifically what the users did and didn't find helpful, so that I know what to keep in the program, and what could be removed or changed to work better. This will give me a clear indication of how effective each usability feature is, so that I know which to improve.

- 10) What did you think of the design of the GUI?
- 11) Did you think the GUI had a minimalistic design?

I will ask questions 10 & 11 so that I can see whether the users found the GUI easy and intuitive to use, or whether it could be designed in an easier to use way. This will also allow me to see whether they like the colour scheme / fonts that are used, or whether these could be changed. Also, a minimalistic design is one of the success criteria, and so asking this will let me know whether the users believe that this has been met or not, as it could be quite subjective.

- 12) Did you encounter any errors whilst using this program?

I will ask this so that I can find out if there are any errors in the program which were not picked up by any previous testing, so that these can be fixed.

- 13) Any other comments on the program?

I will ask this so that the beta testers can give any ideas or thoughts about the program which were not covered by any of the previous questions, so that improvements can be made to the program based off of these.

And finally, another way I will do this is through acceptance testing. This will involve showing the client the program, and evaluating it against the previously agreed success criteria, in order to determine whether the program will meet the needs of the client, and to find out if any improvements or additional features should be added. The following table contains the criteria I will discuss with my client, and how their feedback will be recorded.

#	Criteria	Comments from client	Criterion met? (fully / partially / not)
1	Graphical User Interface (GUI) shows board		
2	GUI has a minimalistic design		
3	GUI uses the colours the end-users have specified		

4	GUI uses distinctive images to represent pieces		
5	User can choose who to play against		
6	User can input their move		
7	Program can switch turns		
8	Show previous move of opponent		
9	AI can play		
11	Can find all possible legal moves for a particular piece		
12	Only makes a move if it is legal		
13	Can determine whether either side is in check		
14	Can tell the user if they are in check		
15	Can determine whether either side is in checkmate		
16	Can inform the user who has won		
17	Allows user to resign		
18	Allows user to undo a move		
19	Allows user to save current board state to a file		
20	Allows user to load previous board state from a file		
21	A way of showing the user a written explanation of the rules		

These correspond to the success criteria, so that every agreed criteria can be checked by the client, to ensure that the program is suitable for them, and so that I know where any changes need to be made.

Additionally, I will ask them whether there are any other features or comments they have / want to be included, in case their requirements for the program change as the program is developed.

Implementation

Stages of development

These are the main stages that I initially plan on implementing, however these may change as the program is developed if it is identified that something else should be added, or if something should be removed / implemented in a different way:

- Creating main data structures / classes
- Creating GUI
- Connecting data structures and GUI
- Allowing user to select moves
- Making moves
- Check if move is pseudo-legal
- Determine check
- Determine checkmate
- Showing user possible moves
- Showing user previous moves
- Taking turns
- Allowing user to select opponent
- Allowing computer to make moves
- Allowing user to surrender
- Allowing user to undo moves
- Showing help window

Creating main data structures

Development

```
public abstract class Piece {

    private int type; // 1 = pawn, 2 = rook, 3 = knight, 4 = bishop, 5 = queen, 6 = king
    private int colour; // 0 = white, 1 = black

    public Piece(int type, int colour) {
        this.type = type;
        this.colour = colour;
    }

    public abstract boolean isValidMove(String move, Piece[][] board);

    @Override
    public abstract String toString();

    public int getColour() {
        return this.colour;
    }

    public int getType() {
        return this.type;
    }

    public void setType(int type) {
        this.type = type;
    }

    public void setColour(int colour) {
        this.colour = colour;
    }

}
```

To begin with, I created a “Piece” class which contains the functionality common to all the different types of pieces. It allows the type & colour of each piece to be stored, and contains methods to get / set these variables. This class is an abstract class, as I do not need to instantiate “Piece” objects directly, but instead instantiate “Pawn”, “Rook”, “Knight”, etc objects, which will inherit from this “Piece” class. I have declared two abstract methods – “isValidMove” & “toString”, so that any classes which inherit from this class are forced to implement these methods. This allows each different type of chess piece to determine whether a move is valid, according to the rules for that piece. The “toString” method will be used to display the piece to the user by setting the text of the button to what this function returns.

I have decided to use a class for this because all of the pieces will have a similar functionality, and have many methods which are common to all of them, and so using a class will mean that I do not have to re-write the same code multiple times. Making it abstract will mean that all of the pieces will have to implement particular methods (toString, isValidMove), which are essential for the program to function, eliminating the possibility of any of the classes not containing these methods, which would cause the program to not work correctly.

```

public class Pawn extends Piece {

    public Pawn(int colour) {
        super(1, colour);
    }

    @Override
    public boolean isValidMove(String move, Piece[][] board) {
        return true;
    }

    @Override
    public String toString() {
        if (this.getColour() == 0) {
            return "♟";
        } else {
            return "♙";
        }
    }

}

```

```

public class Bishop extends Piece {

    public Bishop(int colour) {
        super(4, colour);
    }

    @Override
    public boolean isValidMove(String move, Piece[][] board) {
        return true;
    }

    @Override
    public String toString() {
        if (this.getColour() == 0) {
            return "♝";
        } else {
            return "♗";
        }
    }

}

```

```

public class Queen extends Piece {

    public Queen(int colour) {
        super(5, colour);
    }

    @Override
    public boolean isValidMove(String move, Piece[][] board) {
        return true;
    }

    @Override
    public String toString() {
        if (this.getColour() == 0) {
            return "♛";
        } else {
            return "♕";
        }
    }

}

```

```

public class Rook extends Piece {

    public Rook(int colour) {
        super(2, colour);
    }

    @Override
    public boolean isValidMove(String move, Piece[][] board) {
        return true;
    }

    @Override
    public String toString() {
        if (this.getColour() == 0) {
            return "♜";
        } else {
            return "♖";
        }
    }

}

```

```

public class Knight extends Piece {

    public Knight(int colour) {
        super(3, colour);
    }

    @Override
    public boolean isValidMove(String move, Piece[][] board) {
        return true;
    }

    @Override
    public String toString() {
        if (this.getColour() == 0) {
            return "♞";
        } else {
            return "♘";
        }
    }

}

```

```

public class King extends Piece {

    public King(int colour) {
        super(6, colour);
    }

    @Override
    public boolean isValidMove(String move, Piece[][] board) {
        return true;
    }

    @Override
    public String toString() {
        if (this.getColour() == 0) {
            return "♚";
        } else {
            return "♔";
        }
    }

}

```

Here, I have created classes for each of the different types of chess piece, which all inherit from the “Piece” class. Each constructor takes one parameter – the colour of the piece, and then sets the type and colour of the piece using the constructor of the super class.

Each class also implements the “isValidMove” method, however I have not implemented the logic for this yet, and so as a placeholder, just returns true, so I can test whether the pieces can move at all, i.e. for now, all moves will be considered legal. Once the pieces are able to move around on the board correctly, I can begin to validate these moves by implementing the “isValidMove” function properly.

These classes also all implement the “toString” method, returning a String consisting of a single character representing the type and colour of the piece. This will be used to allow the pieces can be printed to the console, so that I can check whether the program is working correctly, e.g. checking if the pieces are in the right places.

```

public class ChessBoard {

    private Piece[][] board;
    private int currentTurn; // even num = white, odd num = black

    public ChessBoard() {
        this.board = new Piece[][] {
            {new Rook(1), new Knight(1), new Bishop(1), new Queen(1), new King(1), new Bishop(1), new Knight(1), new Rook(1)},
            {new Pawn(1), new Pawn(1)},
            {null, null, null, null, null, null, null, null},
            {new Pawn(0), new Pawn(0)},
            {new Rook(0), new Knight(0), new Bishop(0), new Queen(0), new King(0), new Bishop(0), new Knight(0), new Rook(0)}
        };

        this.currentTurn = 0;
    }

}

```

Then, I started to create the “ChessBoard” class. I added two attributes – a 2D array of type “Piece”, which will store all of the pieces on the board, with null representing an empty square, and a variable of type int, to store the current turn. I used a 2D array of type piece to represent the board, as this replicates the layout of an actual chessboard, making implementing the algorithms for checking whether a move is legal more intuitive, and allows all of the different types of pieces to be held in it. After each move, currentTurn will be incremented by 1, and whether it is even or odd will determine whose turn it is. I have done this as it is easier to increment the value by 1 each time a move is made, rather than for example changing a char from ‘w’ to ‘b’, and this allows how far through the game (i.e. how many moves have been made) to be kept track of, which could be useful in keeping track of whether a move can be undone or not later. In the constructor, I initialised the “board” array, so that it contains the pieces in the correct locations, because at the start of the game the pieces will always need to be in this layout, and the game cannot be played if there are no pieces on the board, and set “currentTurn” to 0, because at the start of the game no moves have been made.

```

import java.util.ArrayList;

public class MoveHistory {

    private ArrayList<String> moves;

    public MoveHistory() {
        this.moves = new ArrayList<>();
    }

    public void add(String move) {
        this.moves.add(move);
    }

    public String pop() {
        String move = this.moves.get(this.moves.size() - 1);
        this.moves.remove(this.moves.size() - 1);
        return move;
    }

    public String peek() {
        return this.moves.get(this.moves.size() - 1);
    }

    public int getSize() {
        return this.moves.size();
    }

}

```

Moves are represented by strings, according to the image on the right, and so are stored in an arraylist of strings. The start position and end position need to be stored so that the squares the piece moved between are known, which are needed to move the correct piece to the correct position if the move is undone. Also, the type of piece that was captured (if any) is stored so that if the move is undone, both the piece that was moved can go back to the previous location, and the piece that was captured can be put back onto the board. I chose to store the moves in a string as this would take up less memory than if a class ‘Move’ was created and objects instantiated for every move made, which is especially important as potentially a large number of moves may need to be stored.

```

public Piece[][] getBoard() {
    return this.board;
}

```

Then, I created the class called “MoveHistory”, in order to keep track of the moves taken in the game, so that I will be able to undo moves later, as in order to undo a move, the previous move(s) taken need to be accessed. This uses an ArrayList to store each move, as it is a dynamic data structure, which stores data in an ordered way, allowing the size of it to change as moves are added / removed, and for the moves to be stored in chronological order, helping with undoing moves (the last move made will be at the end of the arraylist). This class contains methods to add a move to the list (so that moves can be added as they are made), pop a move from the list (remove the item at the end of the list and return it, which will be needed when undoing a move), to peek at an item (returns the item at the end of the list, without removing it, which is needed when showing the user their previous move), and to get the size of the list (to check that there are moves left that can be removed, again useful when a move needs to be undone later).

Then, I added an attribute of type MoveHistory to the ChessBoard class, so that the moves taken can be stored (as it is the ChessBoard class which will handle the functionality of making moves), which is needed so that they can be later undone / shown to the user. I then instantiated it in the constructor of the ChessBoard class:

```

private MoveHistory moveHistory;

this.moveHistory = new MoveHistory();

```

Start Position	End Position	Type of Captured Piece		

Then, I added a function to the ChessBoard class which returns the 2D array representing the board, which will be needed to connect the data structures to the GUI, by iterating through this array to find out which piece is where, and displaying this in the correct place in the GUI to the user, so that they can actually see the board.

Testing

Firstly, I checked that the board is initialised with all of the pieces in the correct positions, by using the “getBoard” method of the “ChessBoard” class, and printing this array of pieces, so that I can see whether the ChessBoard class is working correctly so far, or whether I need to go back and change it.

```
run:  
X ♕ ♗ ♘ ♙ ♖ ♔ X  
i i i i i i i i  
-----  
-----  
-----  
-----  
-----  
-----  
-----  
-----  
BUILT SUCCESSFUL (total time: 0 seconds)
```

```
ChessBoard board = new ChessBoard();  
Piece[][] pieces = board.getBoard();  
  
for(int i = 0; i < 8; i++) {  
    for(int j = 0; j < 8; j++) {  
        if(pieces[i][j] != null) {  
            System.out.print(pieces[i][j] + " ");  
        } else {  
            System.out.print("_ ");  
        }  
    }  
    System.out.println();  
}
```

Here, you can see that this is working correctly – the pieces / empty spaces are displayed in their correct locations.

Then I checked whether the “getColour” and “getType” functions of the “Piece” class were working as expected.

```
run:  
Piece one:  
Type: 2  
Colour: 1  
Piece two:  
Type: 1  
Colour: 0  
BUILT SUCCESSFUL (total time: 0 seconds)
```

```
ChessBoard board = new ChessBoard();  
Piece[][] pieces = board.getBoard();  
  
Piece pieceOne = pieces[0][0]; // Should be a black rook  
System.out.println("Piece one:");  
System.out.println("Type: " + pieceOne.getType());  
System.out.println("Colour: " + pieceOne.getColour());  
  
Piece pieceTwo = pieces[6][5]; // Should be a white pawn  
System.out.println("Piece two:");  
System.out.println("Type: " + pieceTwo.getType());  
System.out.println("Colour: " + pieceTwo.getColour());
```

A type of 2 is a rook, a type of 1 is a pawn, a colour of 1 is black, and a colour of 0 is white, and so this appears to be working correctly.

Then I tested whether “MoveHistory” was working correctly. To begin with, I temporarily added a procedure to print the list of moves, so that I can see if this is working correctly or not. Then, I tested whether moves could be added to the MoveHistory correctly by adding some moves, and printing the list, to check they have been added and that they are stored in the correct order.

```
run:  
1) a  
-----  
1) a  
2) b  
-----  
1) a  
2) b  
3) c  
BUILT SUCCESSFUL (total time: 0 seconds)
```

```
ChessBoard board = new ChessBoard();  
Piece[][] pieces = board.getBoard();  
  
MoveHistory mh = new MoveHistory();  
mh.add("a");  
mh.printMoves();  
System.out.println("-----");  
mh.add("b");  
mh.printMoves();  
System.out.println("-----");  
mh.add("c");  
mh.printMoves();
```

Here, it can be seen that the correct moves are printed, and they are stored in the correct order (the most recent move at the end of the list), and so this is working correctly.

Next, I tested whether moves could be popped from the list correctly. Here, the most recent move (i.e. the move at the end of the list) should be removed.

```
run:  
1) a  
2) b  
3) c  
-----  
1) a  
2) b  
BUILD SUCCESSFUL (total time: 0 seconds)
```

```
ChessBoard board = new ChessBoard();  
Piece[][] pieces = board.getBoard();  
  
MoveHistory mh = new MoveHistory();  
mh.add("a");  
mh.add("b");  
mh.add("c");  
mh.printMoves();  
System.out.println("-----");  
mh.pop();  
mh.printMoves();
```

Here, you can see that the move at the end of the list, "c", is removed correctly.

```
ChessBoard board = new ChessBoard();  
Piece[][] pieces = board.getBoard();  
  
MoveHistory mh = new MoveHistory();  
mh.printMoves();  
System.out.println("-----");  
mh.pop();  
mh.printMoves();
```

```
run:  
-----  
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: -1  
at java.util.ArrayList.elementData(ArrayList.java:422)  
at java.util.ArrayList.get(ArrayList.java:435)  
at testing.MoveHistory.pop(MoveHistory.java:21)  
at testing.Testing.main(Testing.java:13)  
C:\Users\carys cooper\AppData\Local\NetBeans\Cache\8.1\executor-snippets\run.xml:53: Java returned: 1  
BUILD FAILED (total time: 0 seconds)
```

However, when there are no moves in the list, an `IndexOutOfBoundsException` occurs, as it does not check whether the list is empty or not. If the list is empty, no move should be removed in order for the error to be avoided.

```
ChessBoard board = new ChessBoard();  
Piece[][] pieces = board.getBoard();  
  
MoveHistory mh = new MoveHistory();  
mh.printMoves();  
System.out.println("-----");  
mh.peek();  
mh.printMoves();
```

```
run:  
-----  
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: -1  
at java.util.ArrayList.elementData(ArrayList.java:422)  
at java.util.ArrayList.get(ArrayList.java:435)  
at testing.MoveHistory.peek(MoveHistory.java:30)  
at testing.Testing.main(Testing.java:13)  
C:\Users\carys cooper\AppData\Local\NetBeans\Cache\8.1\executor-snippets\run.xml:53: Java returned: 1  
BUILD FAILED (total time: 0 seconds)
```

This same error occurs for the same reason with the "peek" method as well, and so these both need to be changed to check the size of the list first.

I then tested whether the "getSize" method returned the correct length of the list by adding some moves to the list and calling this function.

```
run:  
1) a  
Size: 1  
-----  
1) a  
2) a  
3) a  
4) a  
5) a  
6) a  
Size: 6  
BUILD SUCCESSFUL (total time: 0 seconds)
```

```
ChessBoard board = new ChessBoard();  
Piece[][] pieces = board.getBoard();  
  
MoveHistory mh = new MoveHistory();  
mh.add("a");  
mh.printMoves();  
System.out.println("Size: " + mh.getSize());  
System.out.println("-----");  
mh.add("a");  
mh.add("a");  
mh.add("a");  
mh.add("a");  
mh.add("a");  
mh.printMoves();  
System.out.println("Size: " + mh.getSize());
```

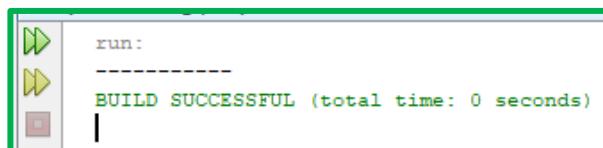
Here you can see that this method is working correctly.

Remedial

```
public String pop() {
    if (this.moves.size() > 0) {
        String move = this.moves.get(this.moves.size() - 1);
        this.moves.remove(this.moves.size() - 1);
        return move;
    }
    return "";
}
```

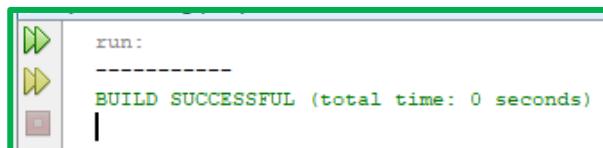
```
public String peek() {
    if (this.moves.size() > 0) {
        return this.moves.get(this.moves.size() - 1);
    }
    return "";
}
```

In order to fix the `IndexOutOfBoundsException` error that occurred for the “peek” and “pop” methods, I changed these so that they check whether there are any moves in the list or not before attempting to remove / get the move from the end of the list, as this will avoid the `indexoutofbounds` error.



A screenshot of a terminal window titled "run:" showing a successful build message: "BUILD SUCCESSFUL (total time: 0 seconds)".

When I tested these in the same way as before, no error occurs, and no moves are printed, as expected.



A screenshot of a terminal window titled "run:" showing a successful build message: "BUILD SUCCESSFUL (total time: 0 seconds)".

Review

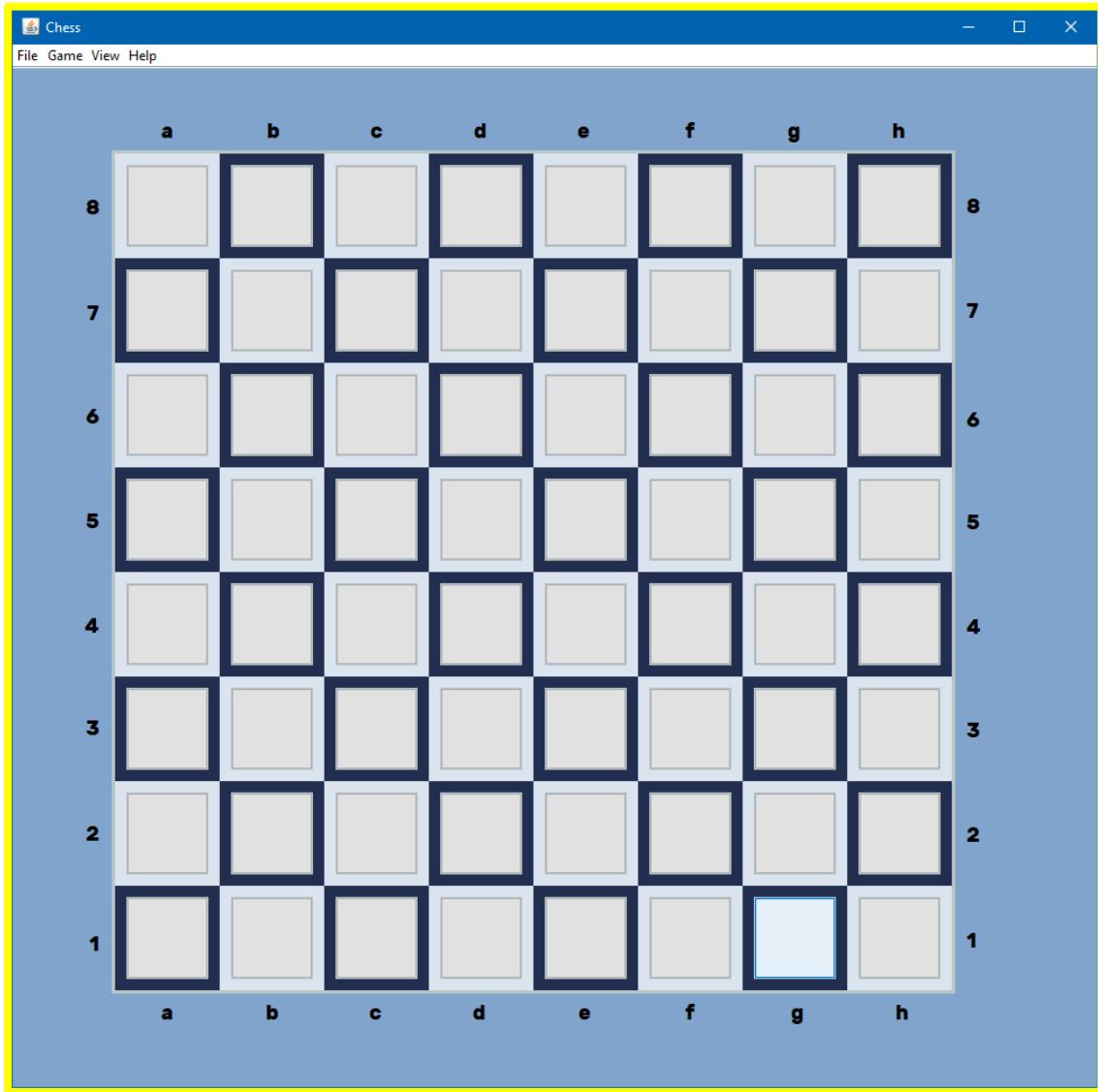
Here, the main data structures / classes that will be used in this program have been created, and the functionality they currently have has been tested.

Now that these have been created, I can begin to create the GUI, and then link these together, which will allow the user to interact with the program via the GUI, and for the data structures to be updated by the user.

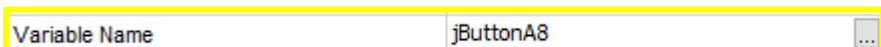
Link to success criteria: Whilst this has not met any of the success criteria yet, the data structures / classes that have been created here will allow me to meet various success criteria later, for example later properly implementing the “isValidMove” methods made here will allow me to meet success criteria #11.

Creating initial GUI

Development



Then, I started to create an outline for the GUI. Firstly, I created a JFrame, which is the main window, so that all of the other components can be held in here. Then, I added a JPanel, and set it to a GridLayout, so that it could contain the 64 JButtons used to represent each square in an 8x8 grid. I did this as it makes it much easier / quicker to add all of the buttons in the correct place, and set them all to the same size, instead of changing the size of each of them & moving them to the correct place individually. I then added a line border to each of the buttons, with alternating colours, in order to represent the chequered board, so that the grid resembles a chessboard for the user. Then, I added 36 JLabels around the edge of the board, showing the name of each row / column, making it easier for the user to select the correct piece to move. Then, I added a JMenuBar to the JFrame, and 4 JMenu's to this menu bar, in order to show the options for the game to the user in an uncluttered yet easy to use way, so that I can add the functionality for these later. This uses the colours white, black, grey and blue, the 4 most liked colours by my end users, so should appeal to them.



I also changed the name of each of the JButtons so that it is clear which button is which, so that their identifiers are more meaningful, making the code easier to read, and reducing the likelihood of accidentally using the wrong button for something in the program, and making it easier to understand when I come back to it later.

```

public class Chess extends javax.swing.JFrame {

    private ChessBoard board;
    private int startPos;
    private int endPos;
    private Color defaultColour;
    private JButton[][] buttons;

    public Chess() {
        initComponents();

        this.board = new ChessBoard();

        this.startPos = -1;
        this.endPos = -1;

        this.defaultColour = this.JPanel2.getBackground();

        this.buttons = new JButton[][] {
            {this.JButtonA8, this.JButtonB8, this.JButtonC8, this.JButtonD8, this.JButtonE8, this.JButtonF8, this.JButtonG8, this.JButtonH8},
            {this.JButtonA7, this.JButtonB7, this.JButtonC7, this.JButtonD7, this.JButtonE7, this.JButtonF7, this.JButtonG7, this.JButtonH7},
            {this.JButtonA6, this.JButtonB6, this.JButtonC6, this.JButtonD6, this.JButtonE6, this.JButtonF6, this.JButtonG6, this.JButtonH6},
            {this.JButtonA5, this.JButtonB5, this.JButtonC5, this.JButtonD5, this.JButtonE5, this.JButtonF5, this.JButtonG5, this.JButtonH5},
            {this.JButtonA4, this.JButtonB4, this.JButtonC4, this.JButtonD4, this.JButtonE4, this.JButtonF4, this.JButtonG4, this.JButtonH4},
            {this.JButtonA3, this.JButtonB3, this.JButtonC3, this.JButtonD3, this.JButtonE3, this.JButtonF3, this.JButtonG3, this.JButtonH3},
            {this.JButtonA2, this.JButtonB2, this.JButtonC2, this.JButtonD2, this.JButtonE2, this.JButtonF2, this.JButtonG2, this.JButtonH2},
            {this.JButtonA1, this.JButtonB1, this.JButtonC1, this.JButtonD1, this.JButtonE1, this.JButtonF1, this.JButtonG1, this.JButtonH1}
        };
    }
}

```

I then added a variable of type “ChessBoard” to the JFrame, so that the board and the GUI can be connected to each other, allowing the user to make moves and update the data structure. I also added a 2D array of buttons, so that the buttons can be accessed much more easily, and correspond to the 2D array of Pieces stored in the ChessBoard (i.e. the piece at [0][0] in the chess board should be displayed on the button at position [0][0] in the buttons array). I also added variables to store the buttons the user selects, and the default background colour of each button, so that whether the user has selected a piece / which piece they have selected can be stored, which is needed to allow moves to be made later, and so that the button background colours can be reset if they are changed at any point.

Then, I added to the constructor of the JFrame, so that the actions which need to be performed every time the program is first started can be added. I initialised a new “Chessboard” object, and stored it in the variable “board”. I then set “startPos” and “endPos” to -1, indicating that a position has not yet been selected by the user, as when the program is first run, the user will not have selected anything. I also set defaultColour equal to default background colour that each button should be. Then, I initialised the 2D array “buttons”, so that each button can correspond to a particular location in the array of pieces in “board”, making it easier to update the GUI and show the board to the user.

```

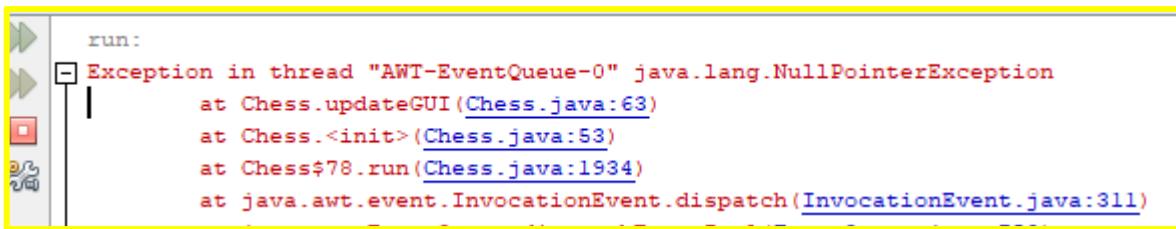
public void updateGUI() {
    Piece[][] cb = this.board.getBoard();

    for(int i = 0; i < cb.length; i++) {
        for(int j = 0; j < cb[i].length; j++) {
            this.buttons[i][j].setText(cb[i][j].toString());
        }
    }
}

```

I then created a method to update the text of the buttons, to reflect the contents of the chessboard, so that the pieces will be shown to the user. I iterated through the 2D array of Pieces using two for loops, for each Piece, setting the corresponding button’s text to a string representing the piece, and added this method to the constructor of the JFrame, after setting the values of the other variables, which should display the pieces to the user, as text on the buttons, when the program is first run.

Testing



```
run:
Exception in thread "AWT-EventQueue-0" java.lang.NullPointerException
 at Chess.updateGUI(Chess.java:63)
 at Chess.<init>(Chess.java:53)
 at Chess$78.run(Chess.java:1934)
 at java.awt.event.InvocationEvent.dispatch(InvocationEvent.java:311)
```

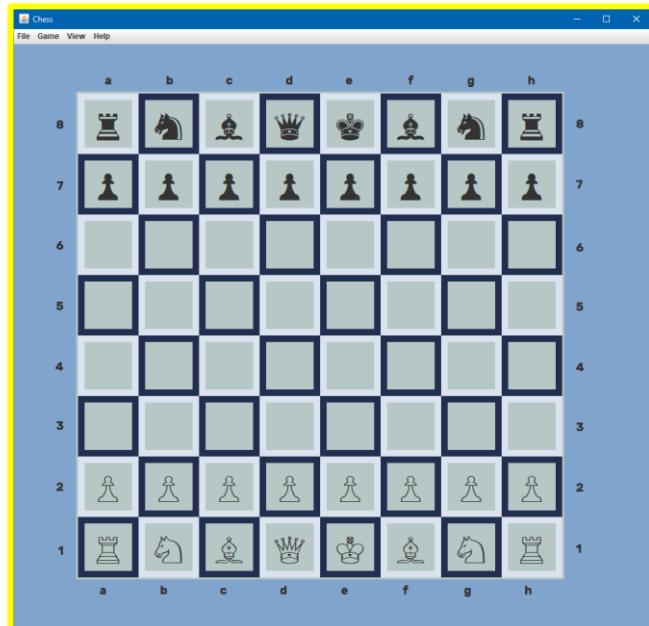
However, when I then ran this, a null pointer exception occurred. Looking at the code, I realised that this was because I had forgotten to check whether the piece was null or not when getting its string. If it is null, the text of the button should just be set to blank, and the “`toString`” method should not be called, as otherwise this null pointer exception will occur.

Remedial

```
public void updateGUI() {
    Piece[][] cb = this.board.getBoard();

    for(int i = 0; i < cb.length; i++) {
        for(int j = 0; j < cb[i].length; j++) {
            if(cb[i][j] != null) {
                this.buttons[i][j].setText(cb[i][j].toString());
            } else {
                this.buttons[i][j].setText("");
            }
        }
    }
}
```

I fixed this error by adding an if statement that checks whether the Piece at that location is null or not. If it is not null, I set the text of the button as before. Otherwise, the text of the button is set to a blank string, so that the button appears empty, showing there is no piece there. Now the method works as intended. I called it in the constructor, and as expected, the window appears correctly, no null pointer exception occurs, and the buttons now show the pieces to the user in their correct positions.



Review

Here, a `JFrame` has been created and buttons added to it to represent the board. Also, this has been linked to the previously created data structures, showing the pieces stored in the array in the `ChessBoard` to the user.

Now that the GUI has been created, and has been connected to the data structures, I can begin to implement the ability for the user to make moves / update the data structures via the GUI, allowing them to actually use the program.

Link to success criteria: #1

Allowing user to select pieces

Development

```
private void jButtonC8ActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
}

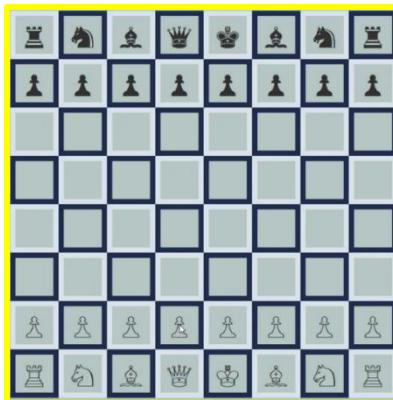
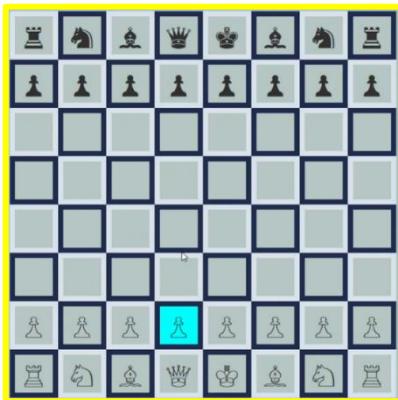
private void jButtonD8ActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
}

private void jButtonE8ActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
}
```

```
public void buttonPressed(int pos) {
    if(this.startPos == -1) {
        this.buttons[pos / 8][pos % 8].setBackground(Color.CYAN);
        this.startPos = pos;
    } else if (this.endPos == -1) {
        this.endPos = pos;
        this.buttons[this.startPos / 8][this.startPos % 8].setBackground(this.defaultColour);
        this.buttons[this.endPos / 8][this.endPos % 8].setBackground(this.defaultColour);
        this.startPos = -1;
        this.endPos = -1;
    }
}
```

```
private void jButtonA8ActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
    buttonPressed(0);
}
```

Testing



Review

Here, the ability to store the location the user selects and for the piece the user selects to be highlighted in another colour has been implemented. So, now the information that the makeMove function will require is being stored when the user selects pieces.

Now that the user is able to select the piece they would like to move and where they would like to move it to, I can implement the ability for this move to actually be made.

Link to success criteria: Whilst this doesn't meet #6 yet, this will allow me to meet this criterion later

Firstly, I added action listeners to each of the JButtons, so that it can be decided what to do when the user presses a particular button in order to enter a move. As all of these buttons will perform the same functionality, I decided to create a method which each of these buttons can call, providing their position as a parameter (so that the specific button which has been pressed is known).

Here I started to write this method. Pos / 8 gives the row, pos % 8 gives the column of the button. If the user has not yet selected a piece (i.e. startPos is -1), it sets the background colour of the button they select to blue, to indicate that this is the piece they have selected, and stores this position. If the user has already selected a piece, it stores the end position that they have selected (which will be needed later so that it is known where the piece should be moved to), resets the button colours, and then resets the start and end position, allowing the user to select a new pair of buttons (i.e. allowing another move to be made). However, I have not yet written the code to move a piece or check for valid moves, so it does not currently move pieces yet / perform its full functionality, so I will update it later.

Then, I added this method to the 'action performed' method of each button, with the correct position for each, so that all of the buttons can make use of this method, allowing the user to select any button to make a move, later, when this is implemented.

Now when a button is pressed, it is highlighted in blue, showing that this piece has been selected as the piece the user would like to move, and then when another piece is then pressed, the background colours of the buttons are reset to default (i.e. the user would have then made their move, so the previously selected piece should no longer be highlighted in blue), and so is working as expected.

Allowing user to make a move

Development

```
public void makeMove(String move) {
    int startPos = Integer.valueOf(move.substring(0, 2));
    int endPos = Integer.valueOf(move.substring(2, 4));

    if (this.board[startPos / 8][startPos % 8].isValidMove(move, this.board)) {
        if (this.board[endPos / 8][endPos % 8] == null) {
            move += "0";
        } else {
            move += this.board[endPos / 8][endPos % 8].getType();
        }

        this.moveHistory.add(move);

        this.board[endPos / 8][endPos % 8] = this.board[startPos / 8][startPos % 8];
        this.board[startPos / 8][startPos % 8] = null;
    }
}
```

Then, I added the “makeMove” method to the “ChessBoard” class. This takes a string representing the move as a parameter, and firstly extracts the start position and end position from this string, for use later in updating the pieces array. It then determines whether this move is valid, using the “isValidMove” method of the piece. If it is a valid move, then a number from 0-6 is added to the end of the string, depending on the type of piece that is captured; 0 = none, 1 = pawn, 2 = rook, 3 = knight, 4 = bishop, 5 = queen, 6 = king. This is to help later in undoing a move, where the piece that was captured should be added back onto the board in its previous position from before it was captured. The move is then added to the move history, so that it can be undone if needed to later. Finally, the pieces themselves in the array move – the piece in startPos moves to endPos, and then the piece at startPos is set to null, as there should now be no piece there.

```
public void buttonPressed(int pos) {
    if(this.startPos == -1 && !(this.buttons[pos / 8][pos % 8].getText().isEmpty())) {
        this.buttons[pos / 8][pos % 8].setBackground(Color.CYAN);

        this.startPos = pos;
    } else if (this.endPos == -1 && this.startPos != -1) {
        this.endPos = pos;

        this.buttons[this.startPos / 8][this.startPos % 8].setBackground(this.defaultColour);
        this.buttons[this.endPos / 8][this.endPos % 8].setBackground(this.defaultColour);

        this.board.makeMove(this.generateMoveString());
        this.updateGUI();

        this.startPos = -1;
        this.endPos = -1;
    }
}
```

```
private String generateMoveString() {
    String move = "";

    if(this.startPos < 10) {
        move += "0";
    }
    move += this.startPos;

    if(this.endPos < 10) {
        move += "0";
    }
    move += this.endPos;

    return move;
}
```

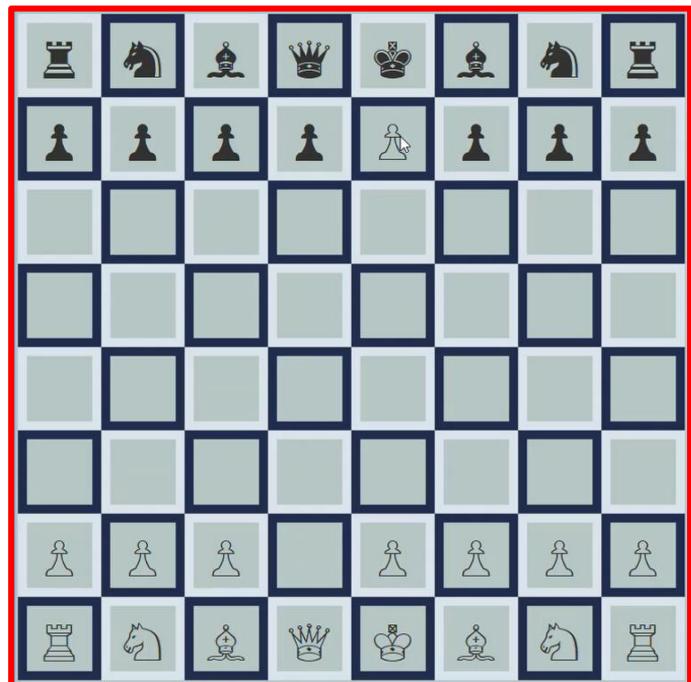
After this, I then realised I needed to check whether the first piece / start pos that the user selects is not empty / is not null, in order to avoid a null pointer exception, and so that the user cannot move an empty piece (as an empty square cannot be moved). I decided to do this in the “buttonPressed” method, as this will prevent the user from selecting an empty square to move in the first place, rather than checking for this after they have selected it. This is done by checking whether the text in the button the user selects is not empty (i.e. whether there is a piece there). Then, I called the “updateGUI” method, to show the new state of the board to the user.

I also altered this method, so that when the user selects pieces, they will actually move using the new “makeMove” method (if the move is determined to be valid) after the user selects an end position. This uses the “generateMoveString” method, which creates a string in the format mentioned before, only without the character at the end representing the captured piece, which is added later in the “makeMove” function, before the move is added to the move history, which allows the start and end positions that the user have selected to be passed to the make move procedure, so that this procedure knows which pieces need to be moved and to where.

```
this.moveHistory.add(move);  
this.currentTurn++;
```

I realised I forgot to increment the “currentTurn” after making the move, so I added this after adding to move to the move history in the “makeMove” method. This will make sure that both users are able to move when turns are implemented later, as otherwise only one player would be able to move, as current turn would always be even, so it would always be whites turn to move.

Testing



When I tested this, this method worked in that the pieces move according to where the user selects, however, since I have not yet properly implemented the “isValidMove” methods for the different types of pieces, currently the pieces can move anywhere on the board. This should fix itself when I implement the methods to validate moves properly later.

Review

Here, the ability for the user to input a move using the buttons on the board has been implemented. The user can currently move their pieces, however no checks are made as to whether these moves should be allowed to be made or not.

Now that the user can move pieces on the board / it has been confirmed that the makeMove procedure is working correctly so far, I can begin to validate these moves by implementing turns (validating that the correct player is moving), by checking whether moves are pseudo-legal (validating that the move follows the rules for the appropriate type of piece), and then by checking for check / check mate (validating that the move is legal).

Link to success criteria: #6

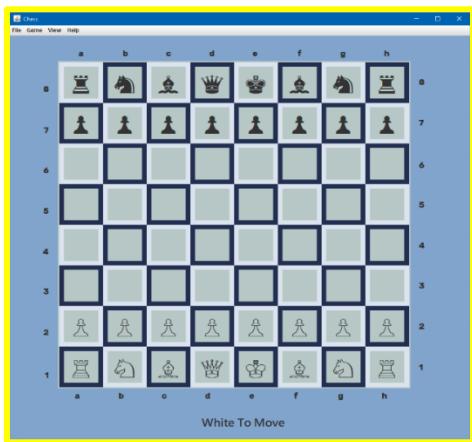
Turn JLabel

Development

Next, I decided to add the JLabel to inform the user of whose turn it currently is.

```
public int getCurrentTurn() {  
    ...  
    return this.currentTurn;  
}
```

Firstly, I added a method to the “ChessBoard” class that returns the current turn, so that this can be accessed in the JFrame, which is needed so that it can be determined what text to show to the user so that they can be told whose turn it is to move.

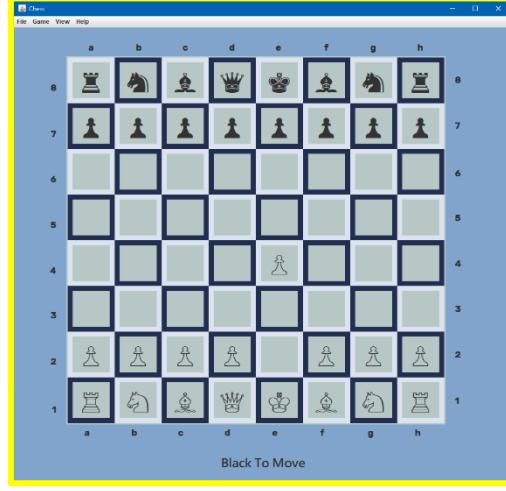
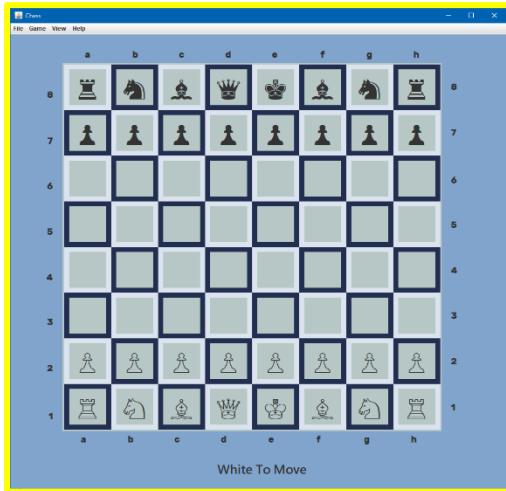


```
if(this.board.getCurrentTurn() % 2 == 0) {  
    this.turnLabel.setText("White To Move");  
} else {  
    this.turnLabel.setText("Black To Move");  
}
```

Then, I added a JLabel to the bottom of the window, where it can easily be seen by the user during the game without getting in the way, which will be used to tell the user whose turn it is to make a move.

Then, I added this if statement to the end of the “updateGUI” method, updating the JLabel with whose turn it is (if current turn is even, then it is white to move, otherwise it is black to move). This should update after each move is made, changing the text at the bottom to say the next player after each move.

Testing



As you can see here, when I tested this, the JLabel updates with whose turn it is correctly, starting with “white to move” and changing to “black to move” after white has made a move, and so seems to be working.

Review

Here, the ability for the program to keep track of whose turn it is and to tell this to the user has been implemented.

Now, the JLabel at the bottom of the screen updates correctly with whose turn it is, however this is not actually enforced yet – the wrong player can still make a move if they wanted to. This will actually be enforced later.

Link to success criteria: starting to meet #7

Showing possible moves

Development

Next, I decided to add the functionality to show the user the possible moves a piece is allowed to make when they select it. This will not only make the program easier to use for those who are unsure of how each piece should move, but will also make it easier to test whether my “isValidMove” functions for the different types of piece are working correctly when I implement them after this, as I will be able to immediately see all the moves the piece thinks are valid, saving time

```
public boolean[][] getValidEndPositions(int pos) {
    boolean[][] validPositions = new boolean[8][8];

    for (int i = 0; i < 64; i++) {
        String move = "";
        if (pos < 10) {
            move += "0";
        }
        move += pos;

        if (i < 10) {
            move += "0";
        }
        move += i;

        if (i != pos && this.board[pos / 8][pos % 8].isValidMove(move, this.board)) {
            validPositions[i/8][i%8] = true;
        }
    }

    return validPositions;
}
```

To begin with, I added the “getValidEndPositions” method to the ChessBoard class. This returns a 2D array of booleans (similar to the board and buttons array, in that a certain position in all 3 arrays will correspond to the same logical piece), where true represents that the selected piece can move to that location, and false represents that the piece cannot move to that location. It iterates through the 2D array of pieces, and checks whether a move to each square on the board would be valid or not. If it is, it sets that location to true in “validPositions”. Otherwise it does nothing, as the default value in this array of booleans is false, so it does not need to be changed. Again, this method will not work properly until the “isValidMove” methods have been fully implemented – currently it will have every position on the board as a valid place to move to, as all moves are considered valid.

```
private void showPossibleEndPositions(int pos) {
    boolean[][] valid = this.board.getValidEndPositions(pos);
    boolean canMakeMove = false;

    for (int i = 0; i < this.buttons.length; i++) {
        for (int j = 0; j < this.buttons[i].length; j++) {
            if (valid[i][j]) {
                this.buttons[i][j].setBackground(Color.YELLOW);
                canMakeMove = true;
            }
        }
    }

    if (!canMakeMove) {
        this.buttons[pos / 8][pos % 8].setBackground(Color.RED);
    }
}
```

Then, I wrote the method that uses “getValidEndPositions” to show the user the possible places they could move their piece to in a different colour on the board. It calls the “getValidEndPositions” method, and then iterates through the 2D array of buttons, setting the background colour to yellow if the value at that location in the array that was created in the “getValidEndPositions” method, “valid”, is equal to true. If there are no valid moves (i.e. all elements in the array were false), it sets the colour of the button to red. This is to improve the usability of the program by letting the user know that piece cannot move, and that they should select a different one.

```

if (this.startPos == -1 && !(this.buttons[pos / 8][pos % 8].getText().isEmpty())) {
    this.buttons[pos / 8][pos % 8].setBackground(Color.CYAN);
    this.showPossibleEndPositions(pos);
    this.startPos = pos;
}

```

```

private void resetButtonColours() {
    for (int i = 0; i < this.buttons.length; i++) {
        for (int j = 0; j < this.buttons[i].length; j++) {
            this.buttons[i][j].setBackground(defaultColour);
        }
    }
}

```

```

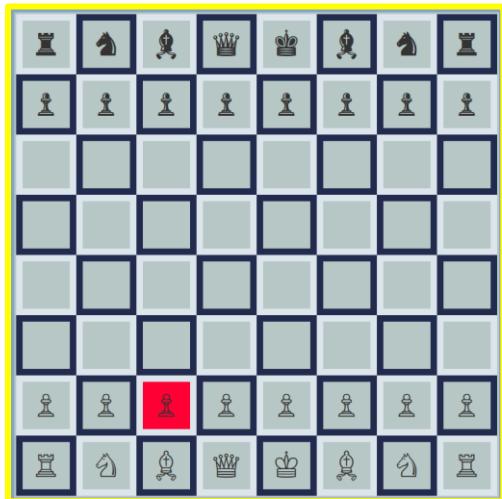
} else if (this.endPos == -1 && this.startPos != -1) {
    this.endPos = pos;

    this.resetButtonColours();
}

```

Testing

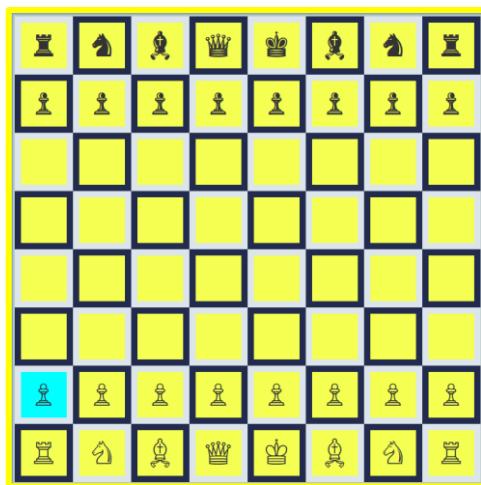
Now, when the user selects a piece, all the squares (except the square of the selected piece) are highlighted in yellow, which is expected as pieces are currently able to move anywhere on the board, and so is working as expected.



I then altered the “buttonPressed” method, so that it calls this method when the user selects the piece they would like to move, so that when the user selects a piece, they are shown the possible moves it can make.

Then, I created a method to reset all button colours back to “defaultColour”, by iterating through the array of buttons, and setting the background of each to “defaultColour”, so that buttons are not permanently changed to blue / yellow, as this could confuse the user.

Finally, I updated the “buttonPressed” method, so that it sets all buttons to the default colour after an end location has been selected by the user so that after the move has been made, the user is no longer shown the moves that the piece could have made.



I then temporarily set the “isValidMove” method of the pawn to return false, to test whether it would be highlighted in red when it was selected, as it would now have no possible moves, which is was, so is working correctly.

Review

Here, the ability for the program to show the user it has determined are valid has been implemented, however the methods for actually working out whether a move is legal or not have not yet been properly implemented.

Next, I will implement the various “isValidMove” functions. The ability to highlight moves in different colours will make these “isValidMove” functions much easier to test.

Link to success criteria: starting to meet #10

Knight move validation

Development

```
@Override
public boolean isValidMove(String move, Piece[][] board) {
    int startPos = Integer.valueOf(move.substring(0, 2));
    int endPos = Integer.valueOf(move.substring(2, 4));

    boolean capturingPieceOfDifferentColour = (board[endPos / 8][endPos % 8] != null)
        && (board[startPos / 8][startPos % 8].getColour() != board[endPos / 8][endPos % 8].getColour());
    boolean movingToEmptySquare = board[endPos / 8][endPos % 8] == null;

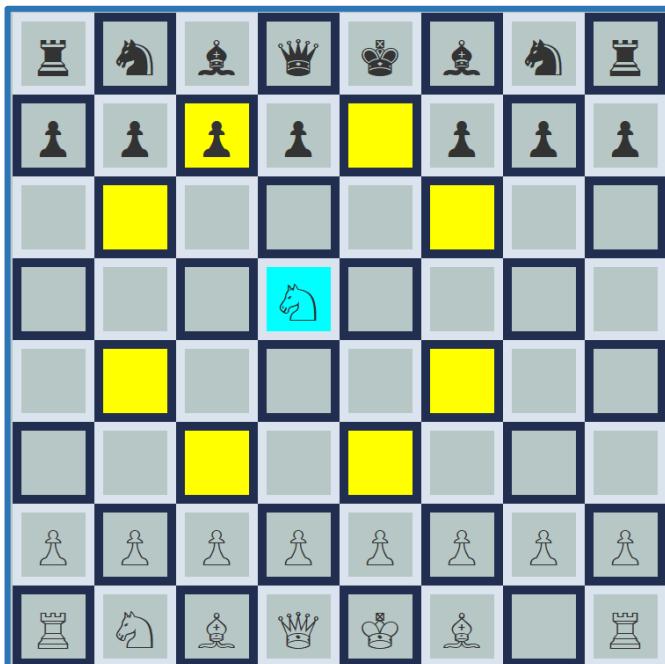
    boolean twoColumnsAway = Math.abs(endPos % 8 - startPos % 8) == 2;
    boolean oneRowAway = Math.abs(endPos / 8 - startPos / 8) == 1;

    boolean twoRowsAway = Math.abs(endPos / 8 - startPos / 8) == 2;
    boolean oneColumnAway = Math.abs(endPos % 8 - startPos % 8) == 1;

    return (capturingPieceOfDifferentColour || movingToEmptySquare)
        && (twoRowsAway && oneColumnAway || twoColumnsAway && oneRowAway);
}
```

Then, I started implementing the “isValidMove” methods, starting with the “Knight” class. Firstly, I extracted the start position and end position from the move string, and stored these in two variables. Then, I declared a series of boolean variables, in order to help determine whether a move is valid. Firstly, I check that if a piece is being captured, then it is not the same colour as the piece that is attacking. Secondly, I check whether the piece is moving to an empty square (i.e. not attacking). I check these because a piece is allowed to do one of two things: either capture an opponent’s piece, or not attack any piece. After that, I check whether the end position is one or two rows or columns away from the start. I check this because a knight is allowed to move two squares in one direction, and then one in another perpendicular to that. Finally, I return a boolean representing whether the move is legal (the piece is attacking a piece of the opposite colour or is not attacking, and has moved twice horizontally and once vertically or twice vertically and once horizontally). However, this method does not include anything in regards to check or checkmate – this only determines whether a move is pseudo-legal. Check will be determined later, in the function that makes the move, as it is not up to the individual pieces to know whether the king is under attack, but for the chessboard to.

Testing



Here is an example of the result of the “showPossibleEndPositions” method for a knight, using this implementation of “isValidMove”, showing that “isValidMove” and “showPossibleEndPositions” is working as expected.

King move validation

Development

```
@Override
public boolean isValidMove(String move, Piece[][] board) {
    int startPos = Integer.valueOf(move.substring(0, 2));
    int endPos = Integer.valueOf(move.substring(2, 4));

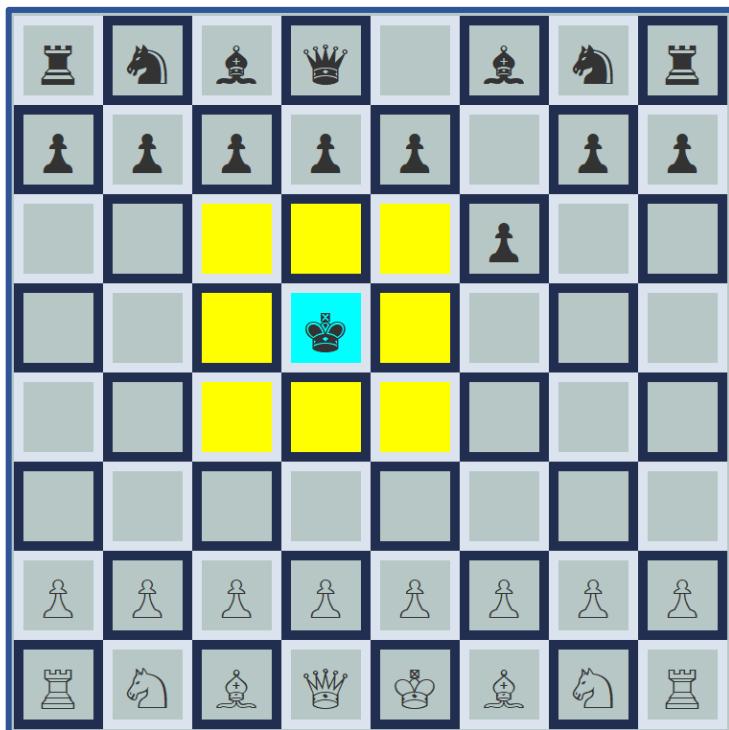
    boolean capturingPieceOfDifferentColour = (board[endPos / 8][endPos % 8] != null)
        && (board[startPos / 8][startPos % 8].getColour() != board[endPos / 8][endPos % 8].getColour());
    boolean movingToEmptySquare = board[endPos / 8][endPos % 8] == null;

    boolean notMoreThanOneRowAway = Math.abs(endPos / 8 - startPos / 8) <= 1;
    boolean notMoreThanOneColumnAway = Math.abs(endPos % 8 - startPos % 8) <= 1;

    return (capturingPieceOfDifferentColour || movingToEmptySquare) && (notMoreThanOneColumnAway && notMoreThanOneRowAway);
}
```

Then I started the “isValidMove” method for the “King” class. Again, I extracted the start position and end position from the move string, and stored these in two variables, and checked that if a piece is being captured, then it is not the same colour as the piece that is attacking & whether the piece is moving to an empty square / is not attacking. Then, I check that the end position is not more than one row away (in either direction), and that it is not more than one column away (in either direction). I check this because the king can only move one square away at a time. Finally, I return a boolean, which is true if a piece of another colour is being captured or it is moving to an empty space, and if the end position is not more than one column away, and if it is not more than one row away.

Testing



Here is an example of the result of the “showPossibleEndPositions” method for a king, showing that this is working correctly.

Rook move validation

Development

```
@Override
public boolean isValidMove(String move, Piece[][] board) {
    int startPos = Integer.valueOf(move.substring(0, 2));
    int endPos = Integer.valueOf(move.substring(2, 4));

    boolean capturingPieceOfDifferentColour = (board[endPos / 8][endPos % 8] != null)
        && (board[startPos / 8][startPos % 8].getColour() != board[endPos / 8][endPos % 8].getColour());
    boolean movingToEmptySquare = board[endPos / 8][endPos % 8] == null;

    boolean inSameRow = endPos / 8 == startPos / 8;
    boolean inSameColumn = endPos % 8 == startPos % 8;

    if ((capturingPieceOfDifferentColour || movingToEmptySquare) && (inSameRow || inSameColumn)) {
        if (inSameRow && endPos % 8 < startPos % 8) { // Moving left
            int col = startPos % 8;
            while (col - 1 >= 0 && board[startPos / 8][col-1] == null) {
                col--;
            }
            return endPos % 8 >= col || capturingPieceOfDifferentColour && endPos % 8 >= col-1;
        } else if (inSameRow && endPos % 8 > startPos % 8) { // Moving right
            int col = startPos % 8;
            while (col + 1 < 8 && board[startPos / 8][col+1] == null) {
                col++;
            }
            return endPos % 8 <= col || capturingPieceOfDifferentColour && endPos % 8 <= col+1;
        } else if (inSameColumn && endPos / 8 < startPos / 8) { // Moving up
            int row = startPos / 8;
            while (row - 1 >= 0 && board[row-1][startPos % 8] == null) {
                row--;
            }
            return endPos / 8 >= row || capturingPieceOfDifferentColour && endPos / 8 >= row-1;
        } else if (inSameColumn && endPos / 8 > startPos / 8) { // Moving down
            int row = startPos / 8;
            while (row + 1 < 8 && board[row+1][startPos % 8] == null) {
                row++;
            }
            return endPos / 8 <= row || capturingPieceOfDifferentColour && endPos / 8 <= row+1;
        }
    }

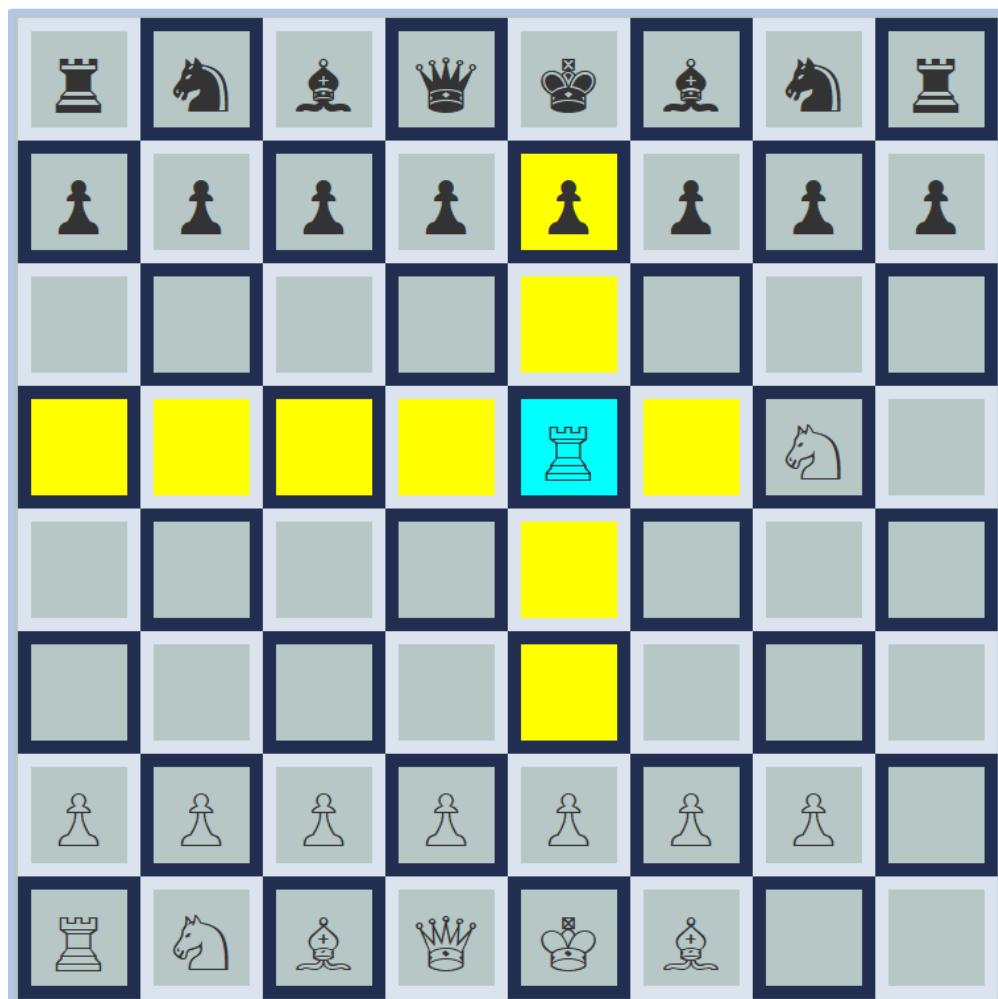
    return false;
}
```

Then I started the “isValidMove” method for the “Rook” class. Again, I extracted the start position and end position from the move string, and stored these in two variables, and checked that if a piece is being captured, then it is not the same colour as the piece that is attacking (as you cannot capture one of your own pieces) & whether the piece is moving to an empty square / is not attacking.

I then created two variables (as this makes the code easier to read rather than doing all of these comparisons in the if statements) to check whether the end position was in the same row or in the same column as the start position (as rooks can only move horizontally or vertically). If the piece is capturing a piece of another colour, or moving to an empty square, and the end position is in the same row or the same column as the start positions, I continue to check whether it is valid or not, and otherwise I return false, as the move is not legal.

I then have a series of if statements, to determine the direction in which the piece is moving (e.g. if the end position is within the same row, and the end column is less than the start column, then the piece is moving to the left). For each direction, I determine the furthest the piece can move in that direction before either reaching another piece, or the end of the board. I check this, as rooks cannot jump over other pieces, so they can only move in a particular direction until they reach another piece, and not further than that. I then determine whether the end position occurs before another piece is reached in that direction (as it can only move up to that piece and not past it), or if the move is a capture, that the end position occurs either before or is equal to the first piece reached in that direction (as it can move onto the other piece in order to capture it). If this is the case, the move is legal, so true is returned. Otherwise, false is returned.

Testing



Here is an example of the result of the “showPossibleEndPositions” method for a rook, showing that this is working correctly.

Bishop move validation

Development

```
public boolean isValidMove(String move, Piece[][] board) {
    int startPos = Integer.valueOf(move.substring(0, 2));
    int endPos = Integer.valueOf(move.substring(2, 4));

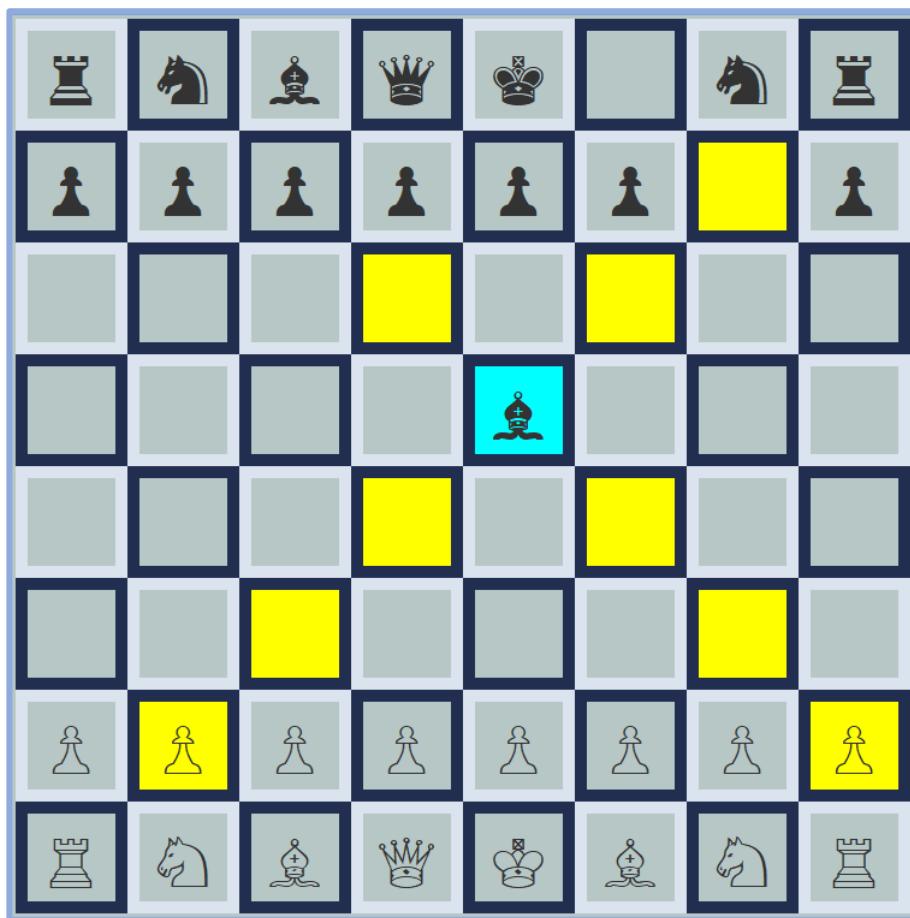
    boolean capturingPieceOfDifferentColour = (board[endPos / 8][endPos % 8] != null)
        && (board[startPos / 8][startPos % 8].getColour() != board[endPos / 8][endPos % 8].getColour());
    boolean movingToEmptySquare = board[endPos / 8][endPos % 8] == null;

    int columnChange = Math.abs(endPos % 8 - startPos % 8);
    int rowChange = Math.abs(endPos / 8 - startPos / 8);
    boolean isDiagonal = columnChange == rowChange;

    if((capturingPieceOfDifferentColour || movingToEmptySquare) && isDiagonal) {
        if(endPos % 8 < startPos % 8 && endPos / 8 < startPos / 8) { // Moving north west
            int col = startPos % 8;
            int row = startPos / 8;
            while(col - 1 >= 0 && row - 1 >= 0 && board[row-1][col-1] == null) {
                col--;
                row--;
            }
            return endPos % 8 >= col && endPos / 8 >= row || capturingPieceOfDifferentColour && endPos % 8 >= col-1 && endPos / 8 >= row-1;
        } else if (endPos % 8 > startPos % 8 && endPos / 8 < startPos / 8) { // Moving north east
            int col = startPos % 8;
            int row = startPos / 8;
            while(col + 1 < 8 && row - 1 >= 0 && board[row-1][col+1] == null) {
                col++;
                row--;
            }
            return endPos % 8 <= col && endPos / 8 >= row || capturingPieceOfDifferentColour && endPos % 8 <= col+1 && endPos / 8 >= row-1;
        } else if (endPos % 8 > startPos % 8 && endPos / 8 > startPos / 8) { // Moving south east
            int col = startPos % 8;
            int row = startPos / 8;
            while(col + 1 < 8 && row + 1 < 8 && board[row+1][col+1] == null) {
                col++;
                row++;
            }
            return endPos % 8 <= col && endPos / 8 <= row || capturingPieceOfDifferentColour && endPos % 8 <= col+1 && endPos / 8 <= row+1;
        } else if (endPos % 8 < startPos % 8 && endPos / 8 > startPos / 8) { // Moving south west
            int col = startPos % 8;
            int row = startPos / 8;
            while(col - 1 >= 0 && row + 1 < 8 && board[row+1][col-1] == null) {
                col--;
                row++;
            }
            return endPos % 8 >= col && endPos / 8 <= row || capturingPieceOfDifferentColour && endPos % 8 >= col-1 && endPos / 8 <= row+1;
        }
    }
    return false;
}
```

Next, I started the “isValidMove” method for the “Bishop” class. I did this in a very similar way to the “Rook” class, only checking directions diagonally rather than horizontally / vertically, as bishops move in a very similar way to rooks, the only difference being that they move diagonally rather than horizontally and vertically. I determined whether the move was diagonal (since a bishop can only move diagonally), and then determined the furthest you could go in a particular direction until you reached another piece (since a bishop cannot jump over any pieces), and then checked that the end position occurred before that piece was reached (or that the end position is at the position of the first piece reached, if the move is a capture).

Testing



Here is an example of the result of the “showPossibleEndPositions” method for a bishop, showing that this is working correctly.

Queen move validation

Development

```
@Override
public boolean isValidMove(String move, Piece[][] board) {
    int startPos = Integer.valueOf(move.substring(0, 2));
    int endPos = Integer.valueOf(move.substring(2, 4));

    boolean capturingPieceOfDifferentColour = (board[endPos / 8][endPos % 8] != null)
        && (board[startPos / 8][startPos % 8].getColour() != board[endPos / 8][endPos % 8].getColour());
    boolean movingToEmptySquare = board[endPos / 8][endPos % 8] == null;

    boolean inSameRow = endPos / 8 == startPos / 8;
    boolean inSameColumn = endPos % 8 == startPos % 8;

    int columnChange = Math.abs(endPos % 8 - startPos % 8);
    int rowChange = Math.abs(endPos / 8 - startPos / 8);

    boolean isDiagonal = columnChange == rowChange;

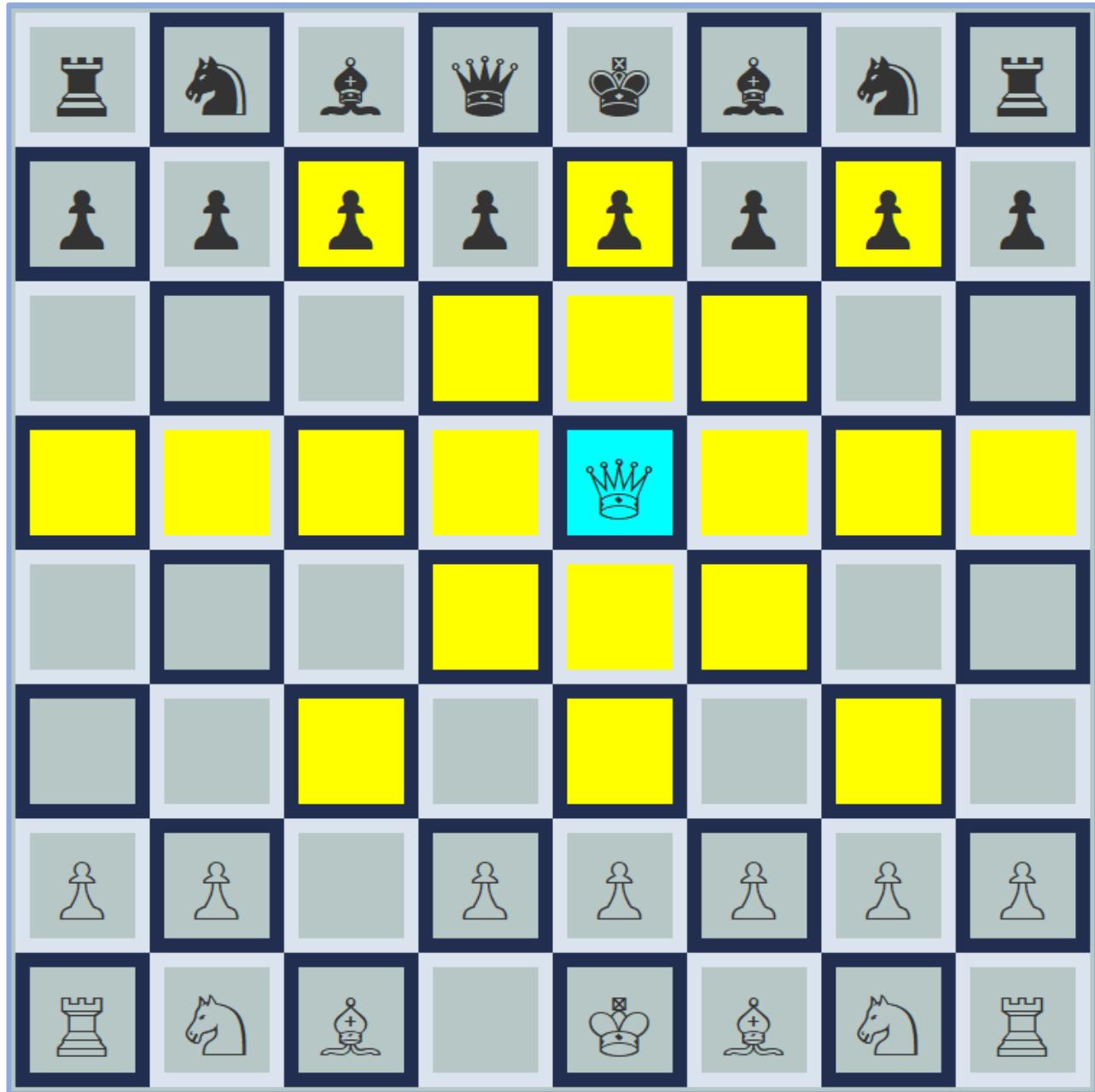
    if((capturingPieceOfDifferentColour || movingToEmptySquare) && (inSameRow || inSameColumn)) {
        if(inSameRow && endPos % 8 < startPos % 8) { // Moving left
            int col = startPos % 8;
            while(col - 1 >= 0 && board[startPos / 8][col-1] == null) {
                col--;
            }
            return endPos % 8 >= col || capturingPieceOfDifferentColour && endPos % 8 >= col-1;
        } else if (inSameRow && endPos % 8 > startPos % 8) { // Moving right
            int col = startPos % 8;
            while(col + 1 < 8 && board[startPos / 8][col+1] == null) {
                col++;
            }
            return endPos % 8 <= col || capturingPieceOfDifferentColour && endPos % 8 <= col+1;
        } else if (inSameColumn && endPos / 8 < startPos / 8) { // Moving up
            int row = startPos / 8;
            while(row - 1 >= 0 && board[row-1][startPos % 8] == null) {
                row--;
            }
            return endPos / 8 >= row || capturingPieceOfDifferentColour && endPos / 8 >= row-1;
        } else if (inSameColumn && endPos / 8 > startPos / 8) { // Moving down
            int row = startPos / 8;
            while(row + 1 < 8 && board[row+1][startPos % 8] == null) {
                row++;
            }
            return endPos / 8 <= row || capturingPieceOfDifferentColour && endPos / 8 <= row+1;
        }
    }
}
```

```
} else if((capturingPieceOfDifferentColour || movingToEmptySquare) && isDiagonal) {
    if(endPos % 8 < startPos % 8 && endPos / 8 < startPos / 8) { // Moving north west
        int col = startPos % 8;
        int row = startPos / 8;
        while(col - 1 >= 0 && row - 1 >= 0 && board[row-1][col-1] == null) {
            col--;
            row--;
        }
        return endPos % 8 >= col && endPos / 8 >= row || capturingPieceOfDifferentColour && endPos % 8 >= col-1 && endPos / 8 >= row-1;
    } else if (endPos % 8 > startPos % 8 && endPos / 8 < startPos / 8) { // Moving north east
        int col = startPos % 8;
        int row = startPos / 8;
        while(col + 1 < 8 && row - 1 >= 0 && board[row-1][col+1] == null) {
            col++;
            row--;
        }
        return endPos % 8 <= col && endPos / 8 >= row || capturingPieceOfDifferentColour && endPos % 8 <= col+1 && endPos / 8 >= row-1;
    } else if (endPos % 8 > startPos % 8 && endPos / 8 > startPos / 8) { // Moving south east
        int col = startPos % 8;
        int row = startPos / 8;
        while(col + 1 < 8 && row + 1 < 8 && board[row+1][col+1] == null) {
            col++;
            row++;
        }
        return endPos % 8 <= col && endPos / 8 <= row || capturingPieceOfDifferentColour && endPos % 8 <= col+1 && endPos / 8 <= row+1;
    } else if (endPos % 8 < startPos % 8 && endPos / 8 > startPos / 8) { // Moving south west
        int col = startPos % 8;
        int row = startPos / 8;
        while(col - 1 >= 0 && row + 1 < 8 && board[row+1][col-1] == null) {
            col--;
            row++;
        }
        return endPos % 8 >= col && endPos / 8 <= row || capturingPieceOfDifferentColour && endPos % 8 >= col-1 && endPos / 8 <= row+1;
    }
}

return false;
```

Then I started the “isValidMove” method for the “Queen” class. A queen can move in any number of spaces any direction (i.e. can move horizontally / vertically, like a rook, and diagonally, like a bishop), but cannot jump over any pieces – and so is essentially a rook and a bishop combined. So, I simply copied the “isValidMethod” for the rook and the bishop, and combined them together, as this was much quicker to do.

Testing



Here is an example of the result of
the “showPossibleEndPositions”
method for a queen, showing that
this is working correctly.

Pawn move validation

Development

```
@Override
public boolean isValidMove(String move, Piece[][] board) {
    int startPos = Integer.valueOf(move.substring(0, 2));
    int endPos = Integer.valueOf(move.substring(2, 4));

    boolean capturingPieceOfDifferentColour = (board[endPos / 8][endPos % 8] != null)
        && (board[startPos / 8][startPos % 8].getColour() != board[endPos / 8][endPos % 8].getColour());
    boolean movingToEmptySquare = board[endPos / 8][endPos % 8] == null;

    boolean isWhite = this.getColour() == 0;

    boolean hasNotMoved = (isWhite && startPos / 8 == 6) || (!isWhite && startPos / 8 == 1);

    boolean oneRowAway = Math.abs(endPos / 8 - startPos / 8) == 1;
    boolean twoRowsAway = Math.abs(endPos / 8 - startPos / 8) == 2;
    boolean inSameColumn = endPos % 8 == startPos % 8;
    boolean oneColumnAway = Math.abs(endPos % 8 - startPos % 8) == 1;

    boolean squareInFrontIsEmpty = (isWhite && (startPos / 8) - 1 >= 0 && board[(startPos / 8) - 1][startPos % 8] == null)
        || (!isWhite && (startPos / 8) + 1 < 8 && board[(startPos / 8) + 1][startPos % 8] == null);

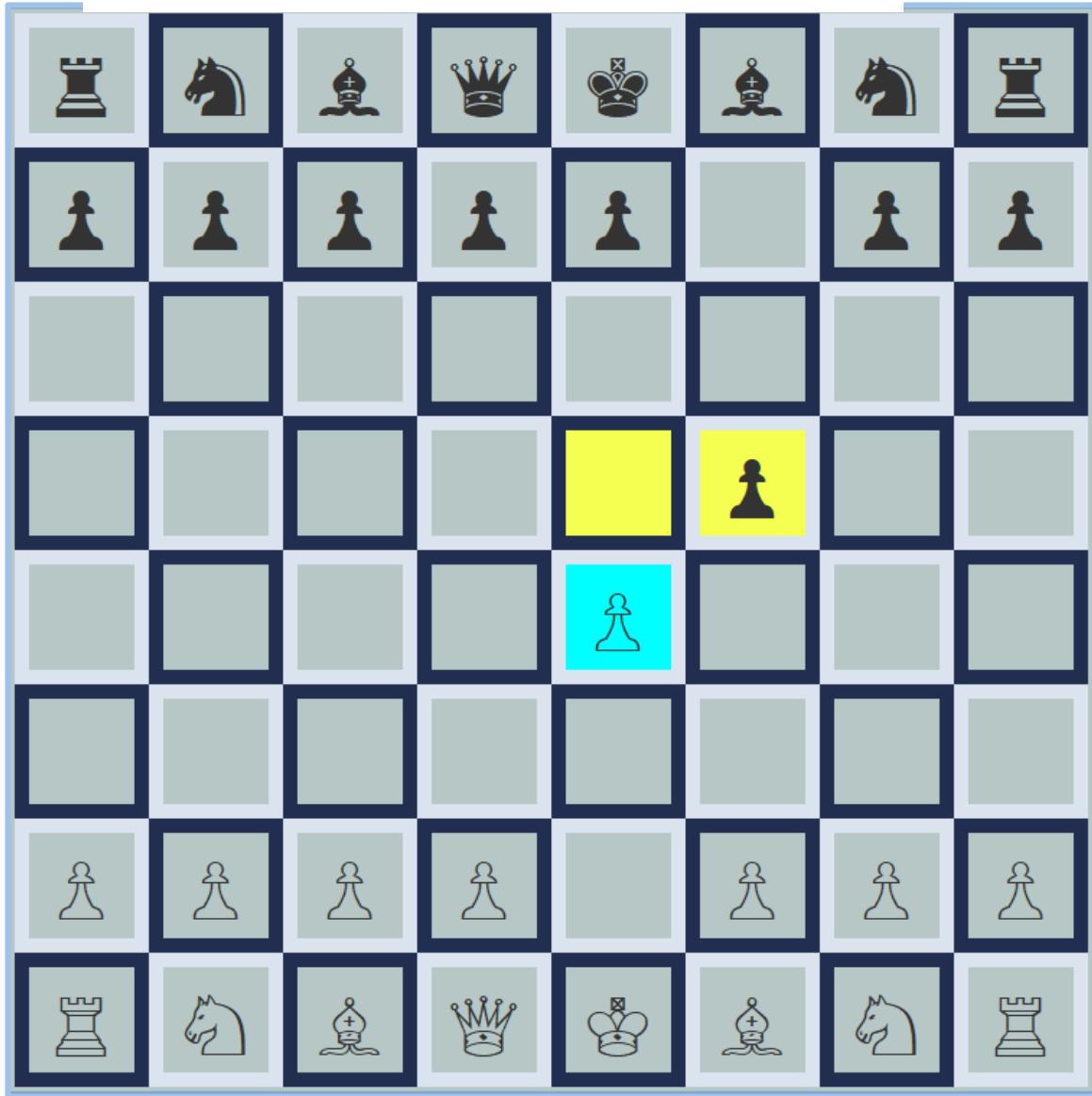
    boolean isMovingForwards = (isWhite && endPos / 8 < startPos / 8) || (!isWhite && endPos / 8 > startPos / 8);

    return (capturingPieceOfDifferentColour && oneColumnAway && oneRowAway || movingToEmptySquare && inSameColumn
        && (oneRowAway || hasNotMoved && twoRowsAway && squareInFrontIsEmpty)) && isMovingForwards;
}
```

Then I started the “isValidMove” method for the “Pawn” class. Again, I extracted the start position and end position from the move string, and stored these in two variables, and checked that if a piece is being captured, then it is not the same colour as the piece that is attacking & whether the piece is moving to an empty square / is not attacking. Then I have a boolean called “isWhite”, which is true if the piece is white. This is so that I can check the piece is moving in the correct direction, as black and white pawns move in opposite directions to each other. I then check whether the piece has moved or not, since pawns which have not moved have the option of moving two squares forwards, whereas pawns that have already moved do not. I then check whether it is moving one or two rows away (i.e. one or two squares forwards), and whether it is in the same column or is one column away. I then check whether the square directly in front of the pawn is empty, since pawns cannot jump over other pieces. I then check whether the piece is moving in the correct direction. This method returns true if the piece is moving forwards, the piece is capturing an opponent’s piece and is one column away and is one row away, or if the piece is moving to an empty square and is in the same column, and is moving one row away, or if it hasn’t moved and the square in front is empty, is moving two rows away.

Testing

Here is a screenshot showing that it is working as expected:



Review

Here, the “isValidMove” methods have been implemented for each of the different types of pieces.

Now that this is working, I will begin to enforce turns, only allowing the correct side to move.

Link to success criteria: #10, #11

Implementing turns

Development

```
public int getColourOfPieceAtPosition(int pos) {  
    return this.board[pos / 8][pos % 8].getColour();  
}
```

I then decided to implement the ability to only allow the correct side to make a move. I added the “getColourOfPieceAtPosition” method to the “ChessBoard” class. This will be used to determine the colour of the piece the user tries to select, which is needed so that it can be checked whether the piece the user has selected is of the correct colour.

```
public void buttonPressed(int pos) {  
    if (this.startPos == -1 && !(this.buttons[pos / 8][pos % 8].getText().isEmpty())  
        && this.board.getColourOfPieceAtPosition(pos) == this.board.getCurrentTurn() % 2) {
```

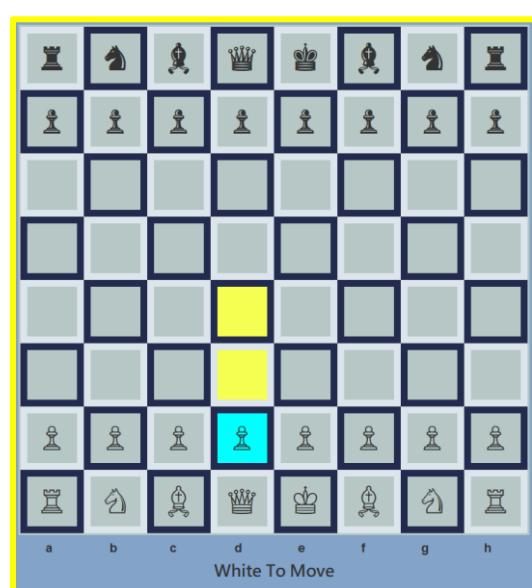
Then I used this in the “buttonPressed” method in the JFrame to only allow the user to select a piece if the colour of that piece matches whose turn it currently is. Now if the wrong side tries to make a move, the move will not be made.

Testing

When I ran this, if you try to select a piece of the wrong colour, nothing happens (i.e. it is not highlighted in blue, and the moves it could make is not shown), however when a piece of the correct colour is selected, it is still highlighted in blue and the possible moves it can make are shown, and so this is working correctly.



Trying to select a black piece (here, the piece highlighted in green) when it is white to move



Selecting a white piece when it is white to move

Review

Now, only the correct side is able to make a move, unlike before, where whilst the current turn was kept track of, any side could move.

Link to success criteria: #7

Showing opponents previous move

Development

I then decided to implement the ability to show the user the previous move of their opponent. To start with, I added the “getLastMove” method to the “ChessBoard” class, which returns the last move if there is one, and if not, returns a blank string. This is needed as it will allow the start and end position of the previous move to be passed to the JFrame, so that the correct buttons can be highlighted in another colour.

```
public String getLastMove() {
    if(this.moveHistory.getSize() > 0) {
        return this.moveHistory.peek();
    } else {
        return "";
    }
}
```

Next, I added this to the end of the “resetButtonColours” procedure, which gets the last move, and if there is one, sets the buttons at the start and end position of the opponent’s previous move to orange.

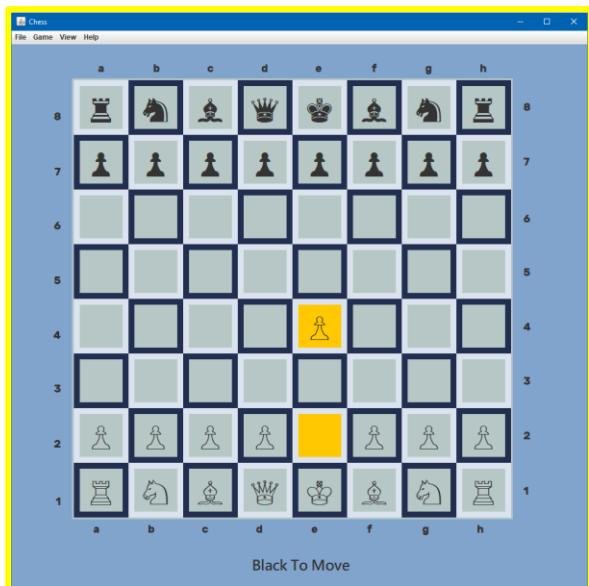
```
String lastMove = this.board.getLastMove();
if(!lastMove.isEmpty()) {
    int startPos = Integer.valueOf(lastMove.substring(0, 2));
    int endPos = Integer.valueOf(lastMove.substring(2, 4));
    this.buttons[startPos / 8][startPos % 8].setBackground(Color.ORANGE);
    this.buttons[endPos / 8][endPos % 8].setBackground(Color.ORANGE);
}
```

Testing

However, when I ran this, it did not work as expected – it was changing the colour of the buttons, but it was doing so too late.

Remedial

In order to fix this, I removed the “resetButtonColours” method from the “buttonPressed” method, and instead called it in the “updateGUI” method. After doing this, it worked as expected.



```
public void updateGUI() {
    Piece[][] cb = this.board.getBoard();

    for (int i = 0; i < cb.length; i++) {
        for (int j = 0; j < cb[i].length; j++) {
            if (cb[i][j] != null) {
                this.buttons[i][j].setText(cb[i][j].toString());
            } else {
                this.buttons[i][j].setText("");
            }
        }
    }

    if (this.board.getCurrentTurn() % 2 == 0) {
        this.turnLabel.setText("White To Move");
    } else {
        this.turnLabel.setText("Black To Move");
    }

    this.resetButtonColours();
}
```

Review

Now, after each move is made, the start and end piece of the last piece moved is highlighted in orange, to show the previous move made.

Overall, the program is beginning to work well, however it currently has no way of ending the game. Next, I will begin to look at determining whether the player is in check and then check mate, which will both allow the game to end and allow the program to check properly whether a move is legal or not.

Link to success criteria: #8

Determining check

Development

Then, I created the “isInCheck” function of the “ChessBoard” class, which returns true if the specified colour is in check, so that later moves will only be made if they do not put the user into check, and so that the user can be told if someone is in check or checkmate.

```
private int determineKingPosition(int colour) {
    for (int i = 0; i < this.board.length; i++) {
        for (int j = 0; j < this.board[i].length; j++) {
            if (this.board[i][j] != null && this.board[i][j].getColour() == colour
                && this.board[i][j].getType() == 6) {
                return i * 8 + j;
            }
        }
    }
    return -1;
}
```

To begin with, I created another function to find the position of a particular coloured king on the board and return it. If the king is not found, it returns -1. This is so that it can be determined whether this particular location is under attack or not – if not it would not know which position to check.

I then used this to find the position of the appropriate king, and stored this in the variable “kingPos”. If for some reason the king is not found, I return false, however this should not happen, as kings cannot be captured – either the player would be in check, and so would move out of check, or would be in checkmate, and so the game would end before the king was captured.

```
public boolean isInCheck(int colour) {
    int kingPos = this.determineKingPosition(colour);

    if (kingPos == -1) {
        return false;
    }
}
```

```
// Checking for pawns
if (colour == 0) { // if white
    // Black pawn attacking white king from top left
    if ((kingPos / 8) - 1 >= 0 && (kingPos % 8) - 1 >= 0 && this.board[(kingPos / 8) - 1][(kingPos % 8) - 1] != null) {
        if (this.board[(kingPos / 8) - 1][(kingPos % 8) - 1].getType() == 1 && this.board[(kingPos / 8) - 1][(kingPos % 8) - 1].getColour() != colour) {
            return true;
        }
    }
    // Black pawn attacking white king from top right
    if ((kingPos / 8) - 1 >= 0 && (kingPos % 8) + 1 < 8 && this.board[(kingPos / 8) - 1][(kingPos % 8) + 1] != null) {
        if (this.board[(kingPos / 8) - 1][(kingPos % 8) + 1].getType() == 1 && this.board[(kingPos / 8) - 1][(kingPos % 8) + 1].getColour() != colour) {
            return true;
        }
    }
} else { // if black
    // White pawn attacking black king from bottom left
    if ((kingPos / 8) + 1 < 8 && (kingPos % 8) - 1 >= 0 && this.board[(kingPos / 8) + 1][(kingPos % 8) - 1] != null) {
        if (this.board[(kingPos / 8) + 1][(kingPos % 8) - 1].getType() == 1 && this.board[(kingPos / 8) + 1][(kingPos % 8) - 1].getColour() != colour) {
            return true;
        }
    }
    // White pawn attacking black king from bottom right
    if ((kingPos / 8) - 1 >= 0 && (kingPos % 8) + 1 < 8 && this.board[(kingPos / 8) + 1][(kingPos % 8) + 1] != null) {
        if (this.board[(kingPos / 8) + 1][(kingPos % 8) + 1].getType() == 1 && this.board[(kingPos / 8) + 1][(kingPos % 8) + 1].getColour() != colour) {
            return true;
        }
    }
}
```

Then, I began to check if there were any pieces attacking the king. Firstly, I did the pawns. I determined what the colour of the pawn was, and then checked to the top left / top right (for white pieces) and to the bottom left / bottom right (for black pieces). Firstly, I checked that these locations were actually on the board, and not off the edge of it, and then I checked whether there was a piece there (i.e. that it wasn’t null). If yes, I checked that the type of the piece was equal to 1 (which is the number representing a pawn), and that it was of the opposite colour to the king. If yes, I returned true, as this means that a pawn is attacking the king – i.e. it is in check.

```

//Checking for kings
// Top left
if ((kingPos / 8) - 1 >= 0 && (kingPos % 8) - 1 >= 0 && this.board[(kingPos / 8) - 1][(kingPos % 8) - 1] != null) {
    if (this.board[(kingPos / 8) - 1][(kingPos % 8) - 1].getType() == 6 && this.board[(kingPos / 8) - 1][(kingPos % 8) - 1].getColour() != colour) {
        return true;
    }
}
// Above
if ((kingPos / 8) - 1 >= 0 && this.board[(kingPos / 8) - 1][kingPos % 8] != null) {
    if (this.board[(kingPos / 8) - 1][kingPos % 8].getType() == 6 && this.board[(kingPos / 8) - 1][kingPos % 8].getColour() != colour) {
        return true;
    }
}
// Top right
if ((kingPos / 8) - 1 >= 0 && (kingPos % 8) + 1 < 8 && this.board[(kingPos / 8) - 1][(kingPos % 8) + 1] != null) {
    if (this.board[(kingPos / 8) - 1][(kingPos % 8) + 1].getType() == 6 && this.board[(kingPos / 8) - 1][(kingPos % 8) + 1].getColour() != colour) {
        return true;
    }
}
// Right
if ((kingPos % 8) + 1 < 8 && this.board[kingPos / 8][(kingPos % 8) + 1] != null) {
    if (this.board[kingPos / 8][(kingPos % 8) + 1].getType() == 6 && this.board[kingPos / 8][(kingPos % 8) + 1].getColour() != colour) {
        return true;
    }
}
// Bottom right
if ((kingPos / 8) + 1 < 8 && (kingPos % 8) + 1 < 8 && this.board[(kingPos / 8) + 1][(kingPos % 8) + 1] != null) {
    if (this.board[(kingPos / 8) + 1][(kingPos % 8) + 1].getType() == 6 && this.board[(kingPos / 8) + 1][(kingPos % 8) + 1].getColour() != colour) {
        return true;
    }
}
// Below
if ((kingPos / 8) + 1 < 8 && this.board[(kingPos / 8) + 1][kingPos % 8] != null) {
    if (this.board[(kingPos / 8) + 1][kingPos % 8].getType() == 6 && this.board[(kingPos / 8) + 1][kingPos % 8].getColour() != colour) {
        return true;
    }
}
// Bottom left
if ((kingPos / 8) + 1 < 8 && (kingPos % 8) - 1 >= 0 && this.board[(kingPos / 8) + 1][(kingPos % 8) - 1] != null) {
    if (this.board[(kingPos / 8) + 1][(kingPos % 8) - 1].getType() == 6 && this.board[(kingPos / 8) + 1][(kingPos % 8) - 1].getColour() != colour) {
        return true;
    }
}
// Left
if ((kingPos % 8) - 1 >= 0 && this.board[kingPos / 8][(kingPos % 8) - 1] != null) {
    if (this.board[kingPos / 8][(kingPos % 8) - 1].getType() == 6 && this.board[kingPos / 8][(kingPos % 8) - 1].getColour() != colour) {
        return true;
    }
}

```

Next, I did the kings. I approached this in the same way as the pawns, only I did not need to check the colour this time, as the direction in which kings can attack does not depend on what colour it is, and instead of just checking two pieces diagonal to the king, I had to check all of the pieces within one square of the king, as this is how a king moves.

```

// Checking for knights
// One right, two up
if ((KingPos / 8) - 2 >= 0 && (KingPos % 8) + 1 < 8 && this.board[(KingPos / 8) - 2][(KingPos % 8) + 1] != null) {
    if (this.board[(KingPos / 8) - 2][(KingPos % 8) + 1].getType() == 3 && this.board[(KingPos / 8) - 2][(KingPos % 8) + 1].getColour() != colour) {
        return true;
    }
}
// Two right, one up
if ((KingPos / 8) - 1 >= 0 && (KingPos % 8) + 2 < 8 && this.board[(KingPos / 8) - 1][(KingPos % 8) + 2] != null) {
    if (this.board[(KingPos / 8) - 1][(KingPos % 8) + 2].getType() == 3 && this.board[(KingPos / 8) - 1][(KingPos % 8) + 2].getColour() != colour) {
        return true;
    }
}
// Two right, one down
if ((KingPos / 8) + 1 < 8 && (KingPos % 8) + 2 < 8 && this.board[(KingPos / 8) + 1][(KingPos % 8) + 2] != null) {
    if (this.board[(KingPos / 8) + 1][(KingPos % 8) + 2].getType() == 3 && this.board[(KingPos / 8) + 1][(KingPos % 8) + 2].getColour() != colour) {
        return true;
    }
}
// One right, two down
if ((KingPos / 8) + 2 < 8 && (KingPos % 8) + 1 < 8 && this.board[(KingPos / 8) + 2][(KingPos % 8) + 1] != null) {
    if (this.board[(KingPos / 8) + 2][(KingPos % 8) + 1].getType() == 3 && this.board[(KingPos / 8) + 2][(KingPos % 8) + 1].getColour() != colour) {
        return true;
    }
}
// One left, two down
if ((KingPos / 8) + 2 < 8 && (KingPos % 8) - 1 >= 0 && this.board[(KingPos / 8) + 2][(KingPos % 8) - 1] != null) {
    if (this.board[(KingPos / 8) + 2][(KingPos % 8) - 1].getType() == 3 && this.board[(KingPos / 8) + 2][(KingPos % 8) - 1].getColour() != colour) {
        return true;
    }
}
// Two left, one down
if ((KingPos / 8) + 1 < 8 && (KingPos % 8) - 2 >= 0 && this.board[(KingPos / 8) + 1][(KingPos % 8) - 2] != null) {
    if (this.board[(KingPos / 8) + 1][(KingPos % 8) - 2].getType() == 3 && this.board[(KingPos / 8) + 1][(KingPos % 8) - 2].getColour() != colour) {
        return true;
    }
}
// Two left, one up
if ((KingPos / 8) - 1 >= 0 && (KingPos % 8) - 2 >= 0 && this.board[(KingPos / 8) - 1][(KingPos % 8) - 2] != null) {
    if (this.board[(KingPos / 8) - 1][(KingPos % 8) - 2].getType() == 3 && this.board[(KingPos / 8) - 1][(KingPos % 8) - 2].getColour() != colour) {
        return true;
    }
}
// One left, two up
if ((KingPos / 8) - 2 >= 0 && (KingPos % 8) - 1 >= 0 && this.board[(KingPos / 8) - 2][(KingPos % 8) - 1] != null) {
    if (this.board[(KingPos / 8) - 2][(KingPos % 8) - 1].getType() == 3 && this.board[(KingPos / 8) - 2][(KingPos % 8) - 1].getColour() != colour) {
        return true;
    }
}

```

Next, I did the knights. Once again, I approached this in the same way as the pawns / kings, only instead of checking squares adjacent to the piece, I checked the squares one away in one direction and two away in another, as this is how a knight moves.

```

// Checking for rooks or queens
int col;
int row;
// Left
col = kingPos % 8;
while (col - 1 >= 0 && this.board[kingPos / 8][col - 1] == null) {
    col--;
}
if (col > 0 && (this.board[kingPos / 8][col - 1].getType() == 2 || this.board[kingPos / 8][col - 1].getType() == 5)
    && this.board[kingPos / 8][col - 1].getColour() != colour) {
    return true;
}
// Right
col = kingPos % 8;
while (col + 1 < 8 && this.board[kingPos / 8][col + 1] == null) {
    col++;
}
if (col < 7 && (this.board[kingPos / 8][col + 1].getType() == 2 || this.board[kingPos / 8][col + 1].getType() == 5)
    && this.board[kingPos / 8][col + 1].getColour() != colour) {
    return true;
}
// Up
row = kingPos / 8;
while (row - 1 >= 0 && this.board[row - 1][kingPos % 8] == null) {
    row--;
}
if (row > 0 && (this.board[row - 1][kingPos % 8].getType() == 2 || this.board[row - 1][kingPos % 8].getType() == 5)
    && this.board[row - 1][kingPos % 8].getColour() != colour) {
    return true;
}
// Down
row = kingPos / 8;
while (row + 1 < 8 && this.board[row + 1][kingPos % 8] == null) {
    row++;
}
if (row < 7 && (this.board[row + 1][kingPos % 8].getType() == 2 || this.board[row + 1][kingPos % 8].getType() == 5)
    && this.board[row + 1][kingPos % 8].getColour() != colour) {
    return true;
}

```

After that, I checked for both rooks and queens that were attacking horizontally or vertically. I check both rooks and queens at the same time, as they can both attack in the same directions, and so there is no need to do them separately, as they would use the same code. I approached this in a different way to the other types of pieces, as these can move any number of squares (in a specified direction, and without jumping over others), whereas the previous pieces could only move certain distances. In each direction (left, right, up, down), I calculated the furthest I could go in that direction before hitting another piece or the edge of the board using a for loop. In each direction, I then checked whether I had reached the end of the board, and if not, whether the type of the piece was equal to 2 or 5 (i.e. a rook or a queen) and whether the piece reached was of the opposite colour to the king. If yes, I returned true, as this means the king is under attack by that piece.

```

// Checking for bishops or queens
// North west
col = kingPos % 8;
row = kingPos / 8;
while (col - 1 >= 0 && row - 1 >= 0 && this.board[row - 1][col - 1] == null) {
    col--;
    row--;
}
if (col > 0 && row > 0 && (this.board[row - 1][col - 1].getType() == 4 || this.board[row - 1][col - 1].getType() == 5)
    && this.board[row - 1][col - 1].getColour() != colour) {
    return true;
}
// North east
col = kingPos % 8;
row = kingPos / 8;
while (col + 1 < 8 && row - 1 >= 0 && this.board[row - 1][col + 1] == null) {
    col++;
    row--;
}
if (col < 7 && row > 0 && (this.board[row - 1][col + 1].getType() == 4 || this.board[row - 1][col + 1].getType() == 5)
    && this.board[row - 1][col + 1].getColour() != colour) {
    return true;
}
// South east
col = kingPos % 8;
row = kingPos / 8;
while (col + 1 < 8 && row + 1 < 8 && this.board[row + 1][col + 1] == null) {
    col++;
    row++;
}
if (col < 7 && row < 7 && (this.board[row + 1][col + 1].getType() == 4 || this.board[row + 1][col + 1].getType() == 5)
    && this.board[row + 1][col + 1].getColour() != colour) {
    return true;
}
// South west
col = kingPos % 8;
row = kingPos / 8;
while (col - 1 >= 0 && row + 1 < 8 && this.board[row + 1][col - 1] == null) {
    col--;
    row++;
}
return col > 0 && row < 7 && (this.board[row + 1][col - 1].getType() == 4 || this.board[row + 1][col - 1].getType() == 5)
    && this.board[row + 1][col - 1].getColour() != colour;

```

Finally, I did the bishops and queens attacking diagonally. I did this in the same way that I did the rooks, only instead of checking left, right, up & down, I checked north west, north east, south east and south west.

Review

I will test whether this is working correctly when I implement the ability for the program to tell the user when they are in check / when it prevents the user making moves that put / keep them in check, which I will do next.

Link to success criteria: #12

Preventing user making moves that put / keep them in check

Development

```
public void undoMove() {
    if (this.moveHistory.getSize() > 0) {
        String move = this.moveHistory.pop();
        this.currentTurn--;

        int startPos = Integer.valueOf(move.substring(0, 2));
        int endPos = Integer.valueOf(move.substring(2, 4));
        int typeOfCapturedPiece = Integer.valueOf(move.substring(4, 5));
        Piece piece = null;

        switch (typeOfCapturedPiece) {
            case 1:
                piece = new Pawn((this.currentTurn - 1) % 2);
                break;
            case 2:
                piece = new Rook((this.currentTurn - 1) % 2);
                break;
            case 3:
                piece = new Knight((this.currentTurn - 1) % 2);
                break;
            case 4:
                piece = new Bishop((this.currentTurn - 1) % 2);
                break;
            case 5:
                piece = new Queen((this.currentTurn - 1) % 2);
                break;
            default:
                break;
        }

        this.board[startPos / 8][startPos % 8] = this.board[endPos / 8][endPos % 8];
        this.board[endPos / 8][endPos % 8] = piece;
    }
}
```

To begin with, I added the procedure “undoMove” to the “ChessBoard” class. I wrote this method so that moves can be temporarily made, it can be determined whether the move caused the user to put/keep themselves in check, and then can be undone after this has been determined. Furthermore, I will later implement a feature to allow the user to be able to select to undo a move if they would like to.

To begin with, I checked that the size of moveHistory was greater than 0 (i.e. whether any moves have been made so far in the game or not). This is so that a move is not attempted to be undone if no moves have actually been made, as this may cause a null pointer exception. If any moves have been made, firstly the most recently added move is removed from the list (as undoing the move means the board will go back to as if the move had never been made, and so the move does not need to be stored in the history) and stored in a variable (so the information about the positions of the pieces can be extracted later). Then, from the string, I get the start / end position and type of piece that was captured, so that it is known which piece to move, where to move it back to, and what type of piece needs to be put back onto the board (if any). I then create a new variable of type Piece, which is needed to store the piece that will be re-placed onto the board, and use a switch statement to set this variable to a new object of the correct type and colour. Then, the location the piece was initially at before it was moved is set equal to the piece that was moved, so that it goes back to its original location. Then, the position that the piece was moved to is set to the piece that was captured (i.e. the piece that was there before the move was made).

```
if (this.isInCheck((this.currentTurn - 1) % 2)) {
    this.undoMove();
}
```

Then, I added an if statement to the end of the “makeMove” procedure, to determine whether the move put a player’s own king into check. If so, I undo the move, as a move is not legal if it puts yourself into check. This is what prevents the user from making a move that puts themselves into check. This is so the user is not allowed to make an illegal move – if the move does put them into check, the undoMove procedure decreases currentTurn by 1, meaning that they will be able to select another move instead – it will not move onto the next player.

Testing

However, when testing I found that whilst a user now cannot make a move that will put them into check, it will still be shown in yellow to them as a valid move when they select a piece to move, which shouldn't happen. This is because I have not yet edited the "getValidEndPositions" method to account for check, and so I will need to update this function.

Remedial

```
public boolean[][] getValidEndPositions(int pos) {
    boolean[][] validPositions = new boolean[8][8];

    for (int i = 0; i < 64; i++) {
        String move = "";
        if (pos < 10) {
            move += "0";
        }
        move += pos;

        if (i < 10) {
            move += "0";
        }
        move += i;

        if (i != pos && this.board[pos / 8][pos % 8].isValidMove(move, this.board)) {
            int startPos = Integer.valueOf(move.substring(0, 2));
            int endPos = Integer.valueOf(move.substring(2, 4));

            if (this.board[endPos / 8][endPos % 8] == null) {
                move += "0";
            } else {
                move += this.board[endPos / 8][endPos % 8].getType();
            }

            this.moveHistory.add(move);
            this.currentTurn++;

            this.board[endPos / 8][endPos % 8] = this.board[startPos / 8][startPos % 8];
            this.board[startPos / 8][startPos % 8] = null;

            if (!this.isInCheck((this.currentTurn - 1) % 2)) {
                validPositions[i / 8][i % 8] = true;
            }
            this.undoMove();
        }
    }

    return validPositions;
}
```

In order to fix this, I modified the "getValidEndPositions" method of the "ChessBoard" class, which is used in the "showPossibleEndPositions" method, which is responsible for highlighting the valid moves in yellow to the user.

Instead of just checking whether a move is pseudo-legal, I also check whether making that move would put a players own king into check (i.e. whether it is a legal move). To do this, I make the move, and then determine whether or not the appropriate king is in check. If it is not in check, then it is a valid move, and so I set the corresponding location in the "validPositions" array to true to indicate this. Then, I undo the move. After each possible move has been checked for that piece, I return the "validPositions" array. This is so that now, if a user makes a move that does put themselves into check, it will not be set as a valid position to move to in the "validPositions" array, and so it will not be highlighted in yellow.

This now works correctly – the user is unable to make a move that would put themselves into check, or that does not take themselves out of check, and is not shown any move that would do so in yellow on the board. Below, I tested whether the “`isInCheck`” method was working correctly:



For example, here the king is not allowed to move to the square directly in front of them, as this would mean the pawn would be attacking the king.



Here, the king is under attack by the rook, and so must make a move that brings them out of check, and so cannot move to the square directly in front of them, as they would still be in check if they did this.



Here, if the pawn (which is highlighted in red, showing it has no possible moves) were to move, the king would be under attack by the queen, and so this pawn cannot move, despite having otherwise valid moves it could take, and so is working as expected.



Here, the king cannot move to the square to the top left, since if it did move there, it would be under attack by the knight.

Review

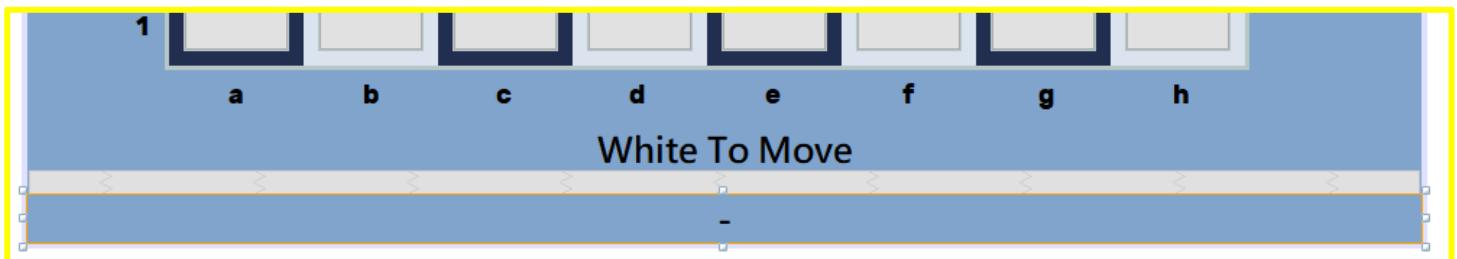
Now, I know that the subroutine for determining whether or not a player is in check is working correctly, and that the user is now unable to make a move that keeps themselves in check.

Whilst the program now responds to a player being in check, it does not actually tell the user this. Therefore, I will implement this next, as this is a feature which my client has said he would like to be included in the program. Also, now that the program is able to determine whether or not someone is in check, I should use this feature to help determine whether or not a player is in checkmate, so that the game is able to end, and a winner determined.

Link to success criteria: #11, #12

Telling the user if they are in check

Development

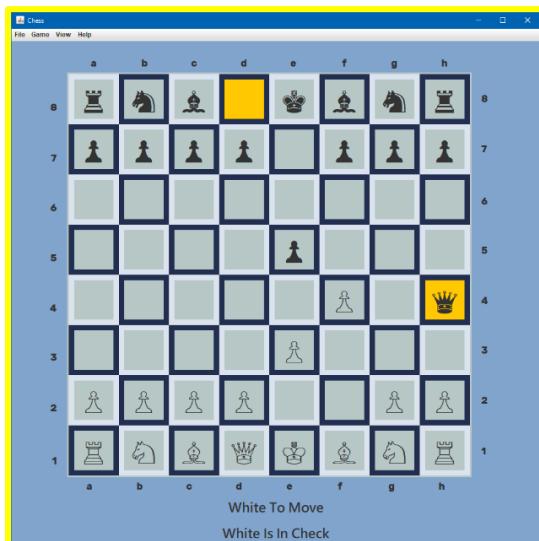


Firstly, I added a `JLabel` below the “turnLabel” (so that I can update this with the appropriate text if a player is in check), and renamed it to “checkLabel” (making the purpose of the label clear and the code easier to read). I will set the text of this to either be blank, “White Is In Check” or “Black Is In Check”, depending on the state of the game, so that the user can be informed if someone is in check / not be shown if no one is in check.

```
if(this.board.isInCheck(0)) {  
    this.checkLabel.setText("White Is In Check");  
} else if(this.board.isInCheck(1)) {  
    this.checkLabel.setText("Black Is In Check");  
} else {  
    this.checkLabel.setText("");  
}
```

Then, I added this if statement to the “updateGUI” method, which determines whether white is in check, black is in check, or no one is in check, and sets the text of the “checkLabel” accordingly.

Testing



When tested this appears to work as expected, showing the correct text depending on who, if anyone, is in check, showing that both this and the “isInCheck” method is working correctly.

Review

Overall, the user is now able to input their moves, they are shown the legal moves the piece they select can make, the program can switch turns, and it can tell the user if they are in check. Whilst the game is now functional, it cannot yet end. Therefore, I will implement the ability to determine checkmate next.

Link to success criteria: #13

Checkmate and ending the game

Development

Then, I decided to implement the feature to determine whether a player is in checkmate, as this is an important part of the game, and the other component of the program needed for this (e.g. determining check) have now been written. My approach was to go through each square on the board, and if a piece was of the correct colour (i.e. the player whose turn it is to move), find all of its possible legal moves. If this list of moves is not empty, then it means that the player is able to make a move, so I return false – the player is not in checkmate. If after checking all of the pieces of the given colour, none of them had any possible moves, I return true, as this means the player is in checkmate, as they have no legal moves left to be made (i.e. a move that does not leave / put them in check).

```
private ArrayList<String> getAllMovesForPiece(int pos) {
    ArrayList<String> moves = new ArrayList<>();

    for (int i = 0; i < 64; i++) {
        String move = "";
        if (pos < 10) {
            move += "0";
        }
        move += pos;

        if (i < 10) {
            move += "0";
        }
        move += i;

        if (i != pos && this.board[pos / 8][pos % 8].isValidMove(move, this.board)) {
            int startPos = Integer.valueOf(move.substring(0, 2));
            int endPos = Integer.valueOf(move.substring(2, 4));

            if (this.board[endPos / 8][endPos % 8] == null) {
                move += "0";
            } else {
                move += this.board[endPos / 8][endPos % 8].getType();
            }

            this.moveHistory.add(move);
            this.currentTurn++;

            this.board[endPos / 8][endPos % 8] = this.board[startPos / 8][startPos % 8];
            this.board[startPos / 8][startPos % 8] = null;

            if (!this.isInCheck((this.currentTurn - 1) % 2)) {
                moves.add(move);
            }
            this.undoMove();
        }
    }

    return moves;
}
```

Firstly, I created the “getAllMovesForPiece” method, which finds all the possible legal moves for a particular piece, and returns these as an ArrayList. To find these moves, a for loop is used to iterate through each square on the board. A “move” string is generated for each, with the first two characters representing the start position (i.e. the location of the piece that the moves are being found for), and the next two representing the end location (i.e. a square on the board which we want to check if it is possible for the piece to be moved to). Then, I check whether moving the piece given as a parameter to this location is pseudo-legal. If it is, I then check that it doesn’t put or keep the king in check, by making the move, determining whether it put the king into check, and then undoing the move. If the move does not keep / put the player in check, then it is a legal move, and so I add it to the arraylist of moves which are valid. After each square on the board has been checked, I return the list of moves.

```

public boolean isInCheckMate(int colour) {
    ArrayList<String> moves;

    for(int i = 0; i < 8; i++) {
        for(int j = 0; j < 8; j++) {
            if(this.board[i][j] != null && this.board[i][j].getColour() == colour) {
                moves = this.getAllMovesForPiece(i * 8 + j);
                if(!moves.isEmpty()) {
                    return false;
                }
            }
        }
    }

    return true;
}

```

Then, I wrote the “isInCheckMate” method. This iterates through each square on the board using a for loop. If a square has a piece in it, and the colour of that piece is equal to the colour of the side being checked for checkmate, then all of the possible moves are generated for that piece, using the previous method. If this list is not empty, then that side is not in checkmate, as they are able to make a legal move, so false is returned. If after checking each piece of the appropriate colour, none of them had any moves (i.e. false was not returned and the function is still running), I return true, as this means there are no legal moves possible for any of their pieces, and so they are in checkmate.

```

private void endGame() {
    this.board = new ChessBoard();
    this.startPos = -1;
    this.endPos = -1;
    this.updateGUI();
}

```

When a player is in checkmate, the game should end, so next I created a procedure to end the game, which will also allow me to test whether the “isInCheckmate” method is working correctly. This sets “board” equal to a new ChessBoard, resets “startPos” and “endPos” to their default values, and finally updates the GUI to show this to the user so that it will appear to them that a new game has been started.

```

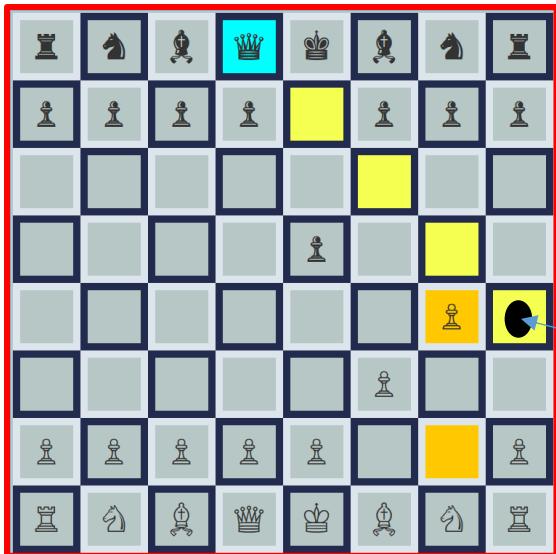
if(this.board.isInCheckMate(0)) {
    this.endGame();
} else if(this.board.isInCheckMate(1)) {
    this.endGame();
}

```

I then add this if statement to the “buttonPressed” method, to determine whether white or black is in checkmate. If they are, the game ends.

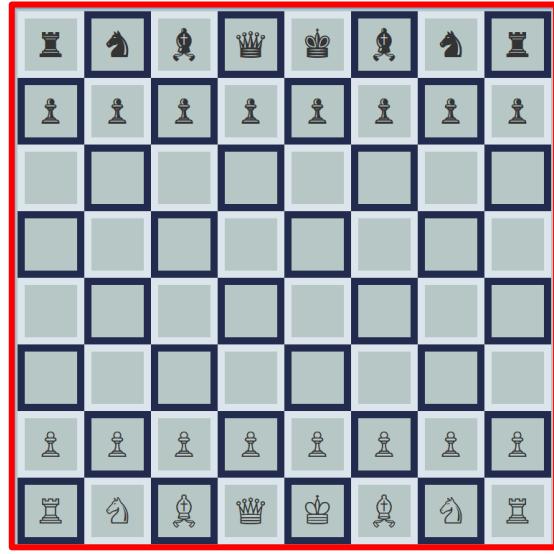
Testing

When I tested this, the “isInCheckmate” method appeared to work correctly, ending the game and resetting the board correctly when a player was in check, however it happens straight away and does not tell the user who won, and so could be confusing for them. To fix this I will make a window appear when the game ends, telling the user who has won, meaning that they will be clear who has won, and that it will not reset immediately – they will be able to look at the board before the next game starts.

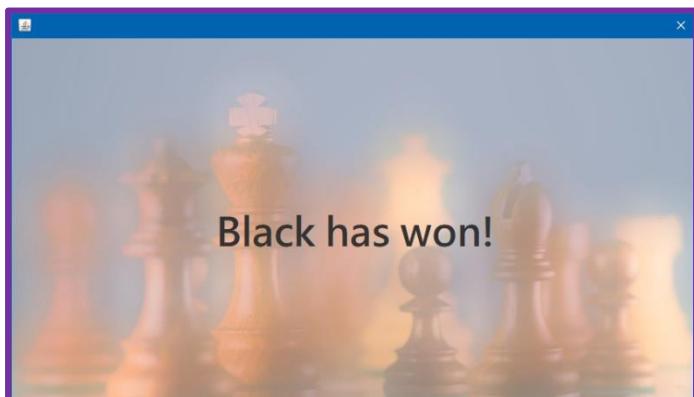


Queen moved here

Remedial



Board immediately
resets



In order to fix this, firstly I created a new `JDialog`, and added two `JLabels` to it. One contains an icon, which is a picture of some chess pieces, and acts as a background image. The other contains text, so that the user can be told who has won the game.

```
public void setWinner(int colour) {
    if(colour == 0) {
        this.jLabel2.setText("White has won!");
    } else {
        this.jLabel2.setText("Black has won!");
    }
}
```

I added a “`setWinner`” method to the `JDialog`, which sets the `JLabel` to either “White has won!” or “Black has won!”, depending on the colour passed to it.

```
if(this.board.isInCheckMate(0)) {
    ShowWinner winnerDialog = new ShowWinner(new javax.swing.JFrame(), true);
    winnerDialog.setWinner(1);
    winnerDialog.setVisible(true);
    this.endGame();
} else if(this.board.isInCheckMate(1)) {
    ShowWinner winnerDialog = new ShowWinner(new javax.swing.JFrame(), true);
    winnerDialog.setWinner(0);
    winnerDialog.setVisible(true);
    this.endGame();
}
```

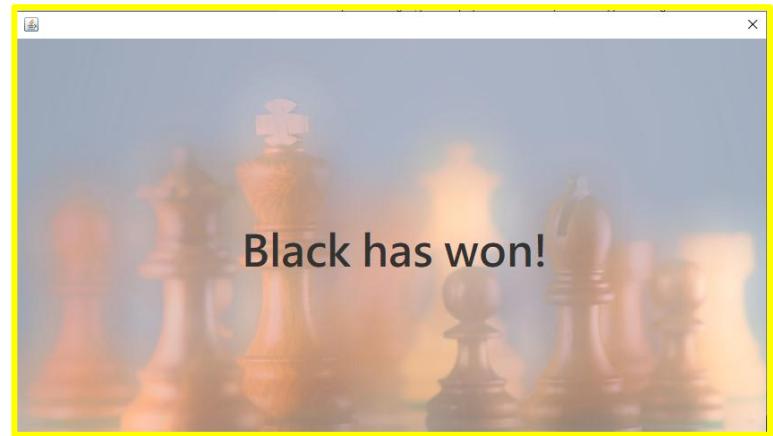
I then added this if statement to the “`buttonPressed`” method, which is meant to set the appropriate winner and display thing to the user, however it resulted in some unexpected behaviour, causing moves to be made when they shouldn’t. This is due to moves not being undone correctly when determining checkmate.

```

if(this.board.isInCheck(0)) {
    this.checkLabel.setText("White Is In Check");
    if(this.board.isInCheckMate(0)) {
        ShowWinner winnerDialog = new ShowWinner(new javax.swing.JFrame(), true);
        winnerDialog.setWinner(1);
        winnerDialog.setVisible(true);
        this.endGame();
    }
} else if(this.board.isInCheck(1)) {
    this.checkLabel.setText("Black Is In Check");
    if(this.board.isInCheckMate(1)) {
        ShowWinner winnerDialog = new ShowWinner(new javax.swing.JFrame(), true);
        winnerDialog.setWinner(0);
        winnerDialog.setVisible(true);
        this.endGame();
    }
} else {
    this.checkLabel.setText("");
}

```

Because a player can only be in checkmate if they are in check, I decided to move this if statement from the “buttonPressed” method, and instead put it in the “updateGUI” method, where it currently checks whether a player is in check or not, and sets the JLabel accordingly, as this makes more sense (check / checkmate being handled in the same place, rather than in different subroutines across the program), and means that it will not need to determine check multiple times. If a particular colour is in check, only then does it determine whether or not it is in checkmate. After moving this, no unexpected moves are made when they shouldn’t be.



Now, when testing this, when a player is in checkmate, a window appears telling the user who has won, suggesting that the “isInCheckmate” and “endGame” methods are now working correctly, the game is now no longer immediately reset (and is instead only reset after the window telling them who has won is closed, so that the user is able to look at the board if they would like, and can clearly see the result), and the user is clearly told who has won in a new window.

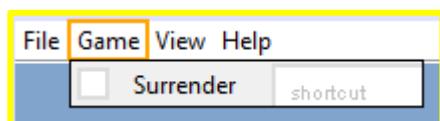
[Review](#)

Now, the program is able to end if someone is in checkmate, and the winner is displayed to the user. Here, most of the essential features of the program have been implemented for it to be functional, so I can start to work on some of the other features the client / end users requested to make it easier to use.

Link to success criteria: #14, #15

Allowing user to end the game

Development

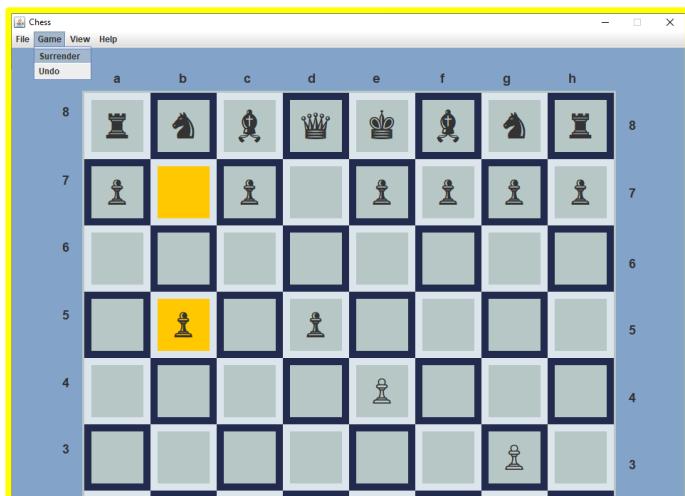


I then added a JMenuItem to the JMenu, and set the text on it to “surrender”, so that the user will be able to press a button to end the game early if they would like to.

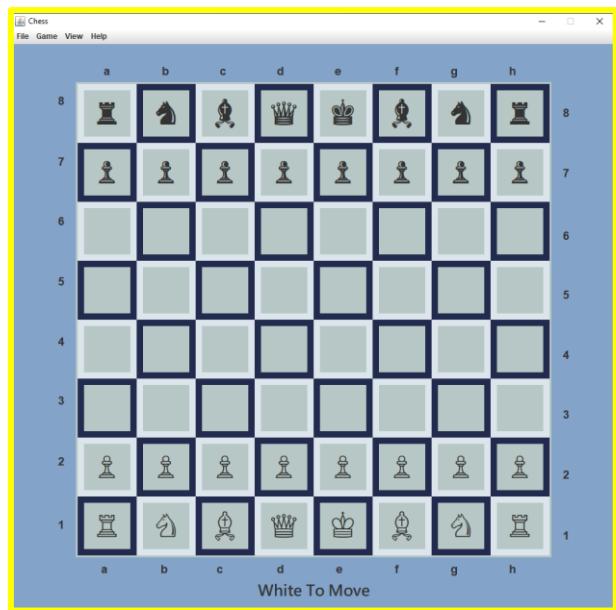
```
private void jMenuItem3ActionPerformed(java.awt.event.ActionEvent evt) {  
    // TODO add your handling code here:  
    this.endGame();  
}
```

When this is pressed, it calls the “endGame” method, so that the board is reset back to its initial state, allowing a new game to be started.

Testing



When tested, when this JMenuItem is pressed, the board and whose turn it is to move is reset correctly, and this is shown to the user.



Review

Now, the user is able to end a game early if they would like to.

Next, I will begin to work on allowing the user to play against the computer. This is because it is an important feature, as the client has said he is expecting the solution to include this, but also relies on the previous features such as check / checking for legal moves.

Link to success criteria: #16

Selecting opponent

Development

Select Mode:

One Player

Two Player

Next, I decided to implement the ability to play against the computer. To start with, I created a new `JDialog` to allow the user to select whether they would like to play against the computer, or against someone else. I added a `JLabel` to act as a background image, by setting it to display this image as an icon. Then, I added another `JLabel` with the text “Select Mode:”, prompting the user for input. Then, I added two `JButtons`, allowing the user to select who they would like to play against – another human or the computer.

```
private boolean playingAgainstComputer;
```

```
public SelectPlayer(java.awt.Frame parent, boolean modal) {
    super(parent, modal);
    initComponents();
    this.playingAgainstComputer = false;
}
```

I then added a boolean called `“playingAgainstComputer”` to the `JDialog`, to store the user’s preference of who they would like to play against. In the constructor I set this to false, so if the user doesn’t select an option and just closes the window, it will default to a two player game, so that even if the user doesn’t select an option, a mode will still be selected (as otherwise an error may occur).

```
private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
    this.playingAgainstComputer = false;
    this.setVisible(false);
    this.dispose();
}

private void jButton2ActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
    this.playingAgainstComputer = true;
    this.setVisible(false);
    this.dispose();
}
```

Depending on which of these buttons is pressed, the variable `“playingAgainstComputer”` is set accordingly, and the `JDialog` is set to not be visible, and is closed.

```
public boolean showDialog() {
    this.setVisible(true);
    return this.playingAgainstComputer;
}
```

I then added a “`showDialog`” method, which sets the window to be visible (allowing the user to select an option), and returns the variable `“playingAgainstComputer”` after it has been set by the user, so that the program will be able to know whether to allow the user to make a move, or whether it should determine and make a move itself.

```

public void selectPlayer(JFrame parent, boolean modal) {
    SelectPlayer selectPlayer = new SelectPlayer(parent, modal);
    this.playingAgainstComputer = selectPlayer.showDialog();
}

```

```
this.board.selectPlayer(this, true);
```

Next, I added the method “selectPlayer” to the “ChessBoard” class. This takes two parameters, the parent JFrame (which in this case will be the main JFrame), and a boolean which determines whether the JDialog will be displayed as modal (i.e. blocks other windows whilst it is open) or not. I then instantiate the JDialog “SelectPlayer”, and call the “showDialog” method (so that the window is shown to the user, allowing them to select an option), setting “playingAgainstComputer” equal to the boolean that this method returns.

I then called this method in the constructor of the JFrame, so that the user is asked for the mode they would like to play when the program is first run. I set the JDialog to be displayed as modal so that the user is not able to start playing the game before the window is closed (i.e. before it has been determined whether they are playing against the computer or not).

I then also called this method in the “endGame” method in the same way, so that the user can choose to play a different mode when they start a new game, rather than always playing against the same opponent, so that they do not have to restart the whole program if they change their mind about who they would like to play against.

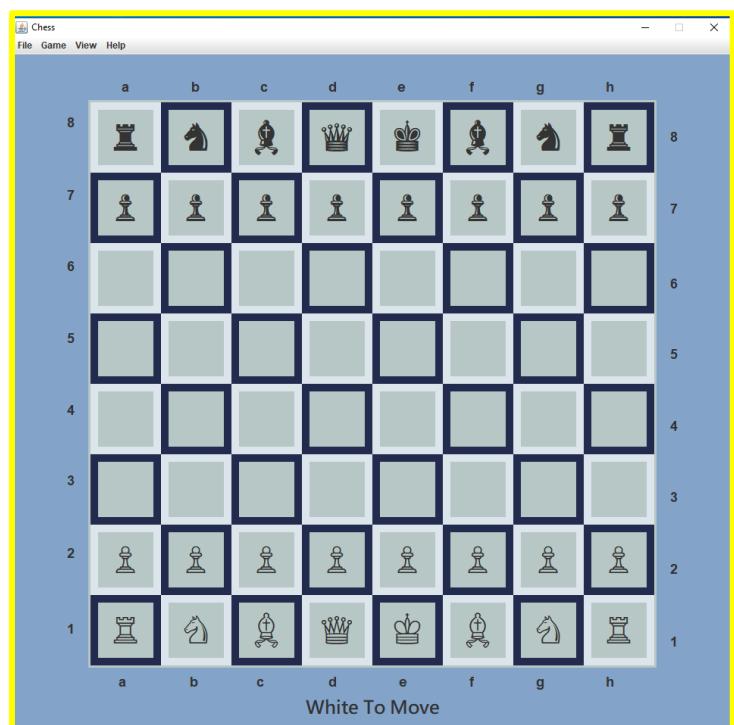
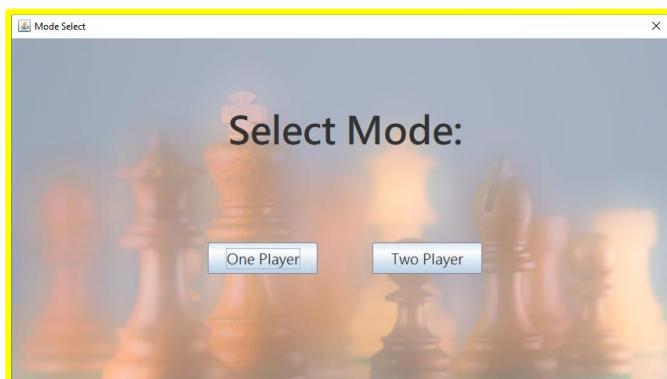
```

public boolean isPlayingAgainstComputer() {
    return this.playingAgainstComputer;
}

```

Testing

When tested, when the program is first run, the user is shown a window, asking them who they would like to play against. After they select an option, the main window showing the board is displayed.



Next, I added the method “isPlayingAgainstComputer” to the “ChessBoard” class, so that the JFrame will be able to determine whether it should wait for the user to input a move (since the JFrame is where user input to the program is handled), or let the computer make a move after the first user has moved.

Review

Now, the user can select who they would like to play against. This means that I can now implement the ability for the computer to actually make a move (if the user selects to play against the computer).

Link to success criteria: #5

Allowing computer to move

Development

```
private ArrayList<String> combineLists(ArrayList<String> a, ArrayList<String> b) {
    if (a.size() < b.size()) {
        for (int i = 0; i < a.size(); i++) {
            b.add(a.get(i));
        }
        return b;
    } else {
        for (int i = 0; i < b.size(); i++) {
            a.add(b.get(i));
        }
        return a;
    }
}
```

Next, I wanted to allow the computer to determine a move to make. To do this, I decided to find all of the black pieces, get their possible moves as a list (using the “getAllMovesForPiece” method), and add these to a list containing all of the moves for the black pieces, and then return this list, so that the computer will then have a set of valid moves to choose from. Firstly, I created a “combineLists” method, which takes two ArrayLists “a” and “b”, and adds the shorter of the two lists to the longer of the two, and then returns the resulting ArrayList – a combination of the two. I created this so that the various arraylists that will be returned when the “getAllMovesForPiece” method is called multiple times can be combined into one long arraylist, as the moves the computer will need to choose from do not need to be stored in separate arrays for each individual piece.

```
public ArrayList<String> getAllMovesForColour(int colour) {
    ArrayList<String> moves = new ArrayList<>();

    for (int i = 0; i < 8; i++) {
        for (int j = 0; j < 8; j++) {
            if (this.board[i][j] != null && this.board[i][j].getColour() == colour) {
                moves = this.combineLists(moves, this.getAllMovesForPiece(i * 8 + j));
            }
        }
    }

    return moves;
}
```

Then, I implemented the “getAllMovesForColour” method, which creates an ArrayList to store all the moves for that colour, and then iterates through each square on the board, checking if it contains a piece of the right colour. If it does, I find all of the possible moves for that piece, and add these to the list storing all the moves for that colour using the “combineLists” method that was just written. After it has checked each square, it returns the list of all possible moves.

```

public void makeComputerMove() {
    ArrayList<String> computerMoves = this.getAllMovesForColour(1);
    Random rand = new Random();
    String move = computerMoves.get(rand.nextInt(computerMoves.size()));
    this.makeMove(move);
}

```

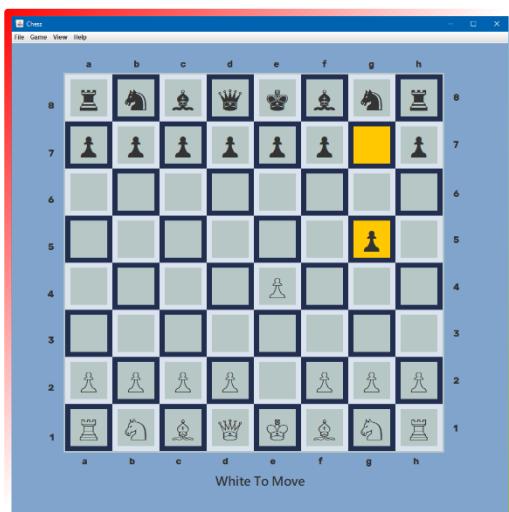
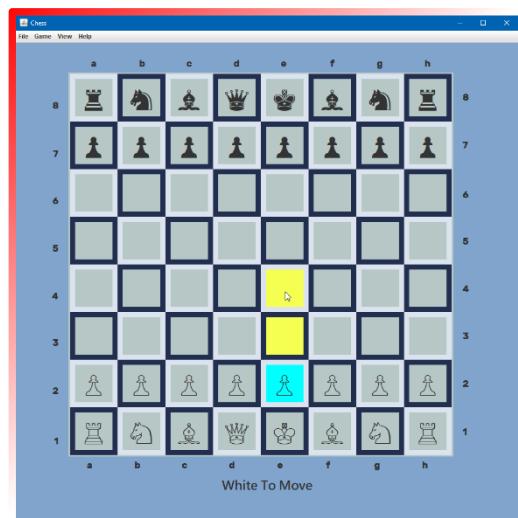
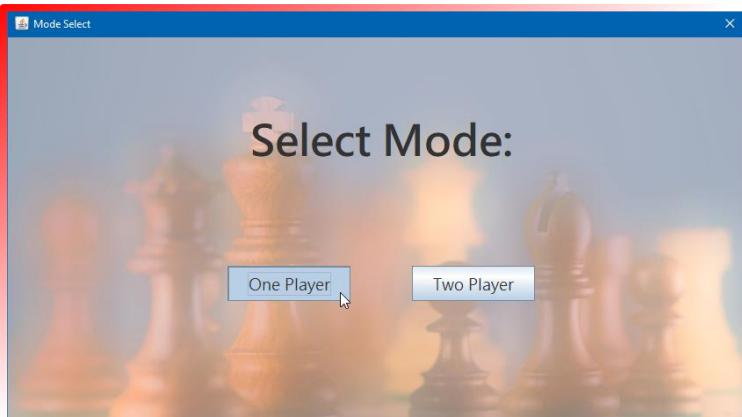
After that, I added the method “makeComputerMove” to the “ChessBoard” class. Currently, this simply determines all of the moves that the computer can make (i.e. all possible legal moves for the black pieces), selects a random one from the list, and then makes that move. I have done this so that I can test that the computer is able to select a move from the list and make the move correctly, before moving onto allowing the computer to determine the best move to make.

```

if(this.board.isPlayingAgainstComputer() && this.board.getCurrentTurn() % 2 == 1) {
    this.board.makeComputerMove();
    this.updateGUI();
}

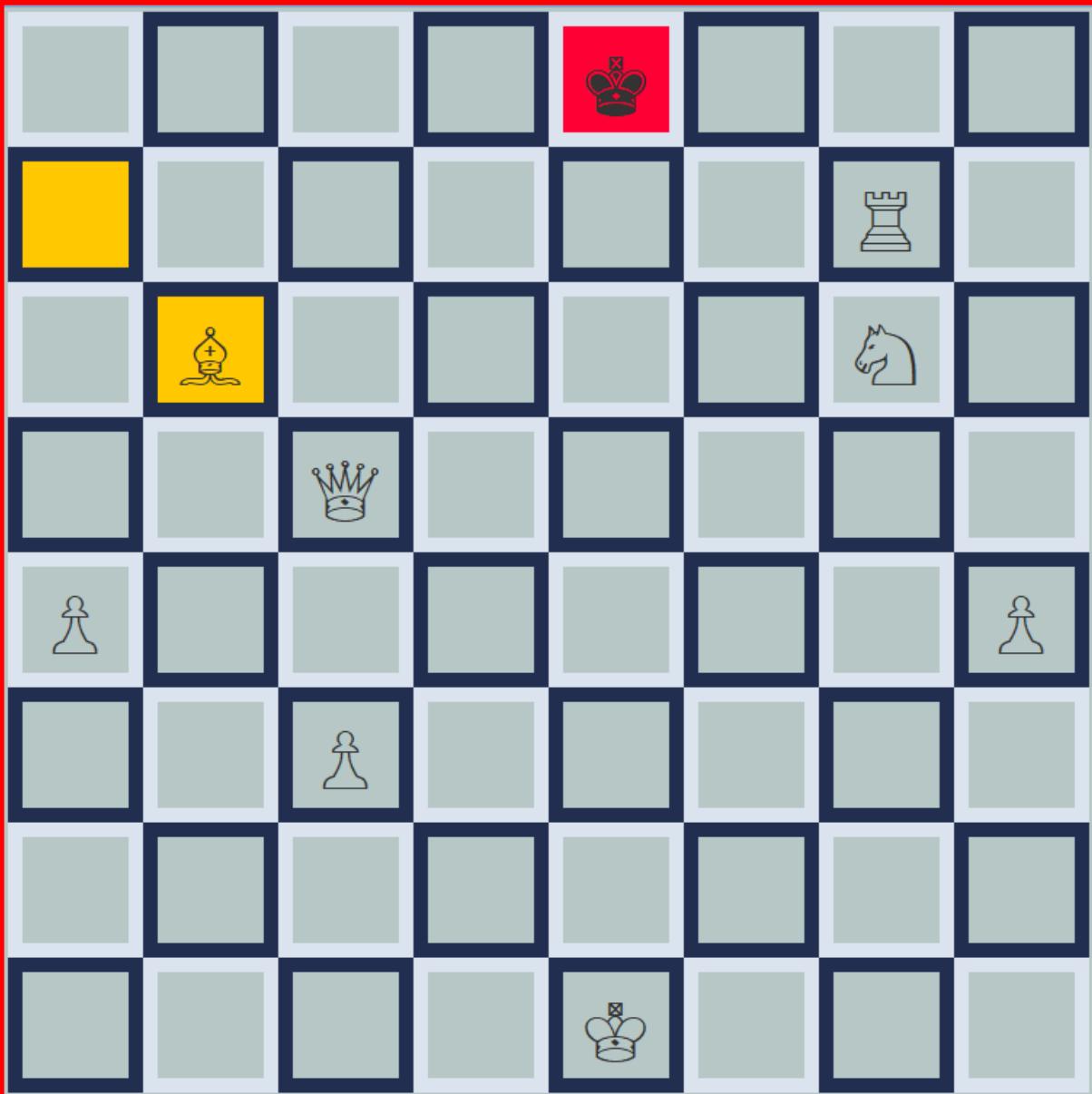
```

Then, I added this if statement to the “buttonPressed” method. After the user has selected the move they want to make, and that move has been made, it is checked whether the user has selected to play against the computer, and whether it is currently black’s move (as the computer always plays as black). If yes, the method “makeComputerMove” is called, as it is the computers turn to move, and then the GUI is updated to show this move to the user.



I then tested that this was working as expected, by selecting to play against the computer, and making a move. As you can see, the computer then made a move straight after I had made a move, and then white is allowed to move again, and so here it works as expected.

```
Exception in thread "AWT-EventQueue-0" java.lang.IllegalArgumentException: bound must be positive  
at java.util.Random.nextInt (Random.java:388)  
at ChessBoard.makeComputerMove (ChessBoard.java:138)
```



However, I noticed that it produced an `IllegalArgumentException`, when the computer had no possible moves to make, but also was not under attack (i.e. unable to move, but not in checkmate). From looking at the code I found that this was because the size of the list of all possible moves will be 0, which is not positive, and the “nextInt” method of the “Random” class only allows positive integers to be passed to it. If the king was under attack (i.e. it was checkmate), the game would end before this error was encountered. In order to fix this, I need to implement a way to determine if the game is in a stalemate – one player has no possible moves left to make, but also is not under attack, and so is not in checkmate – and the ability to end the game if this is the case.

Remedial

```
public boolean currentPlayerHasNoLegalmoves () {  
    return this.getAllMovesForColour(this.currentTurn % 2).isEmpty();  
}
```

To start with, I added the “currentPlayerHasNoLegalMoves” method to the “ChessBoard” class, which generates all of the possible moves for the current player as a list, and checks if this list is empty. If it is empty, the current player has no moves available to them, and so this method returns true. Otherwise, returns false – the current player is able to make a move. This will allow me to determine whether a player is unable to move, even if their king is not under attack.

```

if(this.board.isInCheck(0)) {
    this.checkLabel.setText("White Is In Check");
}
if(this.board.isInCheckMate(0)) {
    ShowWinner winnerDialog = new ShowWinner(new javax.swing.JFrame(), true);
    winnerDialog.setWinner(1);
    winnerDialog.setVisible(true);
    this.endGame();
}
else if(this.board.isInCheck(1)) {
    this.checkLabel.setText("Black Is In Check");
}
if(this.board.isInCheckMate(1)) {
    ShowWinner winnerDialog = new ShowWinner(new javax.swing.JFrame(), true);
    winnerDialog.setWinner(0);
    winnerDialog.setVisible(true);
    this.endGame();
}
} else if (this.board.currentPlayerHasNoLegalmoves()) {
    ShowWinner winnerDialog = new ShowWinner(new javax.swing.JFrame(), true);
    winnerDialog.setWinner(2);
    winnerDialog.setVisible(true);
    this.endGame();
} else {
    this.checkLabel.setText("");
}

```

Then, I modified this if statement in the “updateGUI” method. If none of the players are in check, and the current player has no legal moves, then it is a stalemate, and so I display the window showing the winner, and end the game. Here I set the winner to 2, so now I need to modify the “ShowWinner” JDialog to recognise that a winner of “2” means the game was a stalemate.

```

public void setWinner(int colour) {
    if(colour == 0) {
        this.jLabel2.setText("White has won!");
    } else if (colour == 1) {
        this.jLabel2.setText("Black has won!");
    } else {
        this.jLabel2.setText("Stalemate");
    }
}

```

Here, I modified the “setWinner” method. If colour is not equal to 0 or 1, the text of the JLabel is set to “Stalemate”, informing the user that the game is in stalemate.



Here, the king has no legal moves to make, but is not in check – so it is a stalemate. This is correctly shown to the user, and the IllegalArgumentException no longer occurs, because the game finishes before the computer attempts to make a move if they have none left.

```

private final int[][] pawnEvalBonus = {
    {0, 0, 0, 0, 0, 0, 0, 0},
    {-50, -50, -50, -50, -50, -50, -50, -50},
    {-10, -10, -20, -30, -30, -20, -10, -10},
    {-5, -5, -10, -25, -25, -10, -5, -5},
    {0, 0, 0, -20, -20, 0, 0, 0},
    {-5, 5, 10, 0, 0, 10, 5, -5},
    {-5, -10, -10, 20, 20, -10, -10, -5},
    {0, 0, 0, 0, 0, 0, 0, 0}
};

private final int[][] rookEvalBonus = {
    {0, 0, 0, 0, 0, 0, 0, 0},
    {-5, -10, -10, -10, -10, -10, -10, -5},
    {5, 0, 0, 0, 0, 0, 0, 5},
    {5, 0, 0, 0, 0, 0, 0, 5},
    {5, 0, 0, 0, 0, 0, 0, 5},
    {5, 0, 0, 0, 0, 0, 0, 5},
    {5, 0, 0, 0, 0, 0, 0, 5},
    {0, 0, 0, -5, -5, 0, 0, 0}
};

private final int[][] knightEvalBonus = {
    {50, 40, 30, 30, 30, 30, 40, 50},
    {40, 20, 0, 0, 0, 0, 20, 40},
    {30, 0, -10, -15, -15, -10, 0, 30},
    {30, -5, -15, -20, -20, -15, -5, 30},
    {30, 0, -15, -20, -20, -15, 0, 30},
    {30, -5, -10, -15, -15, -10, -5, 30},
    {40, 20, 0, -5, -5, 0, 20, 40},
    {50, 40, 30, 30, 30, 30, 40, 50}
};

```

```

private final int[][] bishopEvalBonus = {
    {20, 10, 10, 10, 10, 10, 10, 20},
    {10, 0, 0, 0, 0, 0, 0, 10},
    {10, 0, -5, -10, -10, 5, 0, 10},
    {10, -5, -5, -10, -10, -5, -5, 10},
    {10, 0, -10, -10, -10, -10, 0, 10},
    {10, -10, -10, -10, -10, -10, -10, 10},
    {10, -5, 0, 0, 0, 0, -5, 10},
    {20, 10, 10, 10, 10, 10, 10, 20}
};

private final int[][] queenEvalBonus = {
    {20, 10, 10, 5, 5, 10, 10, 20},
    {10, 0, 0, 0, 0, 0, 0, 10},
    {10, 0, -5, -5, -5, -5, 0, 10},
    {5, 0, -5, -5, -5, -5, 0, 5},
    {0, 0, -5, -5, -5, -5, 0, 5},
    {10, -5, -5, -5, -5, -5, 0, 10},
    {10, 0, -5, 0, 0, 0, 0, 10},
    {20, 10, 10, 5, 5, 10, 10, 20}
};

private final int[][] kingEvalBonus = {
    {30, 40, 40, 50, 50, 40, 40, 30},
    {30, 40, 40, 50, 50, 40, 40, 30},
    {30, 40, 40, 50, 50, 40, 40, 30},
    {30, 40, 40, 50, 50, 40, 40, 30},
    {20, 30, 30, 40, 40, 30, 30, 20},
    {10, 20, 20, 20, 20, 20, 20, 10},
    {-20, -20, 0, 0, 0, 0, -20, -20},
    {-20, -30, -10, 0, 0, -10, -30, -20}
};

```

I then decided to begin improving the computer move, by using the minimax algorithm to determine which move is optimal to make, rather than just picking a move at random.

To begin with, I added 6 2D arrays storing the value added for each type of piece depending on its position on the board, using the values described in the design section, to the “ChessBoard” class, so that these values correspond with the position of the piece on the board. This will allow the program to be aware of whether a piece is in a “good” or a “bad” position on the board. I made these arrays all final, since the values in them will not change while the program is running.

```

private int evaluate(Piece[][] board) {
    int eval = 0;
    int score;

    for (int i = 0; i < 8; i++) {
        for (int j = 0; j < 8; j++) {
            if (this.board[i][j] != null) {
                switch (this.board[i][j].getType()) {
                    // is a pawn
                    case 1:
                        score = 100;
                        if(this.board[i][j].getColour() == 0) {
                            score += this.pawnEvalBonus[i][j];
                        } else {
                            score -= this.pawnEvalBonus[7-i][j];
                        }
                        break;
                    // is a rook
                    case 2:
                        score = 500;
                        if(this.board[i][j].getColour() == 0) {
                            score += this.rookEvalBonus[i][j];
                        } else {
                            score -= this.rookEvalBonus[7-i][j];
                        }
                        break;
                    // is a knight
                    case 3:
                        score = 300;
                        if(this.board[i][j].getColour() == 0) {
                            score += this.knightEvalBonus[i][j];
                        } else {
                            score -= this.knightEvalBonus[7-i][j];
                        }
                        break;
                    // is a bishop
                    case 4:
                        score = 300;
                        if(this.board[i][j].getColour() == 0) {
                            score += this.bishopEvalBonus[i][j];
                        } else {
                            score -= this.bishopEvalBonus[7-i][j];
                        }
                        break;
                    // is a queen
                    case 5:
                        score = 900;
                        if(this.board[i][j].getColour() == 0) {
                            score += this.queenEvalBonus[i][j];
                        } else {
                            score -= this.queenEvalBonus[7-i][j];
                        }
                        break;
                    // is a king
                    default:
                        score = 20000;
                        if(this.board[i][j].getColour() == 0) {
                            score += this.kingEvalBonus[i][j];
                        } else {
                            score -= this.kingEvalBonus[7-i][j];
                        }
                        break;
                }

                if (this.board[i][j].getColour() == 0) { // if piece is white
                    score *= -1;
                }

                eval += score;
            }
        }
    }

    return eval;
}

```

Next, I created the evaluation function. This will be used to determine the “value” of the board in a particular position, in order to help decide which move to make, as the computer needs some sort of way of determining the “value” or “score” of a particular move.

Firstly, I declared a variable called “eval” to store the overall evaluation, and set this to 0, and declared a variable called “score” to store the value for each piece.

Then, I iterated through each location in the array, and checked if there was a piece there (i.e. that it wasn’t null). If there is a piece there, a switch statement is then used, to execute different blocks of code depending on the type of the piece.

Here, the variable “score” is set depending on the type of piece. Then, a bonus is added, depending on where the piece is on the board. The arrays storing the evaluation bonuses for each type of piece are for the white pieces – however if this is mirrored and the values multiplied by -1, you get the bonuses for the black pieces. So, here I check what the colour of the piece is so the correct bonus can be added.

Then I check whether the piece is white – if it is, I multiply the score by -1, since the overall evaluation will be the total score of the black pieces minus the total score the white pieces.

Then, I add this score to the overall evaluation of the board.

After that I began to implement the minimax algorithm. I decided to not use alpha-beta pruning at first, and to get the minimax algorithm working first, before adding this in later to improve the speed at which a move is selected if this is working correctly.

```
private String bestMove;

public int minimax(Piece[][][] board, int depth, boolean maximisingPlayer) {
    if (depth == 0 || this.currentPlayerHasNoLegalmoves()) {
        return this.evaluate(board);
    }

    if (maximisingPlayer) {
        int maxEval = Integer.MIN_VALUE;

        ArrayList<String> computerMoves = this.getAllMovesForColour(1);

        for (int i = 0; i < computerMoves.size(); i++) {
            this.makeMove(computerMoves.get(i));
            int eval = this.minimax(this.board, depth-1, false);
            if (eval > maxEval) {
                maxEval = eval;
                this.bestMove = computerMoves.get(i);
            }
            this.undoMove();
        }

        return maxEval;
    } else {
        int minEval = Integer.MAX_VALUE;

        ArrayList<String> computerMoves = this.getAllMovesForColour(0);

        for (int i = 0; i < computerMoves.size(); i++) {
            this.makeMove(computerMoves.get(i));
            int eval = this.minimax(this.board, depth-1, true);
            if (eval < minEval) {
                minEval = eval;
            }
            this.undoMove();
        }

        return minEval;
    }
}

public void makeComputerMove() {
    this.minimax(this.board, 2, true);
    this.makeMove(this.bestMove);
}
```

This is my first attempt at the minimax algorithm, following the pseudocode from earlier, in the design section. I also edited the “makeComputerMove” procedure, so instead of selecting a random move, it simply calls the minimax algorithm in order to determine the best move, and then it makes that move.

Testing

When I tested this, it appeared to work for a depth of 2, however for larger depths, it would not work, and often would not make a move at all, however no runtime errors occurred. In order to investigate this, I edited the "makeComputerMove" procedure and added a print statement, so that I could see which move (if any) was being determined by the minimax algorithm, to see if I could tell why a move was not being made.

```
public void makeComputerMove() {
    this.minimax(this.board, 2, true);
    System.out.println("Move made: " + this.bestMove);
    this.makeMove(this.bestMove);
}
```



The first move the computer makes works correctly.



However, after white has moved for a second time, the computer does not move. The print statement shows that the move that was attempted was moving a piece from location 35 to location 52 (shown in stars on the board).

Looking at this, this is the move that the black knight that moved previously would take. I realised that this meant I was setting the value of “bestMove” incorrectly – it didn’t have the value of the first move that lead to the best evaluation, but a later move. This move it finds is not valid on the original board, and so when it is attempted, it is not made, and so the computer does not move, and it remains as blacks turn to move.

Remedial

```
public String getAIBestMove(Piece[][][] board, int depth) {
    int maxEval = Integer.MIN_VALUE;
    String maxEvalMove = "";

    ArrayList<String> computerMoves = this.getAllMovesForColour(1);

    for (int i = 0; i < computerMoves.size(); i++) {
        this.makeMove(computerMoves.get(i));
        int eval = this.minimax(board, depth-1, false);
        this.undoMove();
        if (eval > maxEval) {
            maxEval = eval;
            maxEvalMove = computerMoves.get(i);
        }
    }

    return maxEvalMove;
}
```

To fix this, I created a function called “getAIBestMove” – this is where the best move will be determined – not in the minimax algorithm, which will only determine the best evaluation. It goes through each possible move the computer can make, makes each move, calls the minimax algorithm to determine the best possible evaluation from that move, and then undoes this move. Then, if this evaluation is larger than the max, it then sets both “maxEval” and “maxEvalMove”. This ensures that the best move determined is the first move that lead to the best evaluation, and not one of the resulting moves.

```

public int minimax(Piece[][] board, int depth, boolean isMaximisingPlayer) {
    if (depth == 0 || this.currentPlayerHasNoLegalmoves()) {
        return this.evaluate(board);
    }

    if (isMaximisingPlayer) {
        int maxEval = Integer.MIN_VALUE;

        ArrayList<String> computerMoves = this.getAllMovesForColour(1);

        for (int i = 0; i < computerMoves.size(); i++) {
            this.makeMove(computerMoves.get(i));
            int eval = this.minimax(this.board, depth-1, false);
            this.undoMove();
            if (eval > maxEval) {
                maxEval = eval;
            }
        }

        return maxEval;
    } else {
        int minEval = Integer.MAX_VALUE;

        ArrayList<String> computerMoves = this.getAllMovesForColour(0);

        for (int i = 0; i < computerMoves.size(); i++) {
            this.makeMove(computerMoves.get(i));
            int eval = this.minimax(this.board, depth-1, true);
            this.undoMove();
            if (eval < minEval) {
                minEval = eval;
            }
        }

        return minEval;
    }
}

```

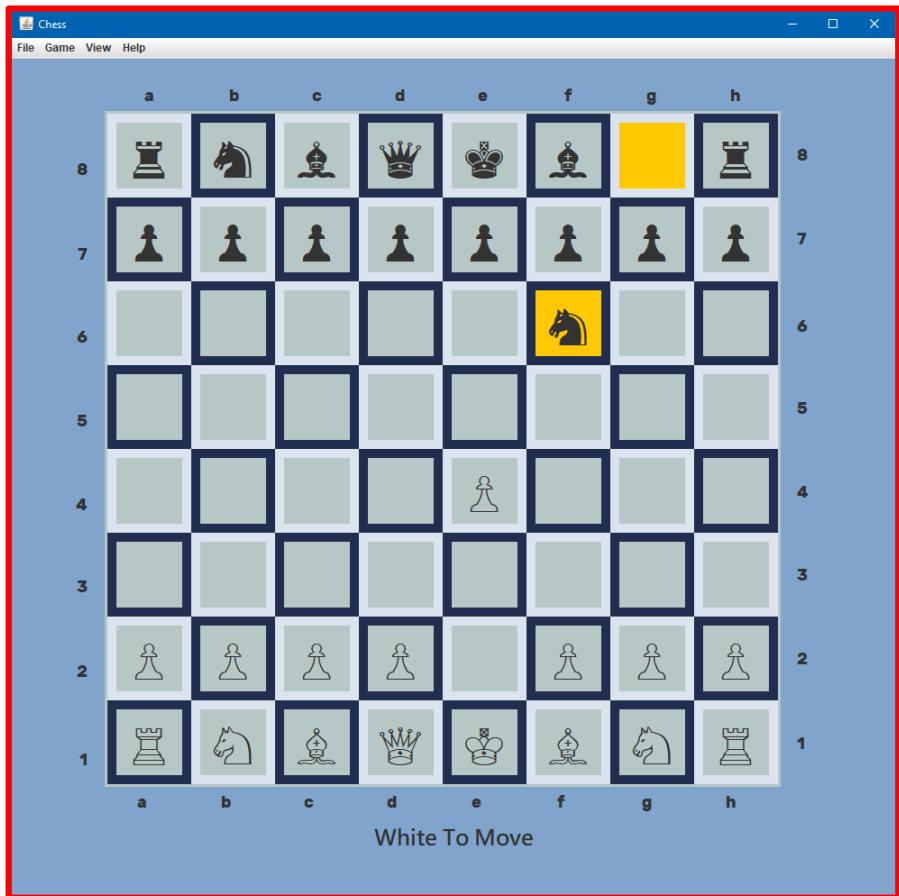
I then removed the “bestMove” variable and edited the “minimax” function so that it no longer used this – this minimax function is now only concerned with what the evaluation is, and not the moves that lead to it, and so the correct move will be returned.

```

public void makeComputerMove() {
    String move = this.getAIBestMove(this.board, 3);
    this.makeMove(move);
}

```

And then lastly, I edited the “makeComputerMove” procedure to use this new “getAIBestMove” function which should work correctly, and to no longer print the move that is made, as this was just for debugging purposes.



Now, whereas before it didn't, the computer now makes a move even when the depth is greater than 2, and so is working correctly.



However, with a depth of three, there is a relatively short but noticeable delay, where the computer is deciding on the best move. So next I decided to next try and speed this up, in order to make the program better for the user, as they will not have to wait so long between each move, which they could find annoying.

```

public int minimax(Piece[][] board, int depth, boolean isMaximisingPlayer, int alpha, int beta) {
    if (depth == 0 || this.currentPlayerHasNoLegalmoves()) {
        return this.evaluate(board);
    }

    if (isMaximisingPlayer) {
        int maxEval = Integer.MIN_VALUE;

        ArrayList<String> computerMoves = this.getAllMovesForColour(1);

        for (int i = 0; i < computerMoves.size(); i++) {
            this.makeMove(computerMoves.get(i));
            int eval = this.minimax(this.board.clone(), depth - 1, false, alpha, beta);
            if (eval > maxEval) {
                maxEval = eval;
            }
            this.undoMove();
            if (alpha < eval) {
                alpha = eval;
            }
            if (beta <= alpha) {
                break;
            }
        }

        return maxEval;
    } else {
        int minEval = Integer.MAX_VALUE;

        ArrayList<String> computerMoves = this.getAllMovesForColour(0);

        for (int i = 0; i < computerMoves.size(); i++) {
            this.makeMove(computerMoves.get(i));
            int eval = this.minimax(this.board.clone(), depth - 1, true, alpha, beta);
            if (eval < minEval) {
                minEval = eval;
            }
            this.undoMove();
            if (beta > eval) {
                beta = eval;
            }
            if (beta <= alpha) {
                break;
            }
        }

        return minEval;
    }
}

```

To do this I implemented alpha-beta pruning into the minimax algorithm, in order to minimise the number of boards that need to be evaluated, by avoiding evaluating boards that do not need to be, which should allow it to run faster, as less computation will be needed. In order to do this, I followed the pseudocode from the design section.

```

public String getAIBestMove(Piece[][][] board, int depth) {
    int maxEval = Integer.MIN_VALUE;
    String maxEvalMove = "";

    ArrayList<String> computerMoves = this.getAllMovesForColour(1);

    for (int i = 0; i < computerMoves.size(); i++) {
        this.makeMove(computerMoves.get(i));
        int eval = this.minimax(board, depth - 1, false, Integer.MIN_VALUE, Integer.MAX_VALUE);
        this.undoMove();
        if (eval > maxEval) {
            maxEval = eval;
            maxEvalMove = computerMoves.get(i);
        }
    }

    return maxEvalMove;
}

```

I then edited the “getAIBestMove” function so that it called the minimax function with the new needed parameters, alpha and beta. Alpha is the “upper limit”, and so it needs to be the smallest possible number at first so whatever the first value is, it will be guaranteed to be bigger, so it can be initially set. Same the other way round for beta – it is the lower limit and so needs to be the largest possible number initially.

When I tested this, again with the same search depth, it still found a move, but did so much more quickly, so appears to be working as expected.

Review

Now, the computer is able to successfully determine a move to make, and then actually make this move, allowing the user to play against the AI.

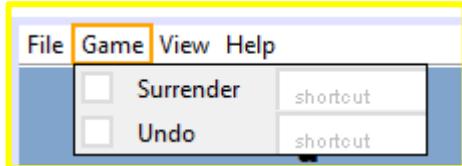
So, overall, the program can now allow the user to move, show the user the moves they can make, determine whether a move is illegal, determine whether the game should end, allow the user to select an opponent, and allow the user to play against the AI.

Link to success criteria: #9

Allowing user to undo moves

Development

```
if(this.board.getCurrentTurn() > 1) {  
    this.board.undoMove();  
    this.board.undoMove();  
    this.updateGUI();  
}
```



Then, I decided to implement the ability for the user to undo a move that they have made. In my program, this will undo both the last move made by the player's opponent, and the last move made by themselves. I added a JMenuItem to the JMenu labelled "Game", with the text "Undo". When this button is pressed, I set this code to run. This firstly checks that at least two moves have been made, and if they have, undos the most recent two moves, and then updates the GUI to show this to the user.

Testing



When just one move had been made so far in the game, nothing is undone and no error occurs, and so this part works as expected.



After two moves have been made, when the undo button is pressed, the last two moves taken in the game are undone, as expected.

Review

Now, the user is able to undo the last move that they made, which was one of the requested features of this program.

Link to success criteria: #17

Pawn promotion

Development

```
public void makeMove(String move) {
    int startPos = Integer.valueOf(move.substring(0, 2));
    int endPos = Integer.valueOf(move.substring(2, 4));

    if (this.board[startPos / 8][startPos % 8] != null && this.board[startPos / 8][startPos % 8].isValidMove(move, this.board)) {
        if (this.board[endPos / 8][endPos % 8] == null) {
            move += "0";
        } else {
            move += this.board[endPos / 8][endPos % 8].getType();
        }

        int colour = this.board[startPos / 8][startPos % 8].getColour();
        boolean pawnPromotion = this.board[startPos / 8][startPos % 8].getType() == 1
            && (endPos / 8 == 0 && colour == 0 || endPos / 8 == 7 && colour == 1);

        if (pawnPromotion) {
            move += "p";
        }

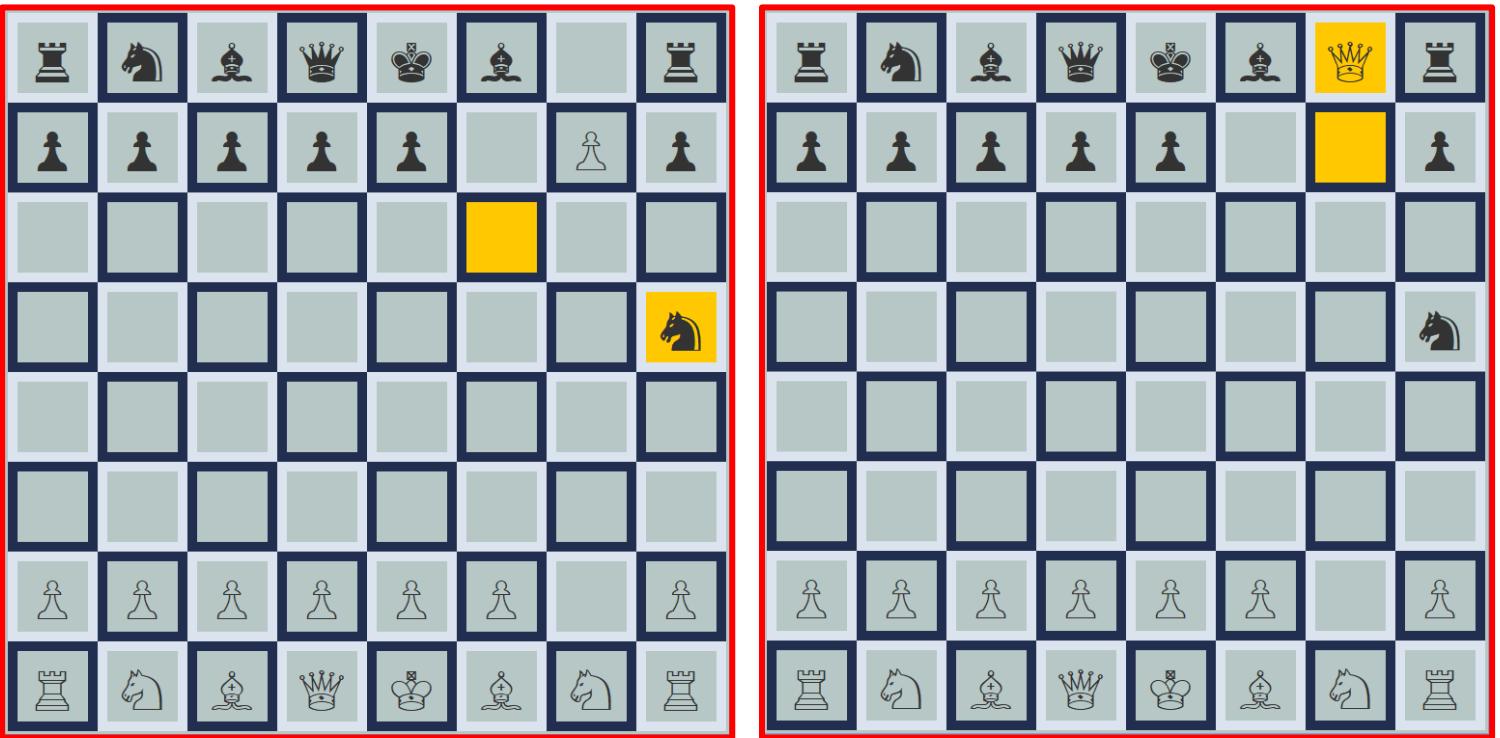
        this.moveHistory.add(move);
        this.currentTurn++;

        if (pawnPromotion) {
            this.board[endPos / 8][endPos % 8] = new Queen(colour);
        } else {
            this.board[endPos / 8][endPos % 8] = this.board[startPos / 8][startPos % 8];
        }
        this.board[startPos / 8][startPos % 8] = null;

        if (this.isInCheck((this.currentTurn - 1) % 2)) {
            this.undoMove();
        }
    }
}
```

I then began to add pawn promotion – which is when a pawn can be promoted to another type of piece when it reaches the opposite side of the board. I edited the “makeMove” procedure to check if the move involves a pawn reaching the opposite side of the board, and if it does, it adds a “p” to the end of the move string to keep track of this, and instead of placing a pawn at the end position, placing a queen. Whether a move involved pawn promotion or not needs to be stored as when this move is undone, the type of the piece will need to be changed from a queen back to a pawn.

Testing



When I tested this, it appeared to work as expected at first – when the pawn reached the other side of the board, it became a queen.



However, when I pressed the button to undo a move, the piece should turn back into a pawn, but it remained a queen. In order to fix this, I need to edit the “undoMove” function so that it can recognise when a move was a pawn promotion, and if it was, not only move the piece back to its original spot, but also change the type of piece back.

Remedial

```
public void undoMove() {
    if (this.moveHistory.getSize() > 0) {
        String move = this.moveHistory.pop();
        this.currentTurn--;

        int startPos = Integer.valueOf(move.substring(0, 2));
        int endPos = Integer.valueOf(move.substring(2, 4));
        int typeOfCapturedPiece = Integer.valueOf(move.substring(4, 5));
        Piece piece = null;

        switch (typeOfCapturedPiece) {
            case 1:
                piece = new Pawn((this.currentTurn - 1) % 2);
                break;
            case 2:
                piece = new Rook((this.currentTurn - 1) % 2);
                break;
            case 3:
                piece = new Knight((this.currentTurn - 1) % 2);
                break;
            case 4:
                piece = new Bishop((this.currentTurn - 1) % 2);
                break;
            case 5:
                piece = new Queen((this.currentTurn - 1) % 2);
                break;
            default:
                break;
        }

        if (move.length() > 5 && move.charAt(5) == 'p') {
            this.board[startPos / 8][startPos % 8] = new Pawn(this.currentTurn % 2);
        } else {
            this.board[startPos / 8][startPos % 8] = this.board[endPos / 8][endPos % 8];
        }

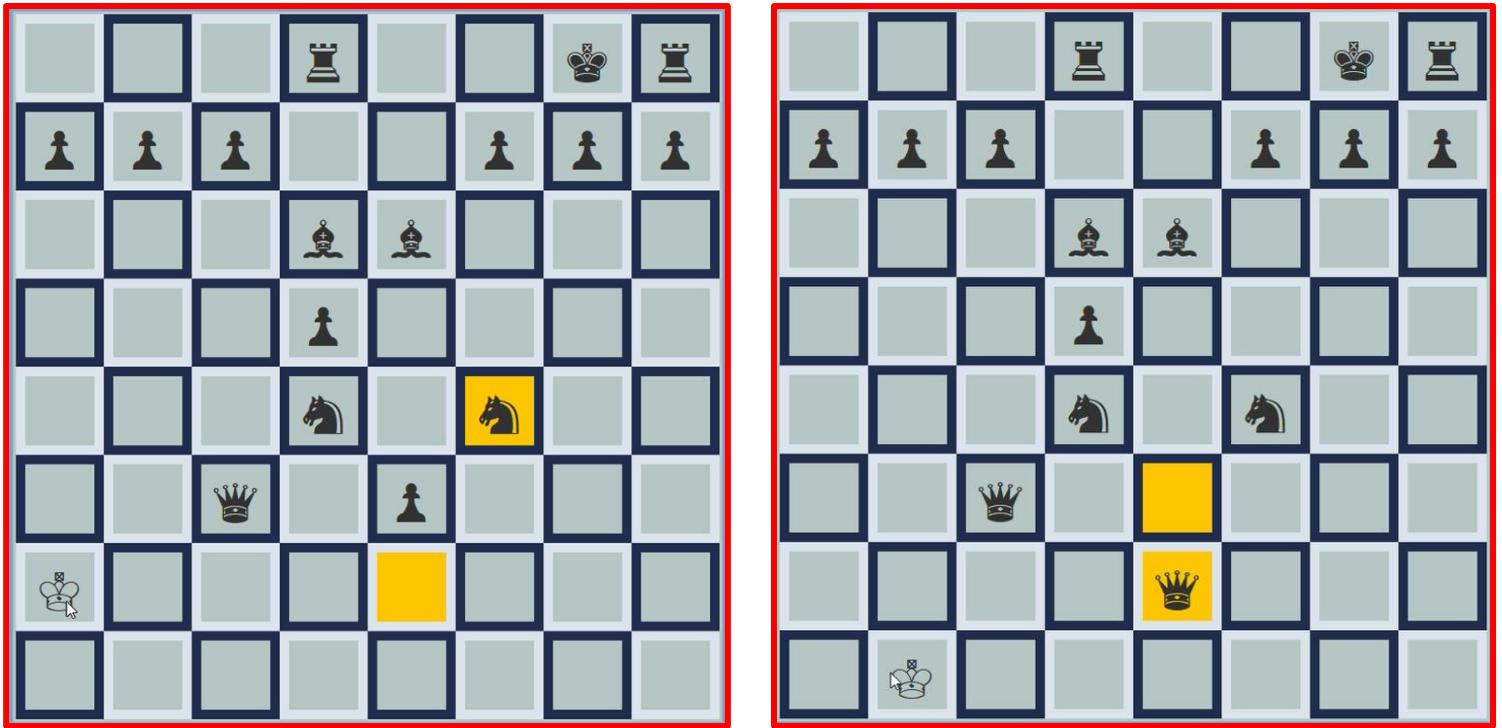
        this.board[endPos / 8][endPos % 8] = piece;
    }
}
```

At the end of the “undoMove” procedure, I check whether the move string is of a great enough length, and contains a “p” in the right place, indicating that the move was a pawn promotion. If it was, I set the piece at the start position of the move to a pawn, otherwise I set it to whatever piece was at the end position. If I didn’t check this, when the move was undone, it would be set to the piece that was at the end position - a queen, when it should be turned back into a pawn.



When I tested this, the move is now undone correctly, converting it back into a pawn.

Testing



When I tested this against a human player, it worked as expected, however when playing against the AI, pawns would be promoted even though they hadn't reached the other side of the board (specifically, when the black pawns reached the 2nd rank). I think this is due to the pawn promotion not being undone correctly when the AI looks ahead at the moves it can make.

Remedial

```
// Move without promoting pawn
private void makeTempMove(String move) {
    int startPos = Integer.valueOf(move.substring(0, 2));
    int endPos = Integer.valueOf(move.substring(2, 4));

    if (this.board[startPos / 8][startPos % 8] != null && this.board[startPos / 8][startPos % 8].isValidMove(move, this.board)) {
        if (this.board[endPos / 8][endPos % 8] == null) {
            move += "0";
        } else {
            move += this.board[endPos / 8][endPos % 8].getType();
        }
    }

    this.moveHistory.add(move);
    this.currentTurn++;

    this.board[endPos / 8][endPos % 8] = this.board[startPos / 8][startPos % 8];
    this.board[startPos / 8][startPos % 8] = null;

    if (this.isInCheck((this.currentTurn - 1) % 2)) {
        this.undoMove();
    }
}
```

To try and fix this I created a procedure called “makeTempMove” which will only be used when looking ahead at moves to make in the minimax algorithm. This works in a very similar way to the “makeMove” procedure, only it does not involve pawn promotion.

```

if (isMaximisingPlayer) {
    int maxEval = Integer.MIN_VALUE;

    ArrayList<String> computerMoves = this.getAllMovesForColour(1);

    for (int i = 0; i < computerMoves.size(); i++) {
        //this.makeMove(computerMoves.get(i));
        this.makeTempMove(computerMoves.get(i));
    }
}

```

```

} else {
    int minEval = Integer.MAX_VALUE;

    ArrayList<String> computerMoves = this.getAllMovesForColour(0);

    for (int i = 0; i < computerMoves.size(); i++) {
        //this.makeMove(computerMoves.get(i));
        this.makeTempMove(computerMoves.get(i));
    }
}

```

```

private String getAIBestMove(Piece[][] board, int depth) {
    int maxEval = Integer.MIN_VALUE;
    String maxEvalMove = "";

    ArrayList<String> computerMoves = this.getAllMovesForColour(1);

    for (int i = 0; i < computerMoves.size(); i++) {
        //this.makeMove(computerMoves.get(i));
        this.makeTempMove(computerMoves.get(i));
    }
}

```

I then edited the “minimax” and “getBestAIMove” functions to use this “makeTempMove” procedure instead.



Now, using this “makeTempMove” procedure, when I tested the program, the pawn is no longer promoted too early, and is still promoted correctly when it reaches the 1st rank.

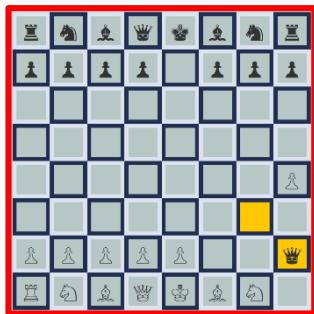
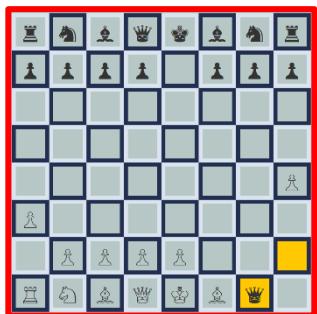
```

private final int[][] pawnEvalBonus = {
    {-900, -900, -900, -900, -900, -900, -900, -900},
    {-50, -50, -50, -50, -50, -50, -50, -50},
    {-10, -10, -20, -30, -30, -20, -10, -10},
    {-10, -10, -20, -30, -30, -20, -10, -10},
    {-5, -5, -10, -25, -25, -10, -5, -5},
    {0, 0, 0, -20, -20, 0, 0, 0},
    {-5, 5, 10, 0, 0, 10, 5, -5},
    {-5, -10, -10, 20, 20, -10, -10, -5},
    {0, 0, 0, 0, 0, 0, 0, 0}
};

```

However, now that the minimax algorithm is not aware of pawn promotion, it does not know of the improvement to the evaluation of a pawn reaching the opposite side of the board, and so has no incentive to promote the pawn. Therefore, I edited the “pawnEvalBonus” so that a pawn receives a large bonus (900, the value of a queen) for reaching the opposite side, to encourage it to promote pawns.

Testing

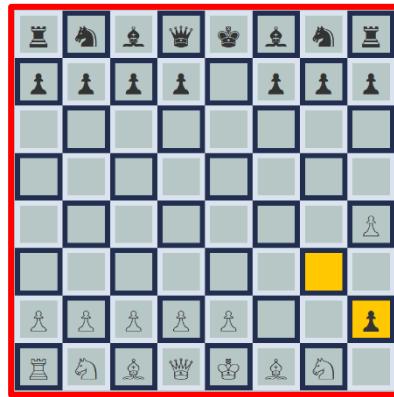


But when I was testing this, although the pawn promotes correctly, when the move is undone, it remains a queen. I think this is because the move is not correctly being recognised as a pawn promotion (i.e. there is no “p” at the end of the move string), as undoing pawn promotion works correctly with human players, but not with the AI.

Remedial

```
public void makeComputerMove() {  
    String move = this.getAIBestMove(this.board, 3);  
  
    int startPos = Integer.valueOf(move.substring(0, 2));  
    int endPos = Integer.valueOf(move.substring(2, 4));  
    int colour = this.board[startPos / 8][startPos % 8].getColour();  
    boolean pawnPromotion = this.board[startPos / 8][startPos % 8].getType() == 1  
        && ((endPos / 8 == 0 && colour == 0) || (endPos / 8 == 7 && colour == 1));  
  
    if (pawnPromotion) {  
        move += "p";  
    }  
  
    this.makeMove(move);  
}
```

So, to try and fix this, I determined whether the move was a pawn promotion, and if it was, added a “p” to the end of the string, to ensure it is correctly recognised as a pawn promotion.



Now when I try this, the queen is changed back to a pawn, and so this is now working correctly.

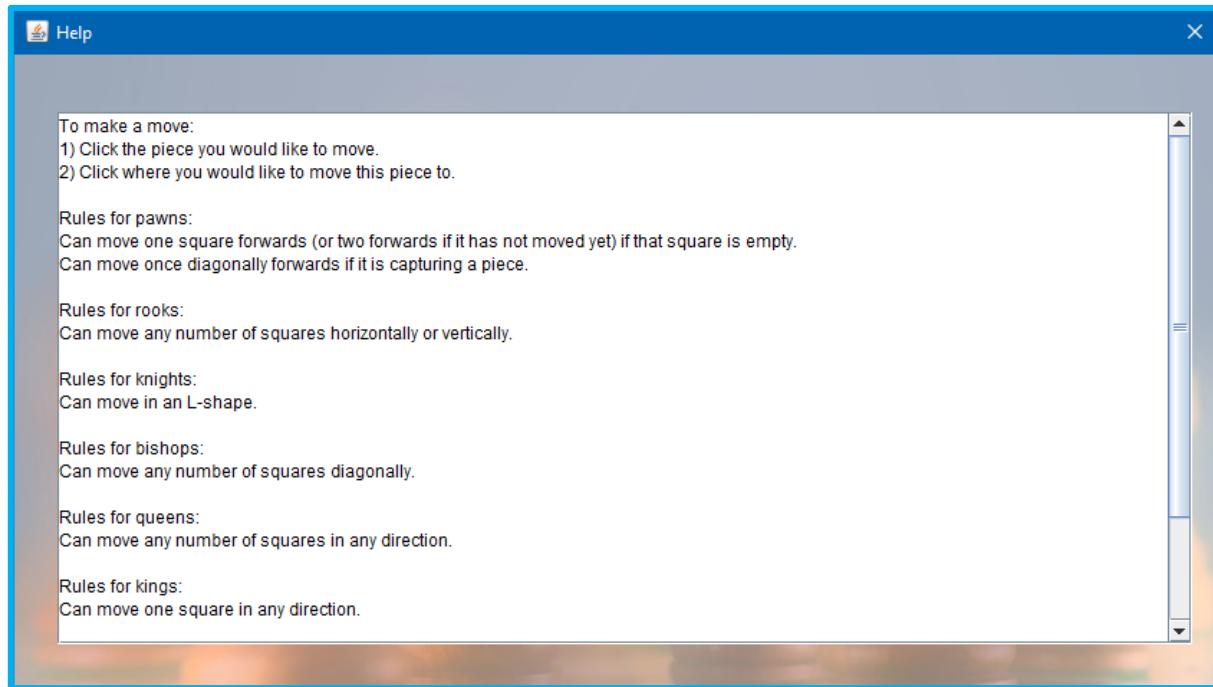
Review

Now, when a pawn reaches the other side of the board, it is promoted, and moves involving pawn promotion are undone correctly. This resulted in some changes being needed in the undo move method and the method for making a computer move in order for pawn promotion to work correctly, and for these other features to work correctly.

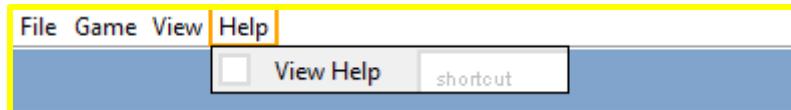
Link to success criteria: #16, #9

Showing user instructions

Development



I then created a “Help” JDialog. I added a JLabel and set it to display an icon to act as a background image. I then added a JTextArea, and set the text of this to be an explanation of how to play the game. I then set it so that this text cannot be edited by the user, by setting it to be disabled.



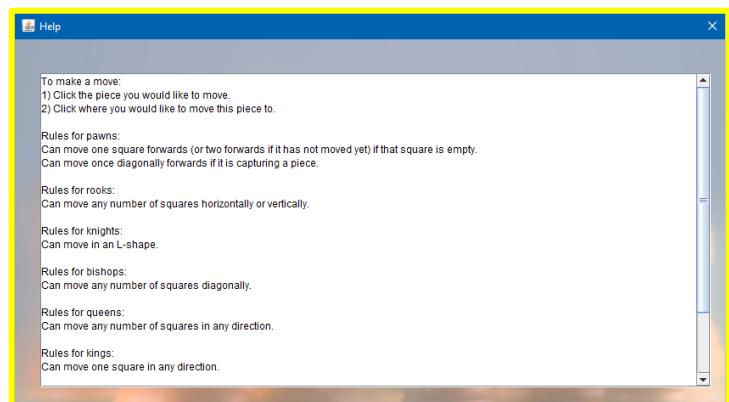
I then added a JMenuItem to the “Help” JMenu, so that the user will be able to press this to see the “Help” JDialog.

```
private void jMenuItem4ActionPerformed(java.awt.event.ActionEvent evt) {  
    // TODO add your handling code here:  
    Help helpDialog = new Help(new javax.swing.JFrame(), false);  
    helpDialog.setVisible(true);  
}
```

I then added the code which will actually display this JDialog to the user when they press the button.

This creates a new Help JDialog, and then sets it to visible. I decided to pass in false to the constructor of the JDialog (indicating that it is not modal), as this help dialog does not need to take priority / prevent input to the other windows in the program whilst it is open. This also allows the user to have the help window open at the same time as they are playing the game, which could be useful to them.

Testing



When tested, when this JMenuItem is pressed, this help dialog appears as expected.

Review

Now, the user is able to view a window containing instructions on how to use the program & the rules of the game.

Now, the program is functioning correctly, and a number of usability features have been implemented, making the program easier to use.

Link to success criteria: #20

Changing colours of GUI

Development

When I surveyed the end users, there was some variety in the colours they said they would like to be in the program. I decided to use blue as the main default colour scheme, as this was one of the most popular colours, however I have decided to implement the ability to change the colour of various elements in the GUI, so that the user can choose different colours if the default ones do not appeal to them, allowing the program to appeal to a wider range of users.

```
private Color defaultColour;
private Color pieceSelectionColour;
private Color validPositionColour;
private Color noAvailableMovesColour;
private Color previousMoveColour;
```

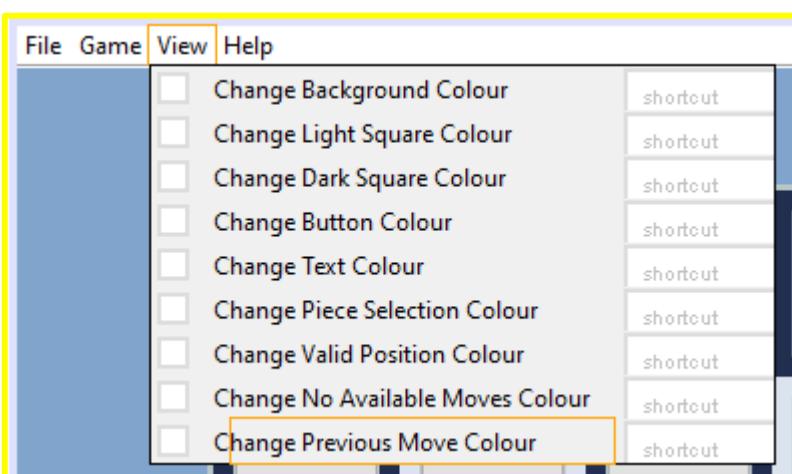
I already had a variable to store the background colour of the buttons ("defaultColour"), but I then added other variables to store the colours that would be used to show the piece which has been selected, valid positions to move a piece to, that a piece has no legal moves, and the previous move made, so that these can be changed by the user if they would like to.

```
this.defaultColour = new Color(182,199,198); // Grey
this.pieceSelectionColour = Color.CYAN;
this.validPositionColour = new Color(244, 255, 82); // Yellow
this.noAvailableMovesColour = new Color(255, 0, 53); // Red
this.previousMoveColour = Color.ORANGE;
```

I then set these variables to their default values in the constructor.

```
this.buttons[i][j].setBackground(this.defaultColour);
this.buttons[pos / 8][pos % 8].setBackground(this.pieceSelectionColour);
this.buttons[i][j].setBackground(this.validPositionColour);
this.buttons[pos / 8][pos % 8].setBackground(this.noAvailableMovesColour);
this.buttons[startPos / 8][startPos % 8].setBackground(this.previousMoveColour);
this.buttons[endPos / 8][endPos % 8].setBackground(this.previousMoveColour);
```

I then changed the various subroutines that used these colours to use these variables instead, so that if the user wants to change the colour of particular things in the GUI, they will be able to, as it will not be hard-coded.



I then added a number of JMenuItem's to the "View" JMenu, which will be used to allow the user to specify what they would like to change the colour of.

```

private void jMenuItem9ActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
    // Change button colour:
    Color newColour = JColorChooser.showDialog(this, "Select Button Colour", new Color(182,199,198));
    if (newColour != null) {
        this.defaultColour = newColour;
        this.updateGUI();
    }
}

```

```

private void jMenuItem11ActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
    // Change piece selection colour:
    Color newColour = JColorChooser.showDialog(this, "Select Piece Selection Colour", Color.CYAN);
    if (newColour != null) {
        this.pieceSelectionColour = newColour;
    }
}

```

```

private void jMenuItem12ActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
    // Change valid position colour:
    Color newColour = JColorChooser.showDialog(this, "Select Valid Position Colour", new Color(244, 255, 82));
    if (newColour != null) {
        this.validPositionColour = newColour;
    }
}

```

```

private void jMenuItem13ActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
    Color newColour = JColorChooser.showDialog(this, "Select No Available Moves Colour", new Color(255, 0, 53));
    if (newColour != null) {
        this.noAvailableMovesColour = newColour;
    }
}

```

```

private void jMenuItem14ActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
    // Change previous move colour:
    Color newColour = JColorChooser.showDialog(this, "Select Previous Move Colour", Color.ORANGE);
    if (newColour != null) {
        this.previousMoveColour = newColour;
    }
}

```

I then added code to determine what should happen when the JMenuItems labelled “Change Button Colour”, “Change Piece Selection Colour”, “Change Valid Position Colour”, “Change No Available Moves Colour” and “Change Previous Move” colour are pressed. They all perform in a very similar way – only each modifies the value of a different variable. The parameters of the “showDialog” method are the parent for the JDialog (which is “this” – the main JFrame), a string for the title of the JDialog, and a “Color”, representing the default colour.

```

private void jMenuItem5ActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
    Color newColour = JColorChooser.showDialog(this, "Select Background Colour", new Color(129,164,205));
    if (newColour != null) {
        this.jPanel1.setBackground(newColour);
    }
}

```

I then added code to determine what should happen when the JMenuItem labelled “Change Background Colour” is pressed. This gets the user to select a colour, and then changes the colour of the JPanel to that colour.

```

private void jMenuItem7ActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
    // Change light square colour:
    Color newColour = JColorChooser.showDialog(this, "Select Light Square Colour", new Color(219, 228, 238));
    if (newColour != null) {
        for (int i = 0; i < 64; i++) {
            if ((i / 8) % 2 == 0 && (i % 8) % 2 == 0 || (i / 8) % 2 != 0 && (i % 8) % 2 != 0) {
                this.buttons[i / 8][i % 8].setBorder(javax.swing.BorderFactory.createLineBorder(newColour, 10));
            }
        }
    }
}

private void jMenuItem8ActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
    // Change dark square colour:
    Color newColour = JColorChooser.showDialog(this, "Select Dark Square Colour", new Color(34, 46, 80));
    if (newColour != null) {
        for (int i = 0; i < 64; i++) {
            if ((i / 8) % 2 == 0 && (i % 8) % 2 != 0 || (i / 8) % 2 != 0 && (i % 8) % 2 == 0) {
                this.buttons[i / 8][i % 8].setBorder(javax.swing.BorderFactory.createLineBorder(newColour, 10));
            }
        }
    }
}

```

I then added code to determine what should happen when the JMenuItem labelled “Change Dark Square Colour” and “Change Light Square Colour” are pressed. This asks the user to select a colour, and then iterates through the array of buttons, changing the appropriate ones (according to the chequered pattern of the board) so that their border is a different colour.

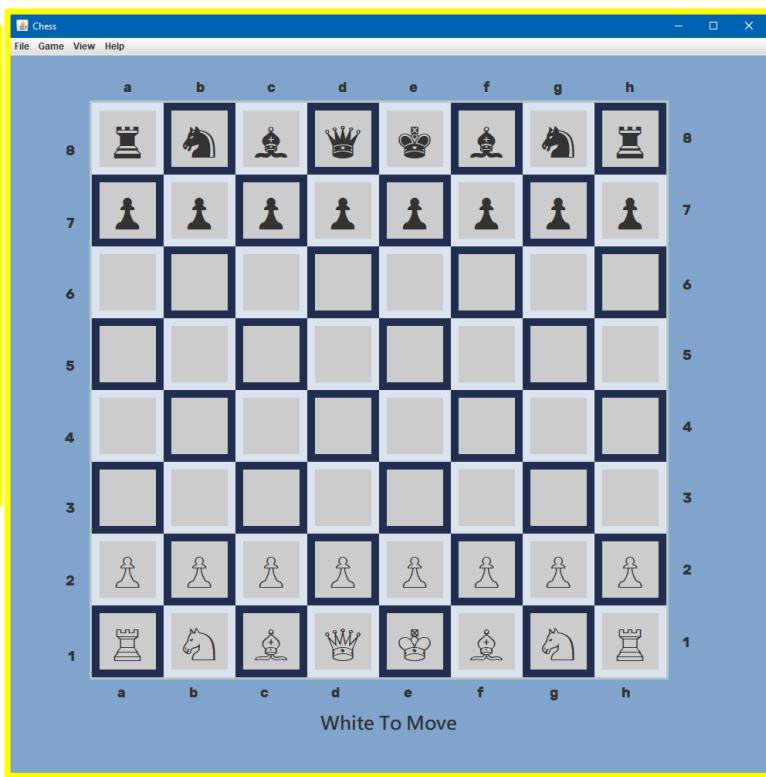
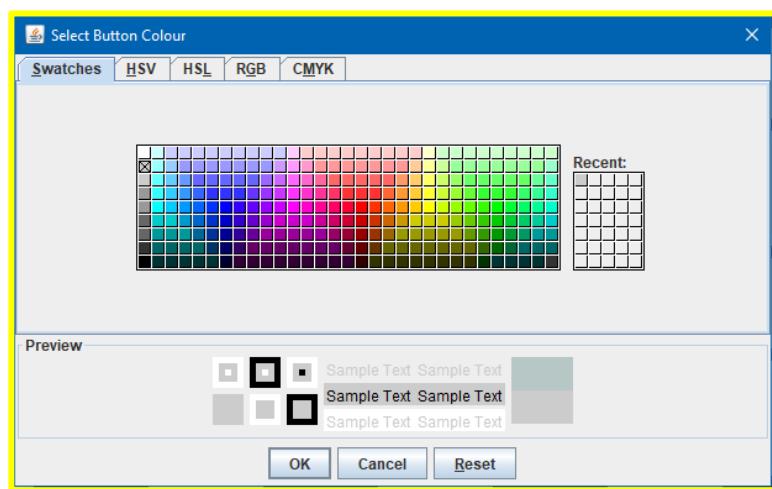
```

private void jMenuItem10ActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
    // Change text colour:
    Color newColour = JColorChooser.showDialog(this, "Select Text Colour", Color.BLACK);
    if (newColour != null) {
        this.jLabel1.setForeground(newColour);
        this.jLabel2.setForeground(newColour);
        this.jLabel3.setForeground(newColour);
        this.jLabel4.setForeground(newColour);
        this.jLabel5.setForeground(newColour);
        this.jLabel6.setForeground(newColour);
        this.jLabel7.setForeground(newColour);
        this.jLabel8.setForeground(newColour);
        this.jLabel9.setForeground(newColour);
        this.jLabel10.setForeground(newColour);
        this.jLabel11.setForeground(newColour);
        this.jLabel12.setForeground(newColour);
        this.jLabel13.setForeground(newColour);
        this.jLabel14.setForeground(newColour);
        this.jLabel15.setForeground(newColour);
        this.jLabel16.setForeground(newColour);
        this.jLabel17.setForeground(newColour);
        this.jLabel18.setForeground(newColour);
        this.jLabel19.setForeground(newColour);
        this.jLabel20.setForeground(newColour);
        this.jLabel21.setForeground(newColour);
        this.jLabel22.setForeground(newColour);
        this.jLabel23.setForeground(newColour);
        this.jLabel24.setForeground(newColour);
        this.jLabel25.setForeground(newColour);
        this.jLabel26.setForeground(newColour);
        this.jLabel27.setForeground(newColour);
        this.jLabel28.setForeground(newColour);
        this.jLabel29.setForeground(newColour);
        this.jLabel30.setForeground(newColour);
        this.jLabel31.setForeground(newColour);
        this.jLabel32.setForeground(newColour);
        this.checkLabel.setForeground(newColour);
        this.turnLabel.setForeground(newColour);
    }
}

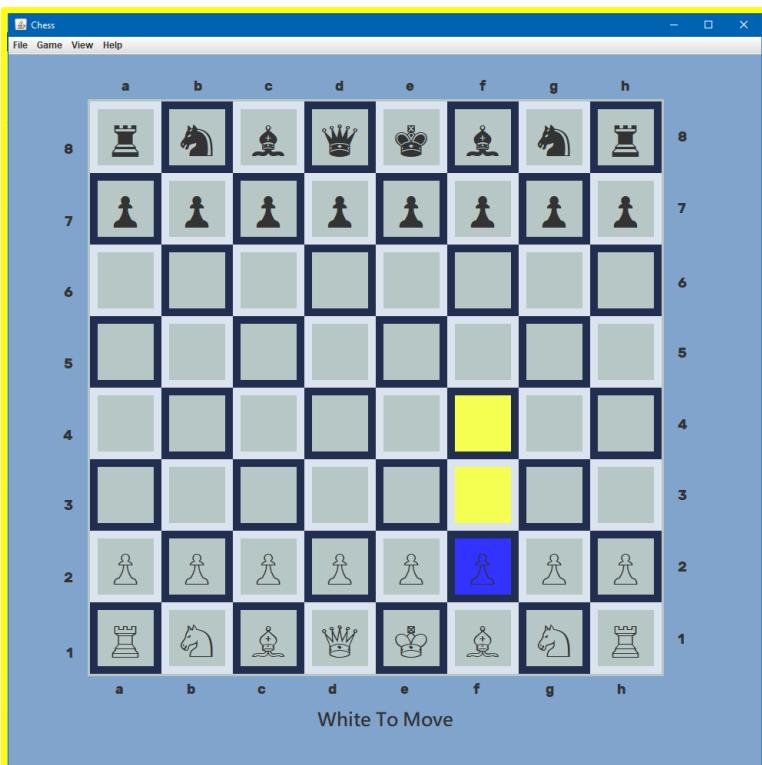
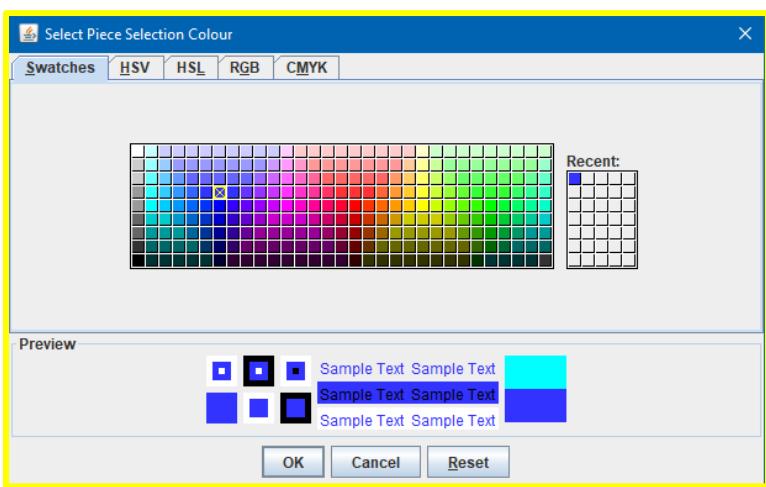
```

I then added code to determine what should happen when the JMenuItem labelled “Change Text Colour” is pressed. This gets the user to select a colour, and then changes the colour of all of the JLabels to that colour.

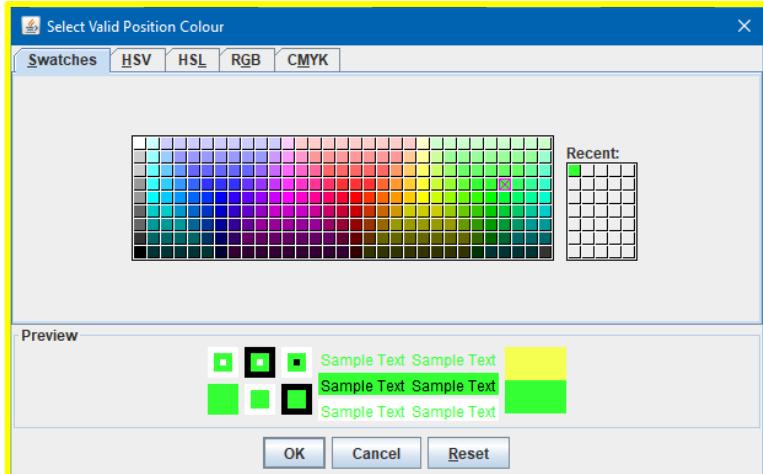
Testing



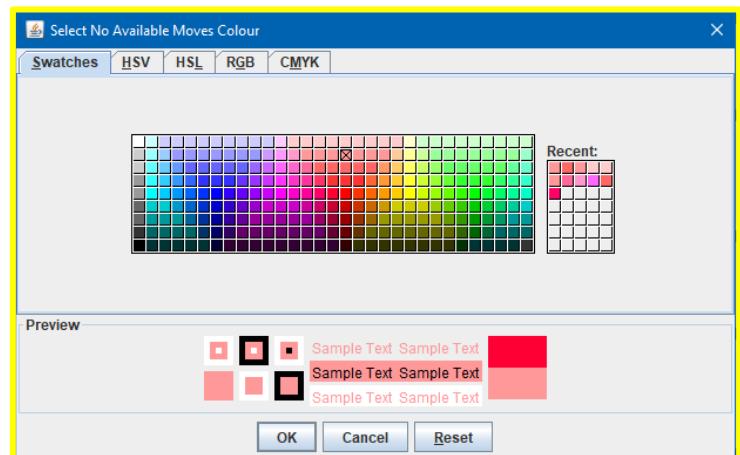
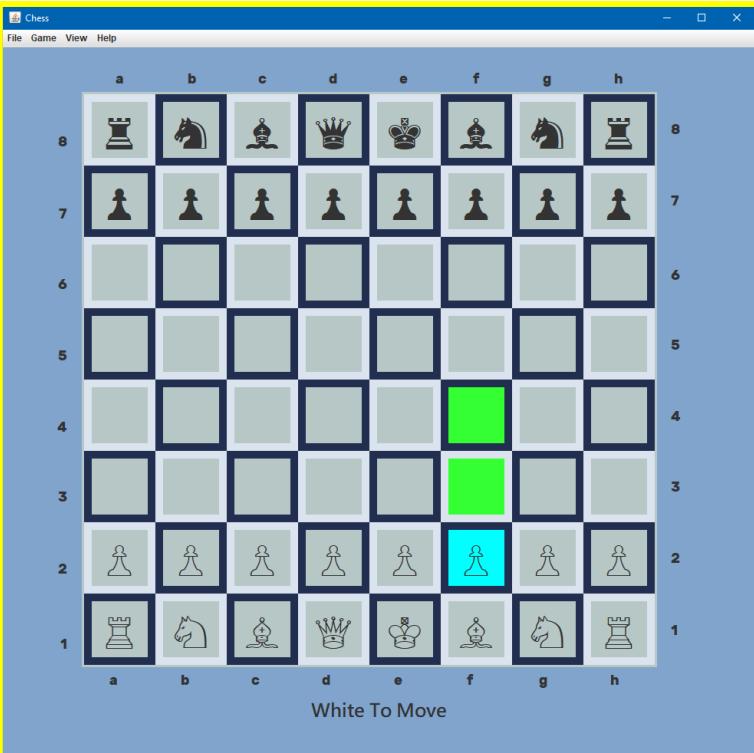
When the “Change Button Colour” JMenuItem is pressed, the user is shown a dialog with the correct title to select a colour, and when they press “OK”, the colour of the buttons is changed to the selected colour, so works as expected.



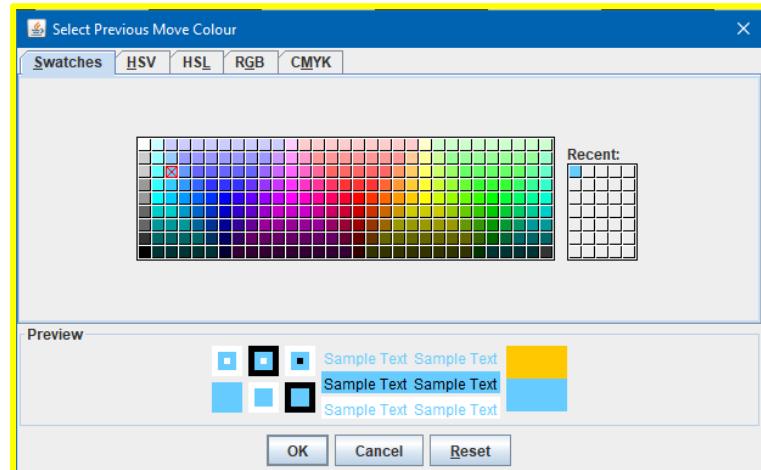
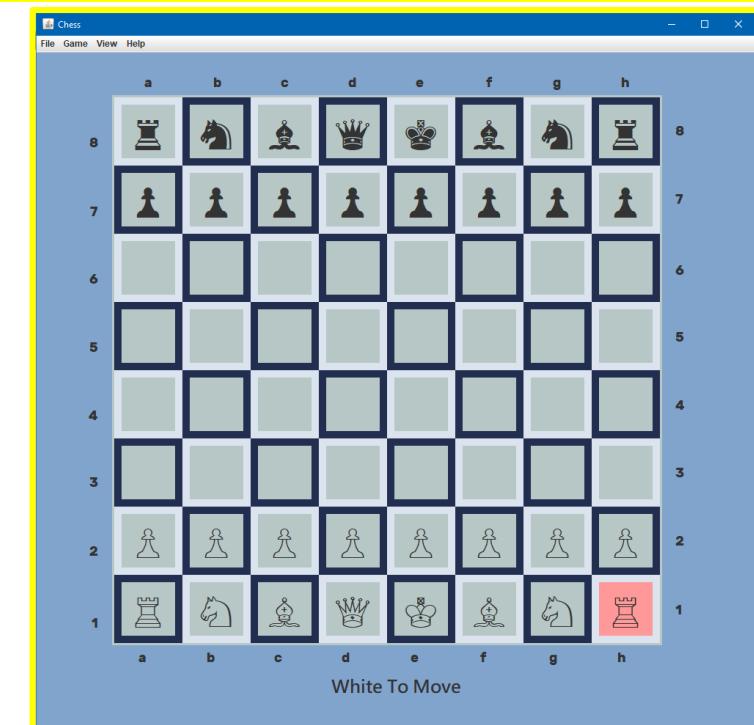
When the “Change Piece Selection Colour” JMenuItem is pressed, the user is shown a dialog with the correct title to select a colour, and when they press “OK”, the colour that is shown to the user when they select a piece is changed, so it is working as expected.



When the “Change Valid Position Colour” JMenuItem is pressed, the user is shown a dialog with the correct title to select a colour, and when they press “OK”, the colour that is shown to the user when they select a piece and are shown the valid moves it can make is changed, so it is working as expected.

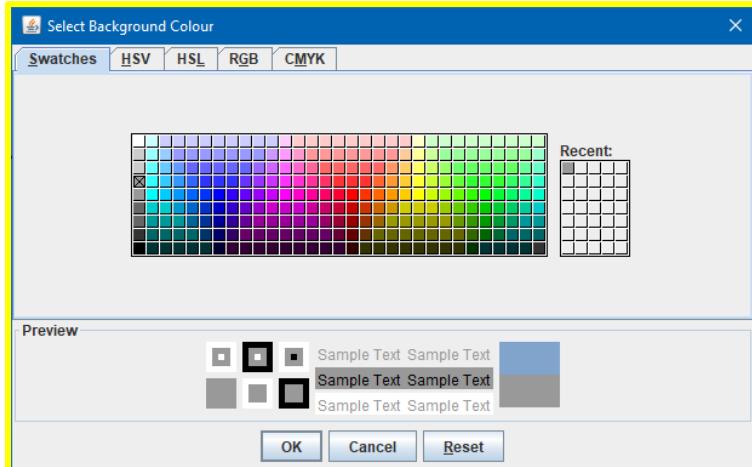


When the “Change No Available Moves Colour” JMenuItem is pressed, the user is shown a dialog with the correct title to select a colour, and when they press “OK”, the colour that is shown to the user when they select a piece with no available legal moves is changed, so it is working as expected.

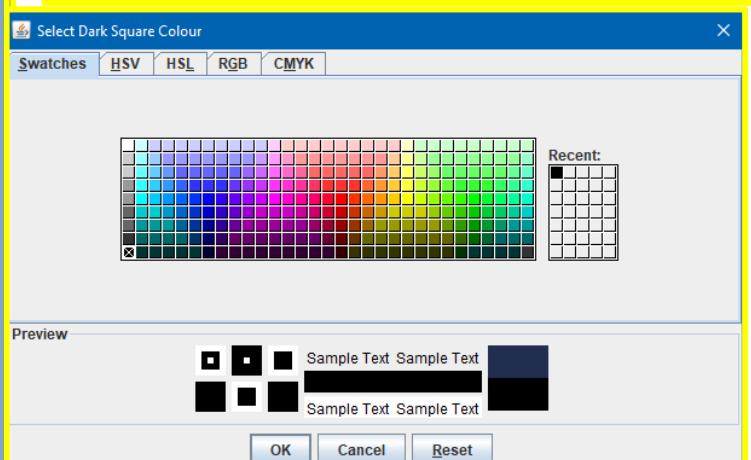
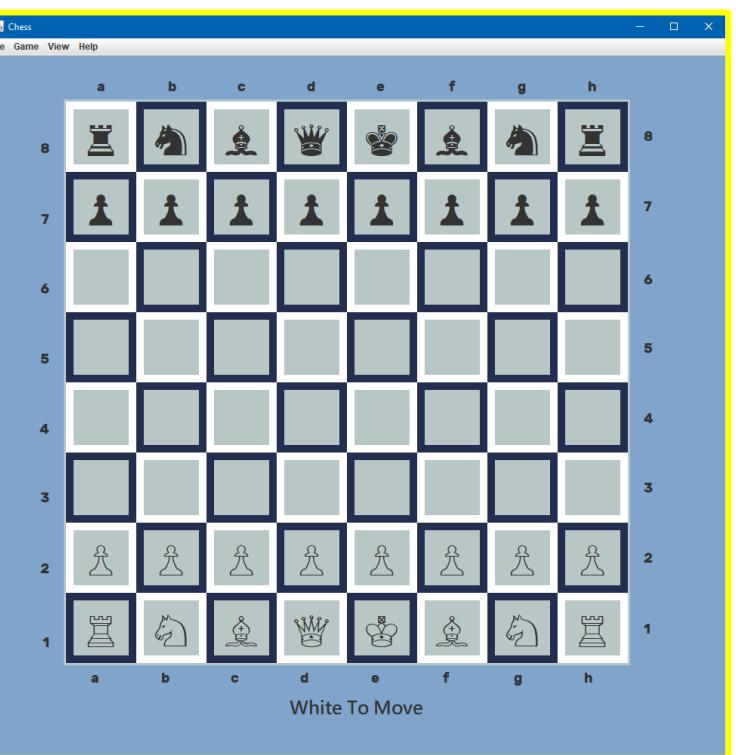
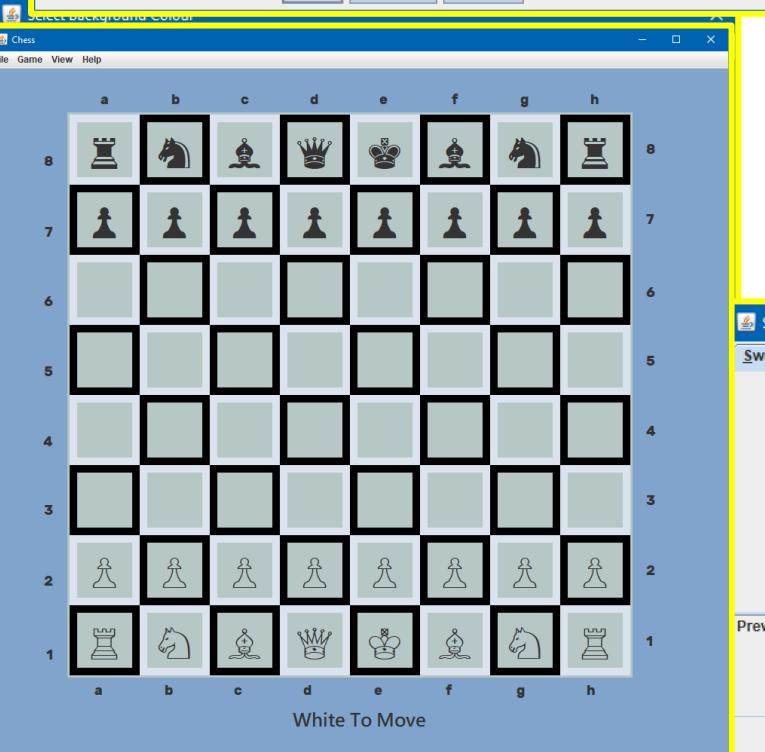
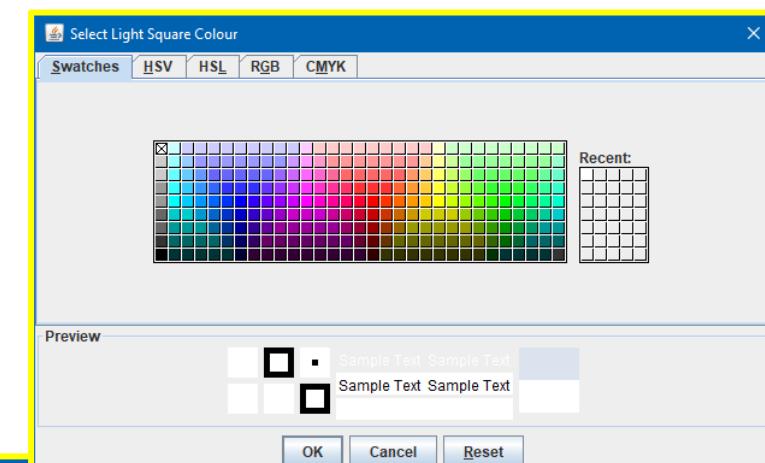
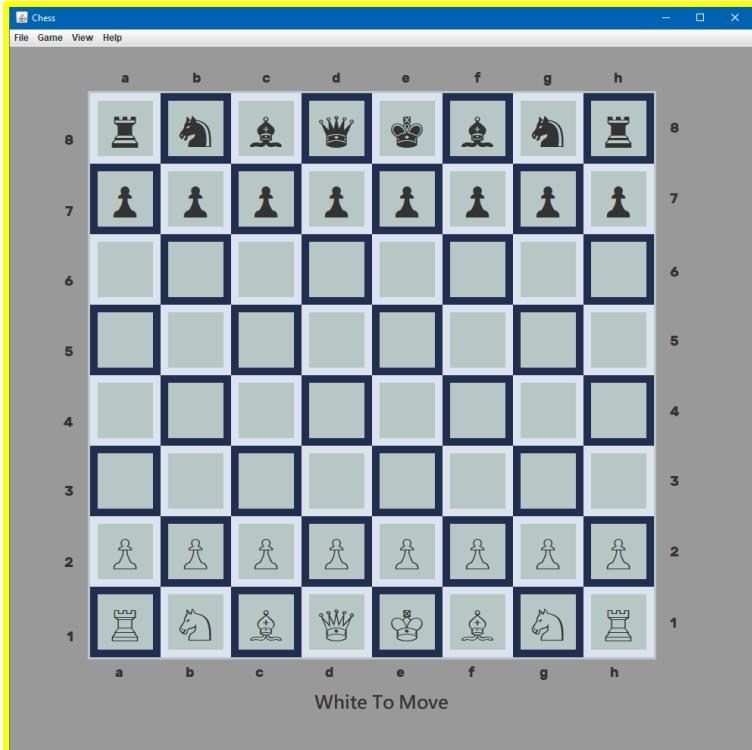


When the “Change Previous Move Colour” JMenuItem is pressed, the user is shown a dialog with the correct title to select a colour, and when they press “OK”, the colour that is shown to the user when they are shown the previous move made is changed, so it is working as expected.

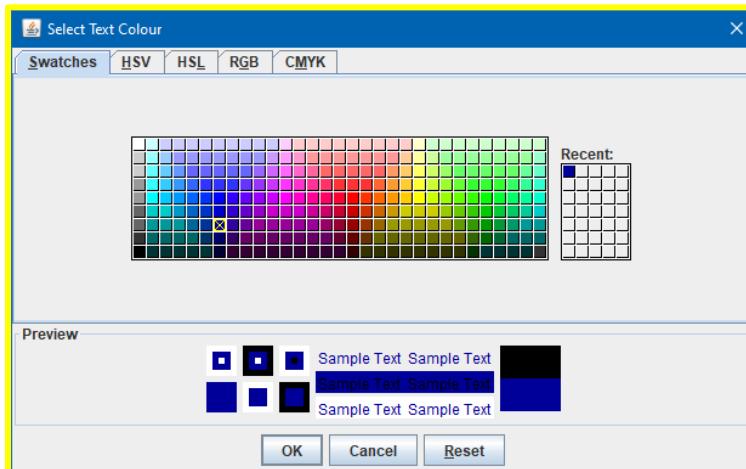




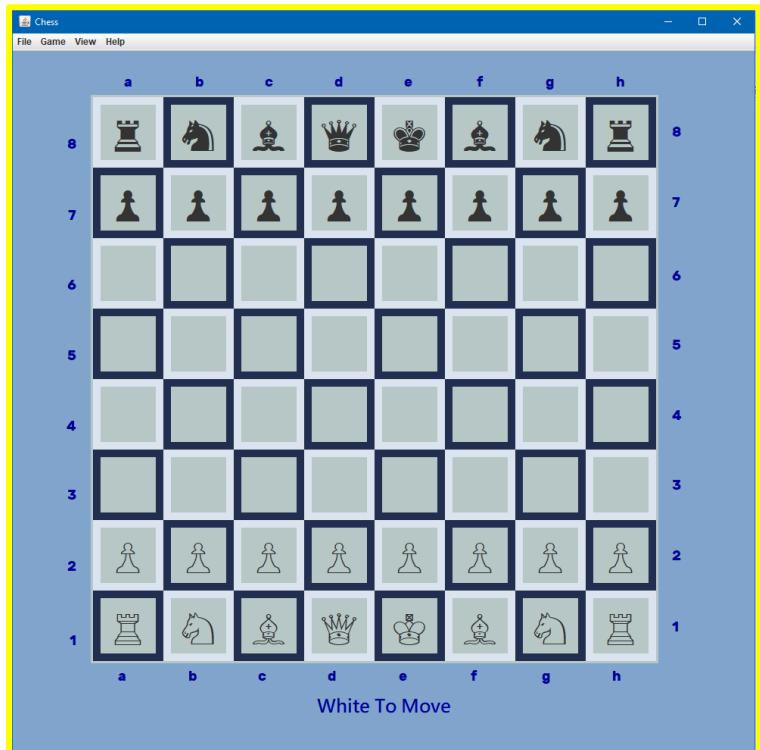
When the “Change Background Colour” JMenuItem is pressed, the user is shown a dialog with the correct title to select a colour, and when they press “OK”, the colour of the JPanel is set accordingly, and so this works as expected.



When the “Change Light/Dark Square Colour” JMenuItem is pressed, the user is shown a dialog with the correct title to select a colour, and when they press “OK”, the colour of the border of the correct buttons is changed correctly, so is working as expected.



When the “Select Text Colour” JMenuItem is pressed, the user is shown a dialog with the correct title to select a colour, and when they press “OK”, the colour of all of the JLabels is set accordingly, and none are missing, and so this works as expected.



Review

Whilst I had not originally planned to include this, now, the user is able to customise the appearance of the program to suit them, making it appeal to a wider range of people.

Link to success criteria: none

Saving games

Development

```
public String getGameString() {
    String game = "";

    int blank; // Used to count number of blank spaces in a row

    for (int i = 0; i < 8; i++) {
        blank = 0;
        for (int j = 0; j < 8; j++) {
            if (this.board[i][j] == null) {
                blank++;
            } else {
                if (blank != 0) {
                    game += blank;
                    blank = 0;
                }
                switch (this.board[i][j].getType()) {
                    // Capital = white, lower-case = black
                    // is a pawn
                    case 1:
                        game += this.board[i][j].getColour() == 0 ? "P" : "p";
                        break;
                    // is a rook
                    case 2:
                        game += this.board[i][j].getColour() == 0 ? "R" : "r";
                        break;
                    // is a knight
                    case 3:
                        game += this.board[i][j].getColour() == 0 ? "N" : "n";
                        break;
                    // is a bishop
                    case 4:
                        game += this.board[i][j].getColour() == 0 ? "B" : "b";
                        break;
                    // is a queen
                    case 5:
                        game += this.board[i][j].getColour() == 0 ? "Q" : "q";
                        break;
                    // is a king
                    case 6:
                        game += this.board[i][j].getColour() == 0 ? "K" : "k";
                        break;
                }
            }
        }
        if (blank != 0) {
            game += blank;
        }
        if (i != 7) {
            game += "/";
        }
    }

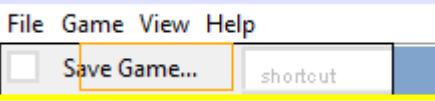
    game += this.currentTurn % 2 == 0 ? " w" : " b"; // If current turn is even - i.e. white, append " w"

    return game;
}
```

I then decided to add in the functionality for the user to load and save a game. Firstly, I created this “getGameString” function, which will return a string representing the layout of the board, so that this can be later written to a file. “p” = pawn, “r” = rook, “n” = knight, “b” = bishop, “q” = queen, “k” = king, uppercase = white, lowercase = black. A number represents that number of blank spaces, and a “/” indicates the end of the row. This function iterates through the array of Pieces and checks the type of Piece (if any) that is there, and adds to the string accordingly.

After this, it then adds either a “w” or a “b” to the end of the string, depending on whether it is white’s turn or black’s turn to move.

And then finally this string is returned.

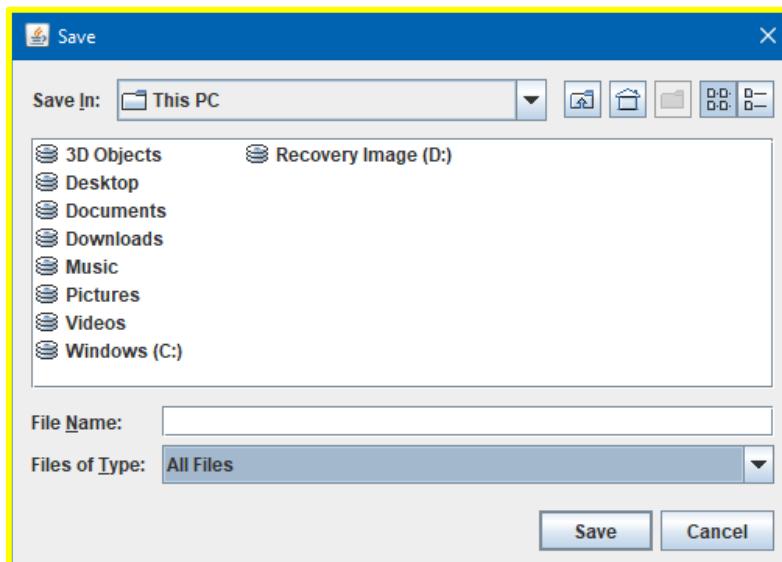


I then wanted the user to be able to save this string to a file, so firstly I added a JMenuItem to the “File” JMenu.

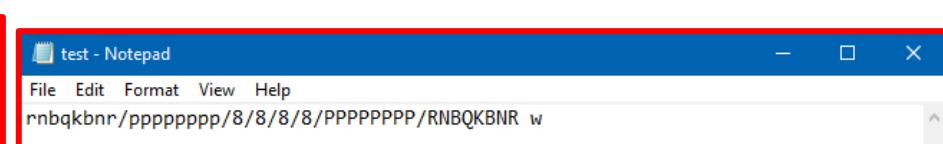
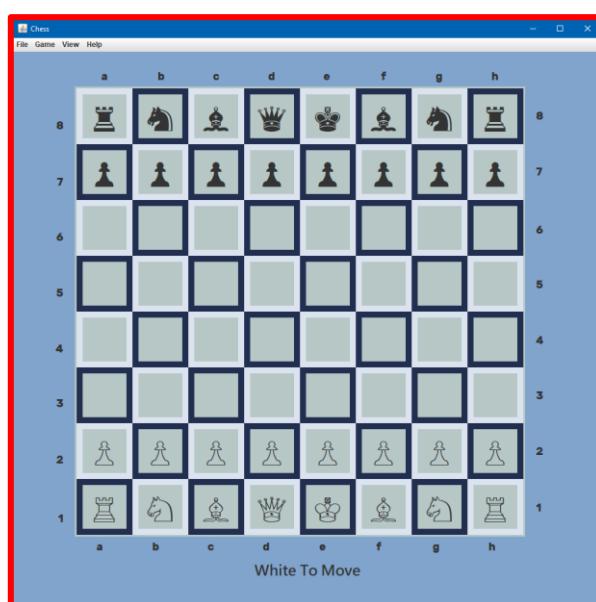
```
private void saveGameMenuItemActionPerformed(java.awt.event.ActionEvent evt) {  
    // TODO add your handling code here:  
    JFileChooser fileChooser = new JFileChooser();  
  
    // Shows dialog to user asking them to select a file, returns JFileChooser.APPROVE_OPTION if they have selected one:  
    boolean userHasSelectedFile = fileChooser.showSaveDialog(null) == JFileChooser.APPROVE_OPTION;  
  
    if (userHasSelectedFile) {  
        File file = fileChooser.getSelectedFile(); // Gets the file the user selected  
        String fen = this.board.getGameString(); // Generates string representing the board  
        try {  
            FileWriter writer = new FileWriter(file);  
            writer.write(fen); // Writes this string to the file  
            writer.close(); // Closes the file  
        } catch (Exception ex) {  
            System.err.println(ex);  
        }  
    }  
}
```

Next, I added the code that should be run when the user clicks this JMenuItem. Firstly, I create a new JFileChooser, and show this dialog to the user. This allows them to select a file to write to. Then, if they selected a file, I generate the String representing the game, and attempt to write this to the selected file. If an exception occurs, this will be printed.

Testing



When I tested this, when the user presses the “Save Game” JMenuItem, they are correctly shown a window, allowing them to select a file.



And when I tried this with the board in this layout, it produced the above string, showing that the program correctly wrote to the chosen file, and that the string to represent the game is also generated correctly – “getGameString” appears to be working as expected.

Loading previously saved games

Development

```
public void loadGameString(String str) {
    if (str.matches("([rnbqkpRNBQKPl-8]{1,8}\\\\/){7} ([rnbqkpRNBQKPl-8]{1,8}) [wb]")) { // Checks string is in correct format
        String[] parts = str.split(" ");
        String[] rows = parts[0].split("/");
        for (int i = 0; i < rows.length; i++) {
            for (int j = 0, col = 0; j < rows[i].length(); j++) {
                switch (rows[i].charAt(j)) {
                    case 'p':
                        this.board[i][col] = new Pawn(1);
                        col++;
                        break;
                    case 'P':
                        this.board[i][col] = new Pawn(0);
                        col++;
                        break;
                    case 'r':
                        this.board[i][col] = new Rook(1);
                        col++;
                        break;
                    case 'R':
                        this.board[i][col] = new Rook(0);
                        col++;
                        break;
                    case 'n':
                        this.board[i][col] = new Knight(1);
                        col++;
                        break;
                    case 'N':
                        this.board[i][col] = new Knight(0);
                        col++;
                        break;
                    case 'b':
                        this.board[i][col] = new Bishop(1);
                        col++;
                        break;
                    case 'B':
                        this.board[i][col] = new Bishop(0);
                        col++;
                        break;
                    case 'q':
                        this.board[i][col] = new Queen(1);
                        col++;
                        break;
                    case 'Q':
                        this.board[i][col] = new Queen(0);
                        col++;
                        break;
                    case 'k':
                        this.board[i][col] = new King(1);
                        col++;
                        break;
                    case 'K':
                        this.board[i][col] = new King(0);
                        col++;
                        break;
                    default:
                        int num = Integer.valueOf(rows[i].substring(j, j + 1));
                        for (int k = 0; k < num; k++) {
                            this.board[i][col] = null;
                            col++;
                        }
                }
            }
        }
        this.moveHistory = new MoveHistory();
        this.currentTurn = parts[1].equals("w") ? 0 : 1;
        if(this.currentTurn % 2 == 1 && this.playingAgainstComputer) {
            this.makeComputerMove();
        }
    }
}
```

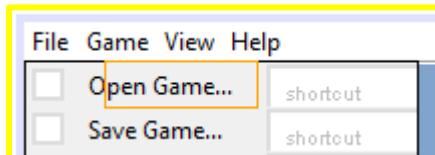
Whilst the user can now save the game to a file, they cannot currently do anything with this, so next I decided to implement the ability to open a game from a file.

To do this I created a procedure to load the game from a string.

Firstly, I check whether the string is in the given format, using a regular expression – 8 blocks of text with any of the characters r, n, b, q, k, p, R, N, B, Q, K, P, 1, 2, 3, 4, 5, 6, 7, 8, with a maximum of 8 and a minimum of 1 character in each, separated by a “/”, and a w or a b at the end, indicating whose turn it is. If I don’t check this, errors such as “IndexOutOfBoundsException” may occur, or an unrecognised character may be in the string, which could potentially cause an error.

Then, I iterate through the string, setting the values in the array of Pieces representing the board, depending on the character in the string passed to the function.

After that, I reset the moveHistory, as this is loading a new game, and so shouldn’t have the history of the current game, and I set the current turn – 0 if the string indicates that it is whites turn to move, and 1 if it is blacks turn to move.

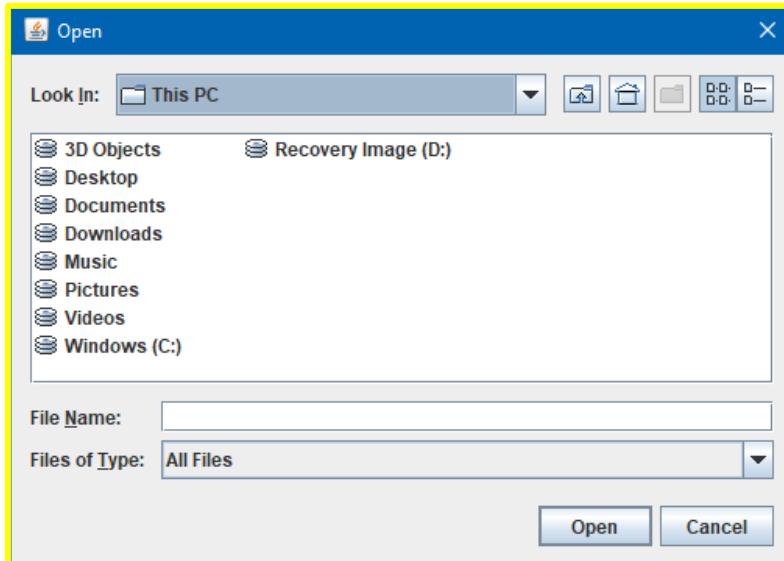


I then wanted the user to be able open a game from a file, so firstly I added a JMenuItem to the “File” JMenu labelled “Open Game...”.

```
private void openGameMenuItemActionPerformed(java.awt.event.ActionEvent evt) {  
    // TODO add your handling code here:  
    JFileChooser fileChooser = new JFileChooser();  
  
    // Shows dialog to user asking them to select a file, returns JFileChooser.APPROVE_OPTION if they have selected one:  
    boolean userHasSelectedFile = fileChooser.showOpenDialog(null) == JFileChooser.APPROVE_OPTION;  
  
    if (userHasSelectedFile) {  
        File file = fileChooser.getSelectedFile(); // Gets the file the user selected  
        try {  
            Scanner reader = new Scanner(file);  
            if(reader.hasNextLine()) { // If the file has a next line  
                String fen = reader.nextLine(); // Get the next line from the file  
                this.board.loadGameString(fen); // Load this string into the board  
            }  
            this.updateGUI();  
        } catch (Exception ex) {  
            System.err.println(ex);  
        }  
    }  
}
```

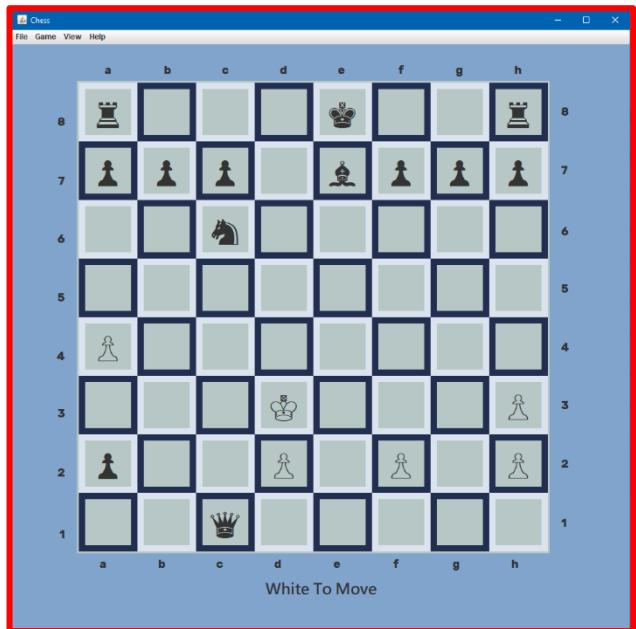
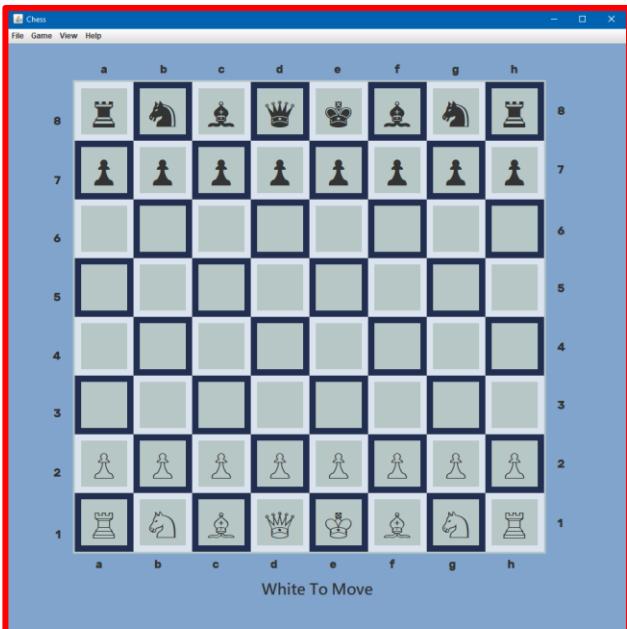
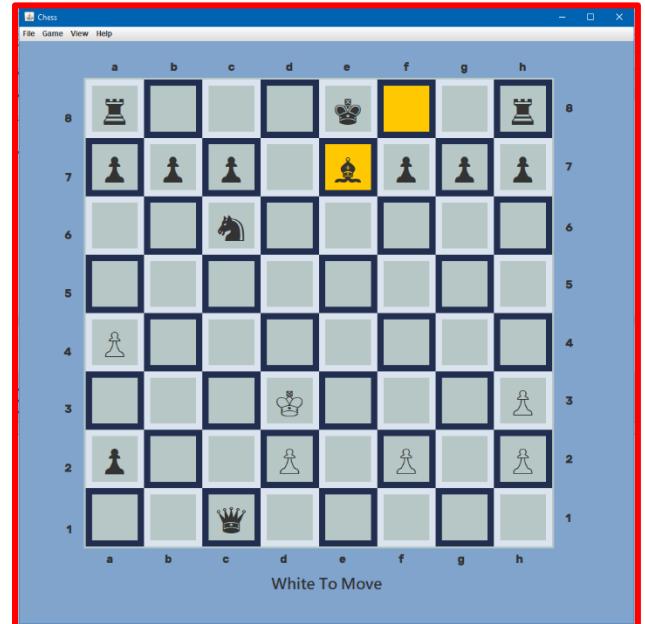
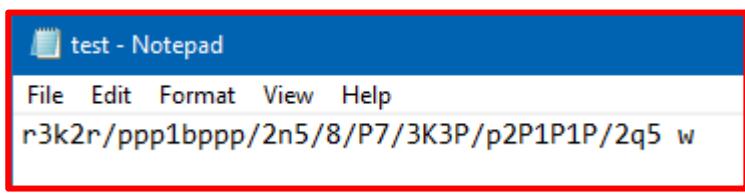
Next I added the code that should be run when the user clicks this JMenuItem. Firstly, I create a new JFileChooser, and show this dialog to the user. This allows them to select a file to write to. Then, if they selected a file, I use a Scanner object to read the first line from this file, and attempt to load this into the chess board. Afterwards, I update the GUI to show the user the new board layout.

Testing



When I tested this, when the user presses the “Open Game” JMenuItem, they are correctly shown a window, allowing them to select a file to open.

Then, to test whether the file will be loaded correctly, I first used the previous save game feature to save the state of this game to a file.



Then, I started a new game, and tried to open the game that was just saved.

As can be seen above, this worked as expected – the game is loaded with the pieces in the correct positions, and the correct player to move, showing that the “loadGameString” procedure is working as expected.

Review

Now that all of the main features that the client and end users had requested have been implemented, I can go back and see if any features need to be changed, added or removed.

Link to success criteria: #18, #19

Changing look of GUI

Development

Since the end users said they thought that the GUI was important, I decided to try and use icons to represent the pieces rather than just text, as I think that these will contrast better with the board and look more aesthetically pleasing, which was something the end-users said they liked in other programs. I also talked to my client, and they agreed that they would like the GUI to use clearer icons and have a more minimalistic design, by removing the grey background colour on each of the buttons, as he thinks this will make the program look better. Also, by using icons, the program will look more consistent between different systems, as I noticed that when running it on different computers, the pieces looked different as the fonts available changed.

```
private String img;
```

Firstly, I added an attribute of type “String” to the “Piece” class, which will store the location of the image to be used to represent the piece.

```
public String getImage() {  
    return this.img;  
}
```

Next, I added methods to set and get this image, so that it can be accessed by the other classes (i.e. the JFrame, so that it can display these images to the user).

```
public void setImage(String img) {  
    this.img = img;  
}
```



```
public Piece(int type, int colour, String img) {  
    this.type = type;  
    this.colour = colour;  
    this.img = img;  
}
```

```
public Bishop(int colour) {  
    super(4, colour, colour == 0 ? "/white-bishop.png" : "/black-bishop.png");  
}
```

```
public King(int colour) {  
    super(6, colour, colour == 0 ? "/white-king.png" : "/black-king.png");  
}
```

```
public Knight(int colour) {
    super(3, colour, colour == 0 ? "/white-knight.png" : "/black-knight.png");
}
```

```
public Pawn(int colour) {
    super(1, colour, colour == 0 ? "/white-pawn.png" : "/black-pawn.png");
}
```

```
public Queen(int colour) {
    super(5, colour, colour == 0 ? "/white-queen.png" : "/black-queen.png");
}
```

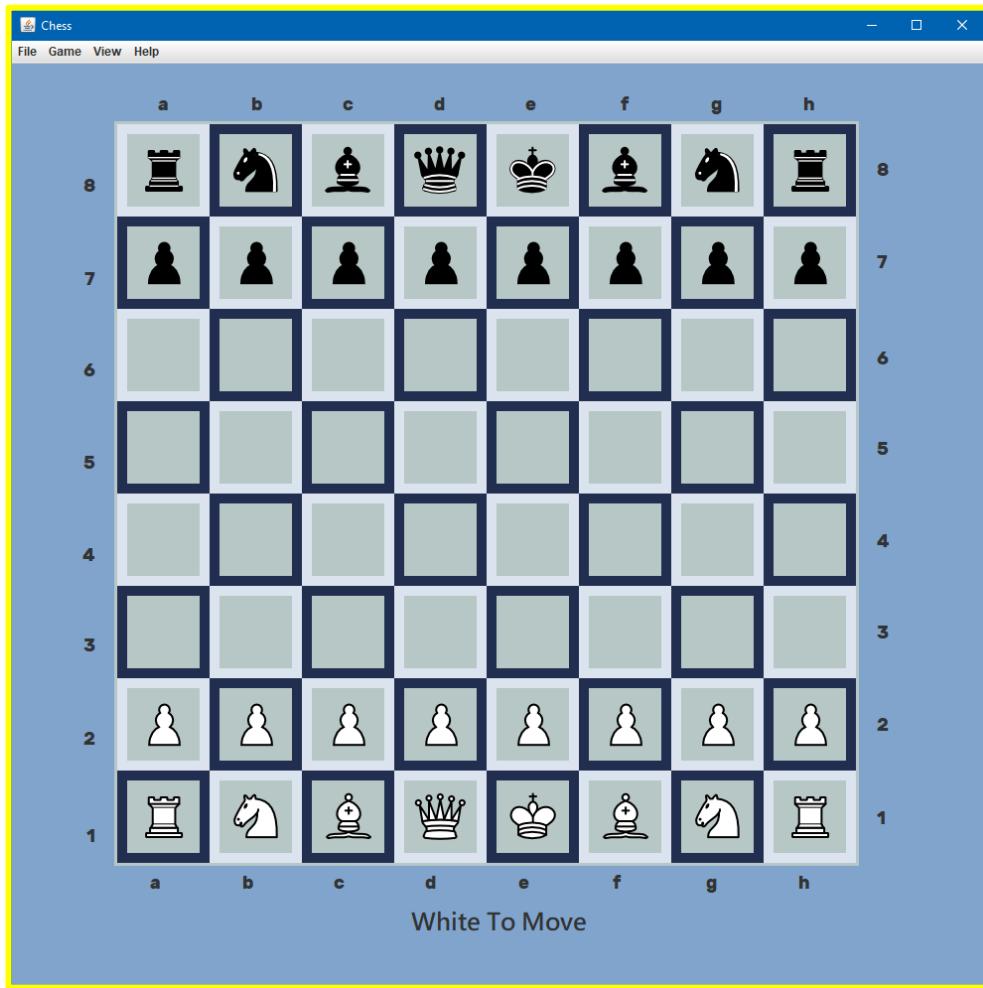
```
public Rook(int colour) {
    super(2, colour, colour == 0 ? "/white-rook.png" : "/black-rook.png");
}
```

I then edited the constructors of each of the different types of pieces to set “img” to the correct value. This sets the value of img to “/white-[type].png” if colour is equal to 0 (i.e. the piece is white), and “/black-[type].png” otherwise (i.e. the piece is black).

```
for (int i = 0; i < cb.length; i++) {
    for (int j = 0; j < cb[i].length; j++) {
        if (cb[i][j] != null) { // If there is a piece at this location
            this.buttons[i][j].setIcon(new javax.swing.ImageIcon(getClass().getResource(cb[i][j].getImage())));
        } else {
            this.buttons[i][j].setIcon(null);
        }
    }
}
```

I then edited the “updateGUI” method so that instead of setting the text of the buttons to represent the piece, the icon is set instead to show these images. Setting the icon to null means that no icon will be shown on that button / that it will appear empty, as there is no piece there.

Testing



When I ran this, the icons appeared correctly, however the user could not make a move. I realised that this was because I was checking whether or not the text of the button was empty, and if it was, not allowing the user to select that piece, however all of the buttons now have no text in them, so none of them can be selected by the user.

Remedial

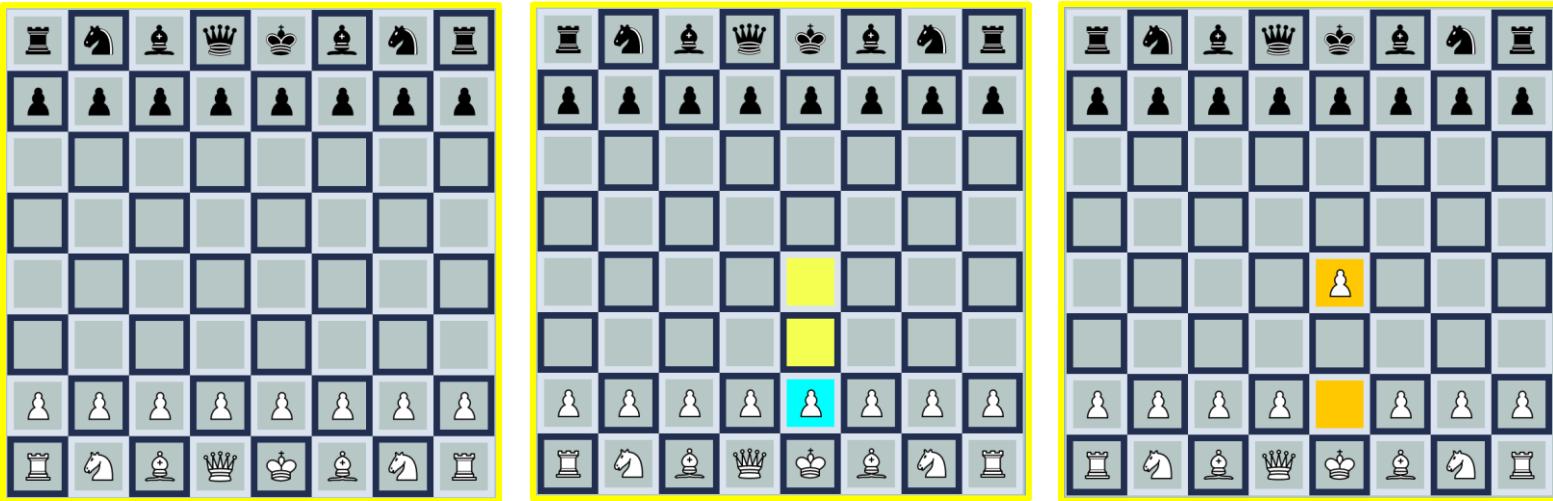
```
public boolean hasPieceAtPosition(int pos) {  
    return this.board[pos / 8][pos % 8] != null;  
}
```

To fix this, I added a function to the "ChessBoard" class which will return true if there is a piece at a particular position, and false otherwise. I will use this to determine whether there is a piece at a particular location or not, rather than relying on the text of the button.

```
// Allows user to select positions on the board and make moves  
public void buttonPressed(int pos) {  
    // If piece to move not chosen and location is not empty and piece belongs to current player  
    if (this.startPos == -1 && this.board.hasPieceAtPosition(pos)  
        && this.board.getColourOfPieceAtPosition(pos) == this.board.getCurrentTurn() % 2) {
```

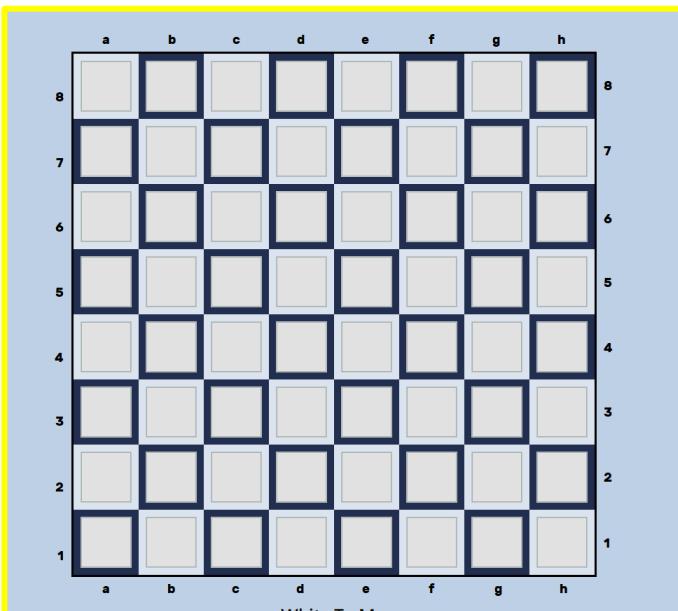
I then changed the "buttonPressed" procedure so that it uses the "hasPieceAtPosition" function to check whether a position is empty, rather than checking if the text of the button is empty, which should fix this error.

Testing



When I tested this, the user could now input their moves, and so this is working correctly.

Development



I next decided to change the colour of the JPanels – one to light blue and the other to black. I changed one to lighter blue to allow the colours of the individual squares to be lighter, which will allow the pieces to stand out more on the board and contrast well, which is something the end users have requested – if the squares are too dark, the pieces will blend into the board. I changed the other to black as it will act as an outline to the board.

I then decided to have the buttons all be one colour, rather than the centre of them being a different colour, as I think this will cause the GUI to look less cluttered & more minimalistic. To do this, I firstly created variables to store the colour of the light and dark squares.

```
this.lightSquareColour = new Color(219, 228, 238);
this.darkSquareColour = new Color(127, 158, 195);
```

```
private Color lightSquareColour;
private Color darkSquareColour;
```

I then chose two new default colours, and set these as the default values.



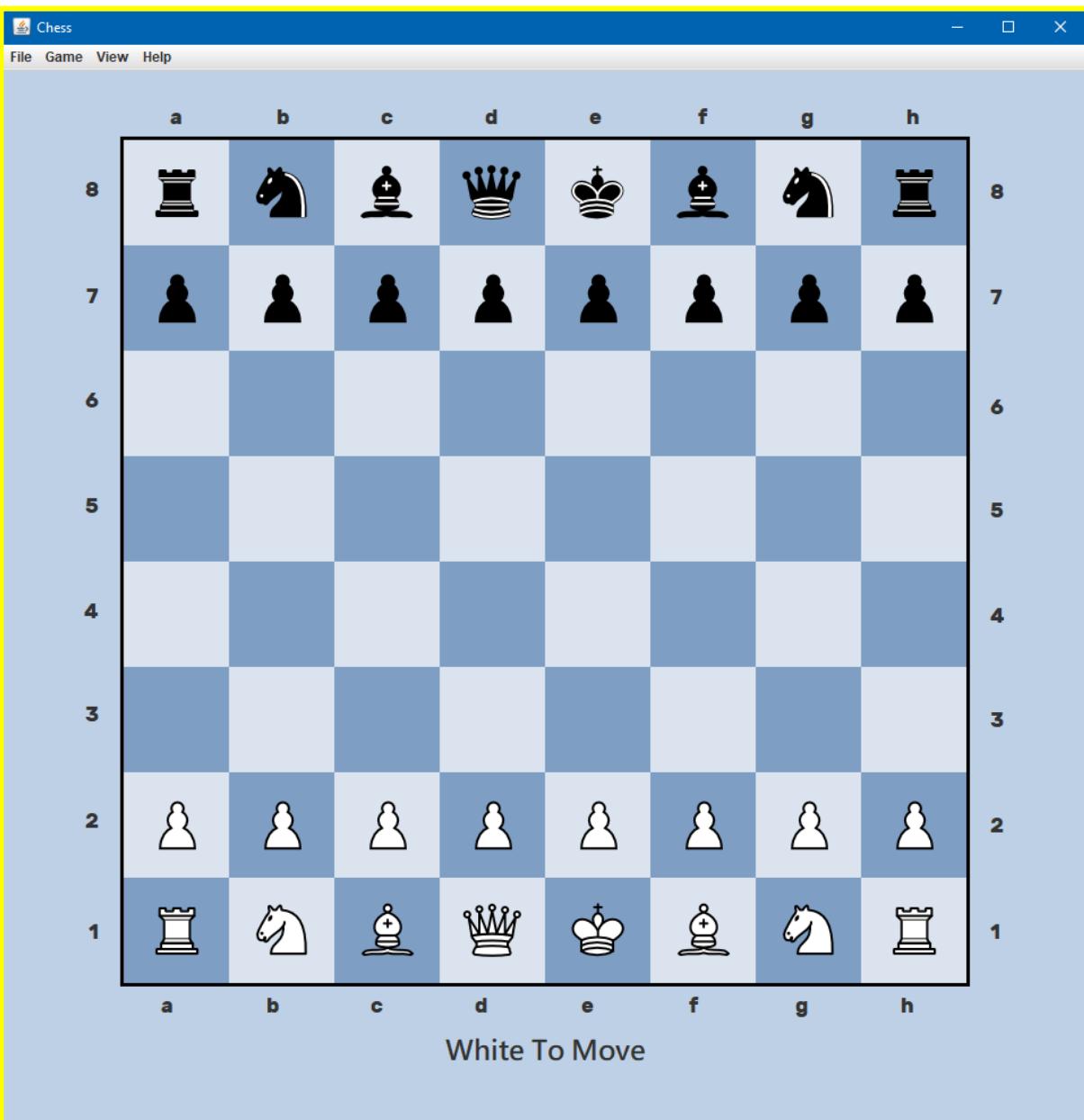
```
// Resets background colour of all buttons:
for (int i = 0; i < this.buttons.length; i++) {
    for (int j = 0; j < this.buttons[i].length; j++) {
        if (i % 2 == 0 && j % 2 == 0 || i % 2 != 0 && j % 2 != 0) {
            this.buttons[i][j].setBackground(this.lightSquareColour);
        } else {
            this.buttons[i][j].setBackground(this.darkSquareColour);
        }
    }
}
```

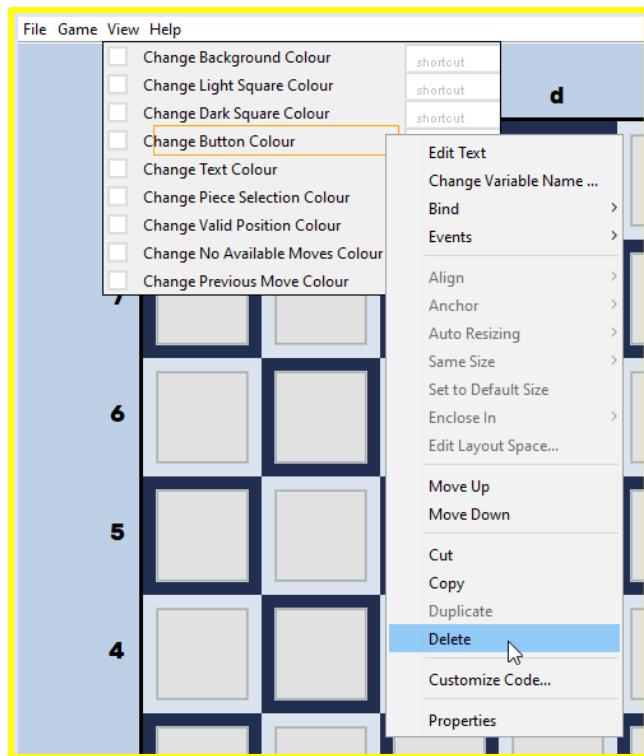
I then changed this for loop so that instead of resetting the background colour to the same value for all buttons, it sets it to the light / dark colour, in the chequered pattern of the board.

```
// Sets border of all buttons
for (int i = 0; i < 64; i++) {
    if ((i / 8) % 2 == 0 && (i % 8) % 2 != 0 || (i / 8) % 2 != 0 && (i % 8) % 2 == 0) {
        this.buttons[i / 8][i % 8].setBorder(javax.swing.BorderFactory.createLineBorder(this.darkSquareColour, 10));
    } else {
        this.buttons[i / 8][i % 8].setBorder(javax.swing.BorderFactory.createLineBorder(this.lightSquareColour, 10));
    }
}
```

After that I added a for loop to the “updateGUI” procedure which loops through all of the buttons, setting their borders in the chequered pattern of the board, so that the colour of the borders can be updated if they change. I also removed the code that did this in the change light / dark square JMenuItems, and called the “updateGUI” method in the action listeners for the light/dark square JMenuItems instead.

Testing





I finally then removed the “Change Button Colour” JMenuItem, and removed the “defaultColour” variable, since these are no longer needed, as the centres of the buttons are no longer different colours.

Review

Now, the GUI has been changed to use different colours / icons to make it more aesthetically pleasing, by removing the grey squares in the middle of each button, and using images, which stand out well with the board, instead of text, which did not stand out well, to represent the pieces.

Link to success criteria: #4

Now that all of the main features that the client and end users had requested have been implemented, and final changes have been made to the GUI, I decided to begin post development testing, in order to check that the program is fully functional.

Evaluation

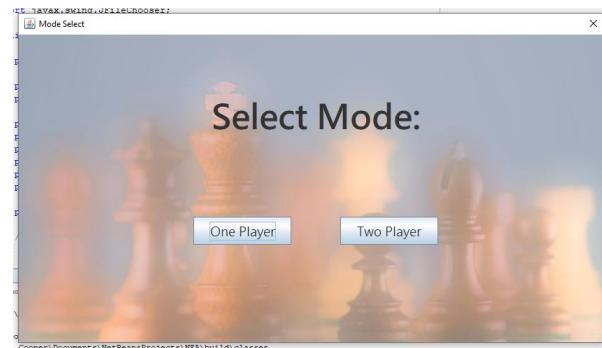
Post-development testing

Black box testing

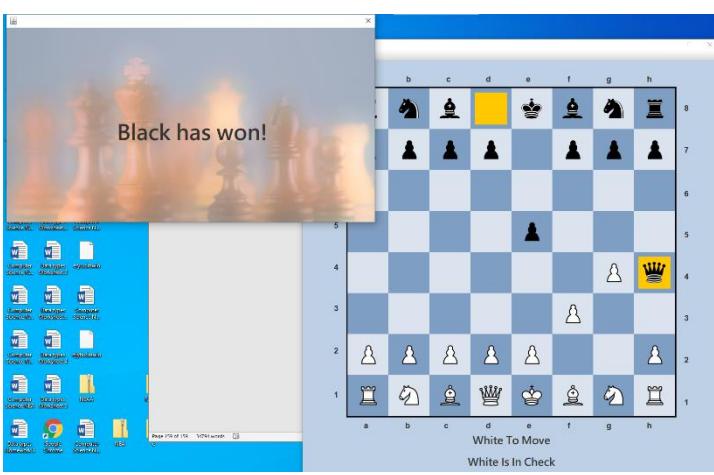
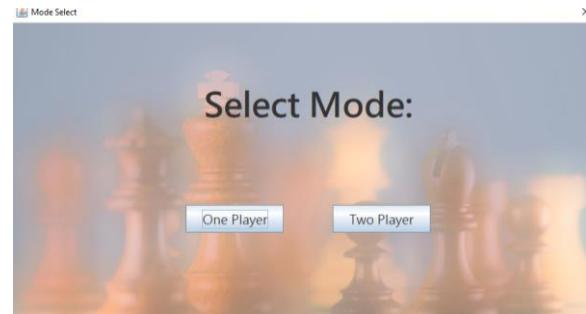
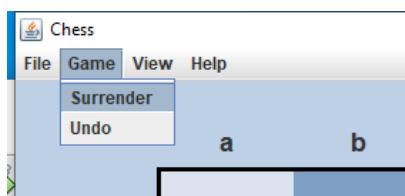
Firstly, I will use black box testing to test whether the program is functioning correctly.

#	Test data	Expected result	Actual result	Changes needed
1	User starts program	Window shown to user asking them to select a player to play against	Window is shown correctly when program is first run	
	User surrenders game		Window is shown correctly when game is surrendered	
	Game is ended due to check / checkmate		Window is shown correctly when game finishes	

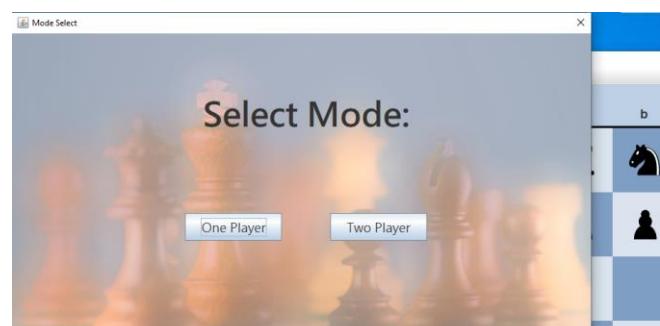
When starting the program, this window is shown to the user.



After surrendering the game, this window is shown.

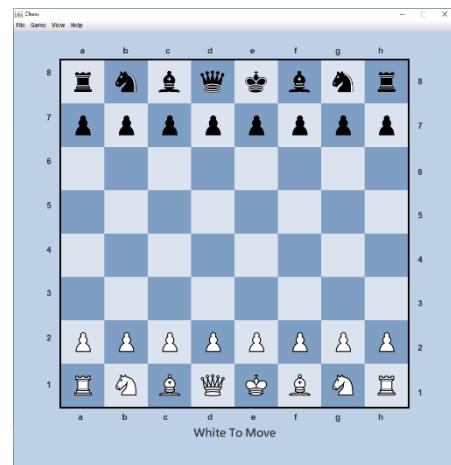
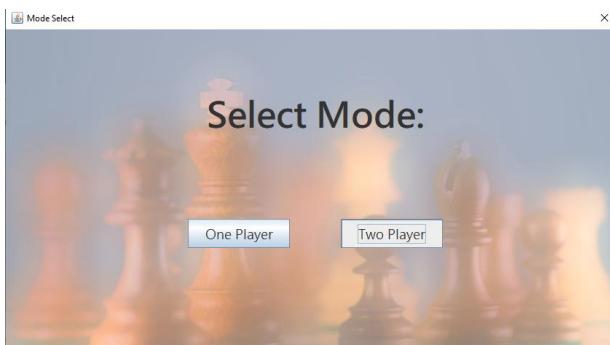


When the game ends due to checkmate, after closing the window saying who has won, the window asking user to select a player is shown.



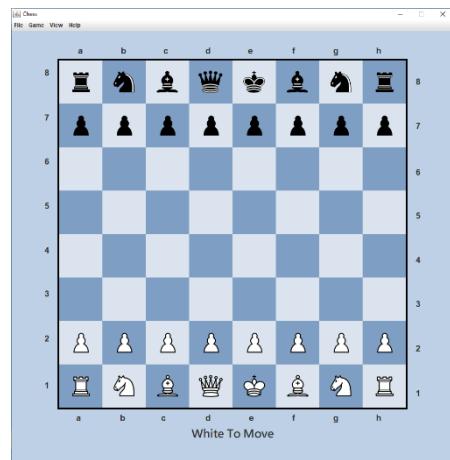
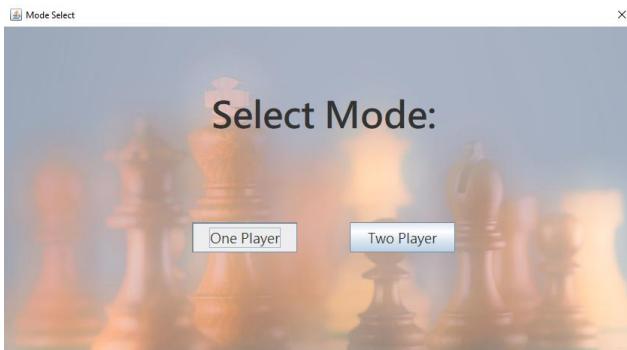
#	Test data	Expected result	Actual result	Changes needed
2	User starts program and selects player to play against	After user has selected who to play against, window is shown, containing 64 JButtons and the JMenuBar.	Window is shown correctly after user selects player	

User starts program and selects to play against someone else



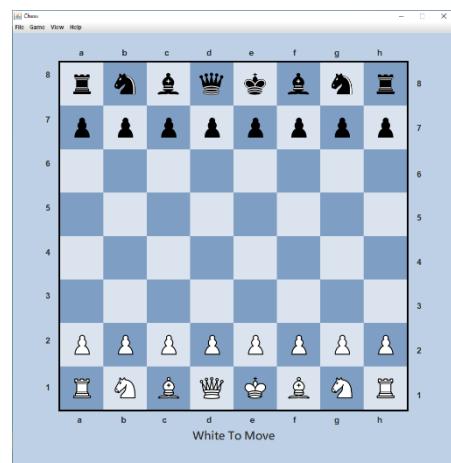
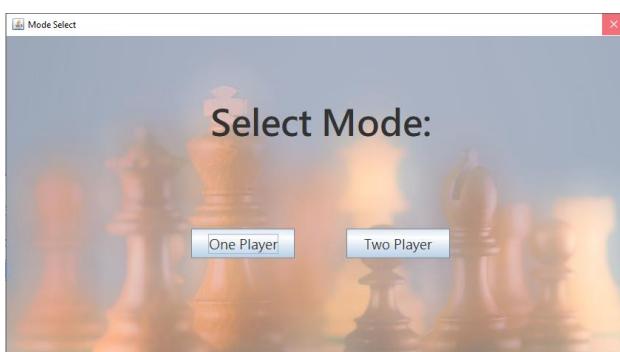
Window is successfully shown

User starts program and selects to play against computer



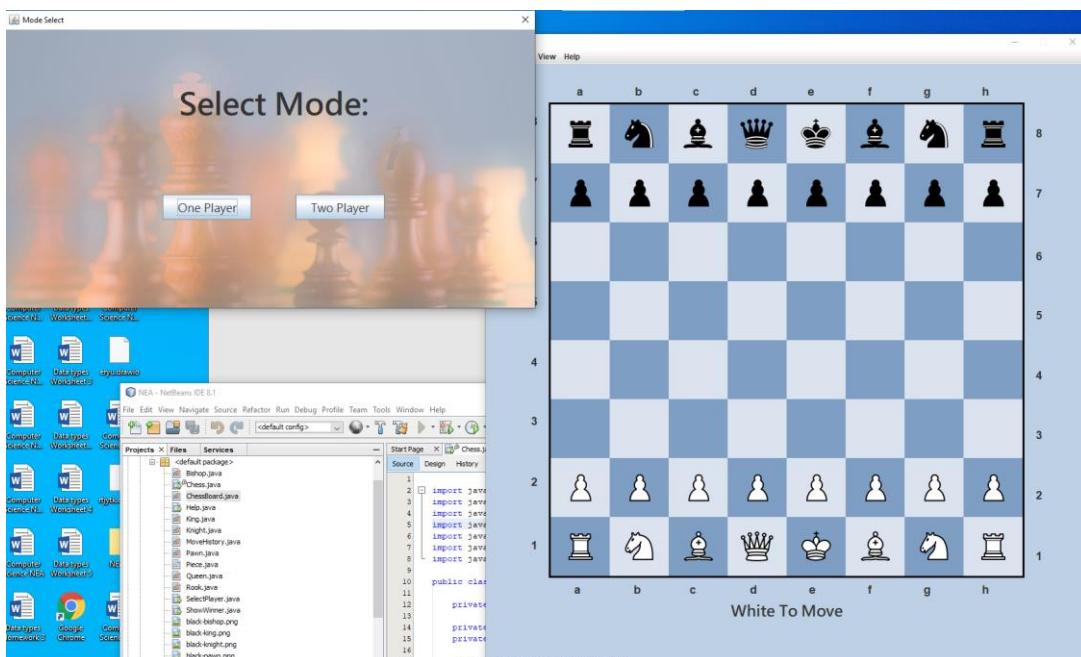
Window is successfully shown

User starts program and doesn't select a player and closes the window

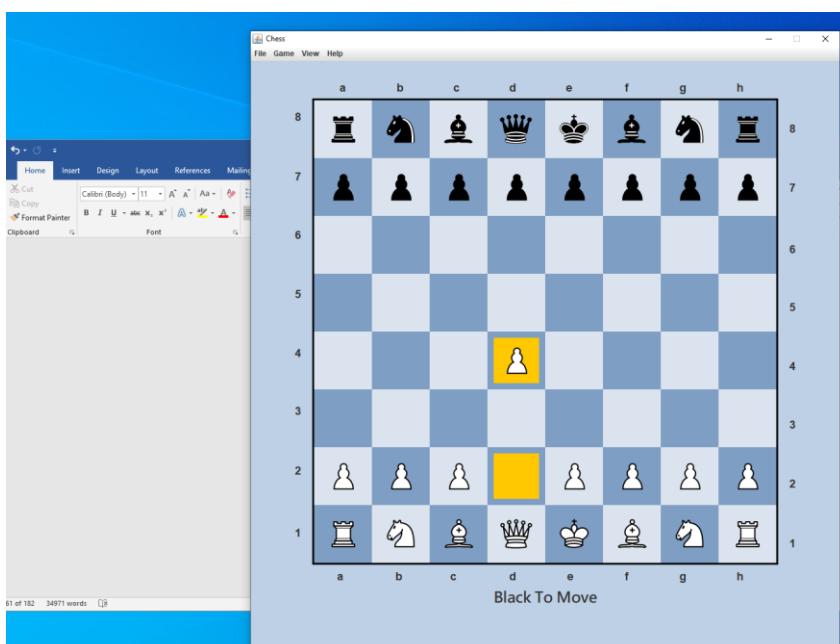


Window is successfully shown

#	Test data	Expected result	Actual result	Changes needed
3	Window is open asking user to select opponent and user tries to enter a move	Moves are unable to be made until window asking user to select player is closed.	Moves cannot be made	
	Window is not open asking user to select opponent and user tries to enter a move	Moves are allowed to be made	Moves can be made	



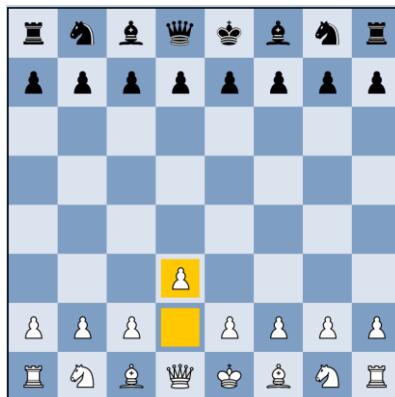
When select opponent window is open, buttons in the main window cannot be pressed, so user cannot make moves.



When select opponent window is not open, buttons in the main window can be pressed, so user is allowed to make moves.

#	Test data	Expected result	Actual result	Changes needed
4	User inputs move that is not pseudo-legal for a pawn	Move should not be made	Move is not made	
	User inputs a move that is pseudo-legal for a pawn	Move should be made	Move is correctly made	

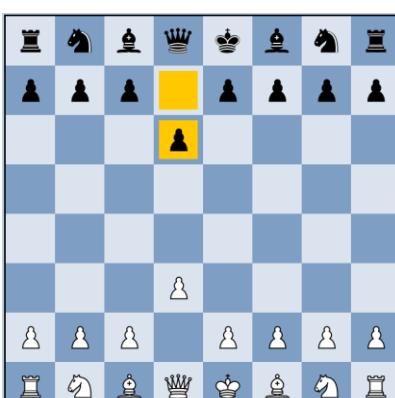
Testing valid inputs:



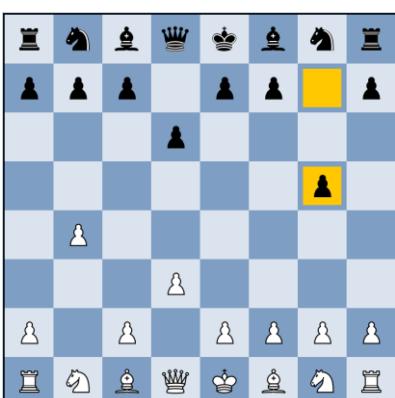
White pawn is allowed to move one square forwards



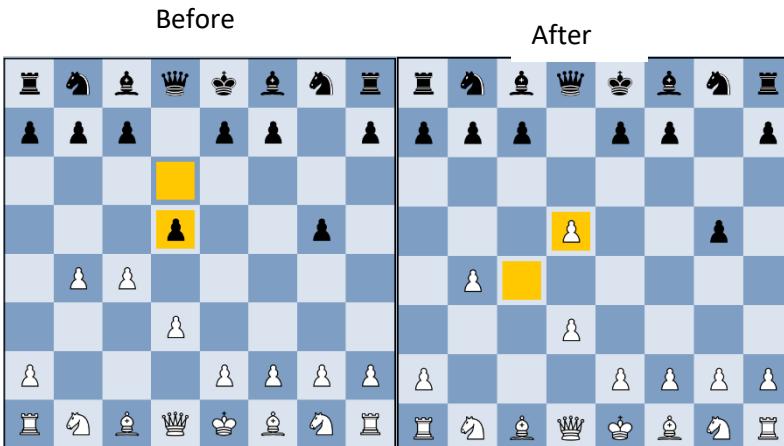
White pawn is allowed to move two squares forwards if it hasn't moved yet



Black pawn is allowed to move one square forwards



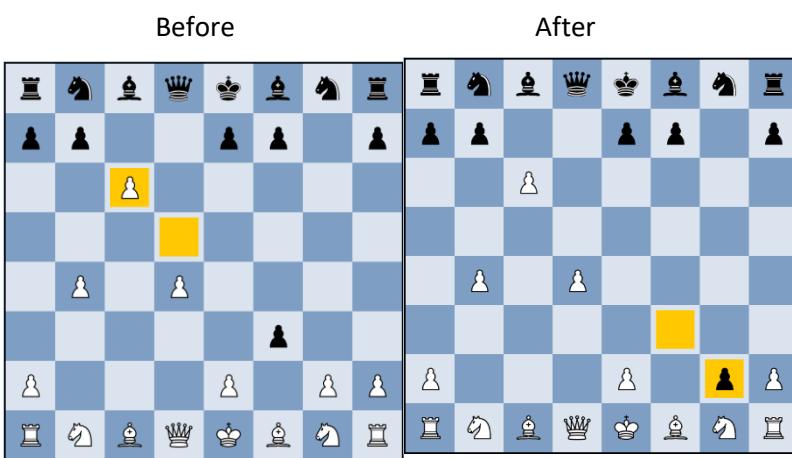
Black pawn is allowed to move two squares forwards if it hasn't moved yet



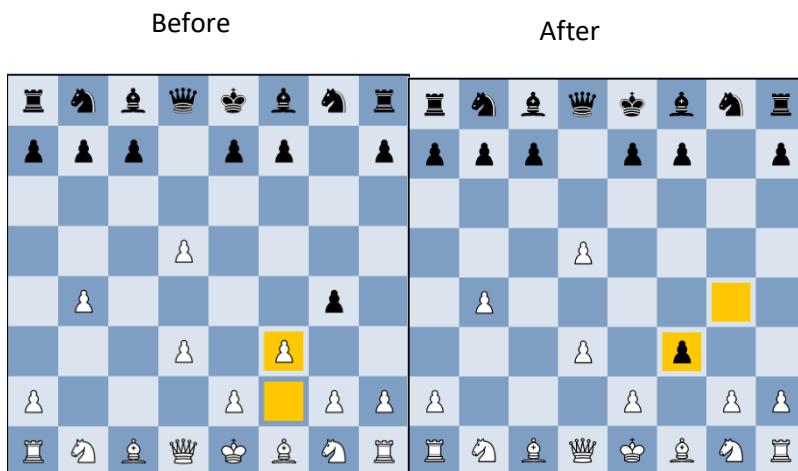
White pawn is allowed to attack to the top right



White pawn is allowed to attack to the top left



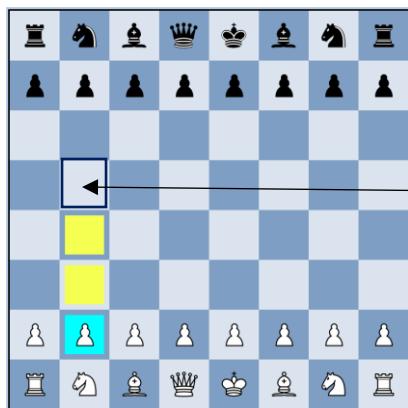
Black pawn is allowed to attack to the bottom right



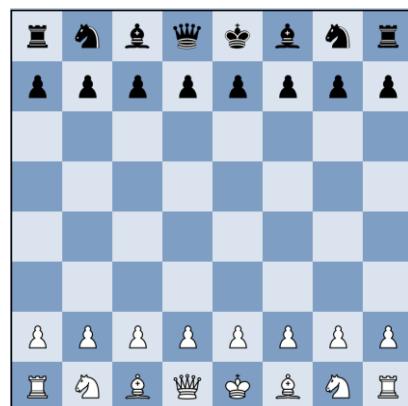
Black pawn is allowed to attack to the bottom left

Testing invalid inputs:

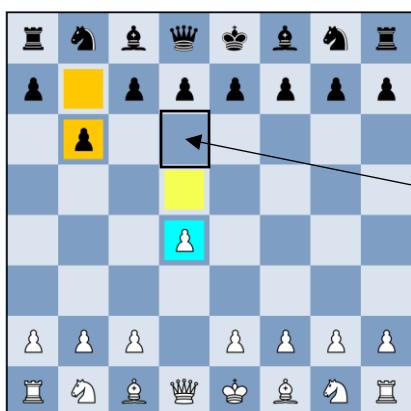
Before



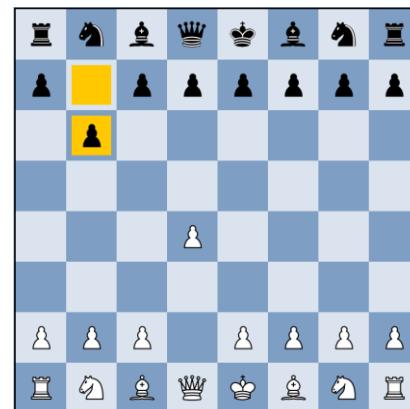
After



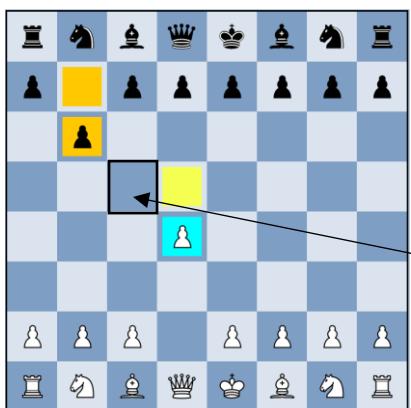
Before



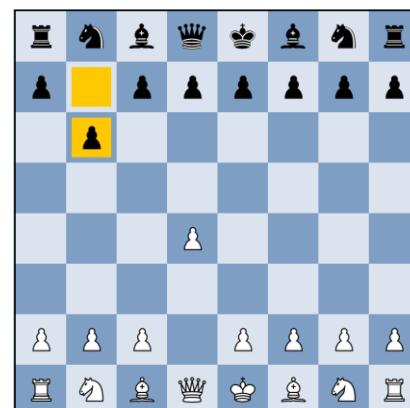
After



Before

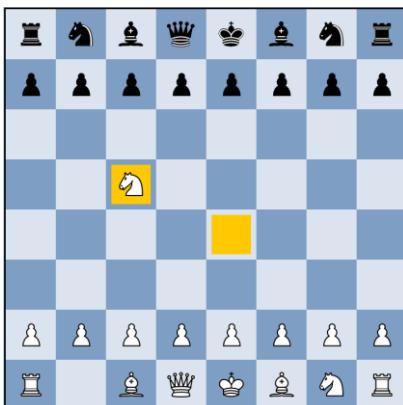


After



#	Test data	Expected result	Actual result	Changes needed
5	User inputs move that is not pseudo-legal for a knight	Move should not be made	Move is not made	
	User inputs a move that is pseudo-legal for a knight	Move should be made	Move is correctly made	

Testing valid inputs:



Testing if knight can move one up and two left



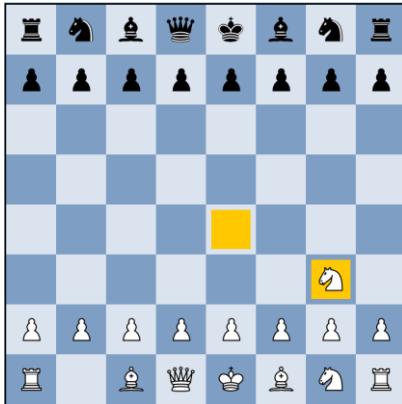
Testing if knight can move two up and one left



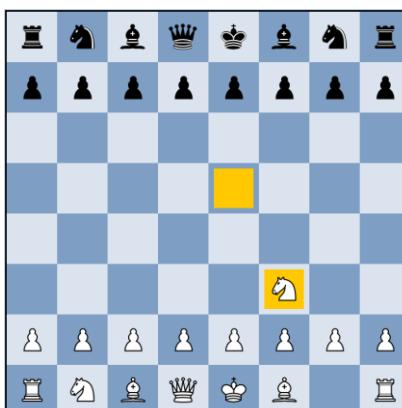
Testing if knight can move two up and one right



Testing if knight can move one up and two right



Testing if knight can move one down and two right



Testing if knight can move two down and one right



Testing if knight can move two down and one left



Testing if knight can move one down and two left

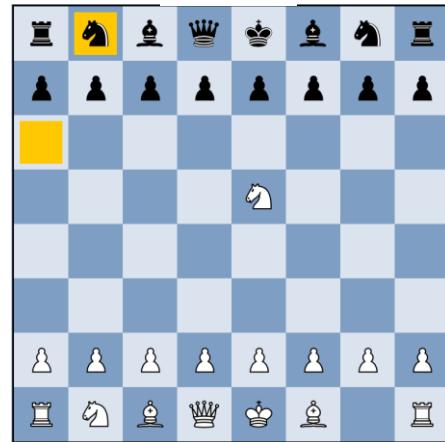
Testing invalid inputs:

Before

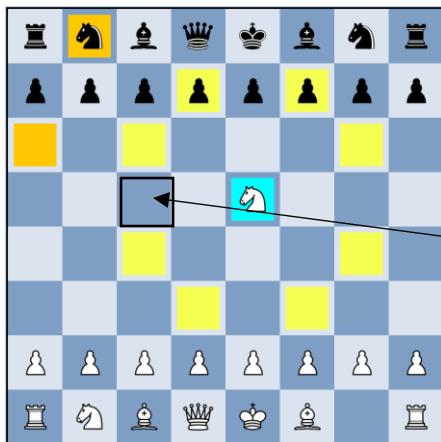


Selecting to move here

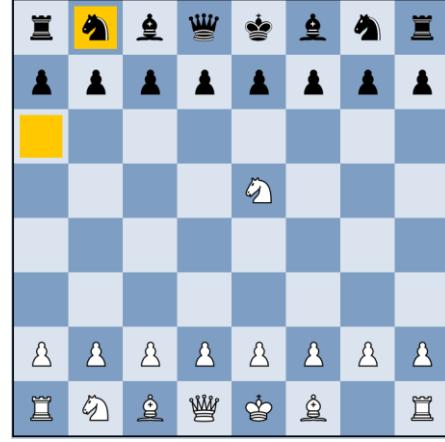
After



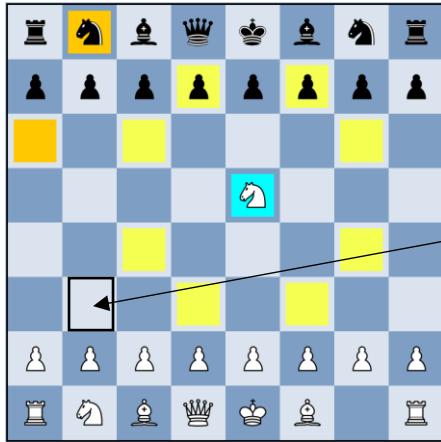
Before



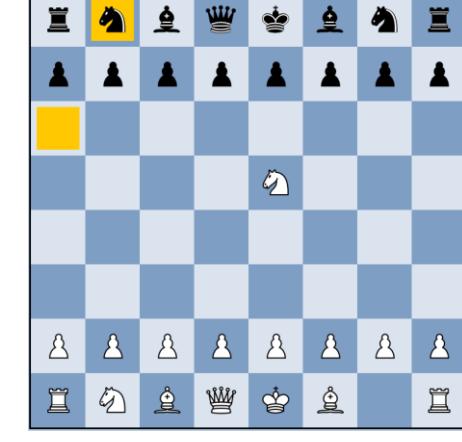
Selecting to move here



Before

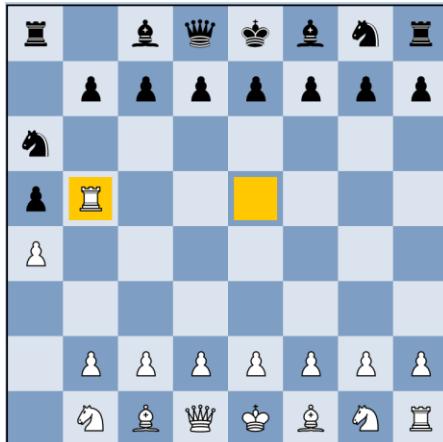


Selecting to move here

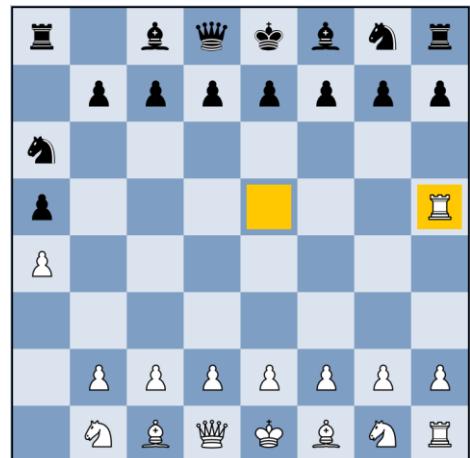


#	Test data	Expected result	Actual result	Changes needed
6	User inputs move that is not pseudo-legal for a rook	Move should not be made	Move is not made	
	User inputs a move that is pseudo-legal for a rook	Move should be made	Move is correctly made	

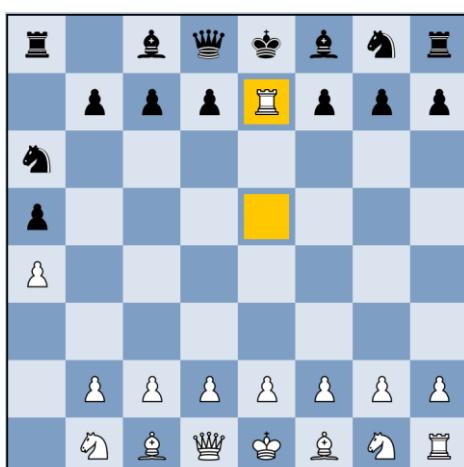
Testing valid inputs:



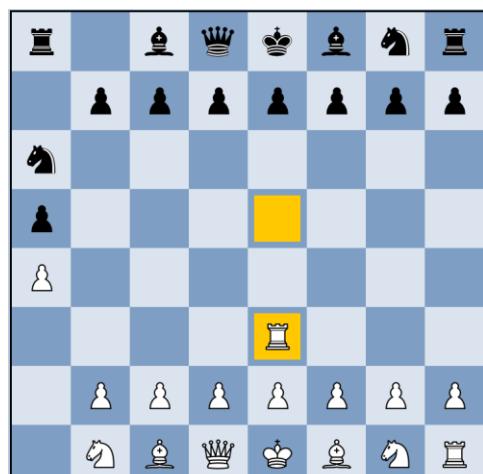
Testing if rook can move left



Testing if rook can move right



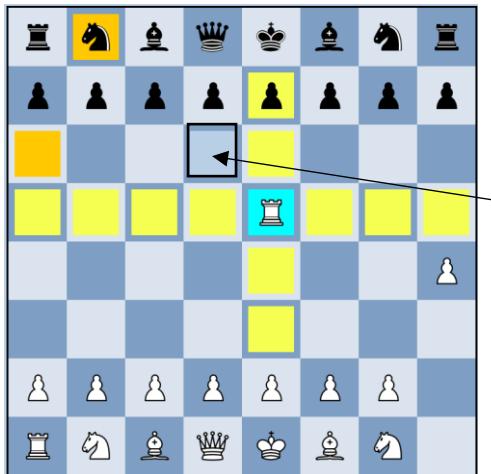
Testing if rook can move up



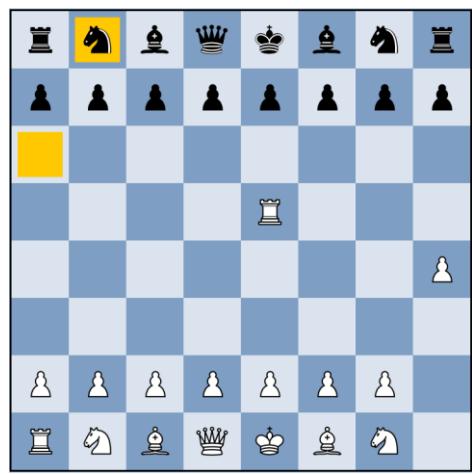
Testing if rook can move down

Testing invalid inputs:

Before

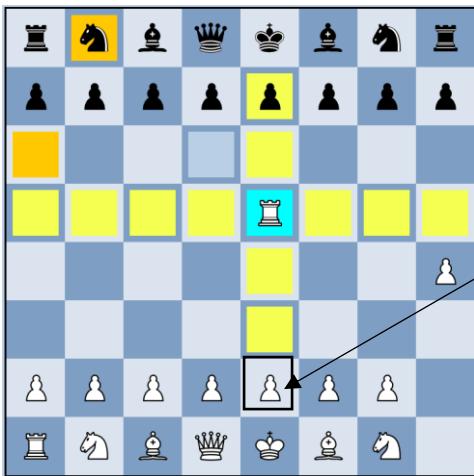


After



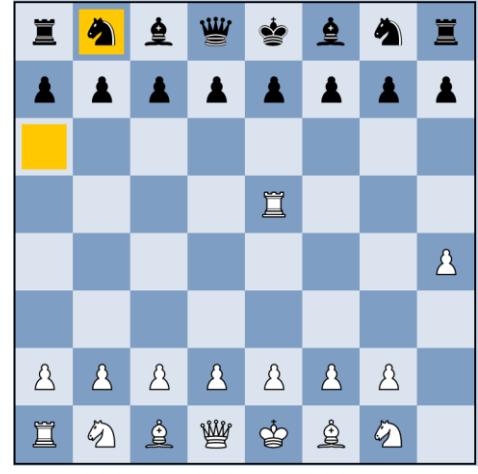
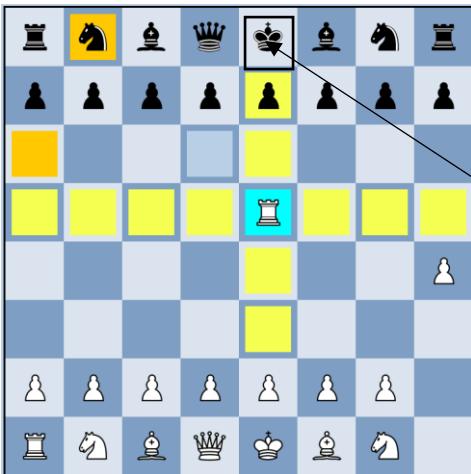
Selecting to move here

Before



Selecting to move here

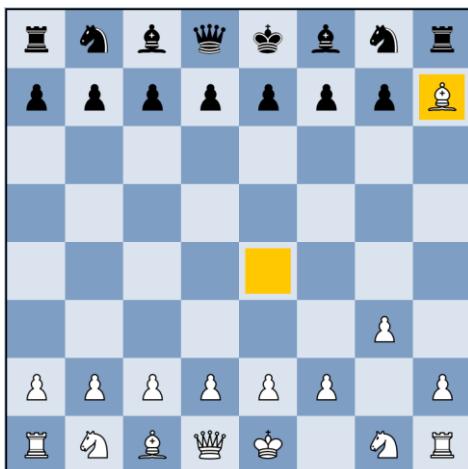
Before



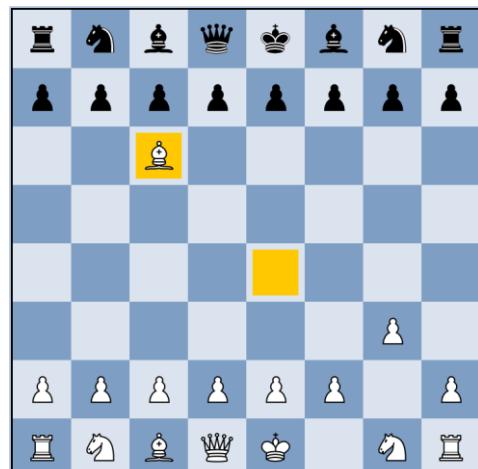
Selecting to move here

#	Test data	Expected result	Actual result	Changes needed
7	User inputs move that is not pseudo-legal for a bishop	Move should not be made	Move is not made	
	User inputs a move that is pseudo-legal for a bishop	Move should be made	Move is correctly made	

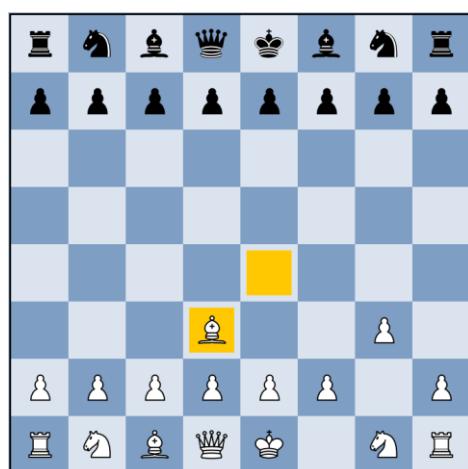
Testing valid inputs:



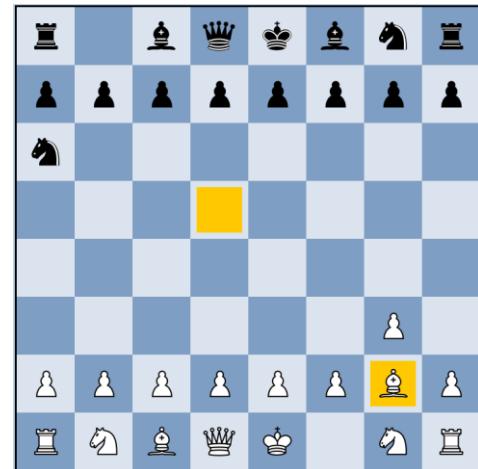
Testing if bishop can move north east



Testing if bishop can move north west

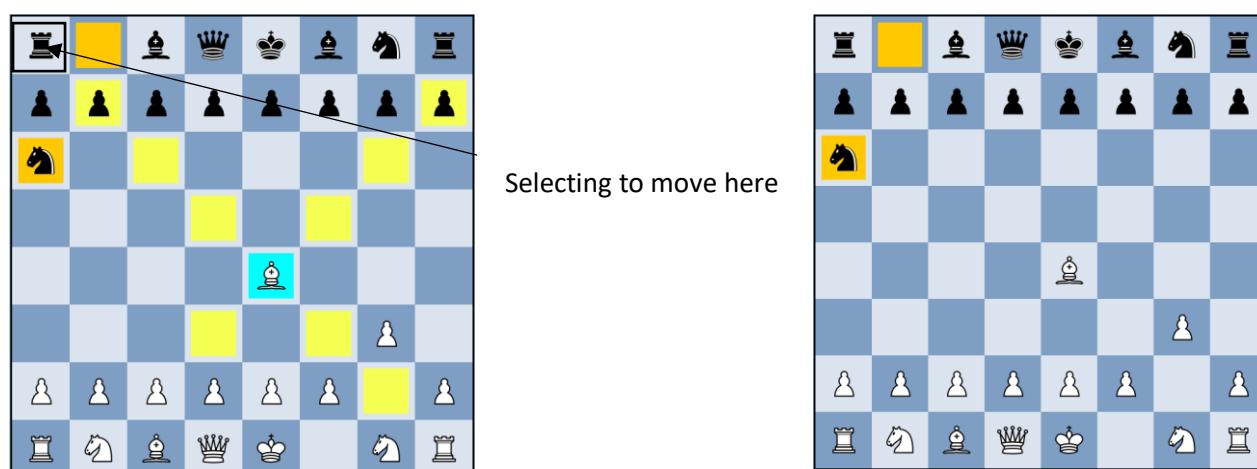
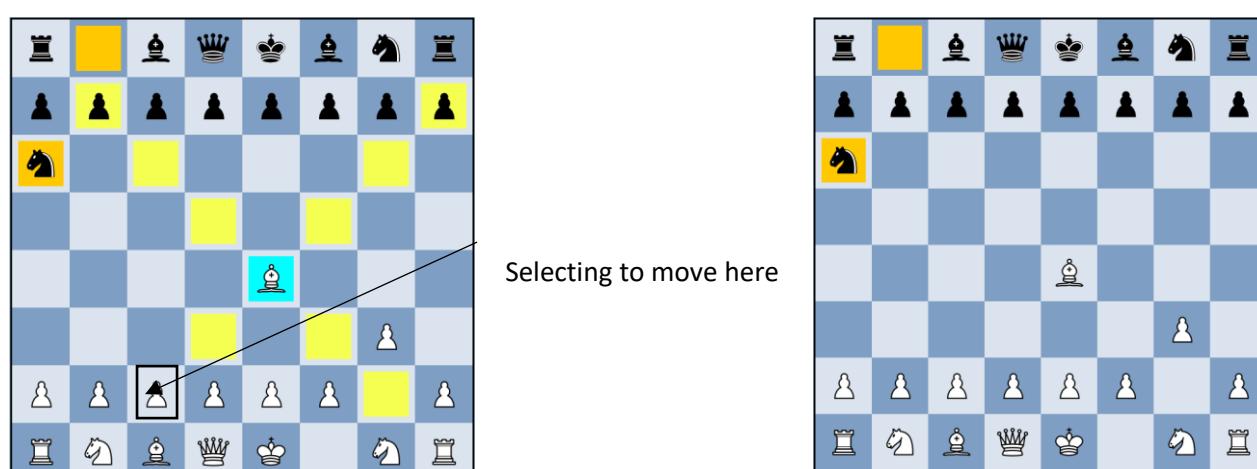
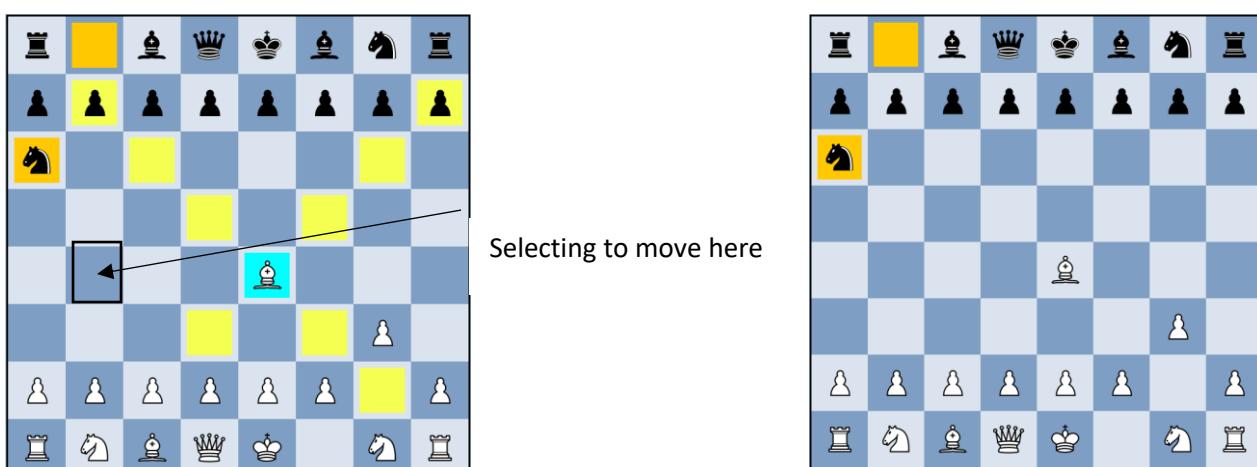
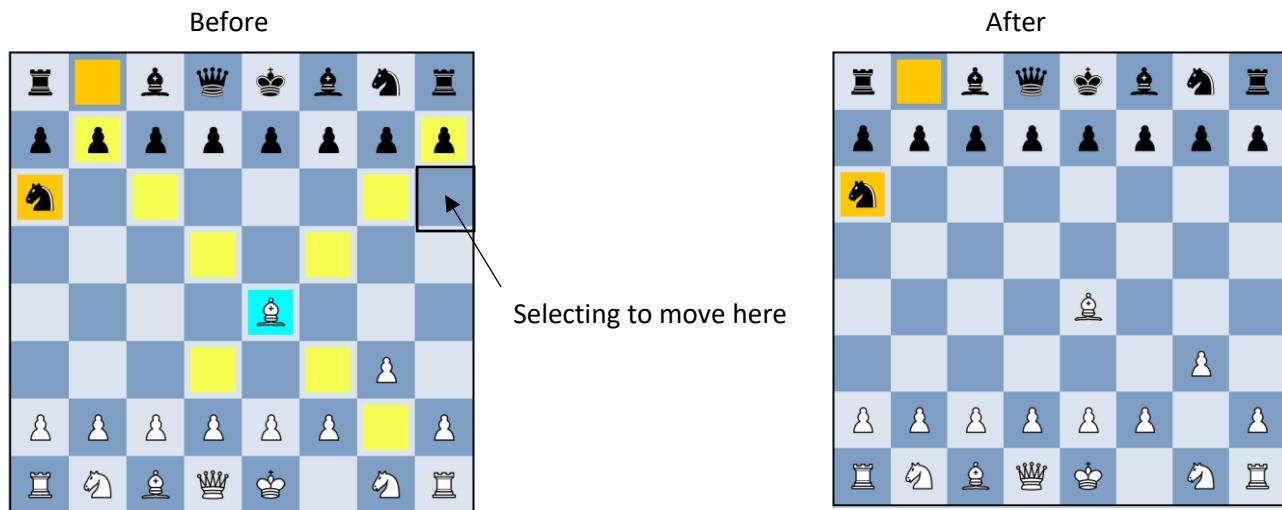


Testing if bishop can move south west



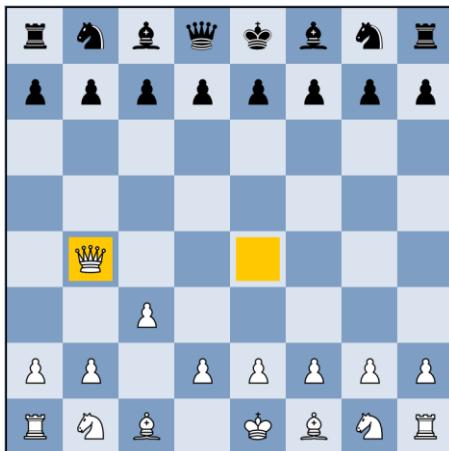
Testing if bishop can move south east

Testing invalid inputs:

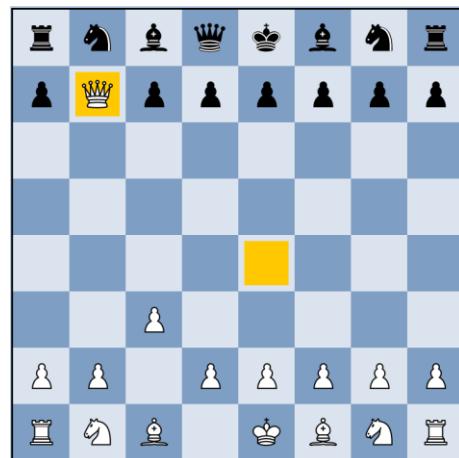


#	Test data	Expected result	Actual result	Changes needed
8	User inputs move that is not pseudo-legal for a queen	Move should not be made	Move is not made	
	User inputs a move that is pseudo-legal for a queen	Move should be made	Move is correctly made	

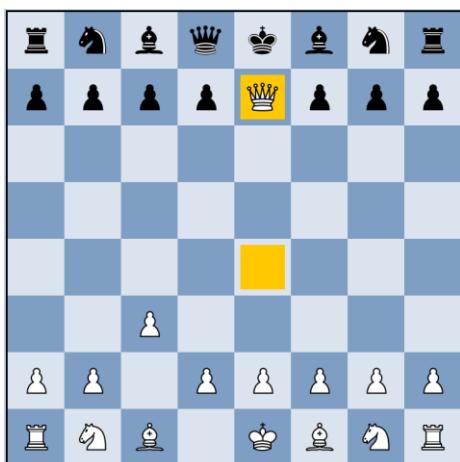
Testing valid inputs:



Testing if queen can move left



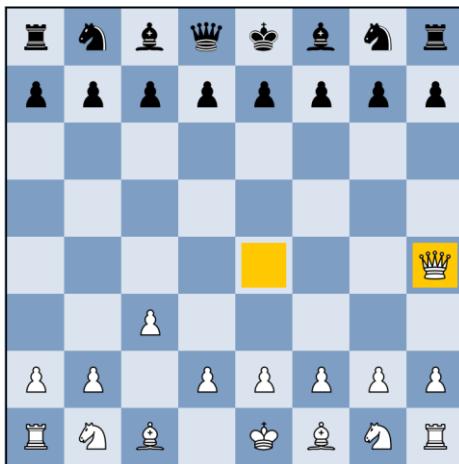
Testing if queen can move north west



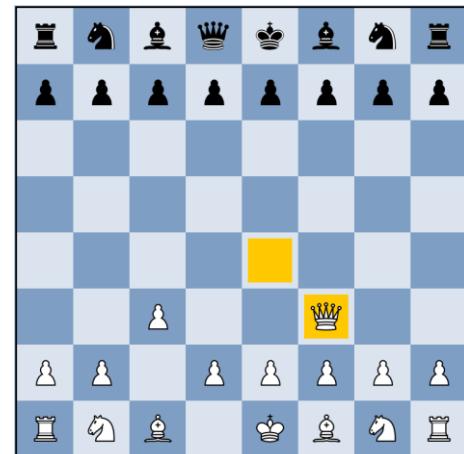
Testing if queen can move up



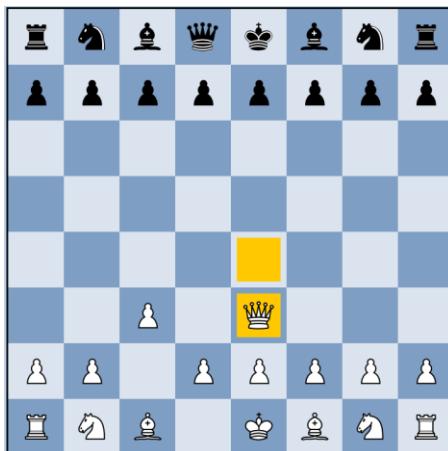
Testing if queen can move north east



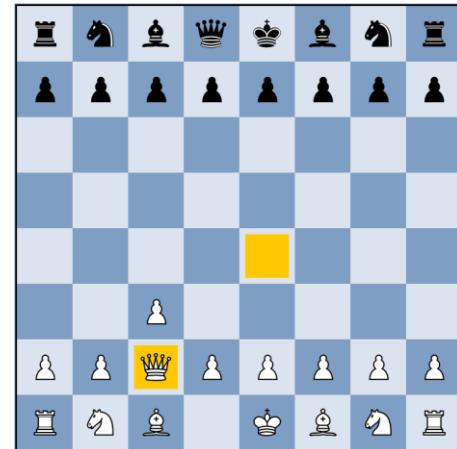
Testing if queen can move right



Testing if queen can move south east

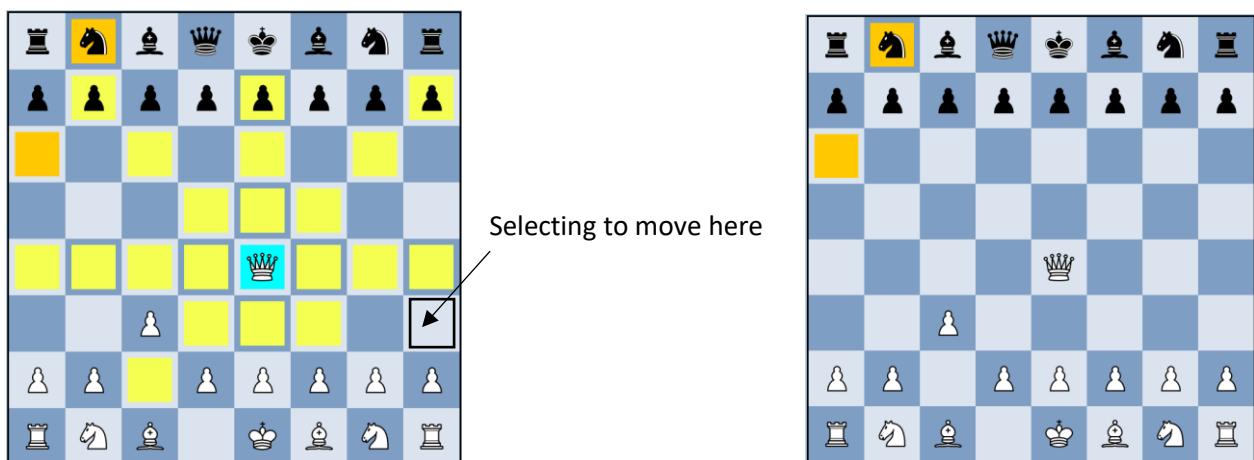
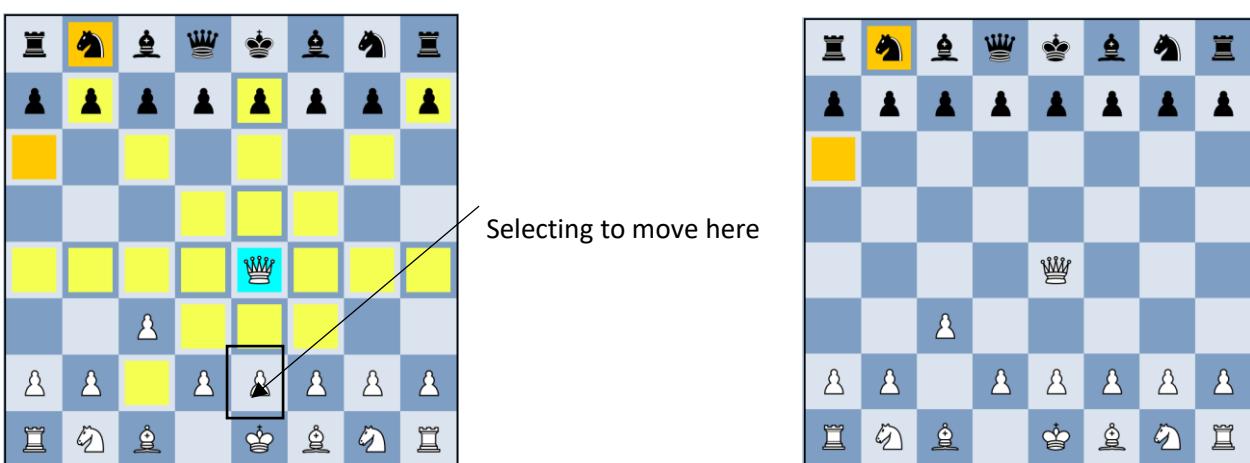
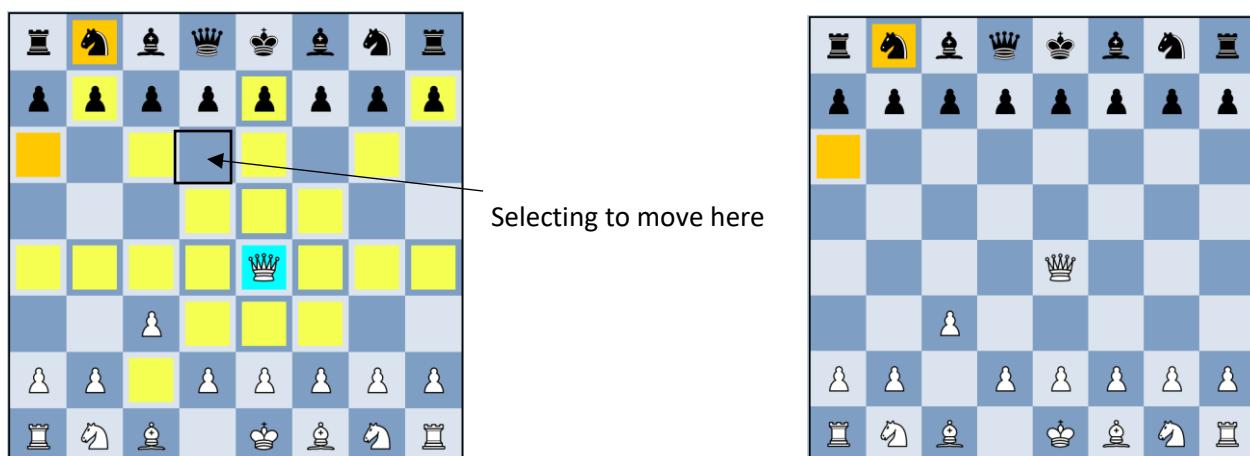
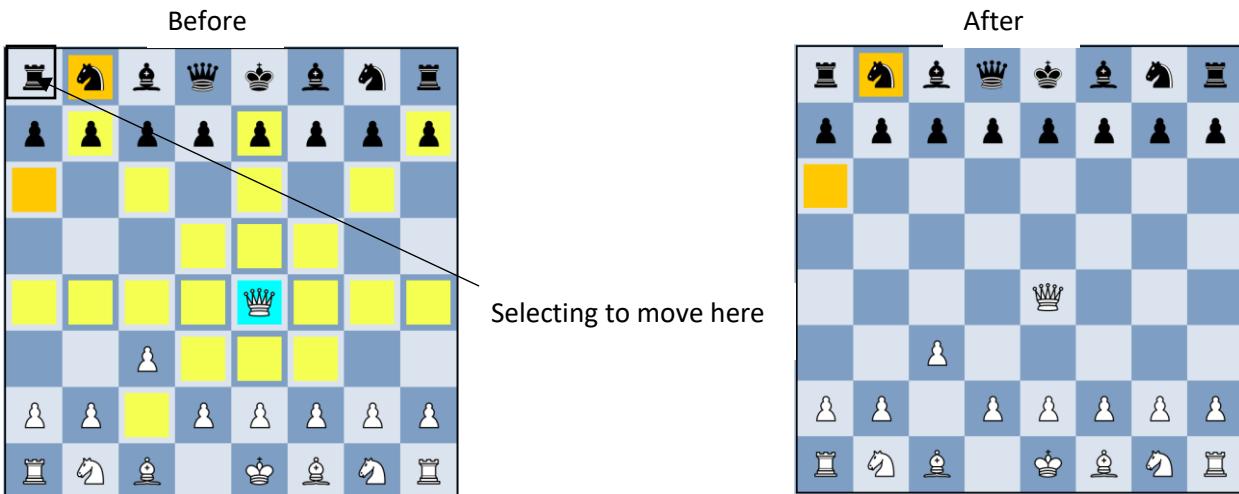


Testing if queen can move down



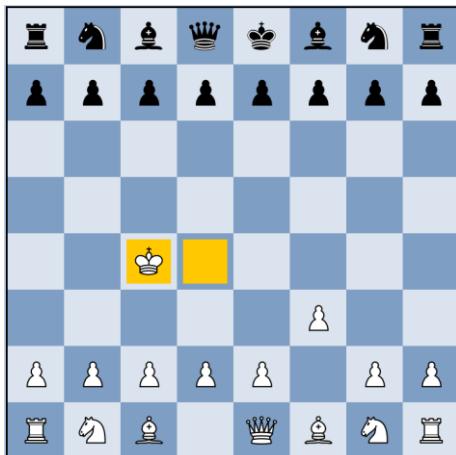
Testing if queen can move south west

Testing invalid inputs:

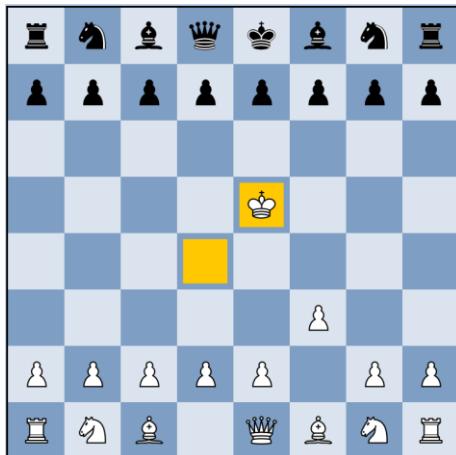


#	Test data	Expected result	Actual result	Changes needed
9	User inputs move that is not pseudo-legal for a king	Move should not be made	Move is not made	
	User inputs a move that is pseudo-legal for a king	Move should be made	Move is correctly made	

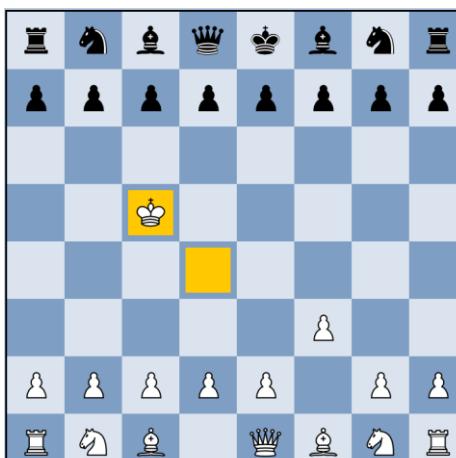
Testing valid inputs:



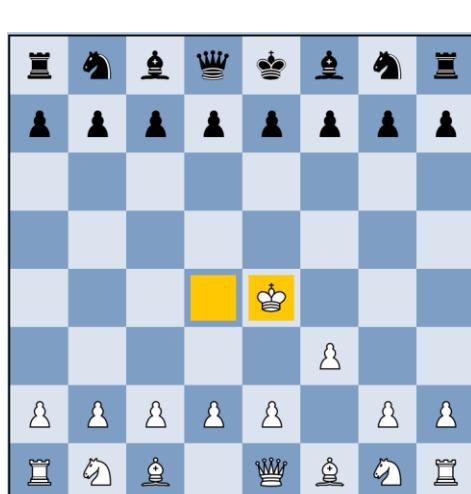
Testing if kings can move left



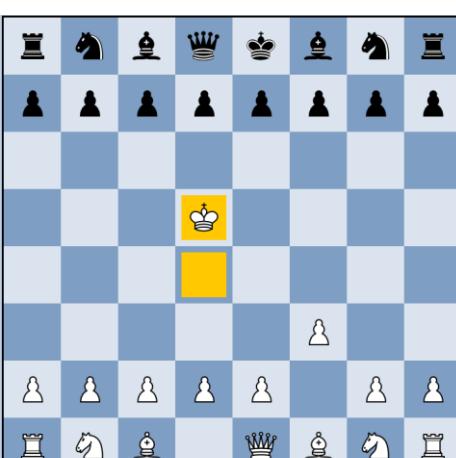
Testing if kings can move north east



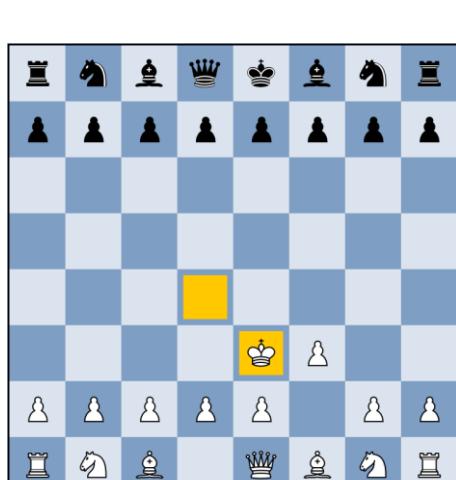
Testing if kings can move north west



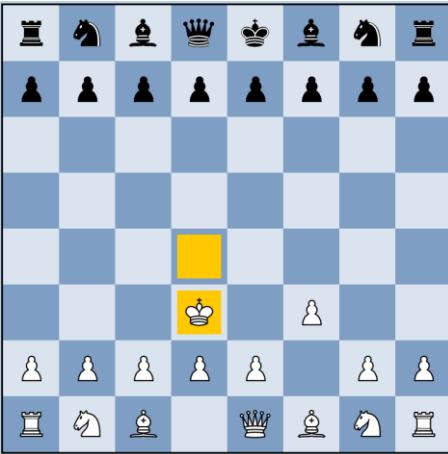
Testing if kings can move right



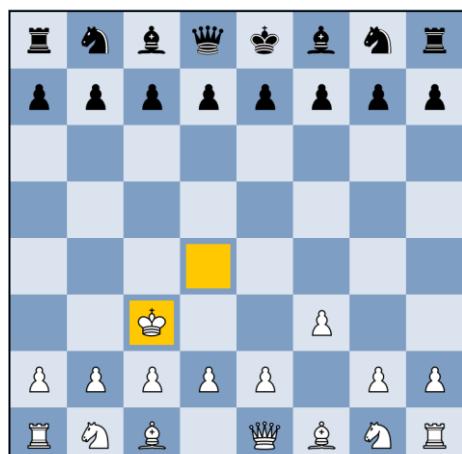
Testing if kings can move up



Testing if kings can move south east



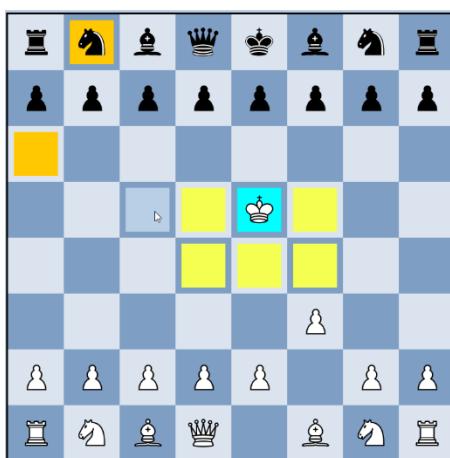
Testing if kings can move down



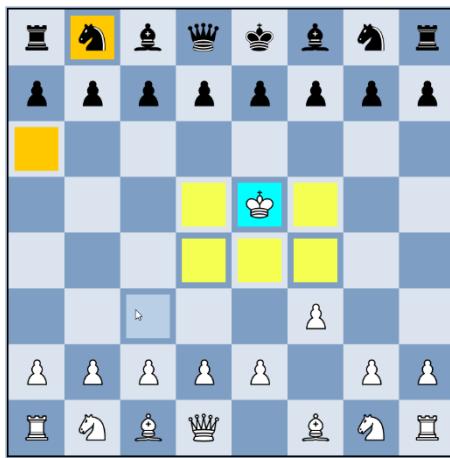
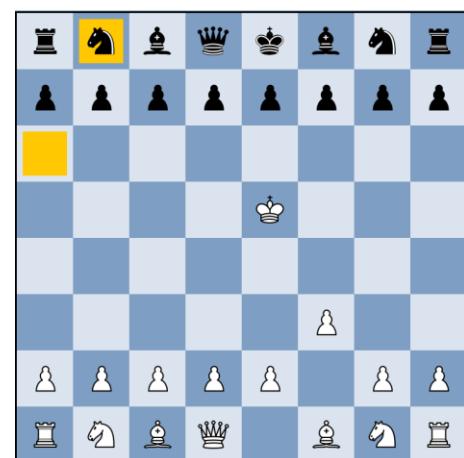
Testing if kings can move south west

Testing invalid inputs:

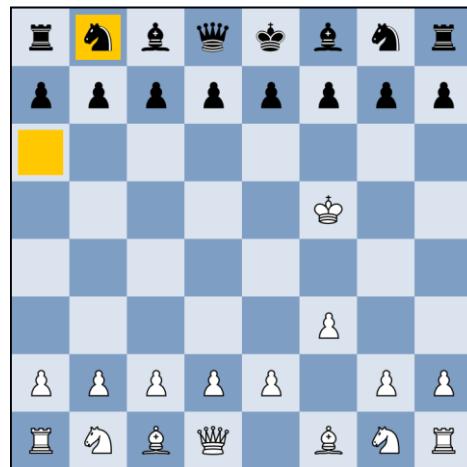
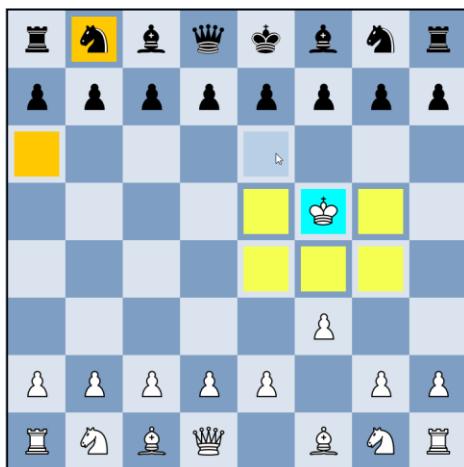
Before



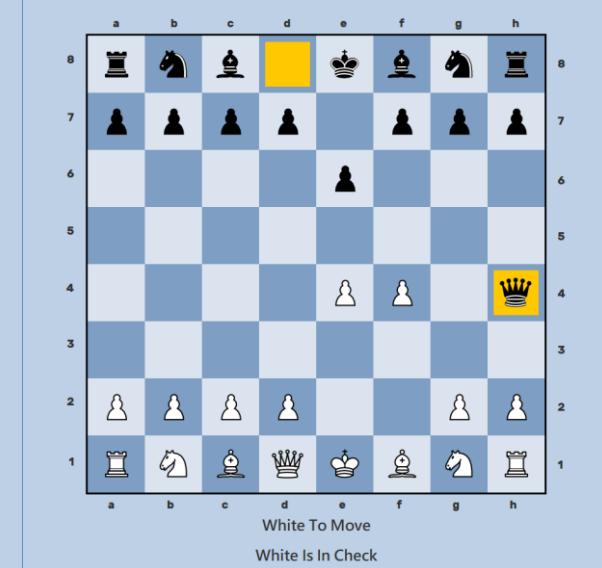
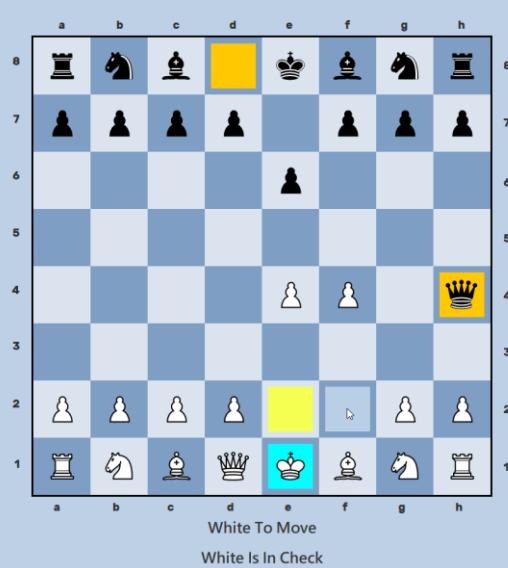
After



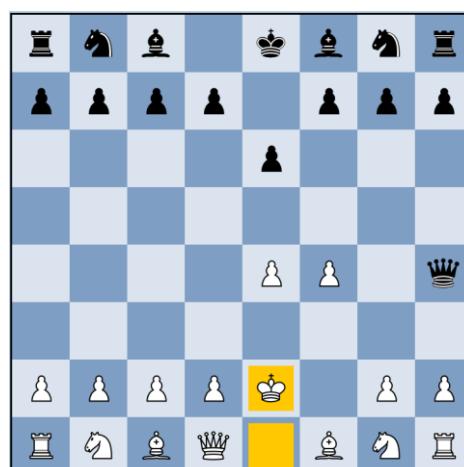
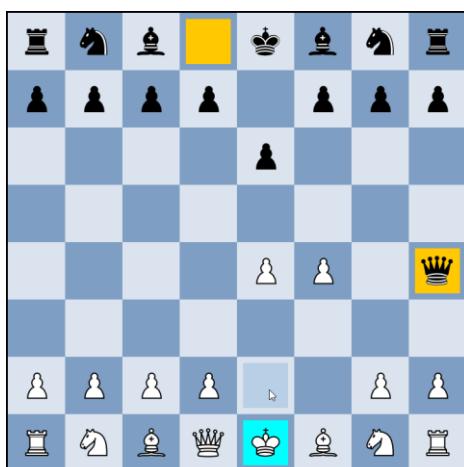
#	Test data	Expected result	Actual result	Changes needed
10	User inputs move that puts themselves into check	Move should not be made	Move is not made	
	User inputs a move that keeps themselves in check	Move should not be made	Move is not made	
	User inputs a legal move (does not put / keep themselves in check)	Move should be made	Move is made	



If the king moved to the square that was clicked, it would be under attack by two pawns, and so would be in check. Therefore, this move should not be allowed to be made. Here, the screenshots show that this move was not made, as expected.

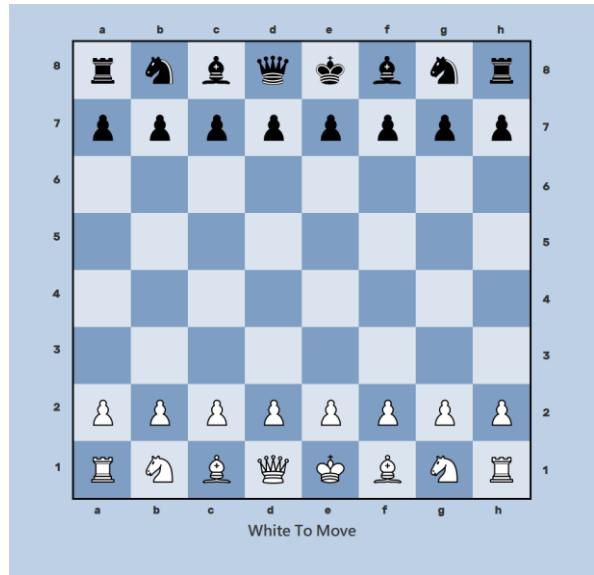
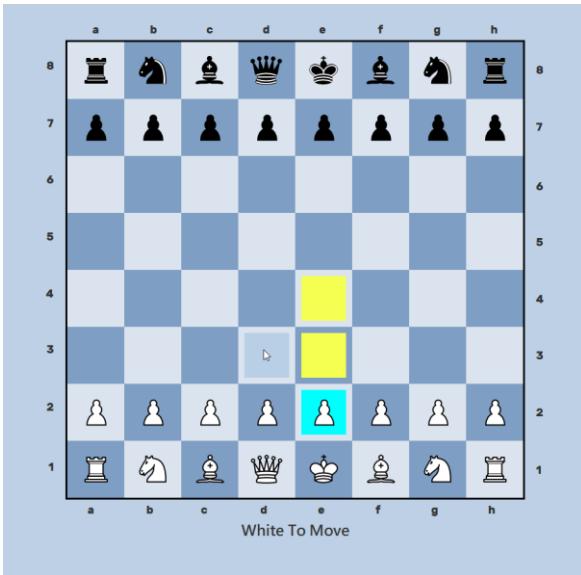


If the king moved to the square that was clicked, it would still be under attack by the queen, so would still be in check. This means that the move should not be made. The screenshots show that this happened correctly – the move was not made.

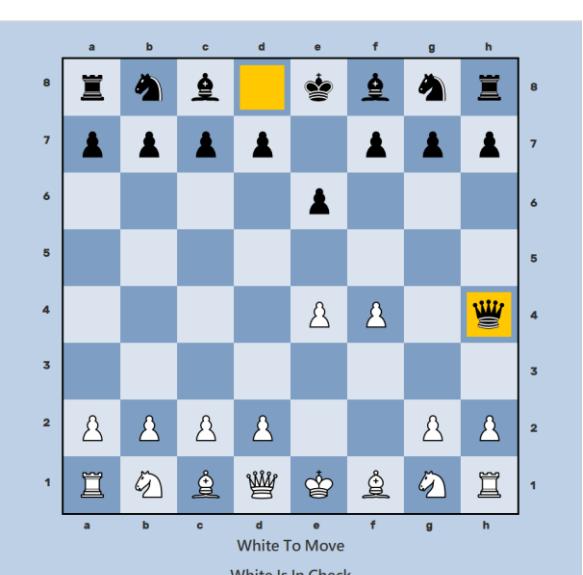
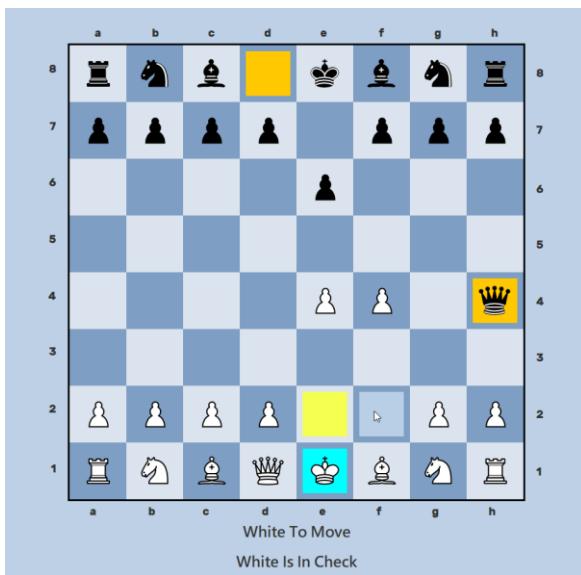


If the king moved to the square that was clicked, it would not be under attack and so would not be in check, so the move should be allowed. Here the screenshots show that the move was correctly made.

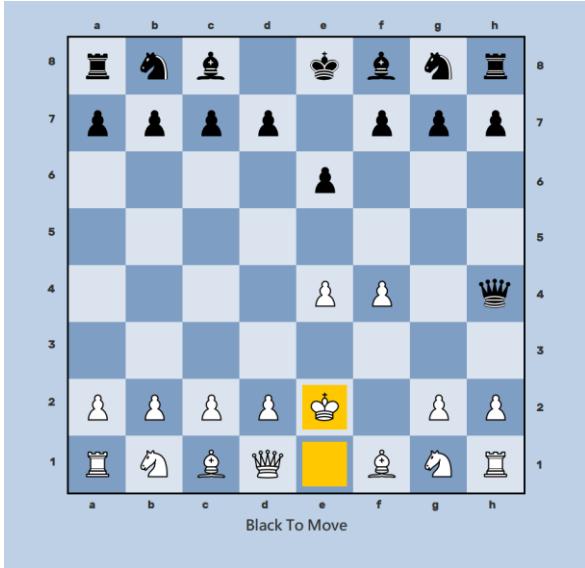
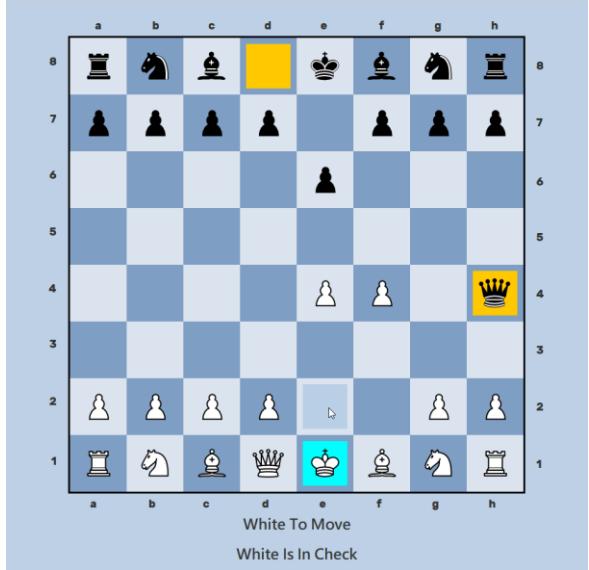
#	Test data	Expected result	Actual result	Changes needed
11	User inputs move that is not pseudo-legal	The same player should be allowed to select a different move	Same player can move again	
	User inputs a move that puts / keeps themselves in check but is pseudo-legal	The same player should be able to make a different move	Same player can move again	
	User inputs legal move	The same user should not be able to move	Same player cannot move	



After trying to make an illegal move (which is legal because it is not pseudo-legal for a pawn), the JLabel at the bottom does not change, and still shows the same user to move, so is working correctly.

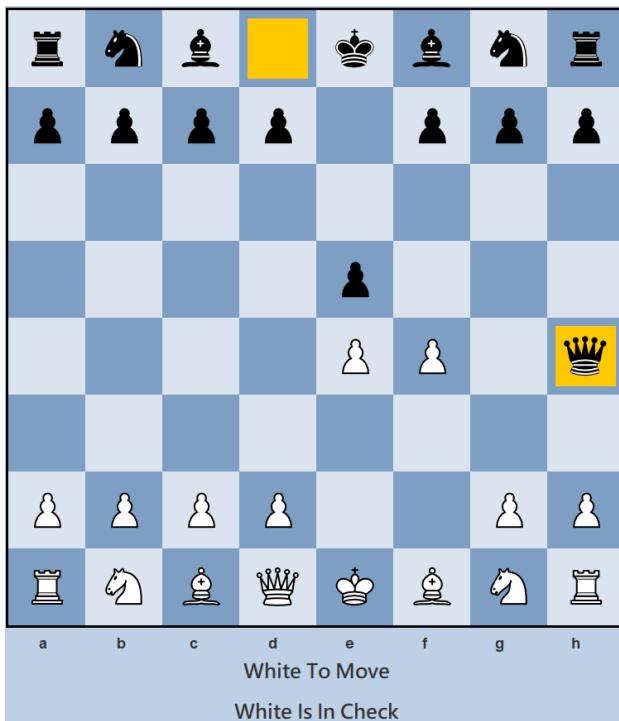


After trying to make an illegal move (which is legal because it keeps white in check), the JLabel at the bottom does not change, and still shows the same user to move, so is working correctly.

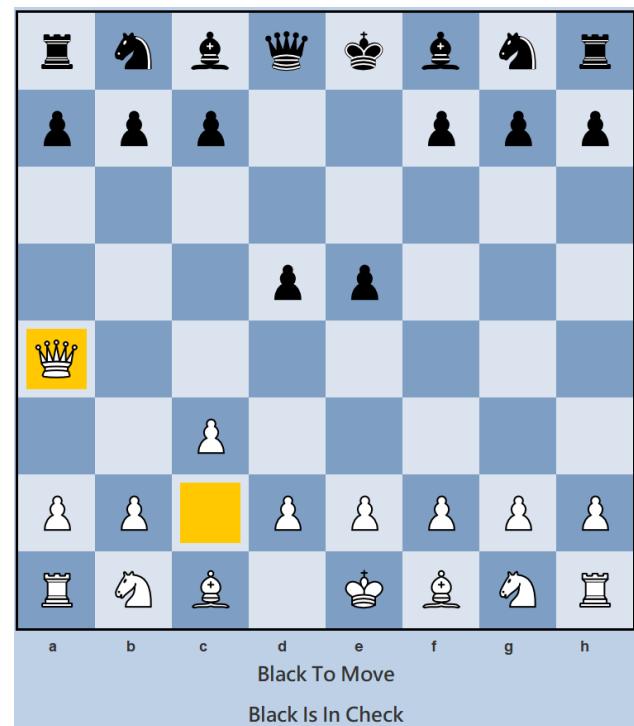


After making a legal move, the JLabel at the bottom changes correctly to show the player to move.

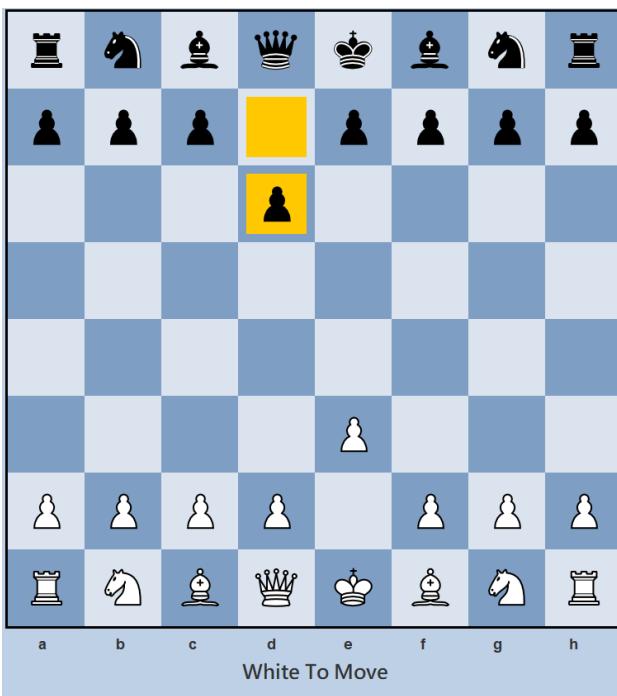
#	Test data	Expected result	Actual result	Changes needed
12	White is in check	JLabel should display text "White Is in check"	When white is in check, JLabel is visible and has text "White is in check"	
	Black is in check	JLabel should display text "Black is in check"	When black is in check, JLabel is visible and has text "Black is in check"	
	Neither player is in check	JLabel should show no text / not be visible to the user	When neither player is in check, JLabel is not visible to user.	



Here, the white king is under attack by the queen, so white is in check, and the JLabel tells this to the user correctly.

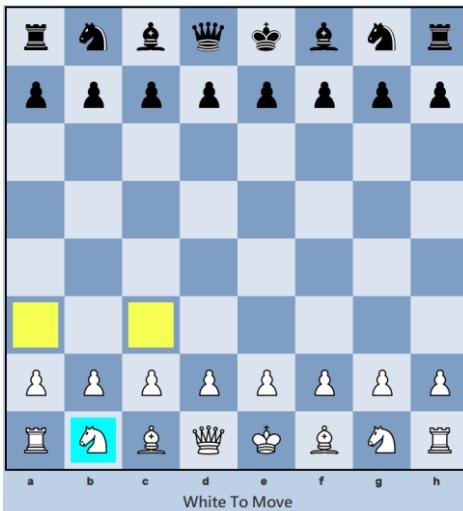


Here, the black king is under attack by the queen, so black is in check, and the JLabel tells this to the user correctly.

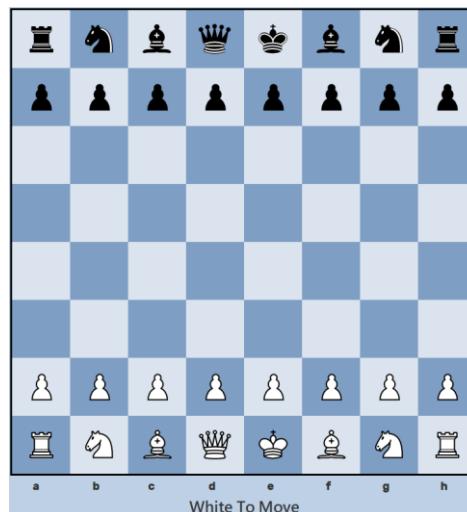
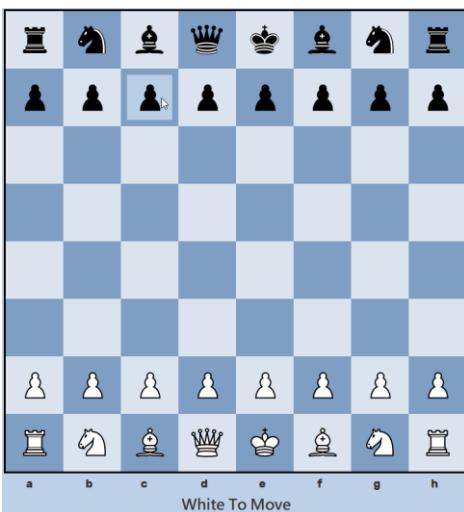


Here, no one is in check, and the JLabel is hidden as expected.

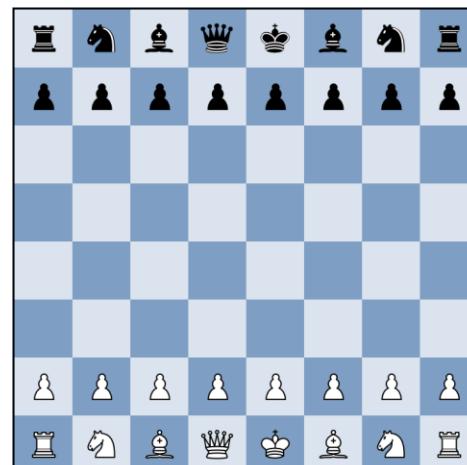
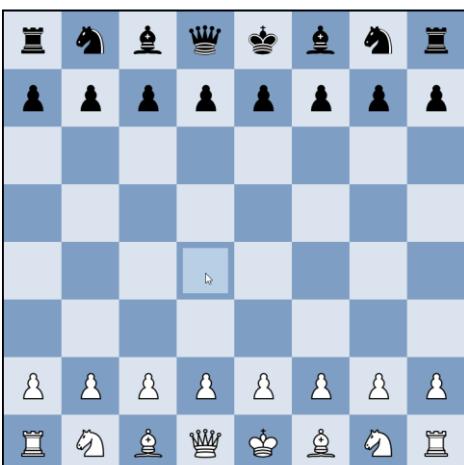
#	Test data	Expected result	Actual result	Changes needed
13	User selects piece of the same colour as the current player to move	Button background colours of piece should be changed to cyan	Background colour is correctly changed	
	User selects piece of the opposite colour as the current player to move	Button background colour of piece should not be changed	Background colour is not changed	
	User selects an empty square	Button background colour should not be changed	Background colour is not changed	



When clicking the piece at b1 when it is whites turn to move, it is correctly highlighted in cyan.

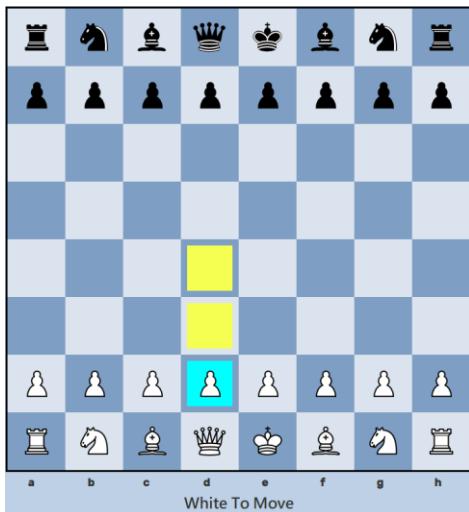


When clicking the piece at c7 when it is whites turn to move, it is not highlighted, which is expected as that piece does not belong to the current player.

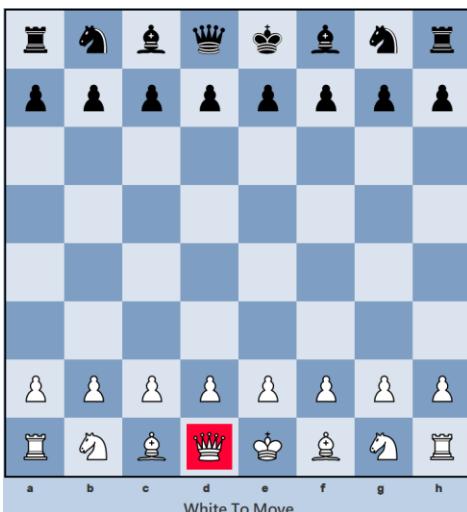


When clicking the piece at d4 when it is whites turn to move, it is not highlighted, which is expected as there is no piece there to be selected.

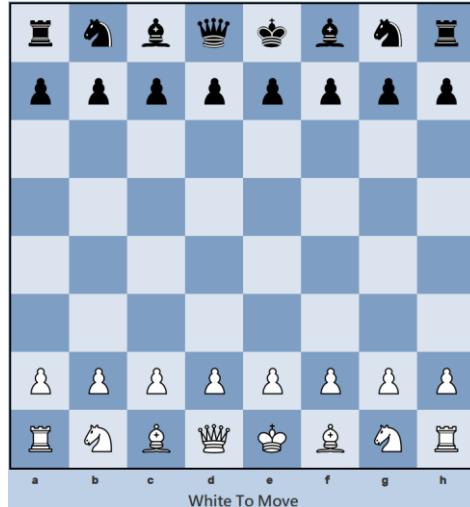
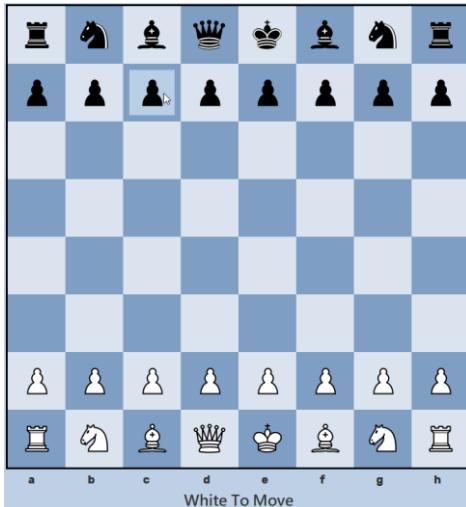
#	Test data	Expected result	Actual result	Changes needed
14	Current player selects one of their own pieces with valid moves	Should change background colour of all buttons representing each of the places the piece can move to to yellow.	Background colour of the correct buttons is changed to yellow	
	Current player selects one of their own pieces with no valid moves	Should not change the background colours of any buttons to yellow.	Background colour of no buttons is changed to yellow	
	Current player selects piece of the other colour	Should not change the background colours of any buttons to yellow.	Background colour of no buttons is changed to yellow	
	Current player selects empty square	Should not change the background colours of any buttons to yellow.	Background colour of no buttons is changed to yellow	



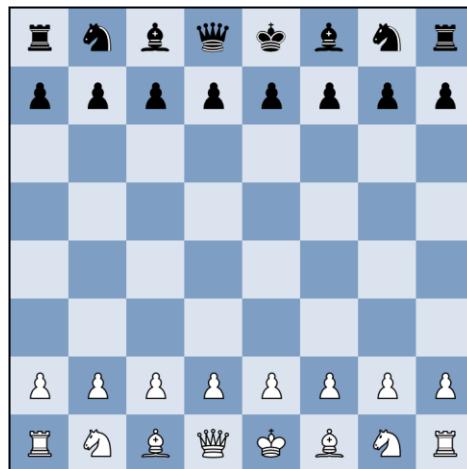
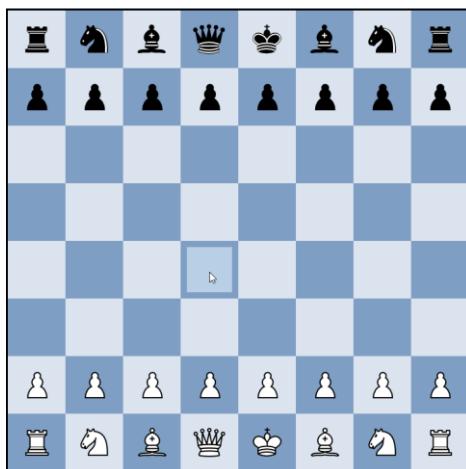
When clicking the piece at d2 when it is whites turn to move, the only two places it can move to are highlighted in yellow, and no squares that shouldn't be are highlighted.



When clicking the piece at d1 when it is whites turn to move, no background colours are changed to yellow, which is expected as this pieces currently has no legal moves it can make.

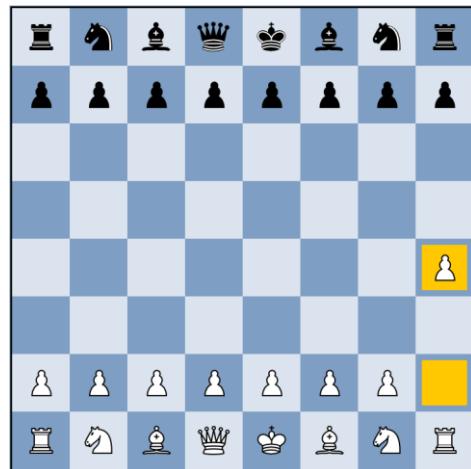
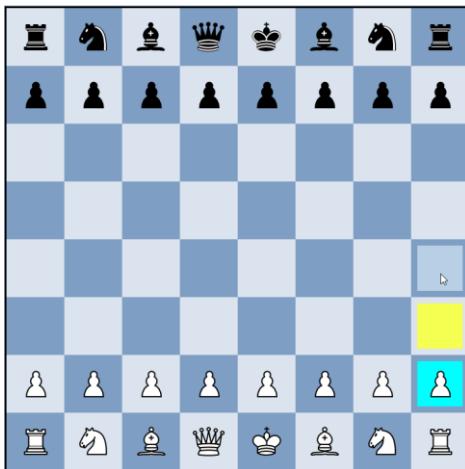


When clicking the piece at c7 when it is whites turn to move, no background colours are changed to yellow, which is expected as this piece does not belong to the current player, and so cannot be moved by them.

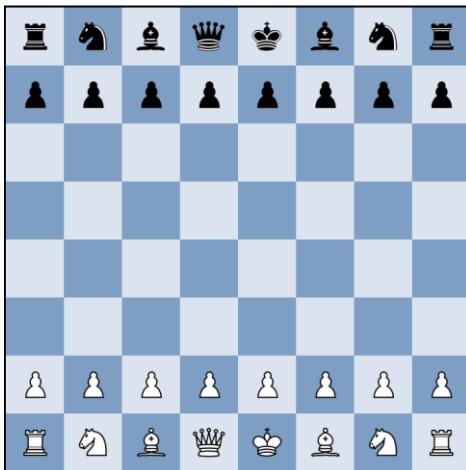


When clicking the piece at d4 when it is whites turn to move, no background colours are changed to yellow, which is expected as there is no piece here, and so it cannot have any valid moves.

#	Test data	Expected result	Actual result	Changes needed
15	Move has been made previously in the game	The background colours of the buttons at the original position of the piece and the position the piece moved to should be changed to orange	Background colour of correct buttons changed to orange	
	No moves have previously been made in the game	No buttons should have their background colour changed to orange	No button background colours changed to orange	

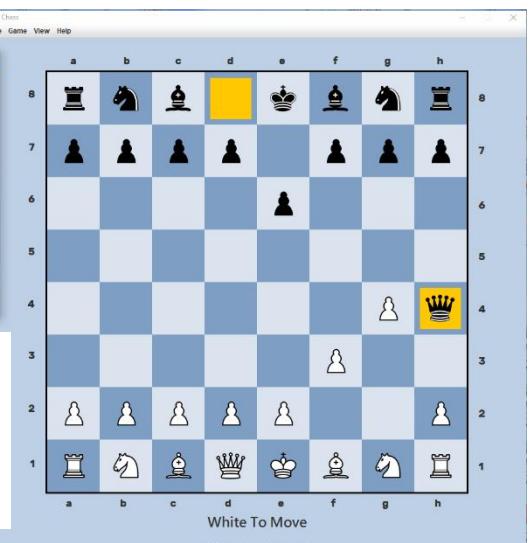
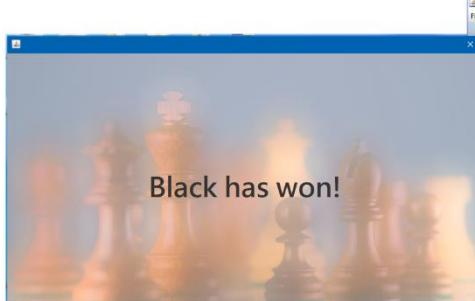
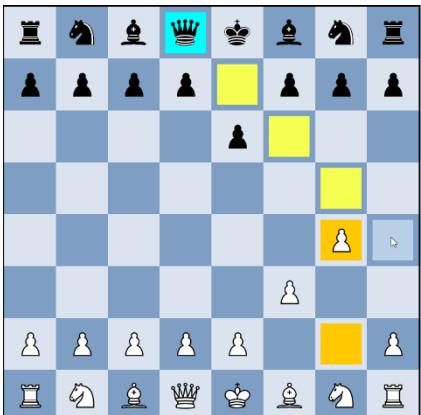


After a move had been made, the square the piece started at, and the square the piece moved to are correctly highlighted in orange.

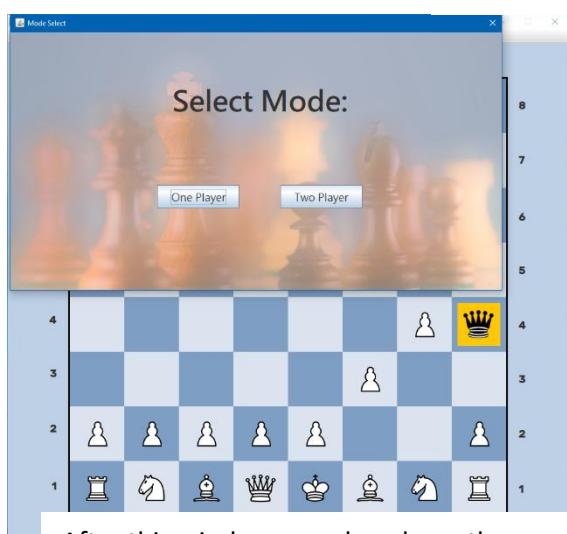


Here, no moves have been made, and so no squares are highlighted in orange.

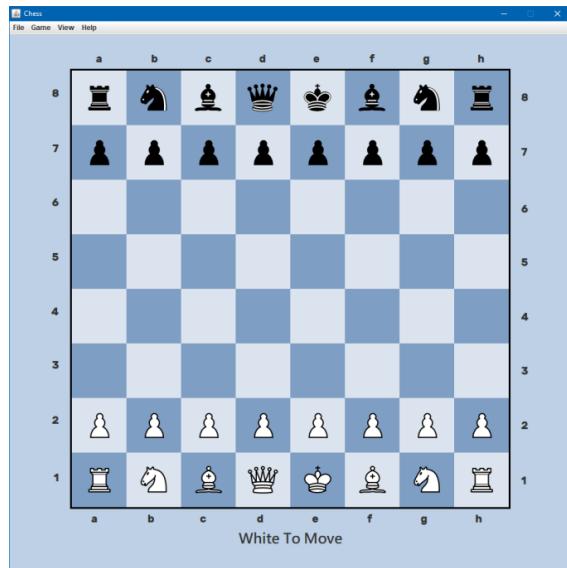
#	Test data	Expected result	Actual result	Changes needed
16	Player is in checkmate	Game should end: User should no longer be able to input moves, board should be reset	Game ends, user can no longer enter moves and after window showing user who won the game is closed, the board is reset.	
	Player is in stalemate	Game should end: User should no longer be able to input moves, board should be reset	Game ends, user can no longer enter moves and after window showing user who won the game is closed, the board is reset.	
	Player is in check but not checkmate	Game should not end: User should be able to input moves, board should not be reset	Game continues as normal	
	No players are in check or checkmate or stalemate	Game should not end: User should be able to input moves, board should not be reset	Game continues as normal	

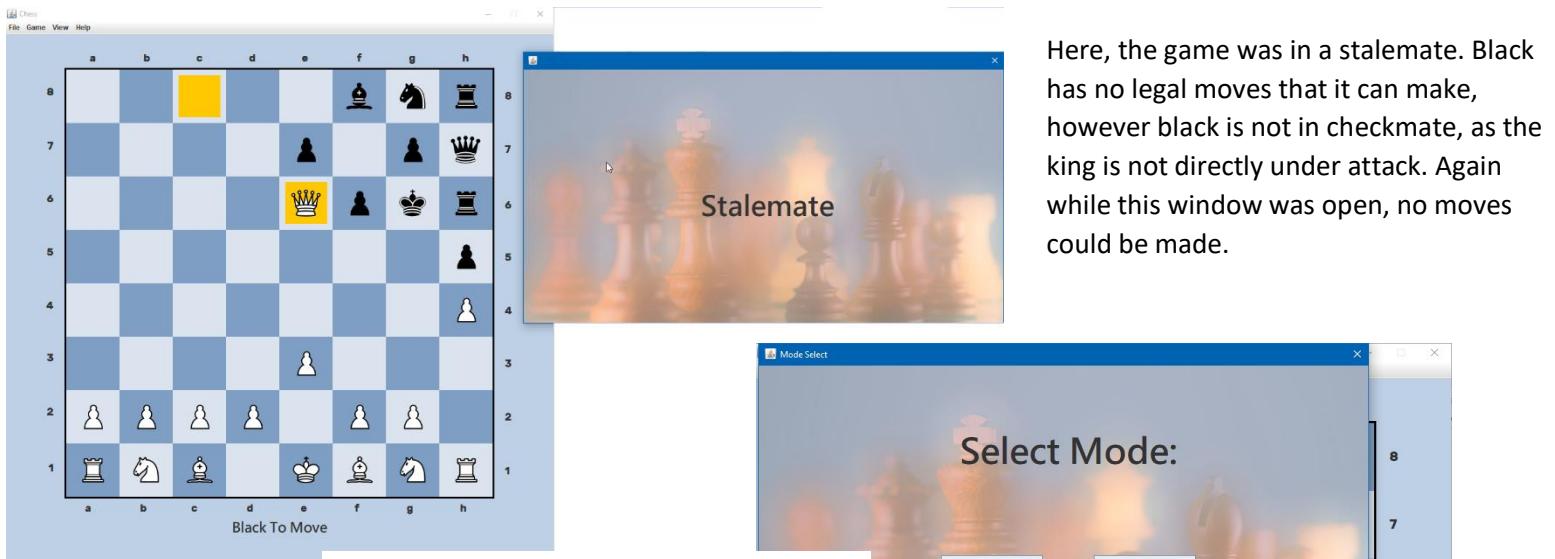


After this move was made, white was in checkmate, so game ends. Moves could not be inputted when the window showing who won was open.

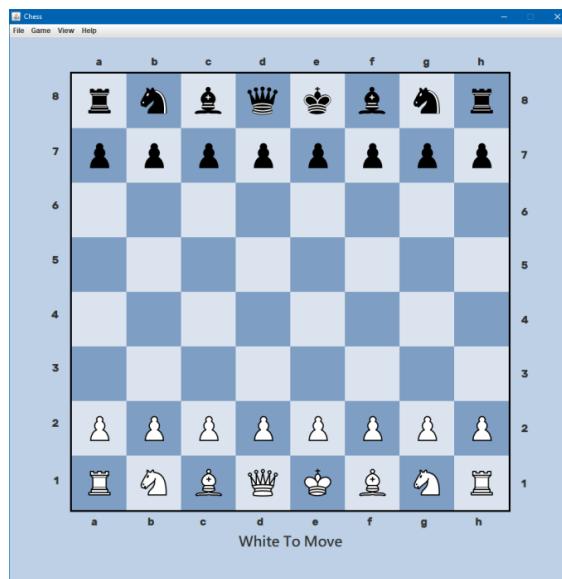
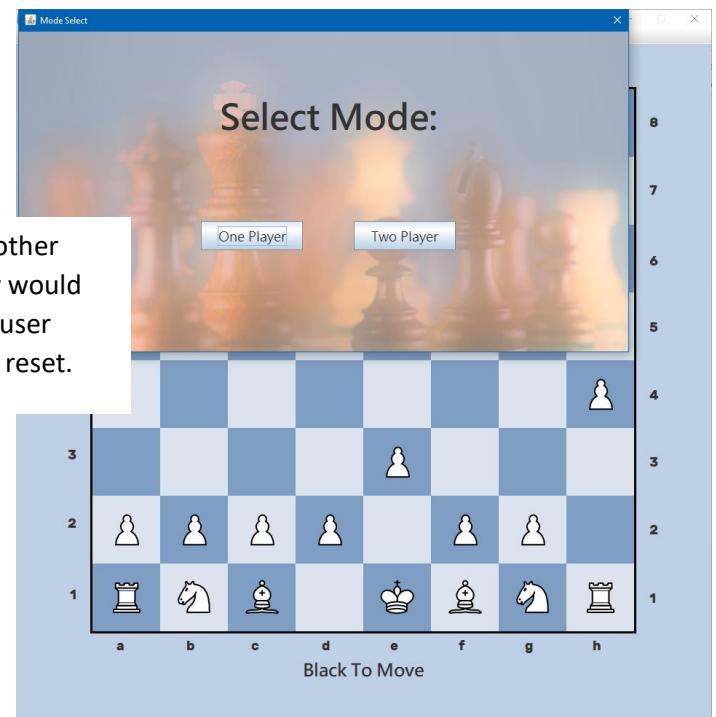


After this window was closed, another appears asking the user who they would like to play against, and after the user selects an opponent, the board is reset.

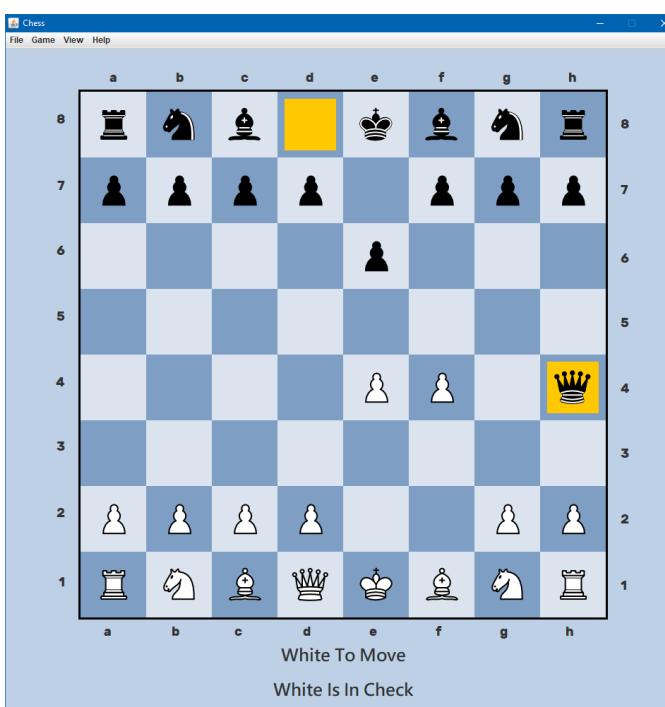


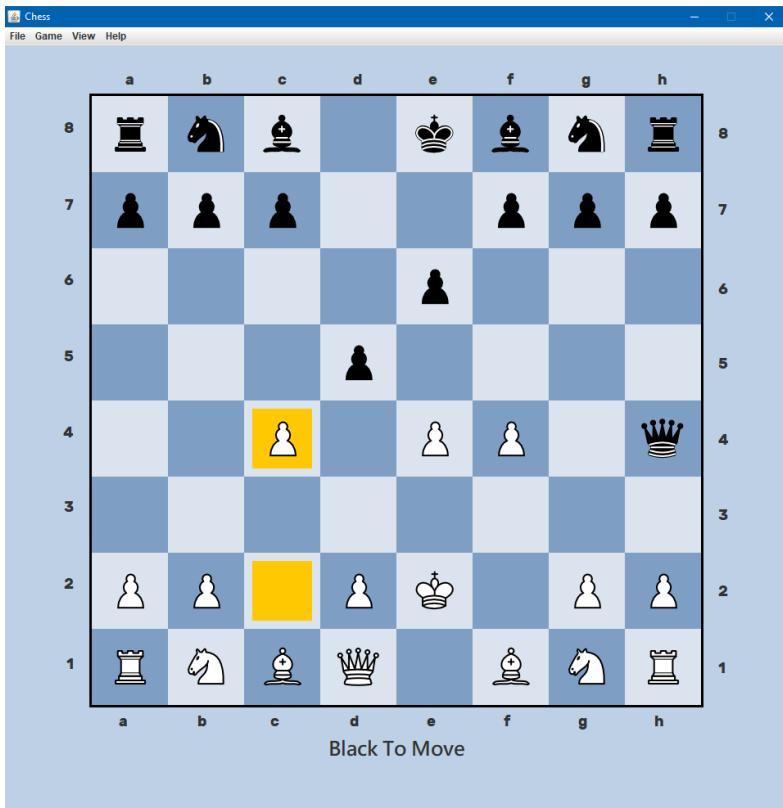


After this window was closed, another appears asking the user who they would like to play against, and after the user selects an opponent, the board is reset.



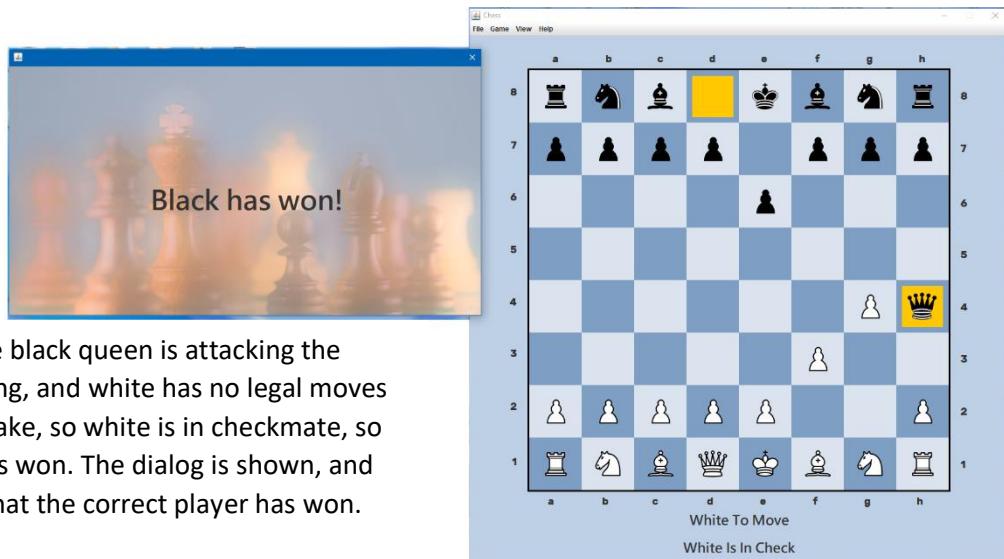
Here, white is in check, but the game does not end, so is working correctly.



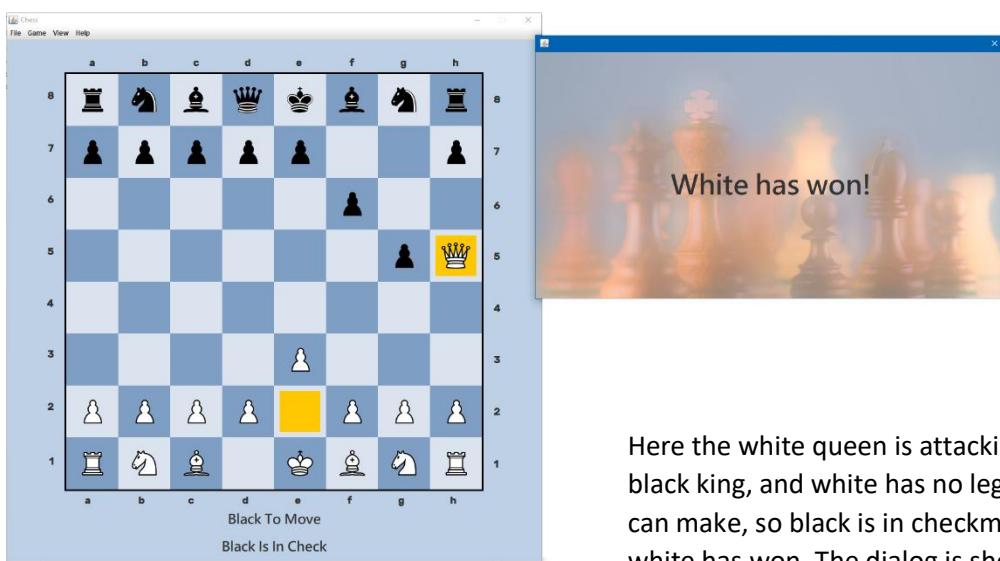


Here, no players are in checkmate / stalemate or check, and the game continues, so is working correctly

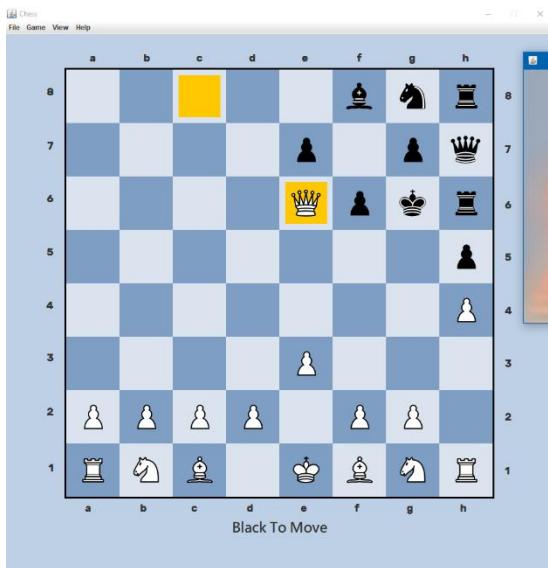
#	Test data	Expected result	Actual result	Changes needed
17	White is in checkmate	Dialog should be shown to user telling them that black has won	Dialog is shown with text "Black has won!"	
	Black is in checkmate	Dialog should be shown to user telling them that white has won	Dialog is shown with text "White has won!"	
	Player is in stalemate	Dialog should be shown telling the user that the game was a draw	Dialog is shown with text "Stalemate"	
	User surrenders game	New game should start	Dialog is shown asking user for opponent for next game	



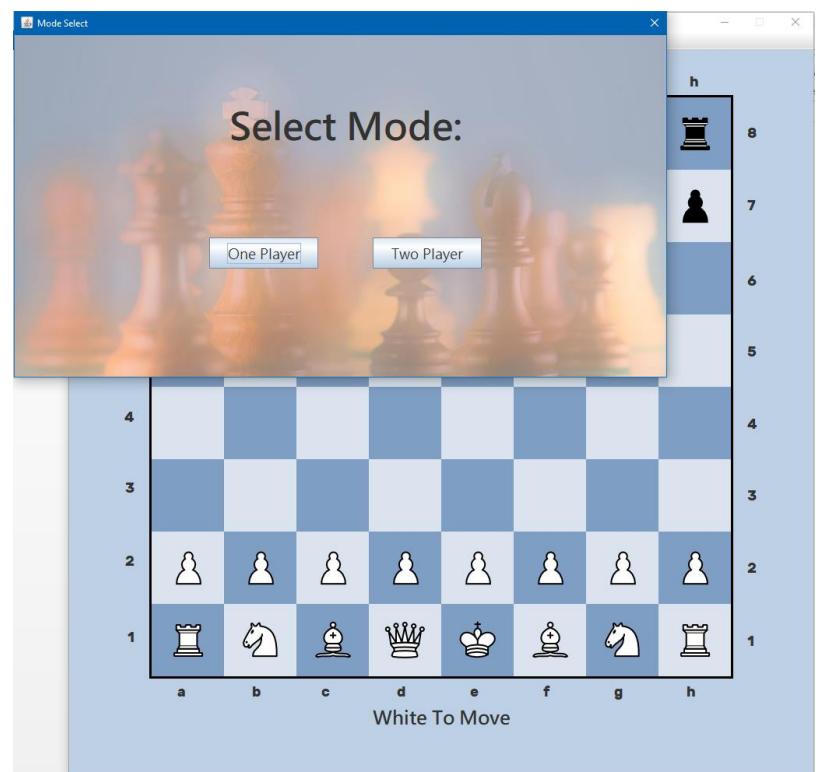
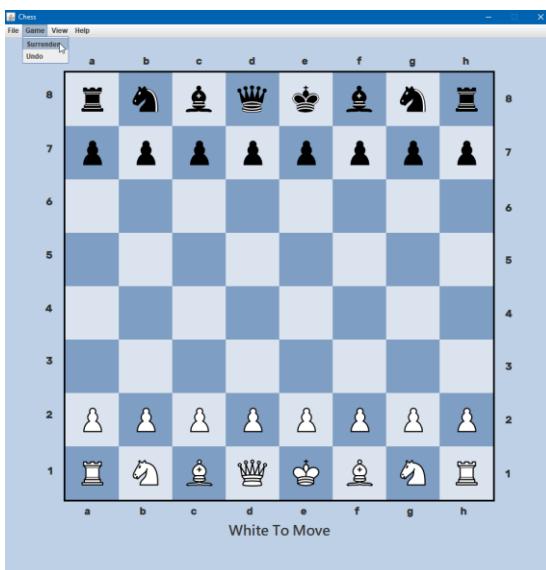
Here the black queen is attacking the white king, and white has no legal moves it can make, so white is in checkmate, so black has won. The dialog is shown, and shows that the correct player has won.



Here the white queen is attacking the black king, and black has no legal moves it can make, so black is in checkmate, so white has won. The dialog is shown, and shows that the correct player has won.

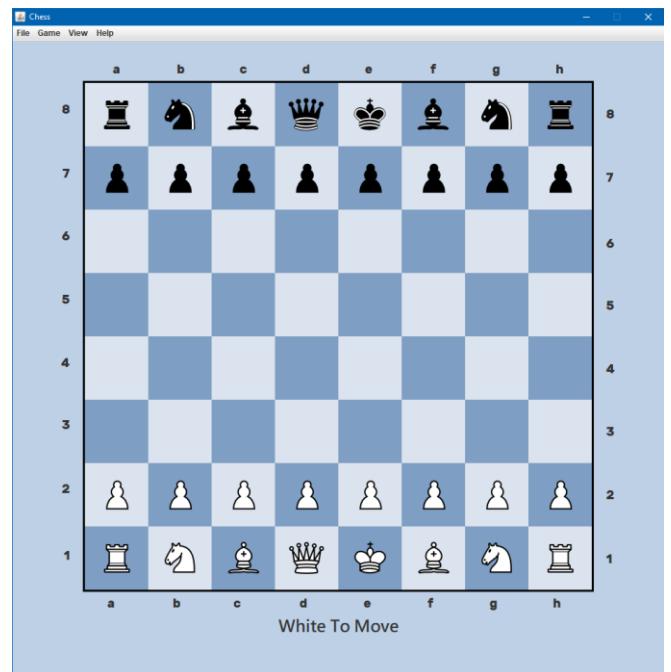
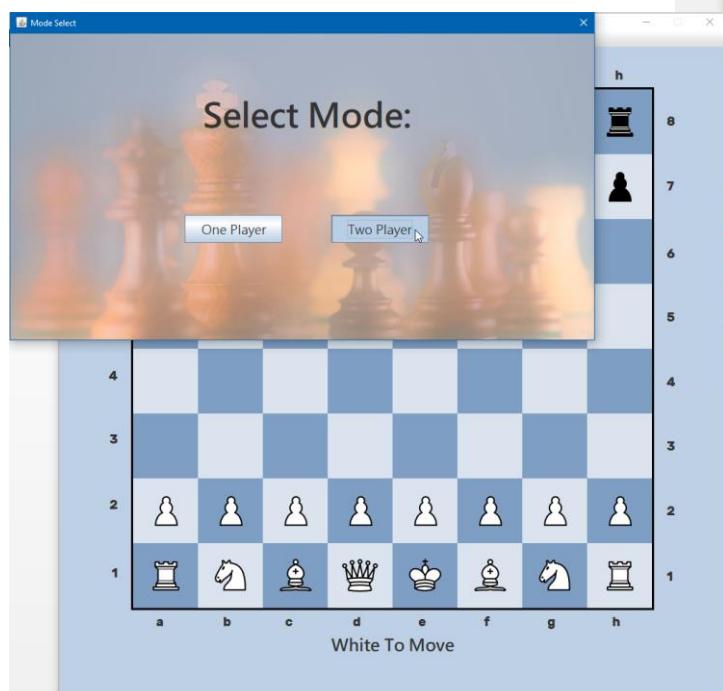
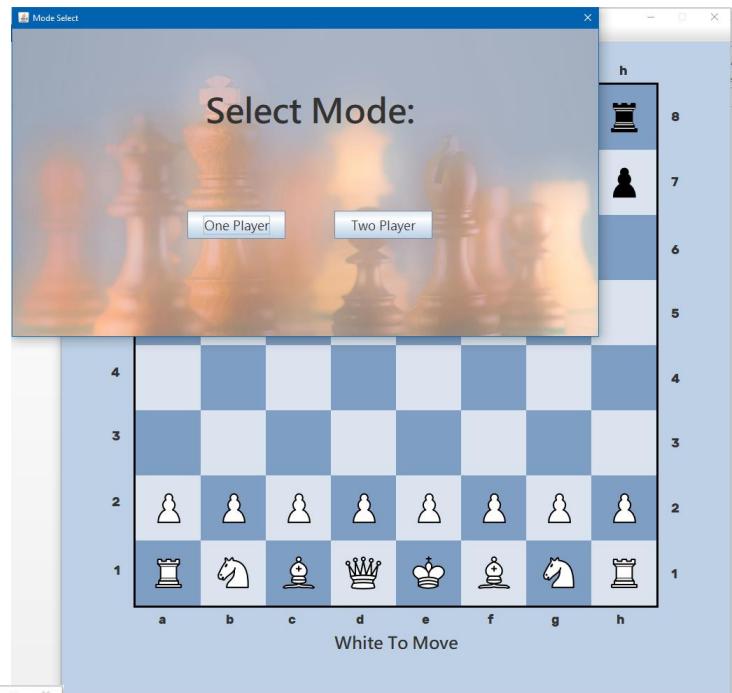
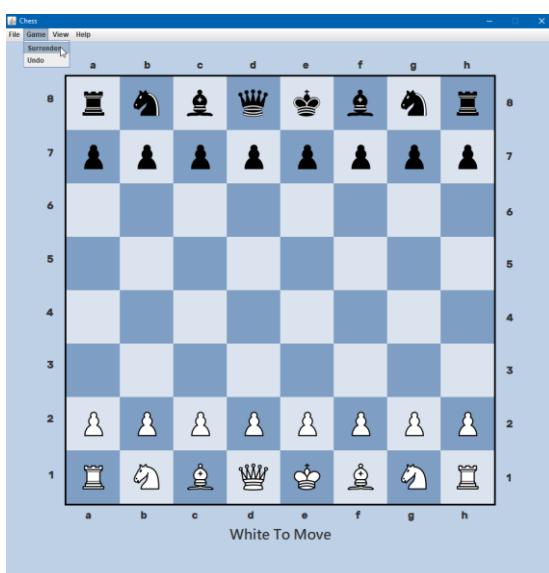


Here black has no legal moves, and yet the black king is not under attack, so it is a stalemate. This is shown correctly in the dialog.



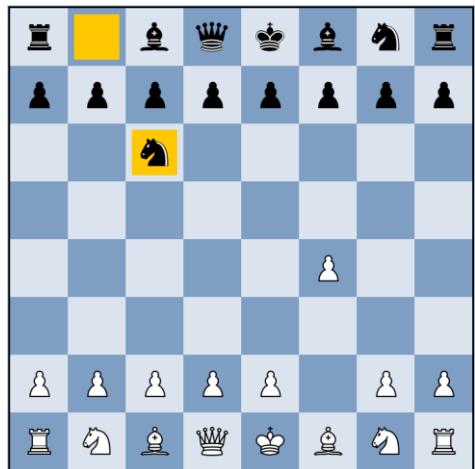
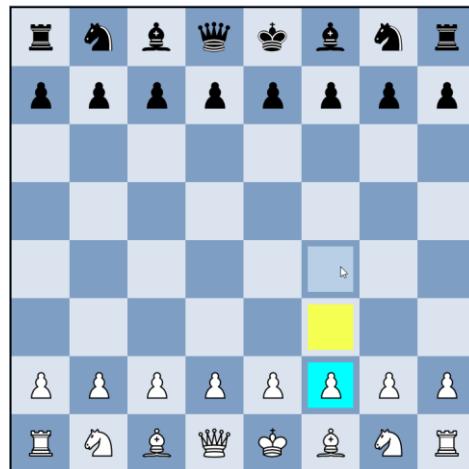
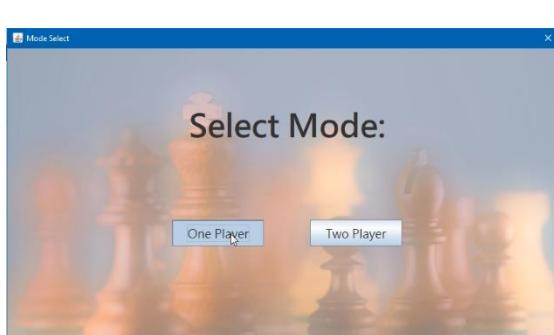
Here, the surrender button is pressed, ending the game. A window is shown asking the user who they would like to play against and then after they select an option a new game starts.

#	Test data	Expected result	Actual result	Changes needed
18	User presses the surrender menu item	Game should end, the board should be reset, and the user asked again who they would like to play against	Game ends and is reset correctly	

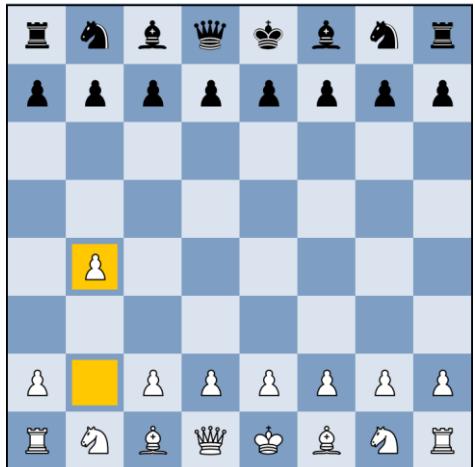
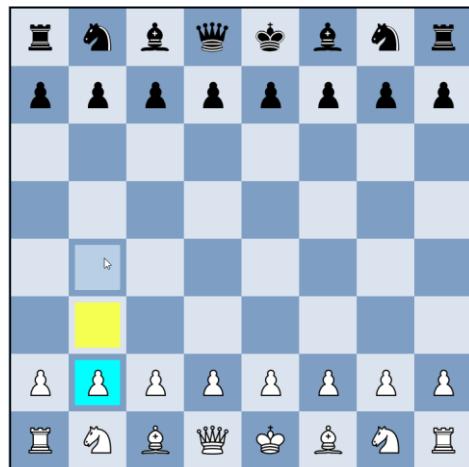
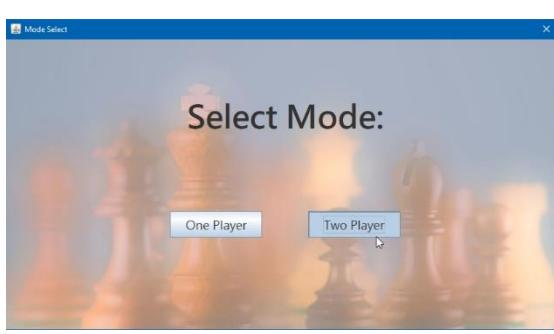


Here, the surrender button is pressed, ending the game. A window is shown asking the user who they would like to play against and then after they select an option a new game starts.

#	Test data	Expected result	Actual result	Changes needed
19	User has selected to play against the computer and has just inputted a valid move.	Computer should make a move and this should be shown to the user	Computer makes a move straight after the user inputs theirs	
	User has selected to play against another person and has just inputted a valid move	Computer should not make a move and should wait for the other player to input a move	Computer doesn't move, waits for user to make a move	

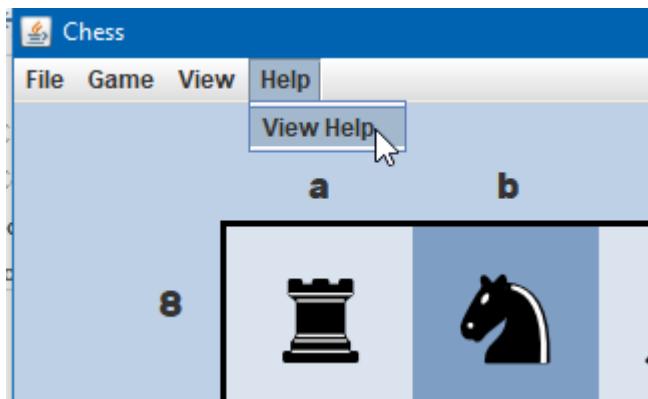


After choosing one player (i.e. to play against the computer), and making a move, the move is made, and then the computer makes its move shortly after, and so this is working correctly.

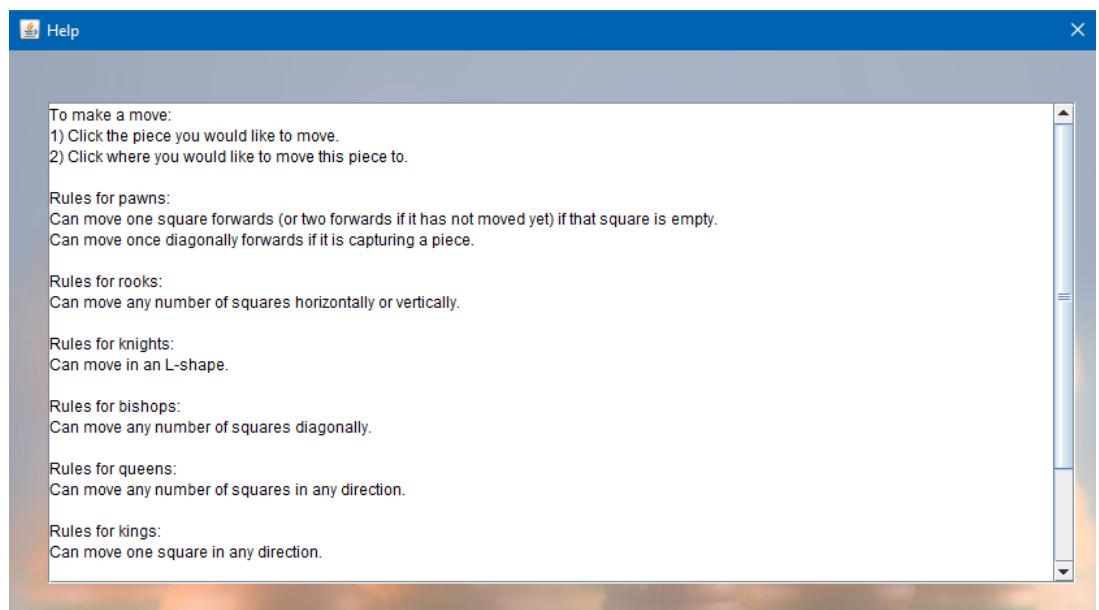


After choosing two player (i.e. to play against another human), and making a move, the move is made, and then the game waits for the other player to input their move – the computer does not take its move, and so this is also working correctly.

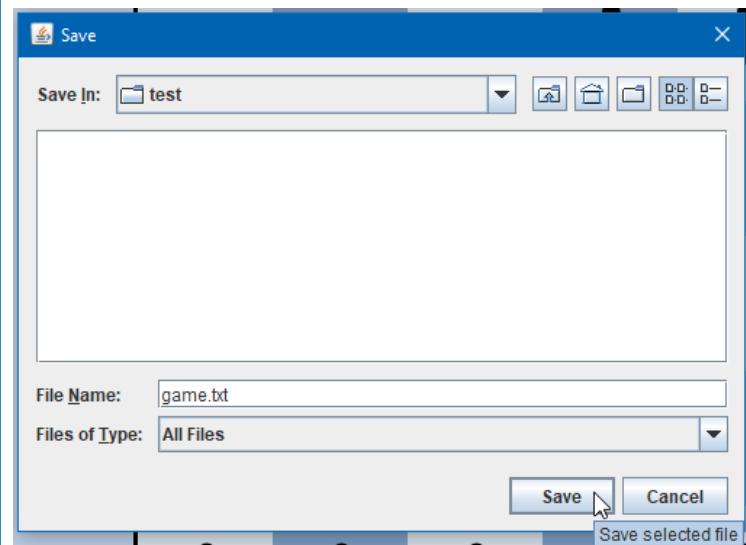
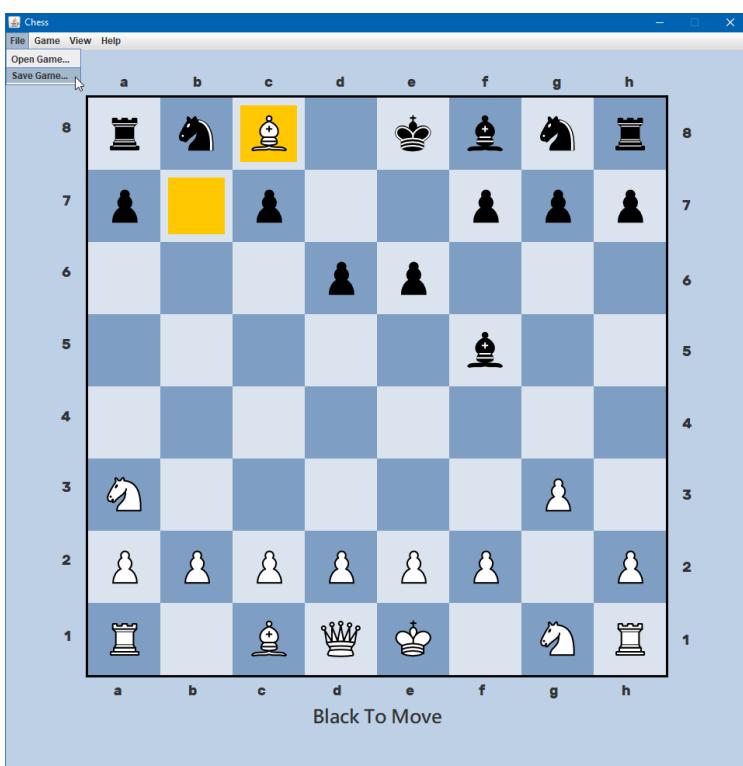
#	Test data	Expected result	Actual result	Changes needed
20	User presses help menu item	Dialog should be shown to user, showing a written explanation of the rules and how to use the program	Help window is shown	



After pressing the help menu item, this window appeared, containing a written explanation of the rules, as expected.



#	Test data	Expected result	Actual result	Changes needed
21	User presses save menu item and selects a valid file	A string representing the board should be written to the selected file	String is correctly saved to file	
	User presses save menu item and selects an invalid file	A dialog should be shown to the user informing them that they selected an invalid file and nothing should be written.	Nothing is written to file but user is not shown an error	Should display error in GUI



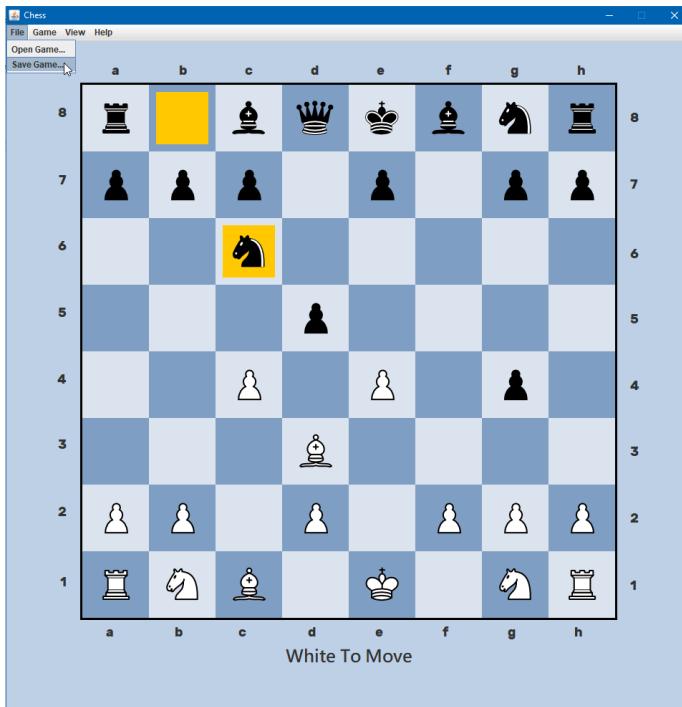
```
game.txt
1 rnbqkbnr/p1p2ppp/3pp3/5b2/8/N5P1/PPPPP1P/R1BQK1NR b
```

Here, pressing save game, and selecting a valid file that could be written to, you can see that the string representing the board is correctly written to the file, and that this string is correct, showing the layout of the pieces and whose turn it is to move at the end.

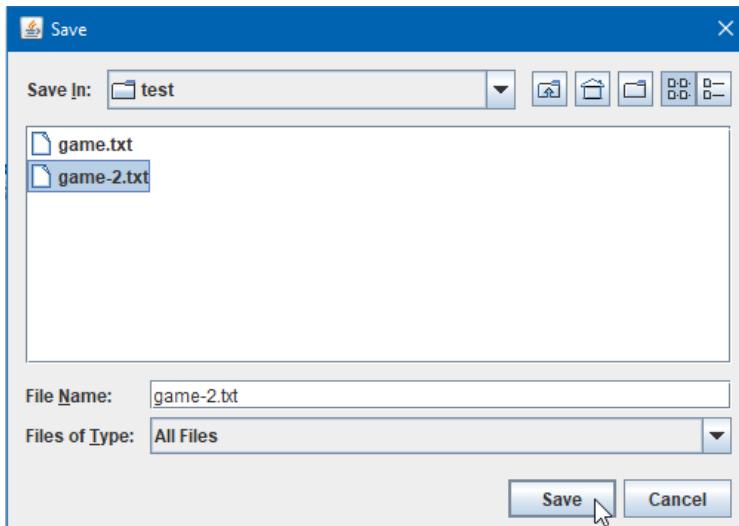
Attributes: Read-only Hidden

[Advanced...](#)

Then, to test an invalid file, I created a text file that was read only.

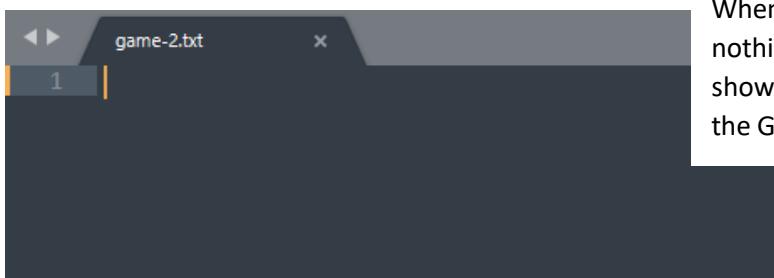


I then attempted to save the game to this file.



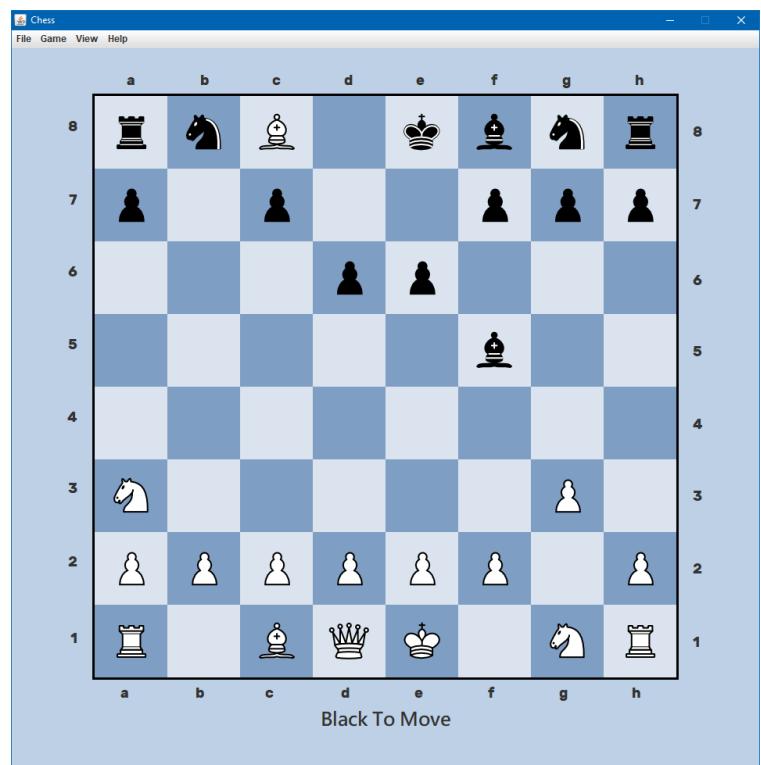
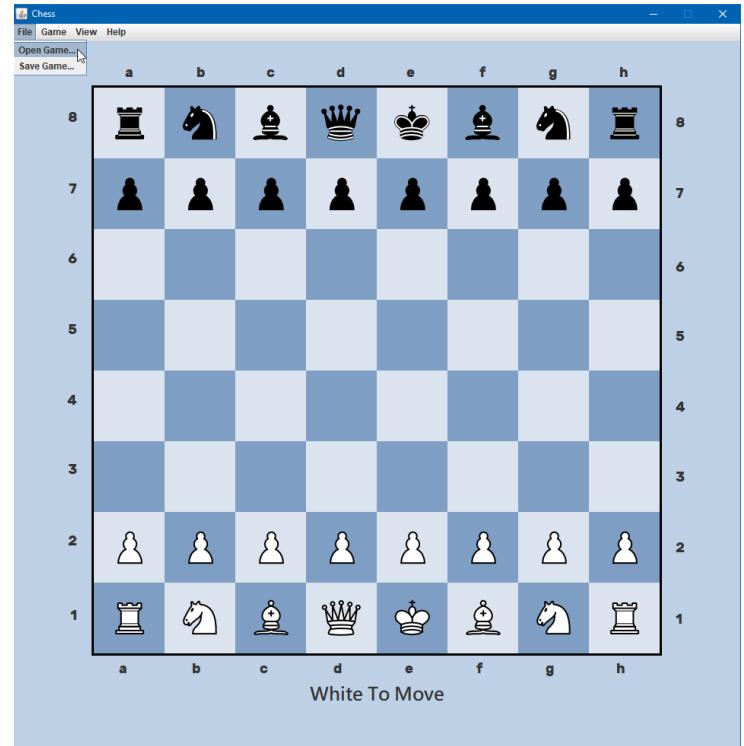
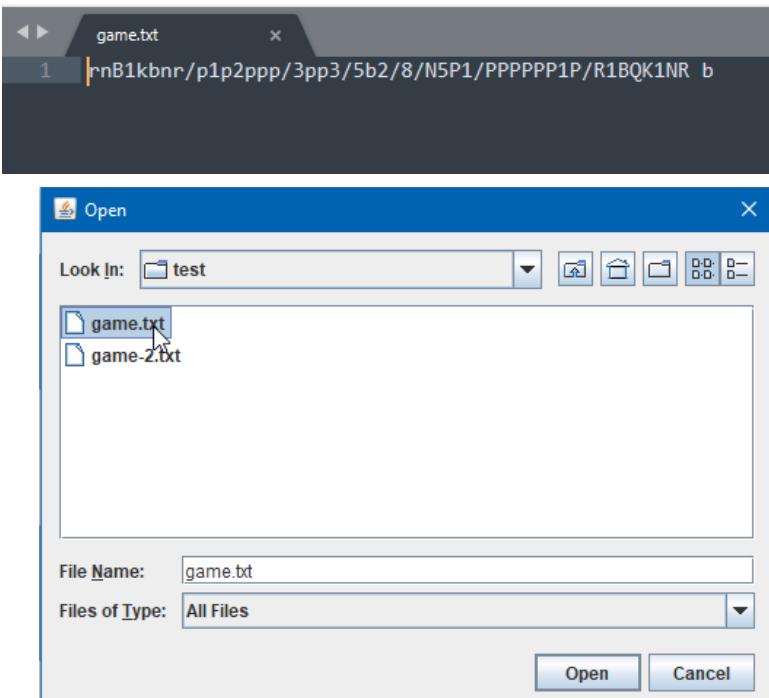
: Output - NEA (run)

```
run:  
java.io.FileNotFoundException: C:\test\game-2.txt (Access is denied)
```



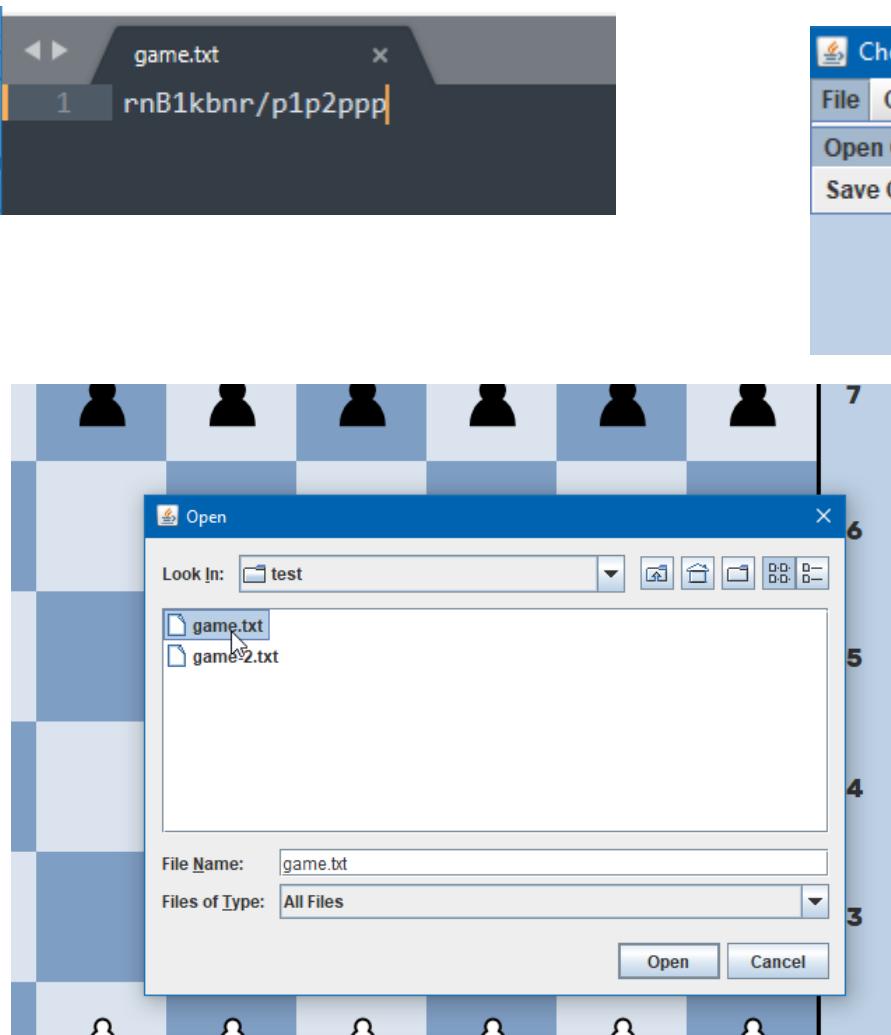
When I did this, as expected, an error occurred and nothing was written. However, whilst an error is shown in the console, there is nothing to indicate in the GUI that there was a problem saving the game.

#	Test data	Expected result	Actual result	Changes needed
22	User selects a valid file containing a string in the correct format	Game should be loaded from file	Game is loaded	
	User selects a valid file that does not contain a string in the correct format	Game should not be loaded and error shown	Not loaded, no error	Should show error
	User does not select a file	Nothing should happen	Not loaded	

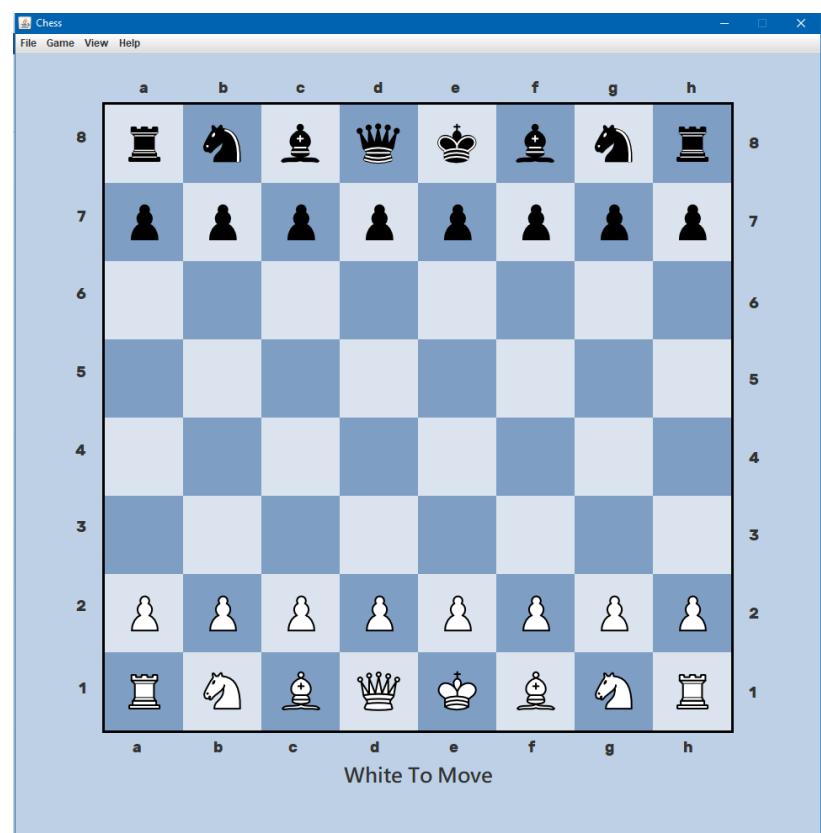


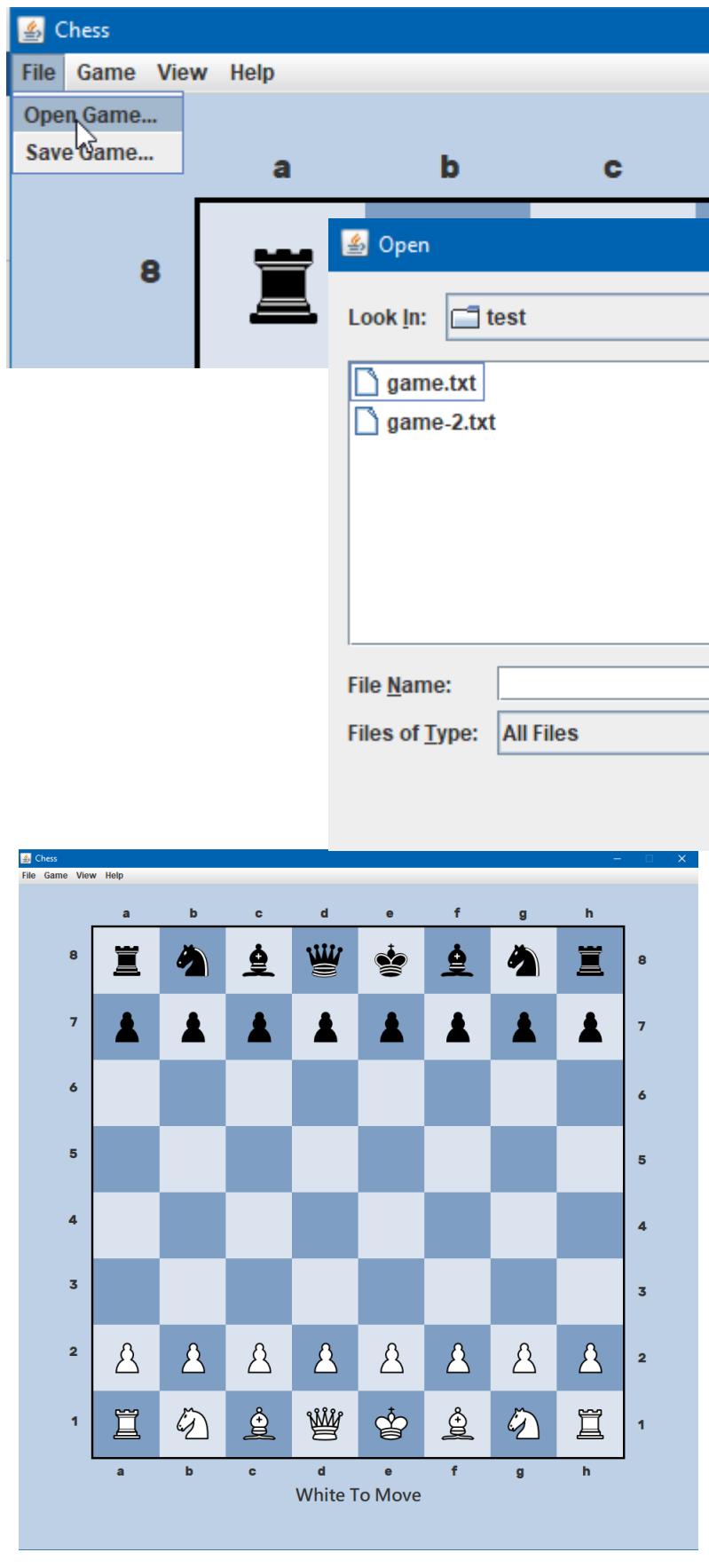
Here, when clicking “open game” the window asking user to select a file is correctly shown.

When selecting the above file, the game was loaded correctly, with all of the pieces in the correct positions, and the correct player to move.



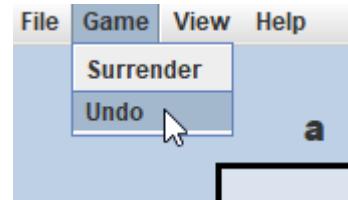
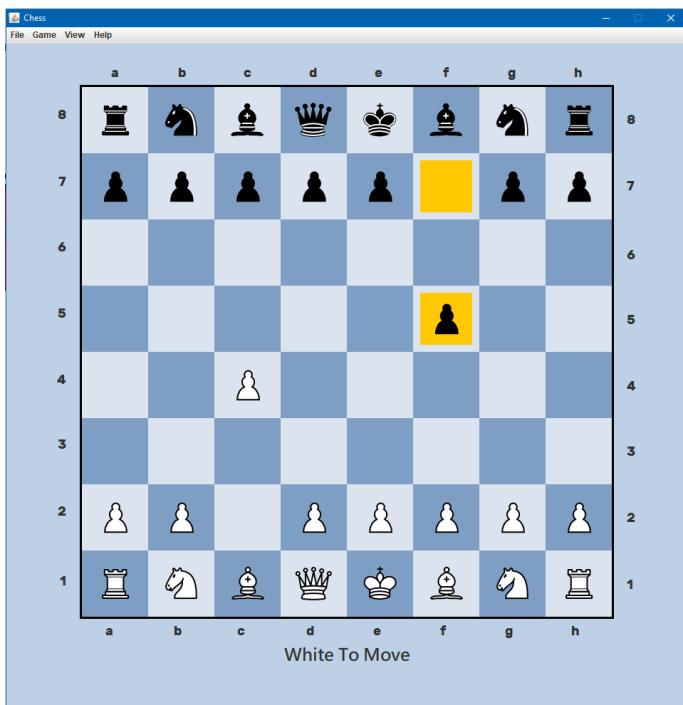
Here, I changed the file that had been previously saved so that it no longer contained a valid string, and tried to open it. It was not attempted to be loaded, as expected, however it also did not show an error to the user, either in the GUI or the console.



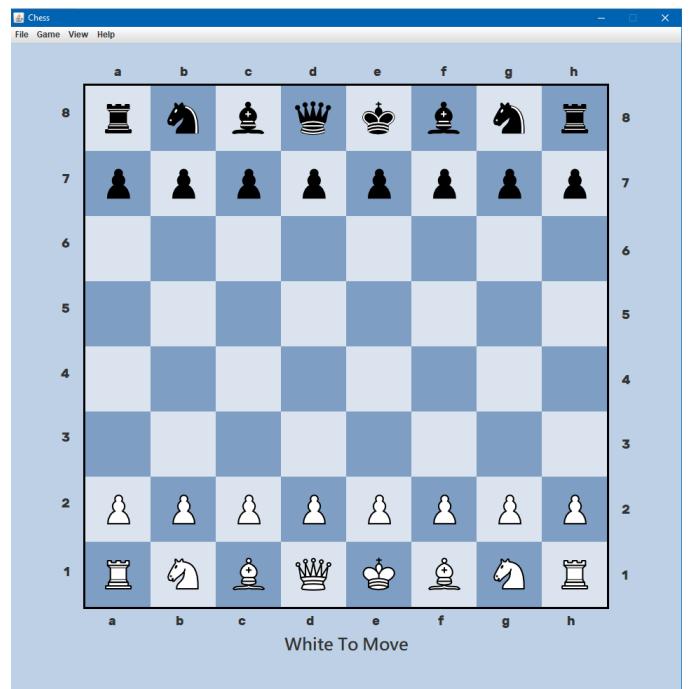


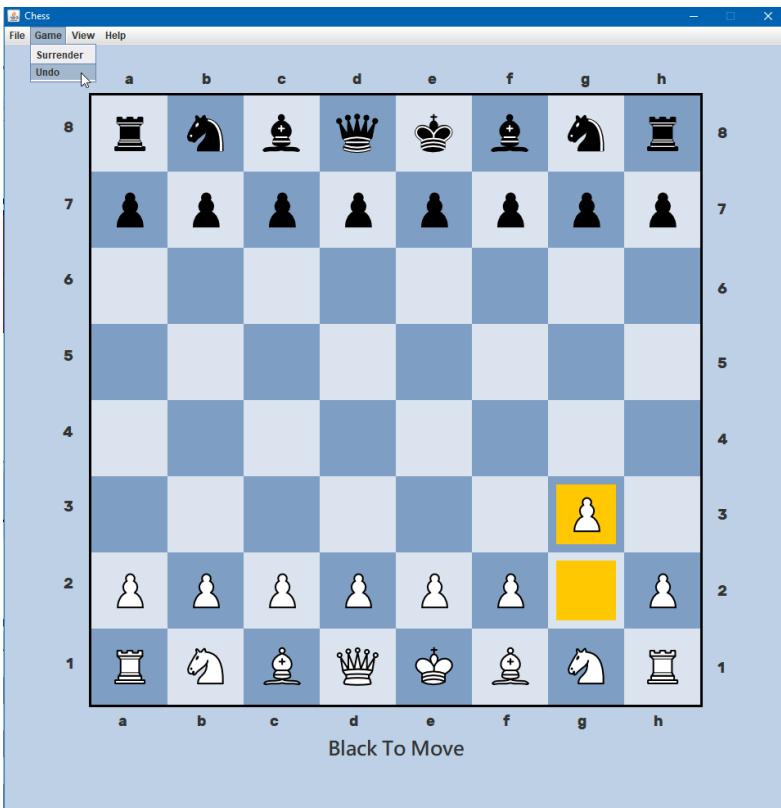
Here, I didn't select a file, and just closed the window. No error occurred, and nothing was attempted to be loaded into the game, as expected.

#	Test data	Expected result	Actual result	Changes needed
23	User presses undo menu item and more than two moves have been made so far in the game	The last two moves that were made should be undone	Last two moves were undone	
	User presses undo menu item and less than two moves have been made so far in the game	No moves should be undone	No moves were undone	

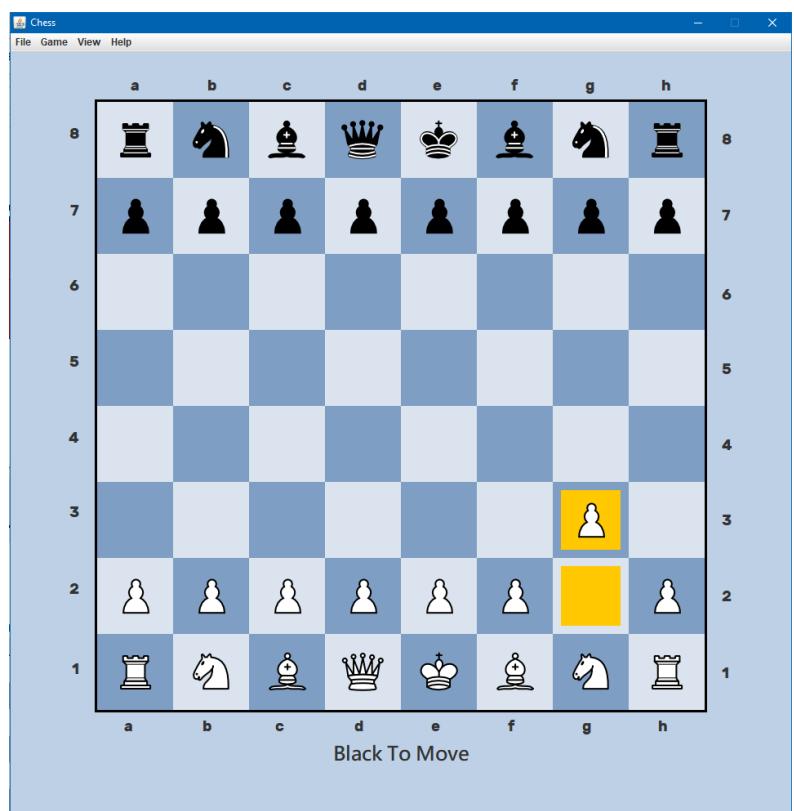


Here, 2 moves have been made, and it is white's turn to move. When I press the undo button, it correctly undos the two moves that had been made. The reason two moves are undone rather than just one is because a player should press the undo button to undo the last move they made, and not just the last move their opponent had made.





Here, less than two moves are made,
and so no moves are undone, as
expected.



Beta Testing

Now that I have tested the functionality of the program, I can begin to test how easy to use the program is for the end users, which will allow me to later determine how successful the various usability features in this program are.

As it would be difficult for me to determine whether the solution is easy to use for the end users, and whether the program appeals to them, I will use a survey to gather feedback from them. This will allow me to determine the degree to which the usability features were successful, and whether any features need to be added or changed or removed, in order for the solution to appeal to the end users / be easier for them to use.

The usability features that have been included in the program are: highlighting the selected piece in another colour, highlighting the places the piece can move to in another colour, highlighting the previous move in another colour, highlighting piece in another colour if it has no legal moves, JLabels informing the user of whose turn it is, JLabels informing the user who (if anyone) is in check, not allowing illegal moves to be made, not allowing the wrong side to move and a help window displaying instructions on how to use the program & the rules of chess.

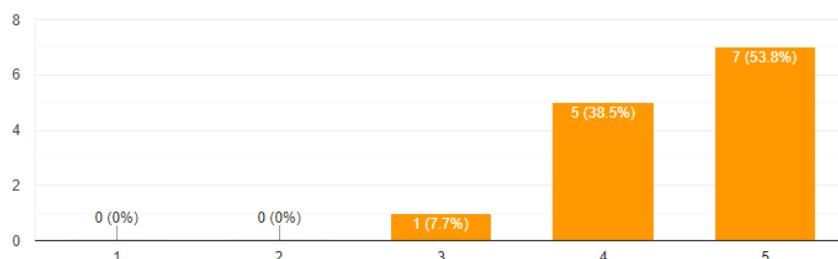
These are the questions which I plan to ask the beta-testers in the survey, after allowing them to try out the program, in order to determine how usable the program is, and whether there are any other errors which have not been found through black box testing.

- 1) How easy did you find the program to use? (1-5)
- 2) Are there any features which could be added that you think would make the program easier to use?
- 3) Was the piece you selected being highlighted in blue a useful feature? (Y/N/Don't know)
- 4) Was the piece you selected being highlighted in red if it had no legal moves a useful feature? (Y/N/Don't know)
- 5) Were the places the selected piece can move to being highlighted in yellow a useful feature? (Y/N/Don't know)
- 6) Was the previous move made being highlighted in orange a useful feature? (Y/N/Don't know)
- 7) Was the text telling you whose turn it was to move a useful feature? (Y/N/Don't know)
- 8) Was the text telling you who was in check a useful feature? (Y/N/Don't know)
- 9) Was the help window containing instruction on how to use the program a useful feature? (Y/N/Don't know)
- 10) What did you think of the design of the GUI?
- 11) Did you think the GUI had a minimalistic design?
- 12) Did you encounter any errors whilst using this program?
- 13) Any other comments on the program?

Below are the results from this survey that I sent to some of the end users:

How easy did you find the program to use?

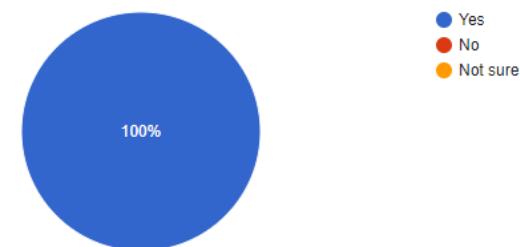
13 responses



From this I can see that the majority of users found the program easy to use, however there could be improvements to make it even easier to use.

Was highlighting in blue the piece you selected a useful feature?

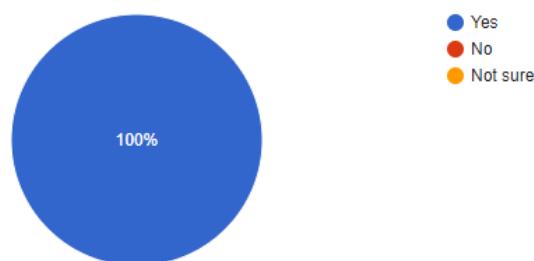
13 responses



This shows that highlighting in blue a piece a user selects is an effective feature and should be kept as it is.

Was highlighting in red a piece with no legal moves a useful feature?

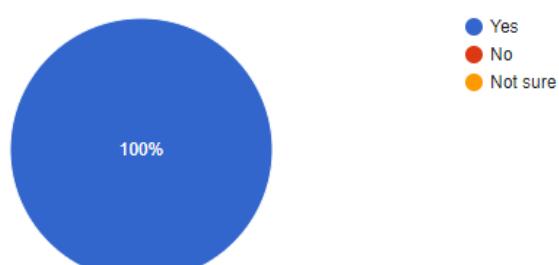
13 responses



This shows that highlighting in red a selected piece with no moves is an effective feature and should be kept as it is.

Was highlighting in yellow the places a piece could move to a useful feature?

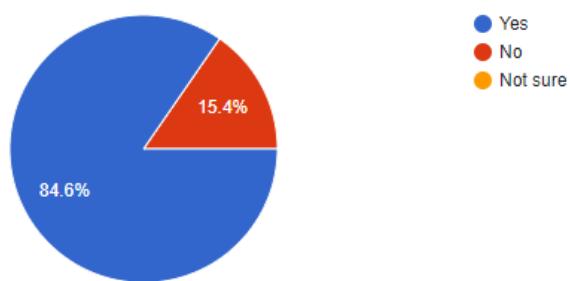
13 responses



This shows that highlighting in yellow the places a piece can move to is an effective feature and should be kept as it is.

Was the text telling you whose turn it was a useful feature?

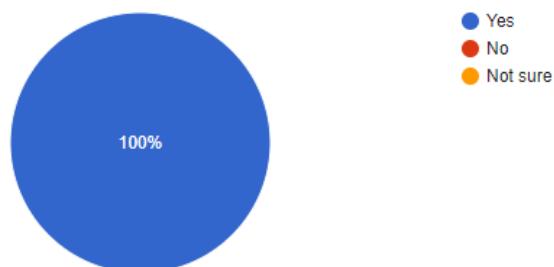
13 responses



This shows that most people thought that the text telling you whose turn it was to move is a useful feature, however a few people did not.

Was the text telling you who was in check a useful feature?

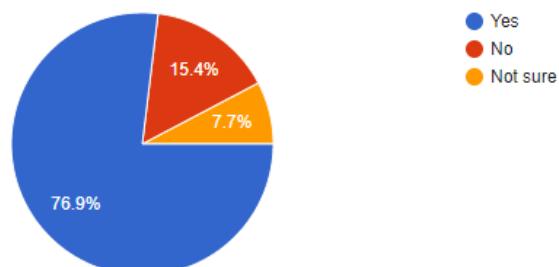
13 responses



This shows the text telling people who was in check is an effective feature and should be kept as it is.

Was the previous move made being highlighted in orange a useful feature?

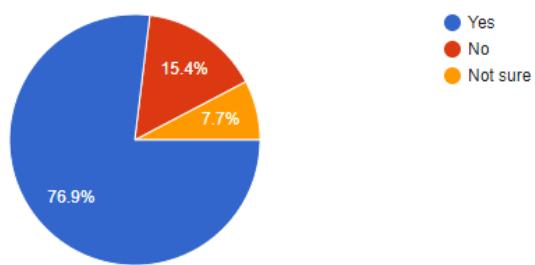
13 responses



This shows that most people thought that the previous move being highlighted in orange is a useful feature, however a few people did not. This suggests that some changes could be made to this feature, or perhaps the option to disable this feature.

Was the window containing instructions on how to use the program a useful feature?

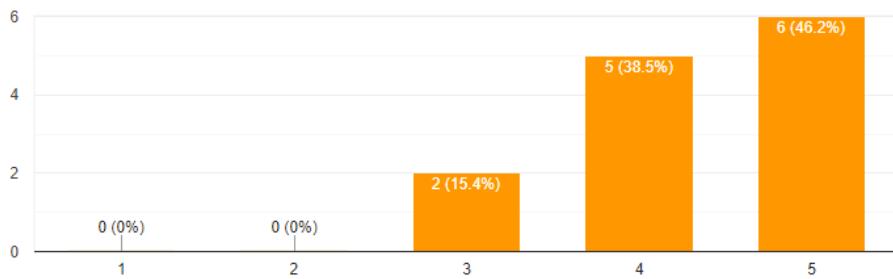
13 responses



This shows that most people thought that the help window is a useful feature, however a few people did not. This could be because a lot of the end users did say that they were already confident with the rules, and so is not needed for them.

How much do you like the design of the GUI?

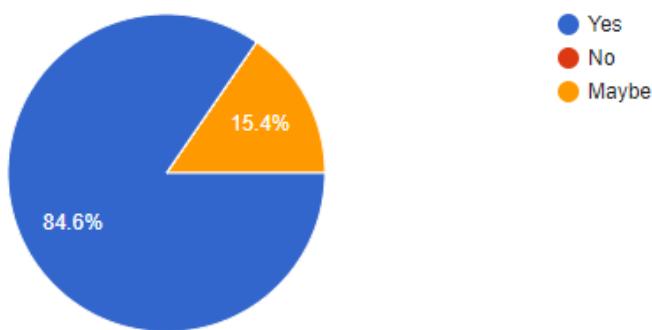
13 responses



This shows that the majority of users liked the design of the GUI, which is good as this was a feature which they considered important.

Did you think the GUI had a minimalist design?

13 responses



This shows that the majority of users thought that the GUI had a minimalist design, which was important as this is something that they said they liked in other programs.

Any other comments on the program?

3 responses

better if it had different difficulties of ai

text in help window shouldn't be scrolled down already

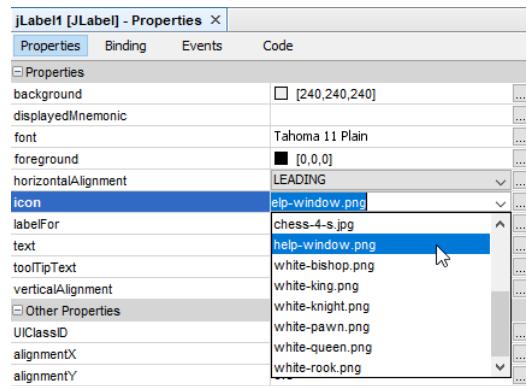
Would be nice if I had the option to change the looks of the pieces

A few of the end users gave suggestions of other features which they would like to be altered or included. These are things which I could implement in the future to make the program more suitable.

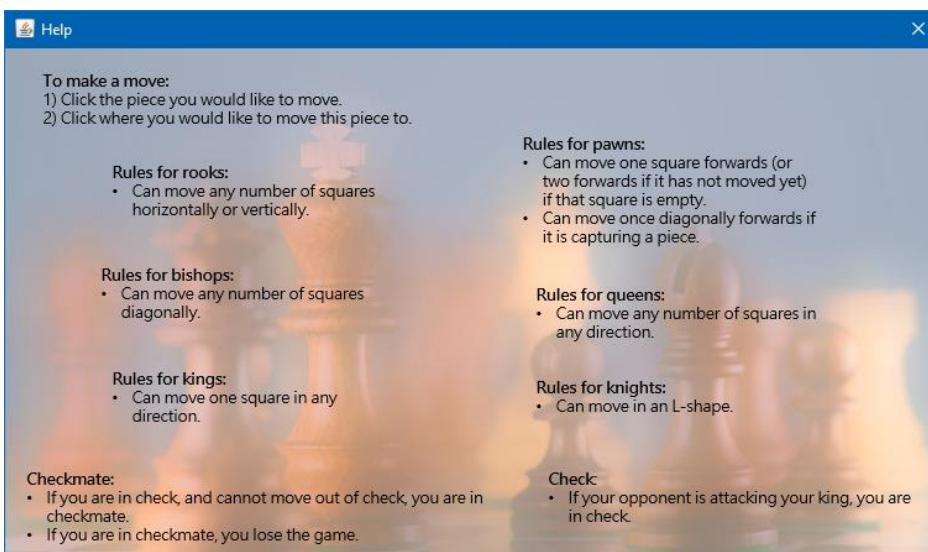
In order to implement different difficulties, I could let the user pick out of "easy", "medium" and "hard", and then set the search depth for the minimax algorithm depending on this – looking less moves ahead to make it easier, or more moves ahead to make it harder.

To fix the help window, I could change the way in which the text is displayed (i.e. by using a different component instead of a text box).

To change the appearance of the pieces, I could create different sets of images for each of the different pieces in different styles, and allow the user to select which they would like to use, and set the value of the "img" variable for the pieces accordingly.



To quickly fix the help window, I created an image containing the help instructions, and set this to be the background of the help window.



When I ran this and pressed the help menu item, the help window now looks like this, fixing the problem of the text box being scrolled down already when the help window is opened, and allows the user to see all of the rules at once without needing to scroll.

Acceptance testing

Now that I have tested both the functionality through black box testing and the usability features of the solution through beta testing, I can meet with my client, and with them determine how fully the success criteria which were agreed earlier on in the project have been met, and whether they have any other comments for each of them. This will allow me to determine how fully the solution meets the client's requirements, and identify any required improvements.

I have had a conversation with my client where we went through each of the success criteria, and determined whether it had been met or not by the solution. I have added his comments in the table below, and how far we agreed each of the criteria had been met. There are some additional features that he mentioned could be included, which are also included in the table.

#	Criteria	Comments from client	Criterion met?
1	Graphical User Interface (GUI) shows board	<i>I agree that this has been well met – after selecting who to play against, I am shown the chess board with all of the pieces in the correct places. I like the colours it uses and how the chequered pattern mimics a real chess board.</i>	Fully
2	GUI has a minimalistic design	<i>The only thing that can be seen when I run the program is the board, some text telling me whose turn it is, and a bar at the top containing various options, and so I would say that the GUI as a whole has a minimalistic design. I also think that the design of the pieces is minimalistic and not overly detailed. I like the simplicity and overall appearance of it.</i>	Fully
3	GUI uses the colours the end-users have specified	<i>I think that a good colour scheme has been used for this program as I like the blue, and so I agree that this has been met. I also like how I can change the colour of various things in the window.</i>	Fully
4	GUI uses distinctive images to represent pieces	<i>I can see the pieces easily and they stand out well against the colours of the board.</i>	Fully
5	User can choose who to play against	<i>When I ran the program, I was able to choose whether to play as "one player" or "two players". This window also appeared after a game ended.</i>	Fully
6	User can input their move	<i>I was able to press a square on the board to pick a piece, and then press another square to choose where to move it to. After this, the piece moved to the correct position.</i>	Fully
7	Program can switch turns	<i>After I moved a white piece, the text at the bottom changed, saying that it was blacks turn to move, and</i>	Fully

		<i>after I moved a black piece, the text at the bottom changed, saying that it was whites turn to move.</i>	
8	Show previous move of opponent	<i>I agree that this has been met - after I moved a piece, I liked how it was highlighted in orange. However, I think it could be better if it showed the path the piece took, and not just its start and end square.</i>	Partially
9	AI can play	<i>I agree that this has been met – if I select to play as one player, after I make a move, the program makes a move itself, allowing me to play against the computer.</i>	Fully
11	Can find all possible legal moves for a particular piece	<i>When I selected a piece to move, all the legal moves it could make were shown in yellow.</i>	Fully
12	Only makes a move if it is legal	<i>When I tried to make a move, it would only allow me to make it if it was legal. Any illegal moves I attempted weren't made.</i>	Fully
13	Can determine whether either side is in check	<i>I agree that this has been met – when I was in check, I wasn't allowed to move a piece if it didn't bring me out of check, and I couldn't make a move that put me into check.</i>	Fully
14	Can tell the user if they are in check	<i>I agree that this has been met - when a side was in check, there was some text at the bottom of the screen telling me which side was in check.</i>	Fully
15	Can determine whether either side is in checkmate	<i>I agree that this has been met as when I was in checkmate, the game ended.</i>	Fully
16	Can inform the user who has won	<i>I agree that this has been met as after the game ended when I was in checkmate, a window popped up telling me who won the game.</i>	Fully
17	Allows user to resign	<i>I agree that this has been met - there was a button I could press in the top menu called surrender that when I pressed ended the game and asked me what mode I would like to select before resetting the board.</i>	Fully
18	Allows user to undo a move	<i>I agree that this has been met – there was a button in the top menu called undo move that when I pressed undid the last two moves made</i>	Fully

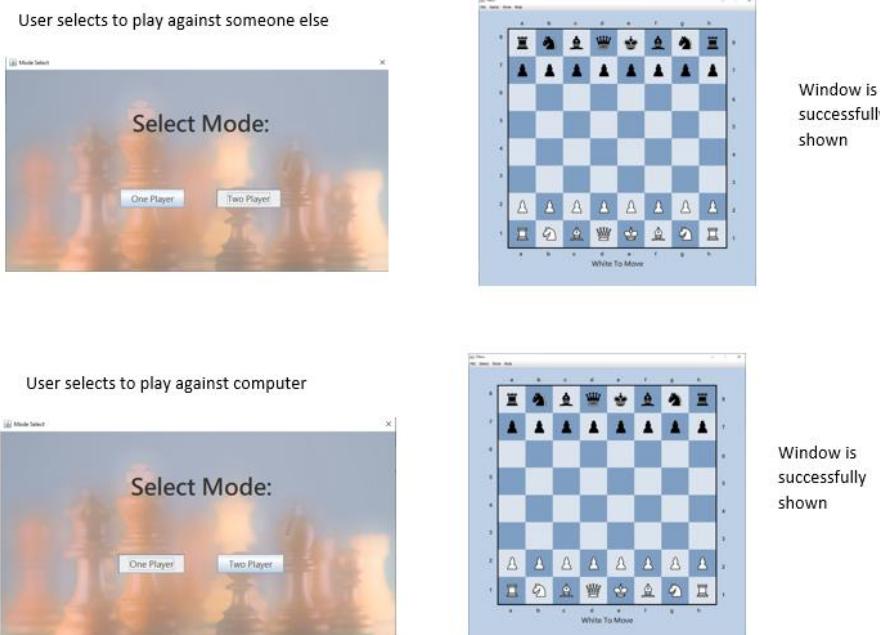
19	Allows user to save current board state to a file	<i>I agree that this has been partially met as it does save the game to a current file, however I think it would be better if it fully used the standard FEN notation to save the game, as this would allow me to also use this in other programs.</i>	Partially
20	Allows user to load previous board state from a file	<i>I agree that this has been partially met as it does allow me to open a game that I previously saved using this program, however I think this could again be improved if it used the standard FEN notation, as it would allow me to open games from other programs as well.</i>	Partially
21	A way of showing the user a written explanation of the rules	<i>I agree that this has been met as there is a button in the top menu called help which when pressed shows a window with an explanation of how to play.</i>	Fully

Evaluation of success criteria and usability features (Usability features in blue)

#	Criteria	Met?
1	Graphical User Interface (GUI) shows board	Fully

One piece of evidence of this is black box test #2, where I tested whether after selecting who to play against, the chess board is shown to the user. As you can see below, this test was successful – after selecting who to play against, the board is shown to the user.

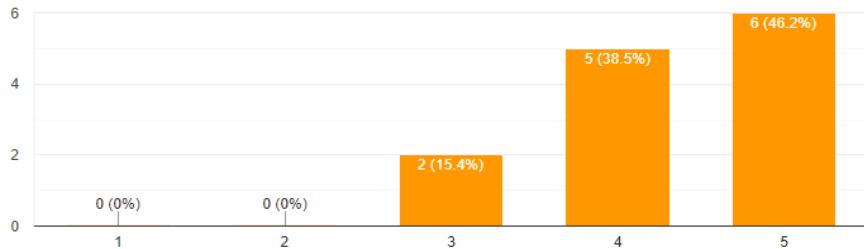
#	Test data	Expected result	Actual result	Changes needed
2	User starts program and selects player to play against	After user has selected who to play against, window is shown, containing 64 JButtons and the JMenuBar	Window is shown correctly after user selects player	



Another piece of evidence for this is the meeting I had with my client during acceptance testing, during which he said that “*after selecting who to play against, I am shown the chess board with all of the pieces in the correct places. I like the colours it uses and how the chequered pattern mimics a real chess board.*” This shows both that the board is shown correctly, the pieces are in the right places, and that the client is happy with the appearance of it.

How much do you like the design of the GUI?

13 responses



I didn't ask the beta testers a question specifically about whether the board is correctly shown or not, however I did ask them how much they liked the design of the GUI / board. This piece of evidence suggests that the board was shown to them correctly, and that they liked the design of it.

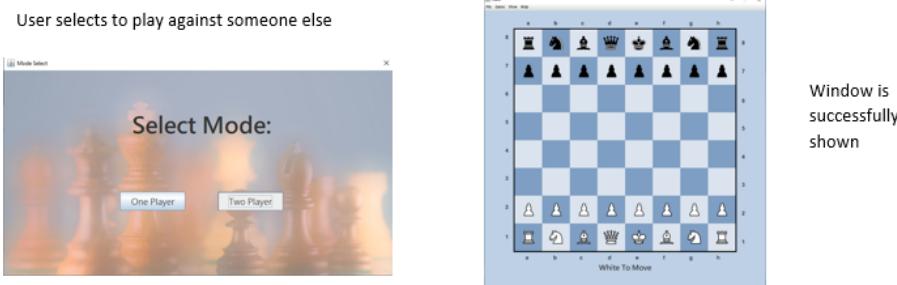
Overall, I think that this criterion has been fully met because, as seen in black box testing, the window is displayed correctly, and as seen from acceptance and beta testing, the client and end users were happy with the aesthetics of the board.

#	Criteria	Met?
2	GUI has a minimalistic design	Fully

Success criterion 2 is one of the usability features of the program – a GUI that is simple / minimalistic will make the program easier to use as the information displayed to the user (for example whose turn it is) will be easier to see, and how the users should interact with the program would be clearer.

Looking at black box test #2 again, whilst it does not explicitly mention the design of the board, the board can be seen in this test. I personally think that this has a fairly minimalistic design, however to be able to determine whether it does or not, and therefore whether this criterion is fully met or not, I need to look at the opinions of the client and the end users.

#	Test data	Expected result	Actual result	Changes needed
2	User starts program and selects player to play against	After user has selected who to play against, window is shown, containing 64 JButtons and the JMenuBar.	Window is shown correctly after user selects player	



During the meeting I had with my client for acceptance testing, we talked about the design of the GUI. He commented that *"The only thing that can be seen when I run the program is the board, some text telling me whose turn it is, and a bar at the top containing various options, and so I would say that the GUI as a whole has a minimalistic design."*, which is evidence that this success criterion has been fully met. He also said that *"the design of the pieces is minimalistic and not overly detailed."*, which is further evidence of this success criterion. One other thing he said was that *"I like the simplicity and overall appearance of it."*. This is evidence that this success criterion has been met because my in my initial meeting with him, he said he would like a simplistic design.

Q: Is there a specific type of design or colour scheme you would like the program to include?

"I want the design of the program to be relatively simple / not too complex, so that it does not distract from the focus of the game, and so that it can be easily seen, even if on a smaller laptop screen."

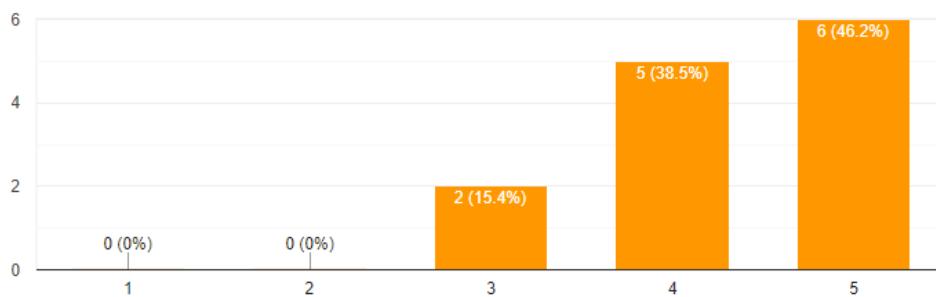
I also asked during beta testing whether the users liked the design of the GUI. Whether they liked the design of the GUI is an indicator of whether this success criterion was met or not because the end users had previously said that they liked the minimalistic design of other programs:

<i>"It looks clean and simple, easy to interact with and navigate."</i>
<i>"The simplicity and the clarity"</i>
<i>"Pieces simple, big board, clarity"</i>
<i>"Clean, easy to tell what's going on, nice pretty colours."</i>
<i>"Minimalistic but still intuitive and full of features"</i>
<i>"neat and very simple to use"</i>
<i>"I like the design and colour scheme of the chess board. Very nice simple layout"</i>
<i>"It shows what is important clearly: the pieces on the board"</i>

As you can see here, the majority of the users were happy with the design of the GUI, suggesting that it has a minimalist design.

How much do you like the design of the GUI?

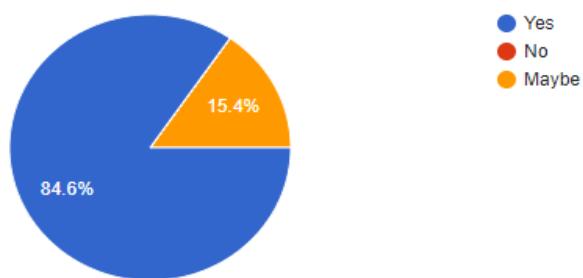
13 responses



I also explicitly asked them whether they thought it was minimalist or not. This shows that this criterion has been met as the majority of them agreed that it was minimalistic, only a small percentage said "maybe", and none of the end users thought that it was not minimalistic.

Did you think the GUI had a minimalist design?

13 responses



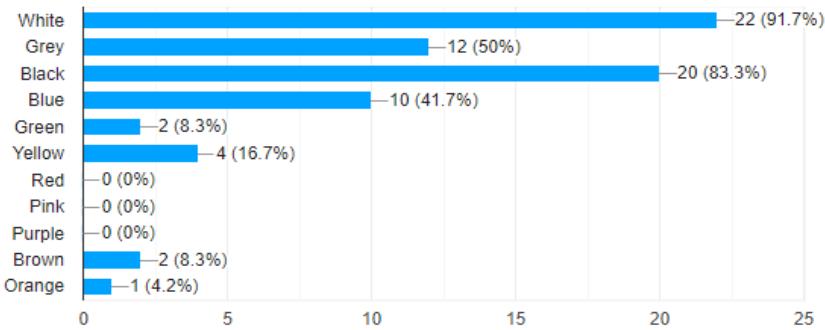
Overall, I think that this success criterion has been fully met, because both the client and end-users agree with me that the GUI has a minimalist design, and both the client and the end users seem to like the design of the GUI as a whole.

#	Criteria	Met?
3	GUI uses the colours the end-users have specified	Fully

In the survey I originally sent to the end users, these are the colours they said they would like to be used:

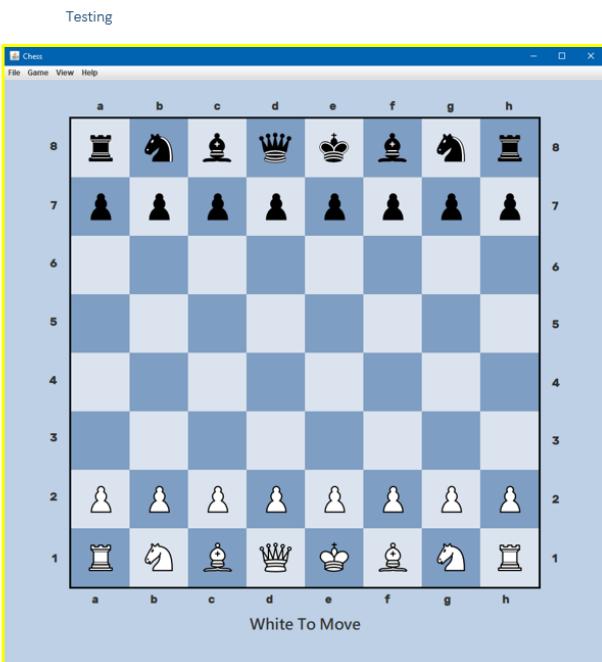
What colour scheme would you like the program to use? (can select multiple colours)

24 responses



The four most popular colours were white, grey, black and blue.

Looking at a screenshot from development, you can clearly see the GUI, and the colours that it uses:



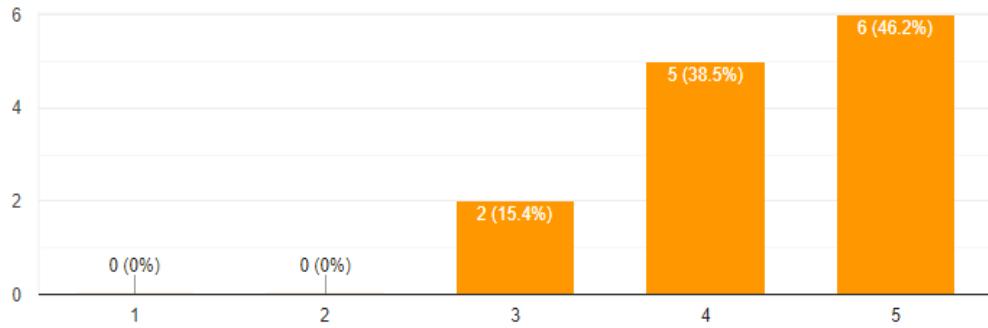
You can see that it uses white and black in the pieces, blue in the background & for the chequered board, black in the outline & text, and light grey for the top menu bar, and so this is evidence that this criterion has been fully met.

During acceptance testing, the client said that "*I think that a good colour scheme has been used for this program as I like the blue, and so I agree that this has been met.*", and so this is further evidence for this success criterion.

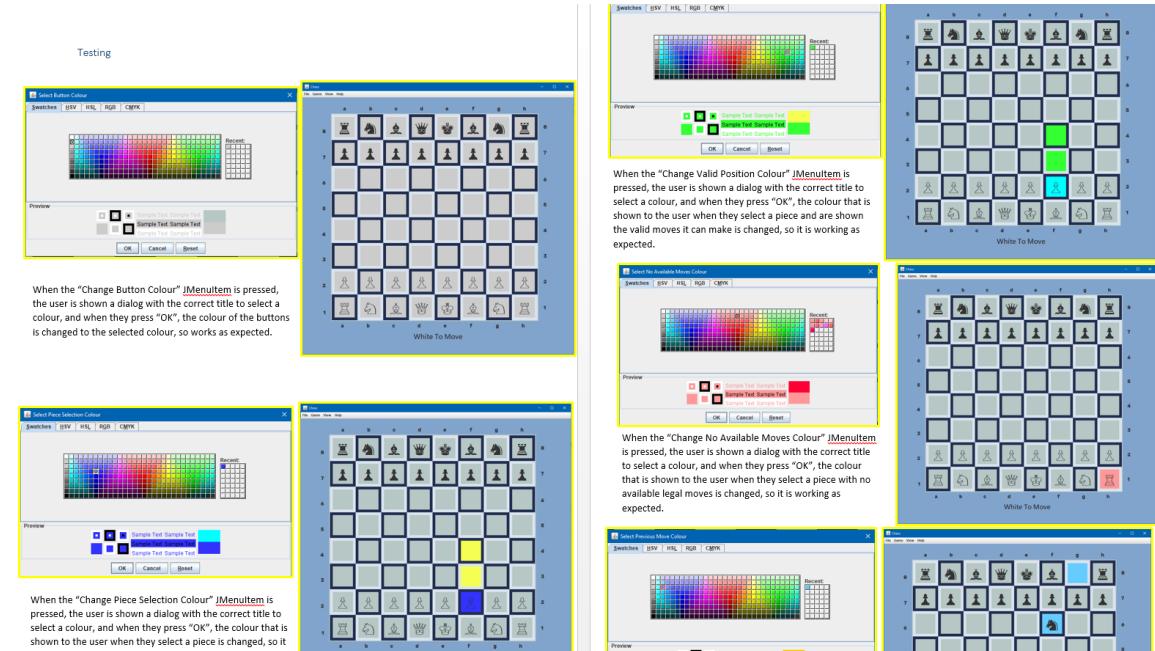
Another piece of evidence again is the question I asked the beta testers of whether they liked the design of the GUI. The majority of them did, and no one strongly disliked it. If they disliked the colours that had been used, I would have expected a lower response from them.

How much do you like the design of the GUI?

13 responses



However, the colours the end users said that they would like to be used in the program varied quite a lot. Therefore, I implemented the ability for them to change these, as seen here from development:

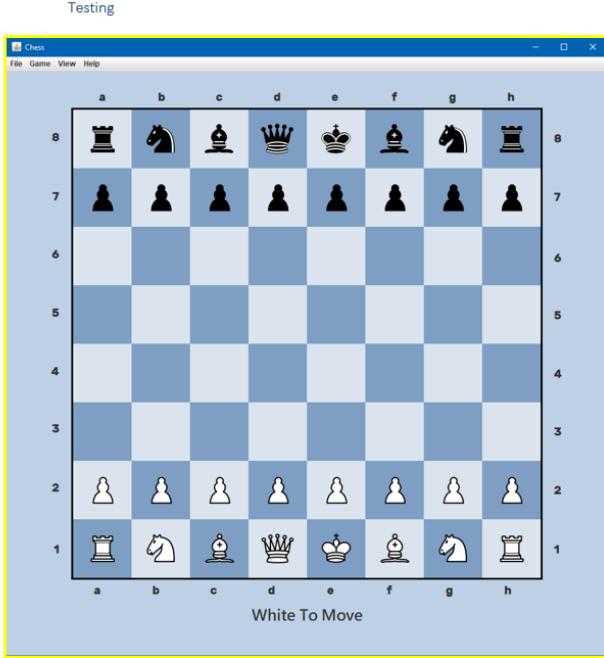


The ability for the user to change the colours used in this program means that they can change it to use the colours that they like, and so means that this success criteria has been fully met.

Overall, I believe that this success criterion has been fully met, as the program uses the four most popular colours requested by the end users, the end users have said that they like the design of the GUI, the client has said that they liked the colour scheme used, and because there is the functionality for the user to change to a different colour scheme if they do not like this one.

#	Criteria	Met?
4	GUI uses distinctive images to represent pieces	Fully

Looking at a screenshot from development, you can clearly see the GUI, and the icons it uses for the pieces:



I think that these contrast well with the colours of the board, however in order to say that this criterion has been fully met, the client / end users should agree.

In the meeting I had with my client, he said that "*I can see the pieces easily and they stand out well against the colours of the board.*". And so this is evidence that this success criterion has been met.

Another piece of evidence is the end users previously stating that they like the icons which were used in another program which look very similar to the ones used in mine, and thought that they were clear / easy to recognise.

Secondly, I asked for their opinions on the following program, from www.mathsisfun.com/games/chess.html:



Here are some of the features that they like about the design:

"the colours behind the piece that you have just played"
"Clear characters"
"The icons"
"is white and brown"
"very simple and good contrast board. nice buttons"
"the chessboard is centre of the screen"
"Clear"
"highlights last move clearly"
"Makes moves clear and obvious, available menu options are intuitive and obvious"
"It looks simply."
"The pieces are very well designed and the board colours are complementary to each other"
"Piece clarity, background colours, simplicity"
"Clean, easy to tell what's going on, nice pretty colours."
"I like the description box showing you where u moved at the bottom"

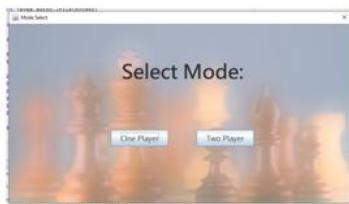
Overall, I think that this criterion has been fully met as the client has said that he thinks the pieces stand out well, and the end users have said the end users have said that pieces very similar to the ones used in my program were clear / well designed, and so easy to recognise.

#	Criteria	Met?
5	User can choose who to play against	Fully

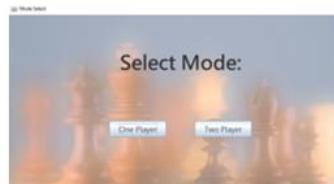
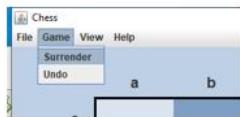
Black box test #1 checks whether the user is shown a window asking them whether they would like to play as one player or two player.

#	Test data	Expected result	Actual result	Changes needed
1	User starts program	Window shown to user asking them to select a player to play against	Window is shown correctly when program is first run	
	User surrenders game		Window is shown correctly when game is surrendered	
	Game is ended due to check / checkmate		Window is shown correctly when game finishes	

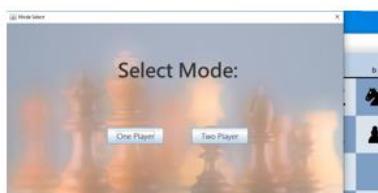
When starting the program, this window is shown to the user.



After surrendering the game, this window is shown.



When the game ends due to checkmate, after closing the window saying who has won, the window asking user to select a player is shown.



This test shows that whenever a new game starts (whether that be because the program is first run, the user ends the previous game themselves by surrendering, or because the previous game ended due to checkmate), the user is shown a window which asks them to select who to play against, and so is evidence that this criteria is met.

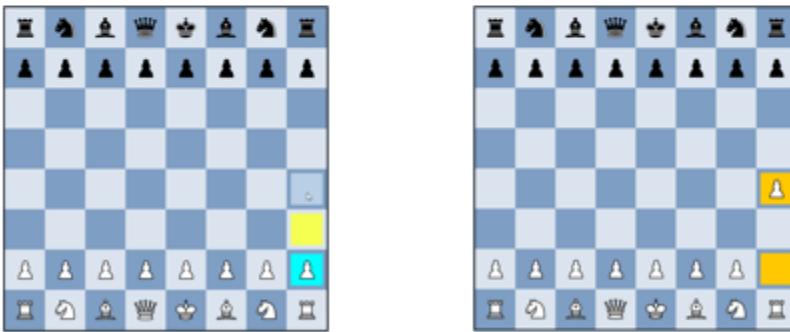
The client also said during our meeting “*When I ran the program, I was able to choose whether to play as “one player” or “two players”. This window also appeared after a game ended.*”, which is further proof that this is working correctly and that this criterion has been fully met.

Overall, I think this criterion has been fully met, as this testing shows that the user is asked at the correct times (when a game is starting) who their opponent will be.

#	Criteria	Met?
6	User can input their move	Fully

Here, in black box test #17, you can see that after the user has selected a piece, when they select the square to move it to, the board is immediately updated to show them the new position of the piece of the board (providing it is a legal move), which is evidence that the user is able to input their moves / that this success criterion has been met.

#	Test data	Expected result	Actual result	Changes needed
17	Move has been made previously in the game	The background colours of the buttons at the original position of the piece and the position the piece moved to should be changed to orange	Background colour of correct buttons changed to orange	
	No moves have previously been made in the game	No buttons should have their background colour changed to orange	No button background colours changed to orange	



After a move had been made, the square the piece started at, and the square the piece moved to are correctly highlighted in orange.

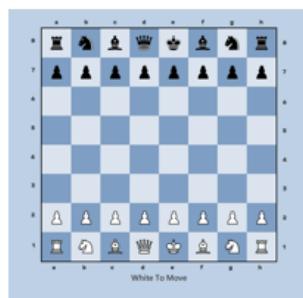
Furthermore, during acceptance testing, my client said that "*I was able to press a square on the board to pick a piece, and then press another square to choose where to move it to. After this, the piece moved to the correct position.*", showing that this is working correctly and that they were able to input their moves when they tested the program, and so is further evidence that this success criterion has been met.

Overall, I think that this success criterion has been fully met as both the black box testing and acceptance testing show that the user is able to input a move.

#	Criteria	Met?
7	Program can switch turns	Fully

Black box test #11 tests whether the program is able to switch turns correctly, which is one of the usability features of the program.

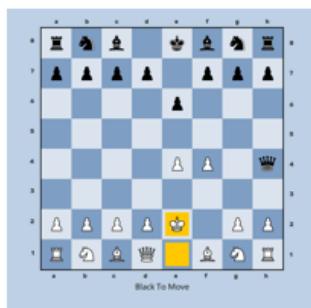
#	Test data	Expected result	Actual result	Changes needed
11	User inputs move that is not pseudo-legal	The same player should be allowed to select a different move	Same player can move again	
	User inputs a move that puts / keeps themselves in check but is pseudo-legal	The same player should be able to make a different move	Same player can move again	
	User inputs legal move	The same user should not be able to move	Same player cannot move	



After trying to make an illegal move (which is legal because it is not pseudo-legal for a pawn), the JLabel at the bottom does not change, and still shows the same user to move, so is working correctly.



After trying to make an illegal move (which is legal because it keeps white in check), the JLabel at the bottom does not change, and still shows the same user to move, so is working correctly.



After making a legal move, the JLabel at the bottom changes correctly to show the player to move.

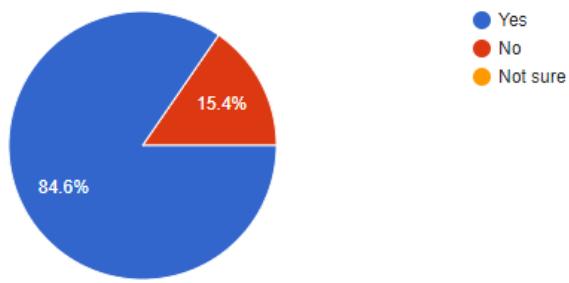
This shows that after a user enters a legal move, the label at the bottom of the screen is correctly changed to show the next player to move, showing that the turn has been switched, which is evidence that this success criterion has been met / that the program can switch turns.

Also, during acceptance testing, my client said that “*After I moved a white piece, the text at the bottom changed, saying that it was blacks turn to move, and after I moved a black piece, the text at the bottom changed, saying that it was whites turn to move.*”, showing that he agreed that the turns switched correctly after a move was made, which is further evidence that this success criterion has been met.

During beta testing, I also asked the end users whether they thought that the text at the bottom of the screen telling them whose turn it is was a useful feature. The majority of them thought that it was a useful feature, again providing evidence that the program is able to switch turns, as if it couldn't, they would not have found the feature useful.

Was the text telling you whose turn it was a useful feature?

13 responses

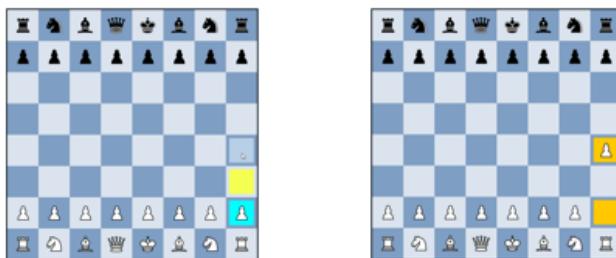


Overall, I think that this success criterion has been fully met because the black box testing shows that this is working as expected, the client agrees that when they used the program it switched turns, and the question from the beta testing suggests that the program was able to switch turns, and that they found the text at the bottom of the screen telling them whose turn it was useful.

#	Criteria	Met?
8	Show previous move of opponent	Partially

Here, in black box test #17, which tests whether the previous move made in the game is highlighted in orange, you can see that the test was successful, and so is evidence that this success criterion has been met and that this usability feature is effective.

#	Test data	Expected result	Actual result	Changes needed
17	Move has been made previously in the game	The background colours of the buttons at the original position of the piece and the position the piece moved to should be changed to orange	Background colour of correct buttons changed to orange	
	No moves have previously been made in the game	No buttons should have their background colour changed to orange	No button background colours changed to orange	

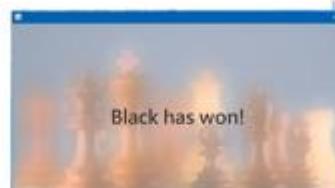


After a move had been made, the square the piece started at, and the square the piece moved to are correctly highlighted in orange.



Here, no moves have been made, and so no squares are highlighted in orange.

There is further evidence of this here, from black box test 18. Again, after the move is made, the start and end position are highlighted in orange, to show the previous move.



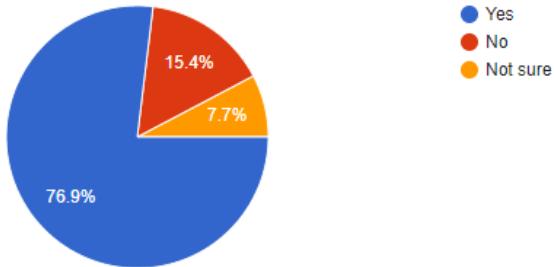
After this move was made, white was in checkmate, so game ends. Moves could not be inputted when the window showing who won was open.



I also asked the beta testers whether they found this feature helpful or not in the survey I gave to them.

Was the previous move made being highlighted in orange a useful feature?

13 responses



Here, you can see that whilst the majority of them did find this usability feature helpful, a small number did not find it useful, or were unsure. This suggests that some improvements could be made to the feature in order to make it more useful. However, I do not want to remove this feature, as many did still find it helpful. So, this is evidence that the feature is partially met, but not that it is fully met.

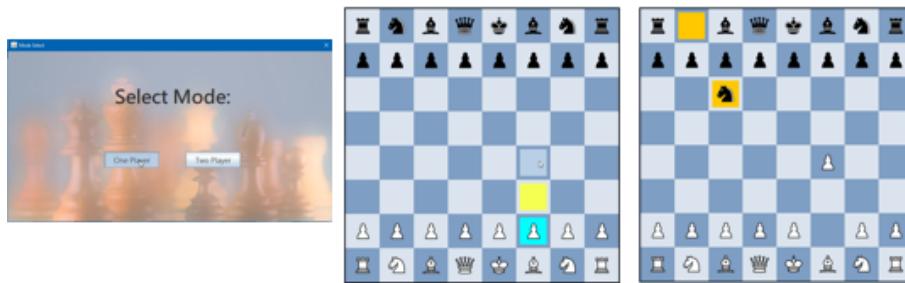
During the meeting with my client, he said that "*I agree that this has been met - after I moved a piece, I liked how it was highlighted in orange. However, I think it could be better if it showed the path the piece took, and not just its start and end square.*". This shows that he also thinks that whilst this feature technically has been included, improvements could be made to it, and so is again evidence that this criterion has only been partially met.

Overall, I think that this criterion has been partially met, as whilst it does work, a few of the clients did not find it helped with the usability of the program, and the client agreed that improvements could be made to it. In order to make this criterion fully met, I could perhaps take the suggestion of my client, and show the path the piece took rather than just the start & end position, or include the option of disabling the feature, so that those who find it helpful can keep it, and those who do not find it helpful do not have to use it.

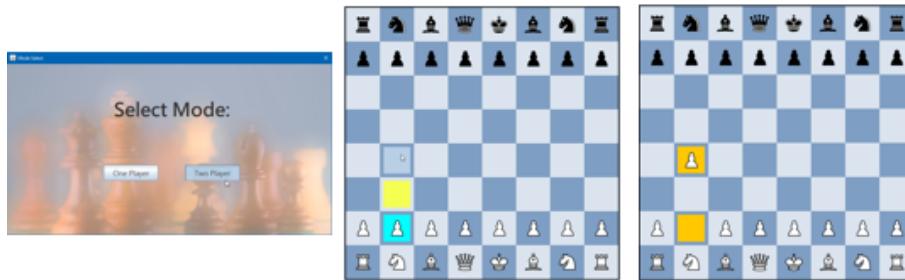
#	Criteria	Met?
9	AI can play	Fully

Here, in black box #21, whether the computer makes a move or not is tested. As you can see below, this black box test was successful – if the user selected to play against the computer, it made a move at the right time, and if they didn't select to play against the computer, it did not make a move, and so is evidence that this criterion has been met.

#	Test data	Expected result	Actual result	Changes needed
21	User has selected to play against the computer and has just inputted a valid move.	Computer should make a move and this should be shown to the user	Computer makes a move straight after the user inputs theirs	
	User has selected to play against another person and has just inputted a valid move	Computer should not make a move and should wait for the other player to input a move	Computer doesn't move, waits for user to make a move	



After choosing one player (i.e. to play against the computer), and making a move, the move is made, and then the computer makes its move shortly after, and so this is working correctly.



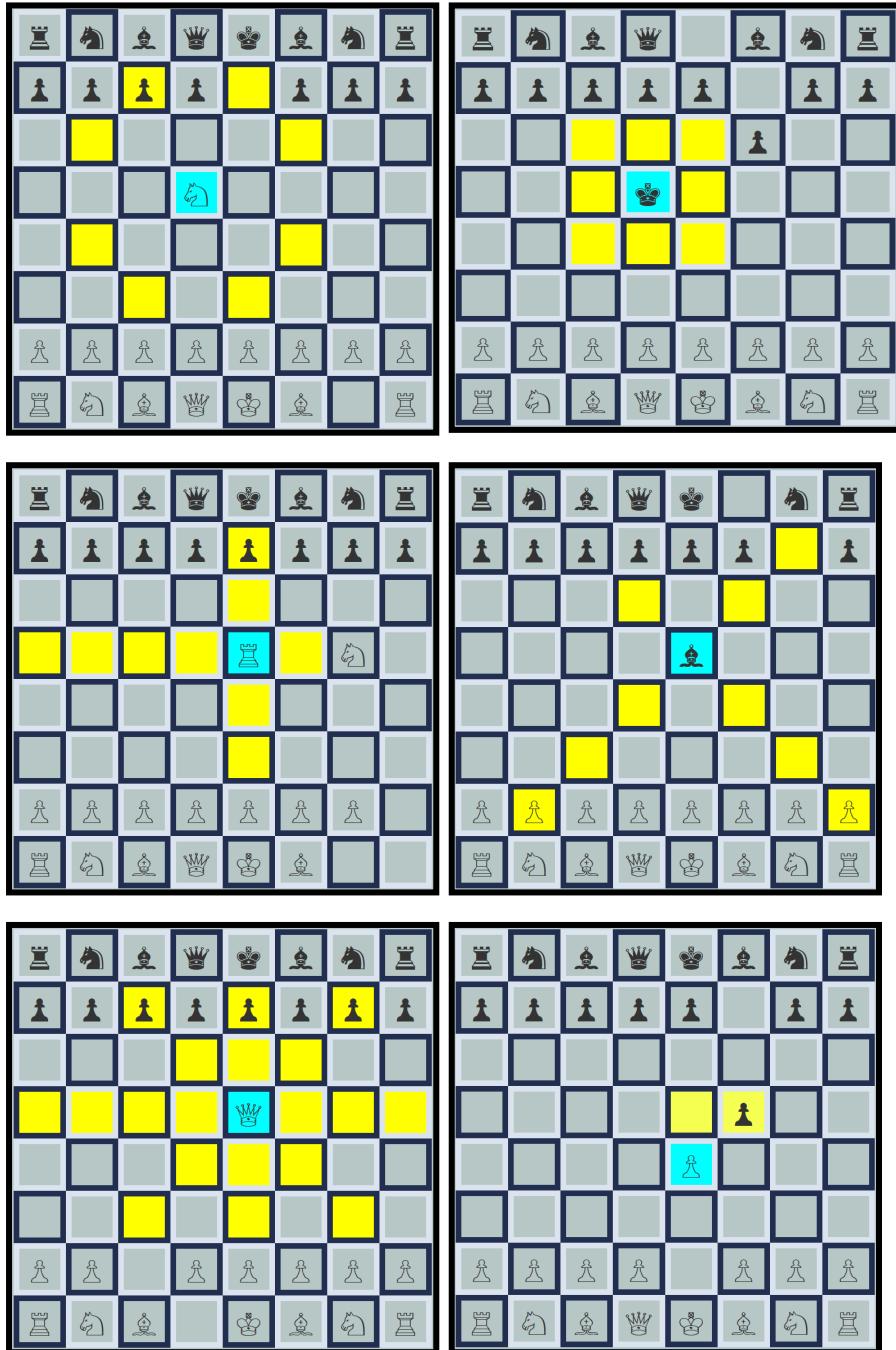
After choosing two player (i.e. to play against another human), and making a move, the move is made, and then the game waits for the other player to input their move – the computer does not take its move, and so this is also working correctly.

Furthermore, during the acceptance testing, my client said that “*if I select to play as one player, after I make a move, the program makes a move itself, allowing me to play against the computer*”, showing that the client was able to successfully play against the AI when testing the program. Therefore, this is more evidence that this criterion has been met.

Overall, I think that this criterion has been fully met, as both the black box test and the clients comments show that the AI is able to make a move, if one player mode is selected.

#	Criteria	Met?
11	Can find all possible legal moves for a particular piece	Fully

From testing during development, you can see that the program is able to find all possible pseudo legal moves for each different type of piece, and show these to the user:



However this only provides evidence that the program checks whether or not a move is pseudo-legal, and not whether it is fully legal (i.e. that it avoids check).

Again from testing during development, this shows evidence that the program also checks whether or not a move is legal, by not showing any moves that put / keep a user in check.

This now works correctly – the user is unable to make a move that would put themselves into check, or that does not take themselves out of check, and is not shown any move that would do so in yellow on the board. Below, I tested whether the “`isInCheck`” method was working correctly:



For example, here the king is not allowed to move to the square directly in front of them, as this would mean the pawn would be attacking the king.



Here, the king is under attack by the rook, and so must make a move that brings them out of check, and so cannot move to the square directly in front of them, as they would still be in check if they did this.



Here, if the pawn (which is highlighted in red, showing it has no possible moves) were to move, the king would be under attack by the queen, and so this pawn cannot move, despite having otherwise valid moves it could take, and so is working as expected.



Here, the king cannot move to the square to the top left, since if it did move there, it would be under attack by the knight.

Black box test #16 shows that the program is able to highlight all of the legal moves a piece can make in yellow, which would only be possible if the program was able to find all possible legal moves for a particular piece, and so is evidence that this success criterion has been met.

#	Test data	Expected result	Actual result	Changes needed
16	Current player selects one of their own pieces with valid moves	Should change background colour of all buttons representing each of the places the piece can move to to yellow.	Background colour of the correct buttons is changed to yellow	
	Current player selects one of their own pieces with no valid moves	Should not change the background colours of any buttons to yellow.	Background colour of no buttons is changed to yellow	
	Current player selects piece of the other colour	Should not change the background colours of any buttons to yellow.	Background colour of no buttons is changed to yellow	
	Current player selects empty square	Should not change the background colours of any buttons to yellow.	Background colour of no buttons is changed to yellow	



When clicking the piece at d2 when it is whites turn to move, the only two places it can move to are highlighted in yellow, and no squares that shouldn't be are highlighted.

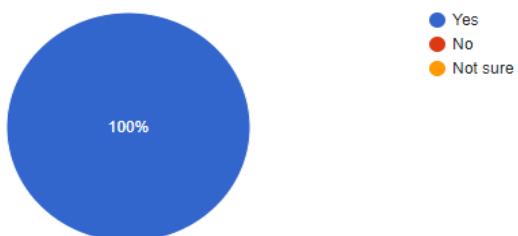


When clicking the piece at d1 when it is whites turn to move, no background colours are changed to yellow, which is expected as this pieces currently has no legal moves it can make.

Furthermore, all of the beta testers agreed that the feature of highlighting legal moves in yellow was useful, providing more evidence that this success criterion has been met, because if it didn't show all of the possible places the piece could move to, then the end users would not have found the feature helpful.

Was highlighting in yellow the places a piece could move to a useful feature?

13 responses



Also, when I spoke to the client, he commented that "*When I selected a piece to move, all the legal moves it could make were shown in yellow.*". This again would only be possible if the program can find all possible legal moves for a particular piece, and so is evidence that this success criterion has been met.

Overall, I believe that this criterion has been fully met as testing done during development, black box testing post development, and the client agree that all of the legal moves a certain piece can make are highlighted in yellow if that piece is selected, which would only be possible if this success criteria had been met. Furthermore, during beta testing, all of those who completed the survey said that they found the feature helpful, showing that not only does it technically work, but that it is also an effective usability feature.

#	Criteria	Met?
12	Only makes a move if it is legal	Fully

To show evidence that this success criterion has been met, I will show all of the black box tests I did for each different type of piece.

#	Test data	Expected result	Actual result	Changes needed
4	User inputs move that is not pseudo-legal for a pawn	Move should not be made	Move is not made	
	User inputs a move that is pseudo-legal for a pawn	Move should be made	Move is correctly made	

Testing valid inputs:

White pawn is allowed to move one square forwards

White pawn is allowed to move two squares forwards if it hasn't moved yet

Black pawn is allowed to move one square forwards

Black pawn is allowed to move two squares forwards if it hasn't moved yet

White pawn is allowed to attack to the top right

White pawn is allowed to attack to the top left

Black pawn is allowed to attack to the bottom right

Black pawn is allowed to attack to the bottom left

Testing invalid inputs:

Before

Selecting to move here

After

Before

Selecting to move here

After

Before

Selecting to move here

After

#	Test data	Expected result	Actual result	Changes needed
6	User inputs move that is not pseudo-legal for a rook	Move should not be made	Move is not made	
	User inputs a move that is pseudo-legal for a rook	Move should be made	Move is correctly made	

Testing valid inputs:



Testing if rook can move left



Testing if rook can move right



Testing if rook can move up



Testing if rook can move down



Testing invalid inputs:

Before



Selecting to move here

After



Selecting to move here



Selecting to move here



#	Test data	Expected result	Actual result	Changes needed
7	User inputs move that is not pseudo-legal for a bishop	Move should not be made	Move is not made	
	User Inputs a move that is pseudo-legal for a bishop	Move should be made	Move is correctly made	

Testing valid inputs:



Testing if bishop can move north east



Testing if bishop can move north west



Testing if bishop can move south west



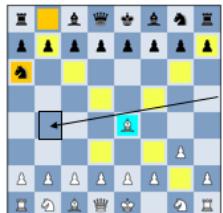
Testing if bishop can move south east



Testing invalid inputs:



Selecting to move here



Selecting to move here



Selecting to move here



Selecting to move here



#	Test data	Expected result	Actual result	Changes needed
8	User inputs move that is not pseudo-legal for a queen	Move should not be made	Move is not made	
	User inputs a move that is pseudo-legal for a queen	Move should be made	Move is correctly made	

Testing valid inputs:



Testing if queen can move left



Testing if queen can move northwest



Testing if queen can move up



Testing if queen can move north east



Testing if queen can move right



Testing if queen can move south east



Testing if queen can move down



Testing if queen can move south west

Testing invalid inputs:



Selecting to move here



Selecting to move here



Selecting to move here



Selecting to move here



#	Test data	Expected result	Actual result	Changes needed
9	User inputs move that is not pseudo-legal for a king	Move should not be made	Move is not made	
	User inputs a move that is pseudo-legal for a king	Move should be made	Move is correctly made	

Testing valid inputs:



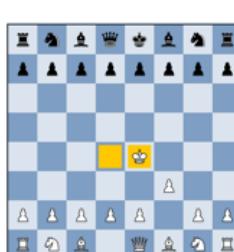
Testing if kings can move left



Testing if kings can move north east



Testing if kings can move north west



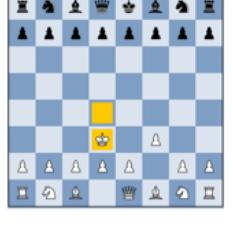
Testing if kings can move right



Testing if kings can move up



Testing if kings can move south east



Testing if kings can move down



Testing if kings can move south west

Testing invalid inputs:



After



These black box tests were all successful, providing evidence that this criterion has been partially met. This only shows that it is partially met, as these only check for pseudo-legal moves, and not fully legal moves.

However, black box #10 tests for legal moves and not just pseudo-legal moves, providing evidence that this criterion has been fully met:

#	Test data	Expected result	Actual result	Changes needed
10	User inputs move that puts themselves into check	Move should not be made	Move is not made	
	User inputs a move that keeps themselves in check	Move should not be made	Move is not made	
	User inputs a legal move (does not put / keep themselves in check)	Move should be made	Move is made	



If the king moved to the square that was clicked, it would be under attack by two pawns, and so would be in check. Therefore, this move should not be allowed to be made. Here, the screenshots show that this move was not made, as expected.



If the king moved to the square that was clicked, it would still be under attack by the queen, so would still be in check. This means that the move should not be made. The screenshots show that this happened correctly – the move was not made.



If the king moved to the square that was clicked, it would not be under attack and so would not be in check, so the move should be allowed. Here the screenshots show that the move was correctly made.

Also, during acceptance testing, the client said “*When I tried to make a move, it would only allow me to make it if it was legal. Any illegal moves I attempted weren’t made*”, providing more evidence that this criterion has been met, as it shows that any legal move he tried to make whilst testing it was allowed, and any illegal move made was not allowed, as expected.

Overall, I think that this criterion has been fully met, as the black box tests show both that the program is able to determine all the pseudo legal moves for a piece, and that it is able to determine whether or not a move cause a player to keep / put themselves into check, which is needed to determine whether a move is legal. Furthermore, the client agreed that he could make illegal moves, and couldn't make any illegal moves.

#	Criteria	Met?
13	Can determine whether either side is in check	Fully

Black box test #14 checks whether the label informing the user who is in check is working correctly, displaying either nothing, “White is in check” or “Black is in check”, depending on the state of the board. This can provide evidence of this success criterion, as in order to correctly tell the user who is in check, it needs to be able to determine whether either side is in check.

#	Test data	Expected result	Actual result	Changes needed
14	White is in check	JLabel should display text “White is in check”	When white is in check, JLabel is visible and has text “White is in check”	
	Black is in check	JLabel should display text “Black is in check”	When black is in check, JLabel is visible and has text “Black is in check”	
	Neither player is in check	JLabel should show no text / not be visible to the user	When neither player is in check, JLabel is not visible to user.	



Here, the white king is under attack by the queen, so white is in check, and the JLabel tells this to the user correctly.



Here, the black king is under attack by the queen, so black is in check, and the JLabel tells this to the user correctly.



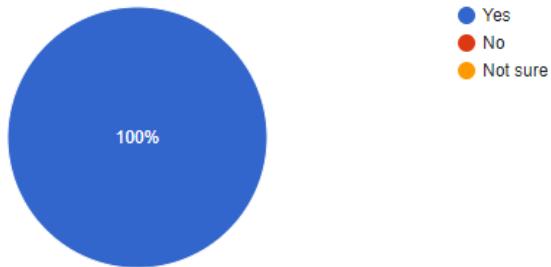
Here, no one is in check, and the JLabel is hidden as expected.

As you can see these black box tests were all successful, showing the correct text depending on the board, providing evidence that it can correctly determine whether either side is in check.

During beta testing, I asked the end users whether the text at the bottom of the screen telling them when someone was in check was a useful feature.

Was the text telling you who was in check a useful feature?

13 responses



All of them agreed that this was a helpful feature of the program. This provides evidence that this criterion has been fully met as it shows that this has been a successful usability feature, and suggests that it works correctly, as they would not have found it such a useful feature if it was not working correctly, and none of them reported any errors with this.

During my meeting with the client, he said that "*when I was in check, I wasn't allowed to move a piece if it didn't bring me out of check, and I couldn't make a move that put me into check.*" Showing that when he tested this, the program was able to determine when a move was illegal due to check, and didn't allow these moves to be made, providing evidence that the program can determine whether either side is in check / that this criterion has been met.

Overall, I think that this criterion has been fully met as both the black box testing and beta testing show that the text at the bottom of the window changed appropriately, which would only be possible if this success criterion had been met, and because the client has said that the program was able to recognise when a move he made put / kept himself in check, which again would only be possible if this success criterion had been met.

#	Criteria	Met?
14	Can tell the user if they are in check	Fully

Black box test #14 checks whether the label informing the user who is in check is working correctly, displaying either nothing, “White is in check” or “Black is in check”, depending on the state of the board.

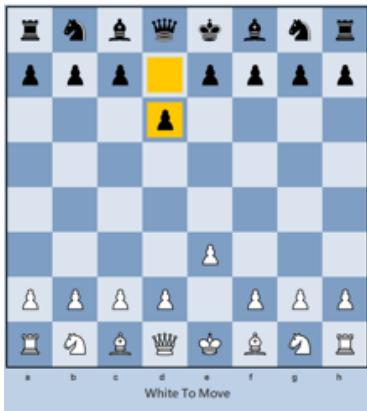
#	Test data	Expected result	Actual result	Changes needed
14	White is in check	JLabel should display text “White is in check”	When white is in check, JLabel is visible and has text “White is in check”	
	Black is in check	JLabel should display text “Black is in check”	When black is in check, JLabel is visible and has text “Black is in check”	
	Neither player is in check	JLabel should show no text / not be visible to the user	When neither player is in check, JLabel is not visible to user.	



Here, the white king is under attack by the queen, so white is in check, and the JLabel tells this to the user correctly.



Here, the black king is under attack by the queen, so black is in check, and the JLabel tells this to the user correctly.



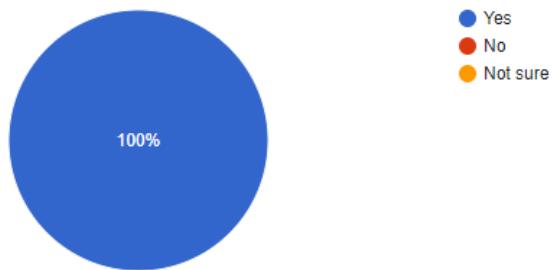
Here, no one is in check, and the JLabel is hidden as expected.

As you can see these black box tests were all successful, showing the correct text depending on the board, providing evidence that this success criterion has been met.

During beta testing, I asked the end users whether the text at the bottom of the screen telling them when someone was in check was a useful feature.

Was the text telling you who was in check a useful feature?

13 responses



All of them agreed that this was a helpful feature of the program. This provides evidence that this criterion has been fully met as it shows that this has been a successful usability feature, and that it works correctly.

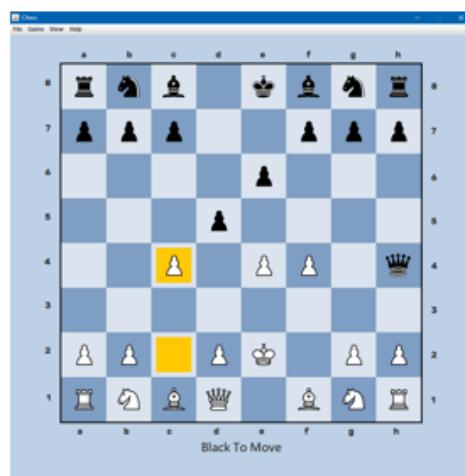
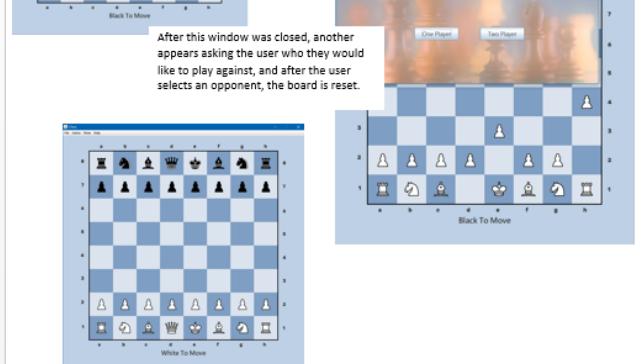
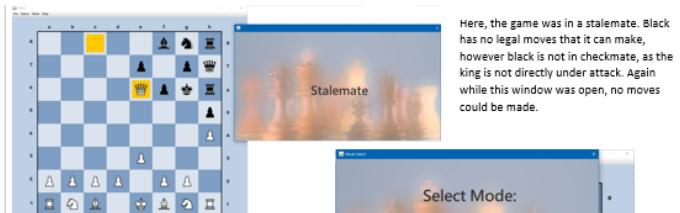
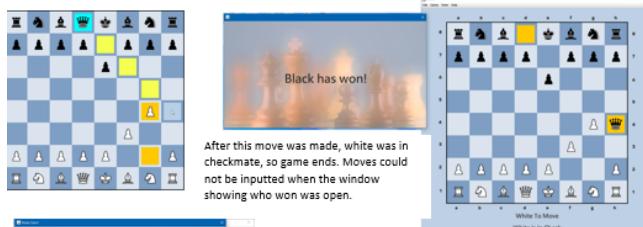
During my meeting with the client, he said that "*when a side was in check, there was some text at the bottom of the screen telling me which side was in check*" Showing that when he tested this, the program told him at the right time when someone was in check, and showed the correct player, providing more evidence that this criterion has been met.

Overall, I think that this criterion has been fully met as the black box testing, beta testing and client all agree that this feature is working correctly.

#	Criteria	Met?
15	Can determine whether either side is in checkmate	Fully

Black box test #19 checks whether the game ends correctly when someone is in checkmate or stalemate. This test was successful, showing that the program correctly ended when someone was in checkmate, providing evidence that it can determine whether either side is in checkmate / that this criterion has been met.

#	Test data	Expected result	Actual result	Changes needed
18	Player is in checkmate	Game should end: User should no longer be able to input moves, board should be reset	Game ends, user can no longer enter moves and after window showing user who won the game is closed, the board is reset.	
	Player is in stalemate	Game should end: User should no longer be able to input moves, board should be reset	Game ends, user can no longer enter moves and after window showing user who won the game is closed, the board is reset.	
	Player is in check but not checkmate	Game should not end: User should be able to input moves, board should not be reset	Game continues as normal	
	No players are in check or checkmate or stalemate	Game should not end: User should be able to input moves, board should not be reset	Game continues as normal	



Also, during acceptance testing, the client said that "*I agree that this has been met as when I was in checkmate, the game ended*", showing that when he tested it, the program was able to correctly recognise when someone was in checkmate, meaning that the game should end. This is more evidence that this criterion has been met.

Overall, I think that this criterion has been fully met as both the black box testing and acceptance testing show that the program is correctly able to determine when a player is in checkmate.

#	Criteria	Met?
16	Can inform the user who has won	Fully

Black box test #19 checks whether the game ends correctly when someone is in checkmate or stalemate, and whether a window is shown to the user telling them who (if anyone) won the game, and so provides evidence for this success criterion.

#	Test data	Expected result	Actual result	Changes needed
18	Player is in checkmate	Game should end: User should no longer be able to input moves, board should be reset	Game ends, user can no longer enter moves and after window showing user who won the game is closed, the board is reset	
	Player is in stalemate	Game should end: User should no longer be able to input moves, board should be reset	Game ends, user can no longer enter moves and after window showing user who won the game is closed, the board is reset	
	Player is in check but not checkmate	Game should not end: User should be able to input moves, board should not be reset	Game continues as normal	
	No players are in check or checkmate or stalemate	Game should not end: User should be able to input moves, board should not be reset	Game continues as normal	

As you can see from this, a window saying black won the game is displayed if white is in checkmate, and a window saying the game ended in a stalemate is displayed if one of the players has no legal moves but is not in checkmate, and so is evidence that this success criterion has been met.

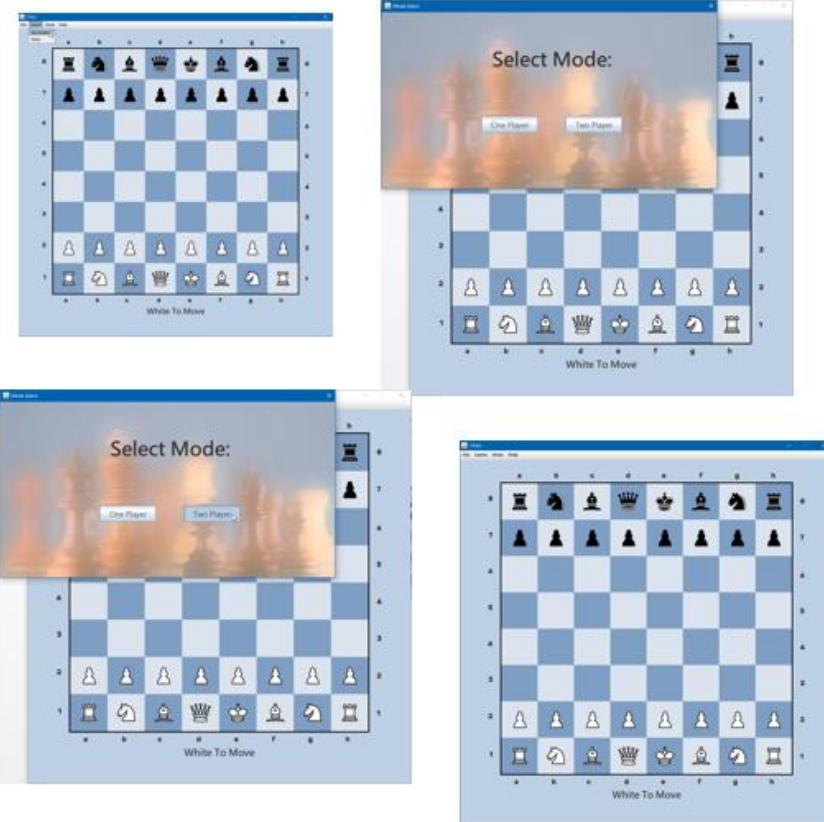
The client confirmed that this criterion had been met when he said "*after the game ended when I was in checkmate, a window popped up telling me who won the game.*"

Overall, I believe that this criterion has been fully met as both the black box testing and acceptance testing provide evidence of this.

#	Criteria	Met?
17	Allows user to resign	Fully

Black box test #20 checks that when the user presses the surrender menu item, the game ends, providing evidence of this success criterion is this allows the user to resign the game. As you can see here, this test as successful, showing that this criterion has been met.

#	Test data	Expected result	Actual result	Changes needed
20	User presses the surrender menu item	Game should end, the board should be reset, and the user asked again who they would like to play against	Game ends and is reset correctly	



During acceptance testing, my client said “*there was a button I could press in the top menu called surrender that when I pressed ended the game and asked me what mode I would like to select before resetting the board*”, providing further evidence that this criterion was met, as pressing this button allowed them to resign / end the game and start a new one.

Overall, I think that this criterion has been fully met, as by pressing a menu item, the user is able to end the game early if they would like, as evidenced by the above tests.

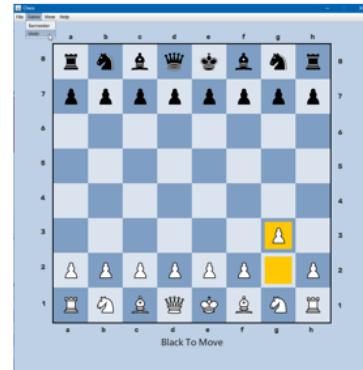
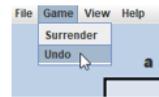
#	Criteria	Met?
18	Allows user to undo a move	Fully

Black box test #25 tests whether moves are undone correctly when the user presses the undo menu item. Here, this test was successful, showing that this criterion has been met.

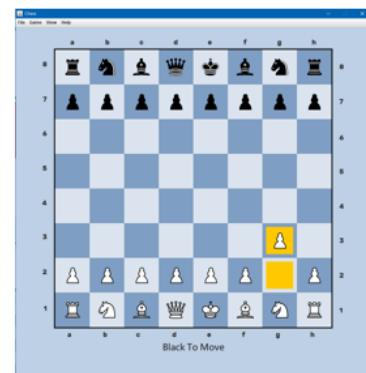
#	Test data	Expected result	Actual result	Changes needed
25	User presses undo menu item and more than two moves have been made so far in the game	The last two moves that were made should be undone	Last two moves were undone	
	User presses undo menu item and less than two moves have been made so far in the game	No moves should be undone	No moves were undone	



Here, 2 moves have been made, and it is white's turn to move. When I press the undo button, it correctly undoes the two moves that had been made. The reason two moves are undone rather than just one is because a player should press the undo button to undo the last move they made, and not just the last move their opponent had made.



Here, less than two moves are made, and so no moves are undone, as expected.



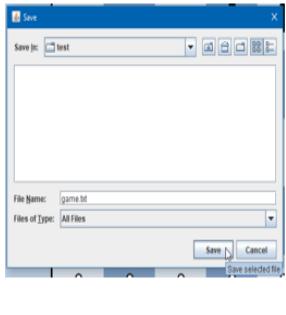
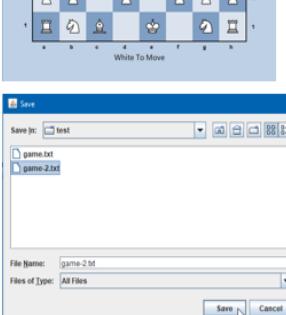
In the meeting I had with my client, he commented that "*there was a button in the top menu called undo move that when I pressed undid the last two moves made*", providing additional evidence that this criterion has been met.

Overall, I think that this success criterion has been fully met, as both of the above tests show that the user is able to undo a move if they press the undo menu item.

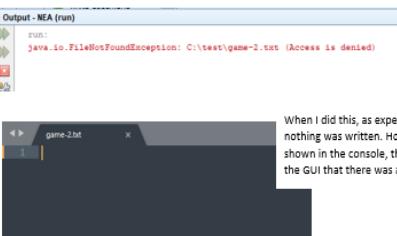
#	Criteria	Met?
19	Allows user to save current board state to a file	Partially

Black box test #21 tests whether the board can be saved to a file, and so provides evidence for this success criterion. As you can see from this, if a valid file is selected, then a string representing the game is written correctly to the file. However, this also shows that if an invalid file is selected, the user is not informed of this, providing evidence that this criterion is only partially met.

#	Test data	Expected result	Actual result	Changes needed
21	User presses save menu item and selects a valid file	A string representing the board should be written to the selected file	String is correctly saved to file	Red
	User presses save menu item and selects an invalid file	A dialog should be shown to the user informing them that they selected an invalid file and nothing should be written.	Nothing is written to file but user is not shown an error	Green

Then, to test an invalid file, I created a text file that was read only.



I then attempted to save the game to this file.

When I did this, as expected, an error occurred and nothing was written. However, whilst an error is shown in the console, there is nothing to indicate in the GUI that there was a problem saving the game.

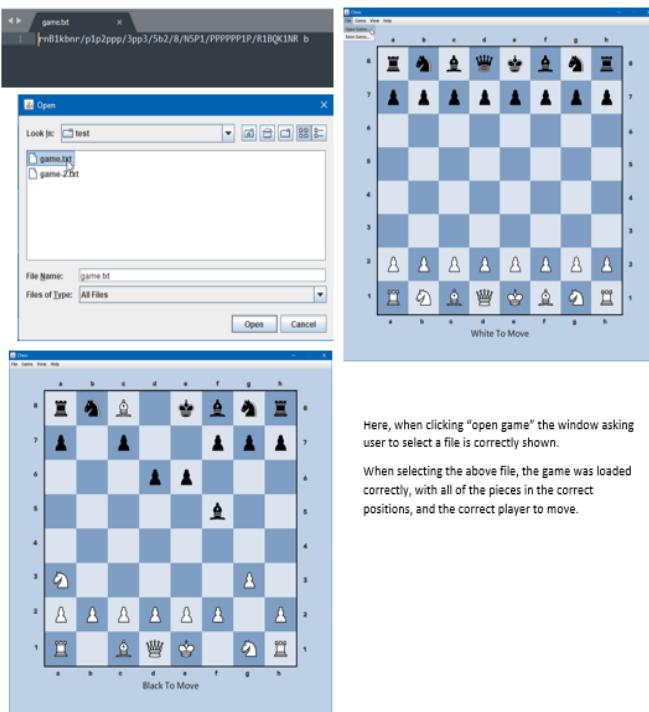
Also, during acceptance testing, the client said that "*it does save the game to a current file, however I think it would be better if it fully used the standard FEN notation to save the game, as this would allow me to also use this in other programs*". This is further evidence that this criterion has only been partially met – the client would like the string to be in a different more standard format, but he agrees that it does still work as it is.

Overall, this criterion has been partially met – whilst it does technically work, there are improvements which could be made. In order to make this fully met, I could change the format in which the game string is generated to be in the standard FEN format, and to display an error dialog to the user if the file cannot be written to.

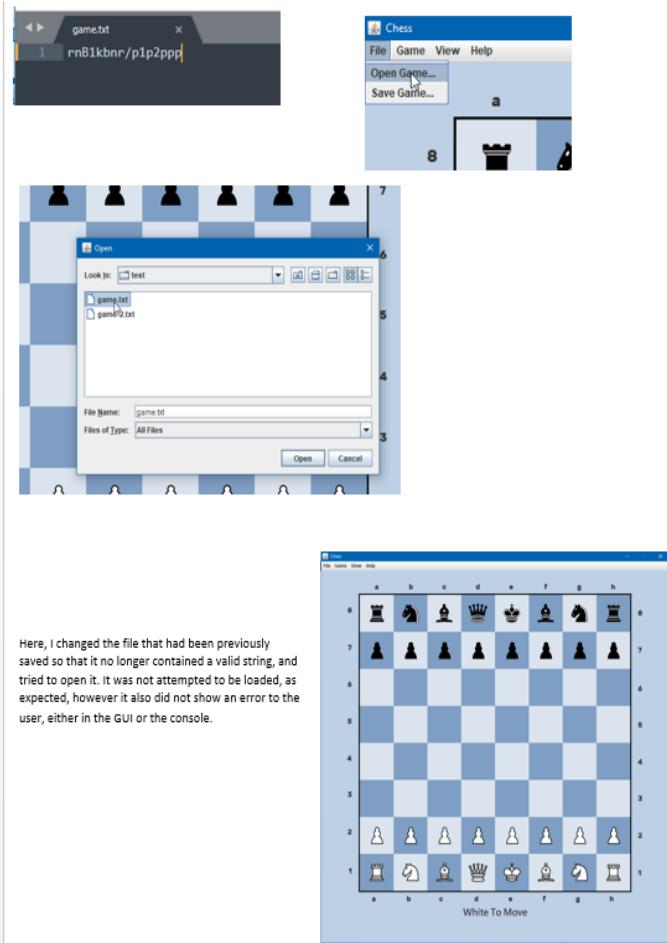
#	Criteria	Met?
20	Allows user to load previous board state from a file	Partially

Black box test #22 tests whether a previous game can be loaded from a file, and so provides evidence for this success criterion. As you can see from this, if a readable file containing a string in the correct format is selected, the game is loaded correctly. However, if an invalid file is used, the game is not loaded, as expected, however no error is shown, and so is evidence that this criterion is only partially met.

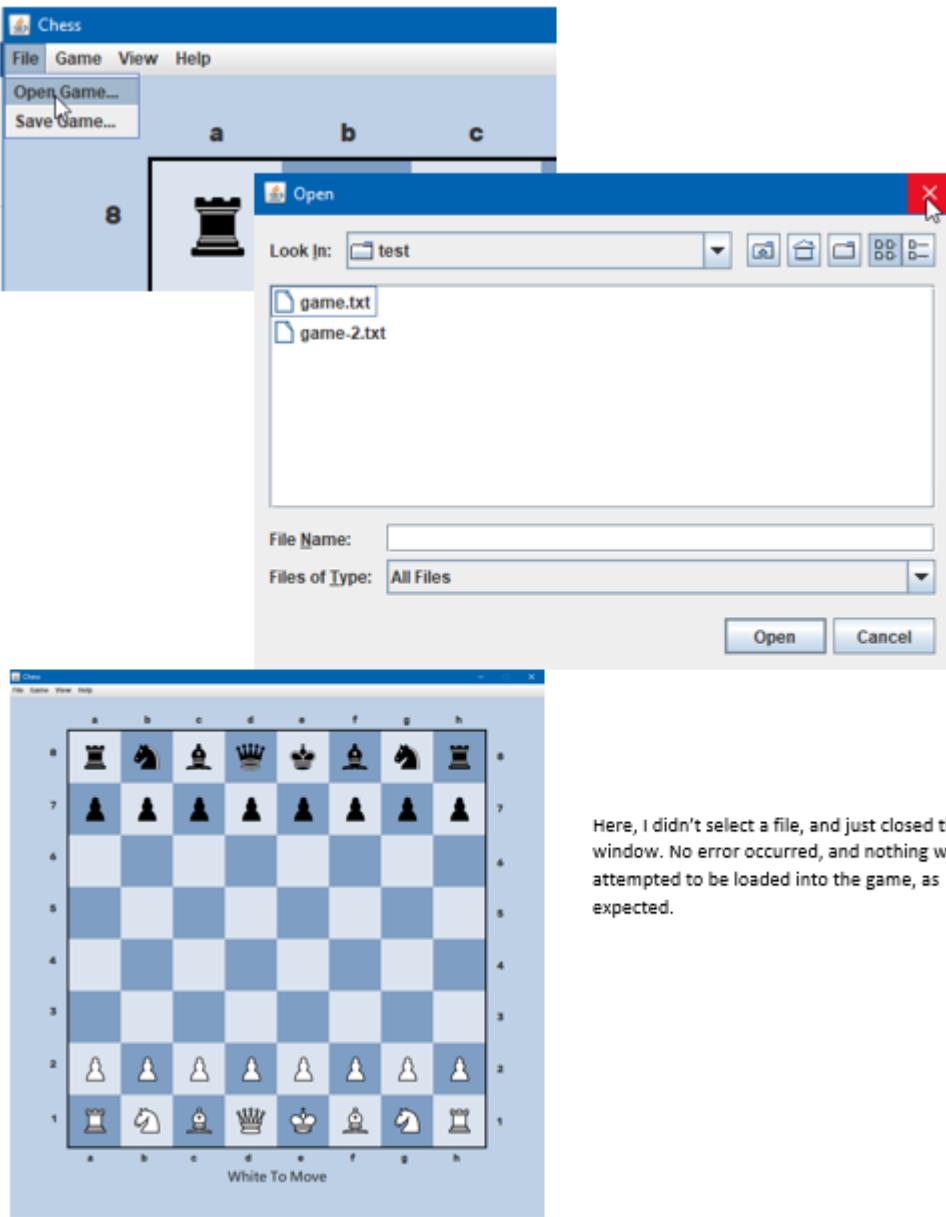
#	Test data	Expected result	Actual result	Changes needed
22	User selects a valid file containing a string in the correct format	Game should be loaded from file	Game is loaded	Red
	User selects a valid file that does not contain a string in the correct format	Game should not be loaded and error shown	Not loaded, no error	Green
	User does not select a file	Nothing should happen	Not loaded	Red



Here, when clicking "open game" the window asking user to select a file is correctly shown. When selecting the above file, the game was loaded correctly, with all of the pieces in the correct positions, and the correct player to move.



Here, I changed the file that had been previously saved so that it no longer contained a valid string, and tried to open it. It was not attempted to be loaded, as expected, however it also did not show an error to the user, either in the GUI or the console.



Here, I didn't select a file, and just closed the window. No error occurred, and nothing was attempted to be loaded into the game, as expected.

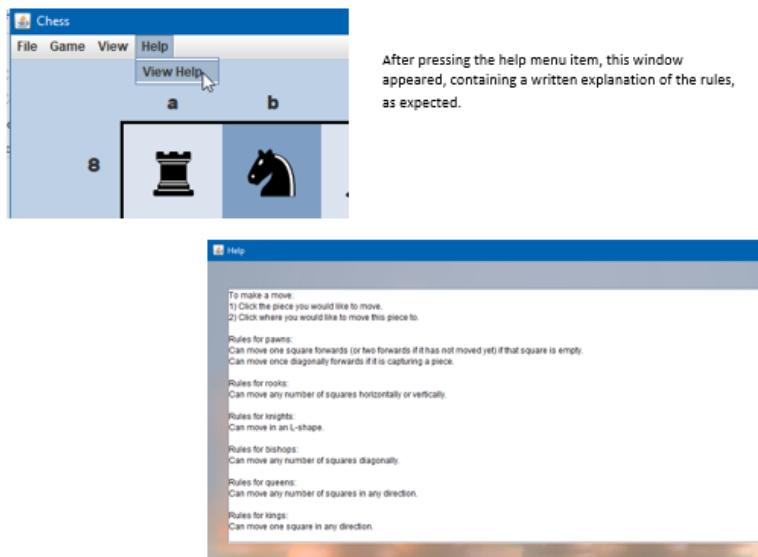
Also, during acceptance testing, the client said that *"it does allow me to open a game that I previously saved using this program, however I think this could again be improved if it used the standard FEN notation, as it would allow me to open games from other programs as well."*. This is further evidence that this criterion has only been partially met – the client would like the string to be in a different more standard format, but he agrees that it does still work as it is.

Overall, this criterion has been partially met – whilst it does technically work, there are improvements which could be made. In order to make this fully met, I could change the format in which the game string is checked to follow to be in the standard FEN format, and to display an error dialog to the user if the file cannot be read from / if the format is invalid.

#	Criteria	Met?
21	A way of showing the user a written explanation of the rules	Fully

Back box test #22 tests whether this is working correctly or not – as you can see, this test was successful and the window appeared correctly, providing some evidence that this criterion has been met.

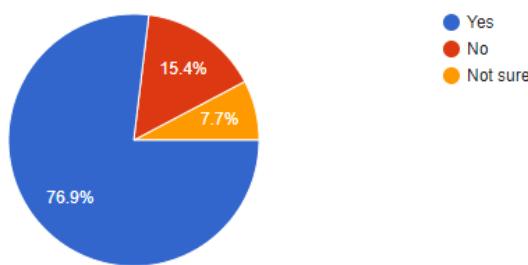
#	Test data	Expected result	Actual result	Changes needed
22	User presses help menu item	Dialog should be shown to user, showing a written explanation of the rules and how to use the program	Help window is shown	



During beta testing, I asked the end users whether they thought the window displaying instructions on how to use the program was a useful feature.

Was the window containing instructions on how to use the program a useful feature?

13 responses



The majority of them thought that it was a useful feature, providing evidence that it is a fairly effective usability feature / that this criterion has been met. Whilst there were a few people who thought this was not a useful feature, this could potentially be because they were already confident with the rules of chess and so did not find it helpful.

However, one person commented that the “text in help window shouldn’t be scrolled down already”. This could be another reason as to why some thought it wasn’t a very useful feature.

I then quickly altered the help window so that it didn’t use a text box, fixing this issue.

Here, you can see a screenshot of where I tested this, providing evidence that this criterion has been met – when I pressed the help menu item, the help window appeared, without the problems of the previous text box.



When I ran this and pressed the help menu item, the help window now looks like this, fixing the problem of the text box being scrolled down already when the help window is opened, and allows the user to see all of the rules at once without needing to scroll.

Future maintenance and limitations

I think that my solution uses clear variable names and has comments throughout, aiding in maintainability if any errors are found, if something needs to be added / changed, or if the software becomes outdated or incompatible with newer systems.

There are a few things which I would've liked to change / add to this program if I had more time. These are:

One thing which could be changed about the program is how it shows the previous move that was made. Instead of just showing the start and end position, it could show the actual path the piece took. I could do this by changing the “resetButtonColours” subroutine. Also, I think it would be better if the user was able to have the option of disabling this feature. This could be achieved by storing a boolean indicating whether the previous move should be highlighted or not, and only changing the background colours of the previous move to orange if this boolean is true.

Another thing which could be changed is how this program saves / loads games. If it used standard FEN notation instead of what it currently uses, the user would be able to open games from another program in this one, or open games from this program in another. This could be achieved by changing the subroutine that generates a string representing the current state of the game and the subroutine which loads a game from a string to recognise FEN notation.

Another feature that could be added is the ability to play against different AI difficulties. This could be implemented by allowing the user to select a difficulty at the start of the game, and depending on the difficulty selected, the AI could look a different number of moves ahead / use a different search depth, looking less moves ahead for an easier AI and more moves ahead for a harder one.

Another feature which could be implemented is the ability to use different icons. There could be a variety of different images for the same type of the piece, with the user being able to select a certain set of icons. The path that the “img” attribute of each piece could change depending on the set of icons the user selects.

Also, the user could be able to press a “hint” button, and have the program determine the best move for them to make. This could be achieved by using the same methods that the computer uses to determine what move to make, but then instead of automatically making the move, informing the user that this is a possible move they could make, and then allowing them to make this move if they would like to.

Full code

Bishop.java

```
public class Bishop extends Piece {

    public Bishop(int colour) {
        super(4, colour, colour == 0 ? "/white-bishop.png" : "/black-bishop.png");
    }

    // Determines if the move is pseudo-legal
    @Override
    public boolean isValidMove(String move, Piece[][] board) {
        // Gets start position and end position as ints from the move string
        int startPos = Integer.valueOf(move.substring(0, 2));
        int endPos = Integer.valueOf(move.substring(2, 4));

        boolean capturingPieceOfDifferentColour = (board[endPos / 8][endPos % 8] != null)
            && (board[startPos / 8][startPos % 8].getColour() != board[endPos / 8][endPos % 8].getColour());
        boolean movingToEmptySquare = board[endPos / 8][endPos % 8] == null;

        int columnChange = Math.abs(endPos % 8 - startPos % 8);
        int rowChange = Math.abs(endPos / 8 - startPos / 8);
        boolean isDiagonal = columnChange == rowChange;

        if ((capturingPieceOfDifferentColour || movingToEmptySquare) && isDiagonal) {
            if (endPos % 8 < startPos % 8 && endPos / 8 < startPos / 8) { // Moving north west
                int col = startPos % 8;
                int row = startPos / 8;
                while (col - 1 >= 0 && row - 1 >= 0 && board[row - 1][col - 1] == null) {
                    col--;
                    row--;
                }
                return endPos % 8 >= col && endPos / 8 >= row || capturingPieceOfDifferentColour && endPos % 8 >= col - 1 && endPos / 8 >= row - 1;
            } else if (endPos % 8 > startPos % 8 && endPos / 8 < startPos / 8) { // Moving north east
                int col = startPos % 8;
                int row = startPos / 8;
                while (col + 1 < 8 && row - 1 >= 0 && board[row - 1][col + 1] == null) {
                    col++;
                    row--;
                }
                return endPos % 8 <= col && endPos / 8 >= row || capturingPieceOfDifferentColour && endPos % 8 <= col + 1 && endPos / 8 >= row - 1;
            } else if (endPos % 8 > startPos % 8 && endPos / 8 > startPos / 8) { // Moving south east
                int col = startPos % 8;
                int row = startPos / 8;
                while (col + 1 < 8 && row + 1 < 8 && board[row + 1][col + 1] == null) {
                    col++;
                    row++;
                }
                return endPos % 8 <= col && endPos / 8 <= row || capturingPieceOfDifferentColour && endPos % 8 <= col + 1 && endPos / 8 <= row + 1;
            } else if (endPos % 8 < startPos % 8 && endPos / 8 > startPos / 8) { // Moving south west
                int col = startPos % 8;
                int row = startPos / 8;
                while (col - 1 >= 0 && row + 1 < 8 && board[row + 1][col - 1] == null) {
                    col--;
                    row++;
                }
                return endPos % 8 >= col && endPos / 8 <= row || capturingPieceOfDifferentColour && endPos % 8 >= col - 1 && endPos / 8 <= row + 1;
            }
        }
        return false;
    }

    @Override
    public String toString() {
        if (this.getColour() == 0) {
            return "♝";
        } else {
            return "♝";
        }
    }
}
```

Chess.java

```
import java.awt.Color;
import java.io.File;
import java.io.FileWriter;
import java.util.Scanner;
import javax.swing.JButton;
import javax.swing.JColorChooser;
import javax.swing.JFileChooser;

public class Chess extends javax.swing.JFrame {

    private ChessBoard board;
    private int startPos; // Location of piece user selects to move, -1 = not yet chosen
    private int endPos; // Location user selects to move piece to, -1 = not yet chosen
    private Color pieceSelectionColour;
    private Color validPositionColour;
    private Color noAvailableMovesColour;
    private Color previousMoveColour;
    private Color lightSquareColour;
    private Color darkSquareColour;

    private JButton[][] buttons;

    /**
     * Creates new form NewJFrame
     */
    public Chess() {
        initComponents();
        this.board = new ChessBoard();
        this.board.selectPlayer(this, true); // Asks user whether they are playing against the computer

        this.startPos = -1;
        this.endPos = -1;

        this.pieceSelectionColour = Color.CYAN;
        this.validPositionColour = new Color(244, 255, 82); // Yellow
        this.noAvailableMovesColour = new Color(255, 0, 53); // Red
        this.previousMoveColour = Color.ORANGE;
        this.lightSquareColour = new Color(219, 228, 238); // Light blue
        this.darkSquareColour = new Color(127, 158, 195); // Dark blue

        this.buttons = new JButton[][]{
            {this.JButtonA8, this.JButtonB8, this.JButtonC8, this.JButtonD8, this.JButtonE8, this.JButtonF8, this.JButtonG8, this.JButtonH8},
            {this.JButtonA7, this.JButtonB7, this.JButtonC7, this.JButtonD7, this.JButtonE7, this.JButtonF7, this.JButtonG7, this.JButtonH7},
            {this.JButtonA6, this.JButtonB6, this.JButtonC6, this.JButtonD6, this.JButtonE6, this.JButtonF6, this.JButtonG6, this.JButtonH6},
            {this.JButtonA5, this.JButtonB5, this.JButtonC5, this.JButtonD5, this.JButtonE5, this.JButtonF5, this.JButtonG5, this.JButtonH5},
            {this.JButtonA4, this.JButtonB4, this.JButtonC4, this.JButtonD4, this.JButtonE4, this.JButtonF4, this.JButtonG4, this.JButtonH4},
            {this.JButtonA3, this.JButtonB3, this.JButtonC3, this.JButtonD3, this.JButtonE3, this.JButtonF3, this.JButtonG3, this.JButtonH3},
            {this.JButtonA2, this.JButtonB2, this.JButtonC2, this.JButtonD2, this.JButtonE2, this.JButtonF2, this.JButtonG2, this.JButtonH2},
            {this.JButtonA1, this.JButtonB1, this.JButtonC1, this.JButtonD1, this.JButtonE1, this.JButtonF1, this.JButtonG1, this.JButtonH1}
        };
    }

    this.updateGUI();
```

```

// Updates text of buttons & JLabels, resets button colours, displays dialog if game has ended
public void updateGUI() {
    Piece[][] cb = this.board.getBoard(); // Gets the array of Pieces
    // Sets icon of buttons to show the board to the user
    for (int i = 0; i < cb.length; i++) {
        for (int j = 0; j < cb[i].length; j++) {
            if (cb[i][j] != null) { // If there is a piece at this location
                this.buttons[i][j].setIcon(new javax.swing.ImageIcon(getClass().getResource(cb[i][j].getImage())));
            } else {
                this.buttons[i][j].setIcon(null);
            }
        }
    }

    // Sets JLabel telling user whose turn it is
    if (this.board.getCurrentTurn() % 2 == 0) {
        this.turnLabel.setText("White To Move");
    } else {
        this.turnLabel.setText("Black To Move");
    }

    this.resetButtonColours();

    // Determines whether the game has ended
    if (this.board.isInCheck(0)) { // If white is in check
        this.checkLabel.setText("White Is In Check");
        if (this.board.isInCheckMate(0)) {
            ShowWinner winnerDialog = new ShowWinner(new javax.swing.JFrame(), true);
            winnerDialog.setWinner(1);
            winnerDialog.setVisible(true);
            this.endGame();
        }
    } else if (this.board.isInCheck(1)) { // If black is in check
        this.checkLabel.setText("Black Is In Check");
        if (this.board.isInCheckMate(1)) {
            ShowWinner winnerDialog = new ShowWinner(new javax.swing.JFrame(), true);
            winnerDialog.setWinner(0);
            winnerDialog.setVisible(true);
            this.endGame();
        }
    } else if (this.board.currentPlayerHasNoLegalmoves()) {
        ShowWinner winnerDialog = new ShowWinner(new javax.swing.JFrame(), true);
        winnerDialog.setWinner(2);
        winnerDialog.setVisible(true);
        this.endGame();
    } else {
        this.checkLabel.setText("");
    }
}

// Allows user to select positions on the board and make moves
public void buttonPressed(int pos) {
    // If piece to move not chosen and location is not empty and piece belongs to current player
    if (this.startPos == -1 && this.board.hasPieceAtPosition(pos)
        && this.board.getColourOfPieceAtPosition(pos) == this.board.getCurrentTurn() % 2) {
        this.buttons[pos / 8][pos % 8].setBackground(this.pieceSelectionColour); // Highlight selected piece in different colour
        this.showPossibleEndPositions(pos);

        this.startPos = pos;
    } else if (this.endPos == -1 && this.startPos != -1) { // If piece to move is chosen but location to move it to is not chosen
        this.endPos = pos;

        this.board.makeMove(this.generateMoveString());
        this.updateGUI();

        this.startPos = -1;
        this.endPos = -1;

        // If user is playing against the computer and it is blacks turn to move, allow computer to move
        if (this.board.isPlayingAgainstComputer() && this.board.getCurrentTurn() % 2 == 1) {
            this.board.makeComputerMove();
            this.updateGUI();
        }
    }
}

```

```

// Generates string representing move using startPos and endPos
private String generateMoveString() {
    String move = "";

    // Each position should be two digits long so add leading "0" if < 10
    if (this.startPos < 10) {
        move += "0";
    }
    move += this.startPos;

    if (this.endPos < 10) {
        move += "0";
    }
    move += this.endPos;

    return move;
}

// Highlights places piece can move to in different colour
private void showPossibleEndPositions(int pos) {
    boolean[][] validPositions = this.board.getValidEndPositions(pos);
    boolean canMakeMove = false;

    for (int i = 0; i < this.buttons.length; i++) {
        for (int j = 0; j < this.buttons[i].length; j++) {
            if (validPositions[i][j]) {
                this.buttons[i][j].setBackground(this.validPositionColour); // Highlight valid end position in different colour
                canMakeMove = true;
            }
        }
    }

    if (!canMakeMove) {
        this.buttons[pos / 8][pos % 8].setBackground(this.noAvailableMovesColour); // Highlight piece in different colour
    }
}

// Resets button colours to default
private void resetButtonColours() {
    // Resets background colour of all buttons:
    for (int i = 0; i < this.buttons.length; i++) {
        for (int j = 0; j < this.buttons[i].length; j++) {
            if (i % 2 == 0 && j % 2 == 0 || i % 2 != 0 && j % 2 != 0) {
                this.buttons[i][j].setBackground(this.lightSquareColour);
            } else {
                this.buttons[i][j].setBackground(this.darkSquareColour);
            }
        }
    }

    // Sets border of all buttons
    for (int i = 0; i < 64; i++) {
        if ((i / 8) % 2 == 0 && (i % 8) % 2 != 0 || (i / 8) % 2 != 0 && (i % 8) % 2 == 0) {
            this.buttons[i / 8][i % 8].setBorder(javax.swing.BorderFactory.createLineBorder(this.darkSquareColour, 10));
        } else {
            this.buttons[i / 8][i % 8].setBorder(javax.swing.BorderFactory.createLineBorder(this.lightSquareColour, 10));
        }
    }

    String lastMove = this.board.getLastMove(); // Last move made, empty = move has not yet been made
    if (!lastMove.isEmpty()) {
        int startPos = Integer.valueOf(lastMove.substring(0, 2)); // First two chars = start position
        int endPos = Integer.valueOf(lastMove.substring(2, 4)); // Next two chars = end position
        // Highlights previous move in different colour:
        this.buttons[startPos / 8][startPos % 8].setBackground(this.previousMoveColour);
        this.buttons[endPos / 8][endPos % 8].setBackground(this.previousMoveColour);
    }
}

// Resets the board / restarts the game
private void endGame() {
    this.board = new ChessBoard();
    this.board.selectPlayer(this, true); // Asks user whether they are playing against the computer
    this.startPos = -1;
    this.endPos = -1;
    this.updateGUI();
}

```

```
/**  
 * This method is called from within the constructor to initialise the form.  
 * WARNING: Do NOT modify this code. The content of this method is always  
 * regenerated by the Form Editor.  
 */  
@SuppressWarnings("unchecked")  
Generated Code  
  
private void helpMenuItemActionPerformed(java.awt.event.ActionEvent evt) {  
    // TODO add your handling code here:  
    // Shows help dialog to user  
    Help helpDialog = new Help(new javax.swing.JFrame(), false);  
    helpDialog.setVisible(true);  
}  
  
private void bkgdColMenuItemActionPerformed(java.awt.event.ActionEvent evt) {  
    // TODO add your handling code here:  
    // Asks user to select a colour  
    Color newColour = JColorChooser.showDialog(this, "Select Background Colour", new Color(189, 208, 229));  
    if (newColour != null) { // If they picked a colour  
        // Sets colour of JPanel acting as background to this colour  
        this.backgroundPanel.setBackground(newColour);  
    }  
}  
  
private void jButtonA8ActionPerformed(java.awt.event.ActionEvent evt) {  
    // TODO add your handling code here:  
    buttonPressed(0);  
}  
  
private void jButtonB8ActionPerformed(java.awt.event.ActionEvent evt) {  
    // TODO add your handling code here:  
    buttonPressed(1);  
}  
  
private void jButtonC8ActionPerformed(java.awt.event.ActionEvent evt) {  
    // TODO add your handling code here:  
    buttonPressed(2);  
}  
  
private void jButtonD8ActionPerformed(java.awt.event.ActionEvent evt) {  
    // TODO add your handling code here:  
    buttonPressed(3);  
}  
  
private void jButtonE8ActionPerformed(java.awt.event.ActionEvent evt) {  
    // TODO add your handling code here:  
    buttonPressed(4);  
}  
  
private void jButtonF8ActionPerformed(java.awt.event.ActionEvent evt) {  
    // TODO add your handling code here:  
    buttonPressed(5);  
}  
  
private void jButtonG8ActionPerformed(java.awt.event.ActionEvent evt) {  
    // TODO add your handling code here:  
    buttonPressed(6);  
}  
  
private void jButtonH8ActionPerformed(java.awt.event.ActionEvent evt) {  
    // TODO add your handling code here:  
    buttonPressed(7);  
}
```

```
private void jButtonA7ActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
    buttonPressed(8);
}

private void jButtonB7ActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
    buttonPressed(9);
}

private void jButtonC7ActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
    buttonPressed(10);
}

private void jButtonD7ActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
    buttonPressed(11);
}

private void jButtonE7ActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
    buttonPressed(12);
}

private void jButtonF7ActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
    buttonPressed(13);
}

private void jButtonG7ActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
    buttonPressed(14);
}

private void jButtonH7ActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
    buttonPressed(15);
}
```

```
private void jButtonA6ActionPerformed(java.awt.event.ActionEvent evt) {  
    // TODO add your handling code here:  
    buttonPressed(16);  
}  
  
private void jButtonB6ActionPerformed(java.awt.event.ActionEvent evt) {  
    // TODO add your handling code here:  
    buttonPressed(17);  
}  
  
private void jButtonC6ActionPerformed(java.awt.event.ActionEvent evt) {  
    // TODO add your handling code here:  
    buttonPressed(18);  
}  
  
private void jButtonD6ActionPerformed(java.awt.event.ActionEvent evt) {  
    // TODO add your handling code here:  
    buttonPressed(19);  
}  
  
private void jButtonE6ActionPerformed(java.awt.event.ActionEvent evt) {  
    // TODO add your handling code here:  
    buttonPressed(20);  
}  
  
private void jButtonF6ActionPerformed(java.awt.event.ActionEvent evt) {  
    // TODO add your handling code here:  
    buttonPressed(21);  
}  
  
private void jButtonG6ActionPerformed(java.awt.event.ActionEvent evt) {  
    // TODO add your handling code here:  
    buttonPressed(22);  
}  
  
private void jButtonH6ActionPerformed(java.awt.event.ActionEvent evt) {  
    // TODO add your handling code here:  
    buttonPressed(23);  
}
```

```
private void jButtonA5ActionPerformed(java.awt.event.ActionEvent evt) {  
    // TODO add your handling code here:  
    buttonPressed(24);  
}  
  
private void jButtonB5ActionPerformed(java.awt.event.ActionEvent evt) {  
    // TODO add your handling code here:  
    buttonPressed(25);  
}  
  
private void jButtonC5ActionPerformed(java.awt.event.ActionEvent evt) {  
    // TODO add your handling code here:  
    buttonPressed(26);  
}  
  
private void jButtonD5ActionPerformed(java.awt.event.ActionEvent evt) {  
    // TODO add your handling code here:  
    buttonPressed(27);  
}  
  
private void jButtonE5ActionPerformed(java.awt.event.ActionEvent evt) {  
    // TODO add your handling code here:  
    buttonPressed(28);  
}  
  
private void jButtonF5ActionPerformed(java.awt.event.ActionEvent evt) {  
    // TODO add your handling code here:  
    buttonPressed(29);  
}  
  
private void jButtonG5ActionPerformed(java.awt.event.ActionEvent evt) {  
    // TODO add your handling code here:  
    buttonPressed(30);  
}  
  
private void jButtonH5ActionPerformed(java.awt.event.ActionEvent evt) {  
    // TODO add your handling code here:  
    buttonPressed(31);  
}
```

```
private void jButtonA4ActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
    buttonPressed(32);
}

private void jButtonB4ActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
    buttonPressed(33);
}

private void jButtonC4ActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
    buttonPressed(34);
}

private void jButtonD4ActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
    buttonPressed(35);
}

private void jButtonE4ActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
    buttonPressed(36);
}

private void jButtonF4ActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
    buttonPressed(37);
}

private void jButtonG4ActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
    buttonPressed(38);
}

private void jButtonH4ActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
    buttonPressed(39);
}
```

```
private void jButtonA3ActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
    buttonPressed(40);
}

private void jButtonB3ActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
    buttonPressed(41);
}

private void jButtonC3ActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
    buttonPressed(42);
}

private void jButtonD3ActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
    buttonPressed(43);
}

private void jButtonE3ActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
    buttonPressed(44);
}

private void jButtonF3ActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
    buttonPressed(45);
}

private void jButtonG3ActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
    buttonPressed(46);
}

private void jButtonH3ActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
    buttonPressed(47);
}
```

```
private void jButtonA2ActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
    buttonPressed(48);
}

private void jButtonB2ActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
    buttonPressed(49);
}

private void jButtonC2ActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
    buttonPressed(50);
}

private void jButtonD2ActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
    buttonPressed(51);
}

private void jButtonE2ActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
    buttonPressed(52);
}

private void jButtonF2ActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
    buttonPressed(53);
}

private void jButtonG2ActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
    buttonPressed(54);
}

private void jButtonH2ActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
    buttonPressed(55);
}
```

```
private void jButtonA1ActionPerformed(java.awt.event.ActionEvent evt) {  
    // TODO add your handling code here:  
    buttonPressed(56);  
}  
  
private void jButtonB1ActionPerformed(java.awt.event.ActionEvent evt) {  
    // TODO add your handling code here:  
    buttonPressed(57);  
}  
  
private void jButtonC1ActionPerformed(java.awt.event.ActionEvent evt) {  
    // TODO add your handling code here:  
    buttonPressed(58);  
}  
  
private void jButtonD1ActionPerformed(java.awt.event.ActionEvent evt) {  
    // TODO add your handling code here:  
    buttonPressed(59);  
}  
  
private void jButtonE1ActionPerformed(java.awt.event.ActionEvent evt) {  
    // TODO add your handling code here:  
    buttonPressed(60);  
}  
  
private void jButtonF1ActionPerformed(java.awt.event.ActionEvent evt) {  
    // TODO add your handling code here:  
    buttonPressed(61);  
}  
  
private void jButtonG1ActionPerformed(java.awt.event.ActionEvent evt) {  
    // TODO add your handling code here:  
    buttonPressed(62);  
}  
  
private void jButtonH1ActionPerformed(java.awt.event.ActionEvent evt) {  
    // TODO add your handling code here:  
    buttonPressed(63);  
}
```

```
private void surrenderMenuItemActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
    this.endGame();
}

private void undoMenuItemActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
    if (this.board.getCurrentTurn() > 1) { // If two or more moves have been made
        this.board.undoMove();
        this.board.undoMove();
        this.updateGUI();
    }
}

private void saveGameMenuItemActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
    JFileChooser fileChooser = new JFileChooser();

    // Shows dialog to user asking them to select a file, returns JFileChooser.APPROVE_OPTION if they have selected one:
    boolean userHasSelectedFile = fileChooser.showSaveDialog(null) == JFileChooser.APPROVE_OPTION;

    if (userHasSelectedFile) {
        File file = fileChooser.getSelectedFile(); // Gets the file the user selected
        String fen = this.board.getGameString(); // Generates string representing the board
        try {
            FileWriter writer = new FileWriter(file);
            writer.write(fen); // Writes this string to the file
            writer.close(); // Closes the file
        } catch (Exception ex) {
            System.err.println(ex);
        }
    }
}

private void lightSquareColMenuItemActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
    // Change light square colour:
    Color newColour = JColorChooser.showDialog(this, "Select Light Square Colour", new Color(219, 228, 238));
    if (newColour != null) {
        this.lightSquareColour = newColour;
        this.updateGUI();
    }
}
```

```
private void darkSquareColMenuItemActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
    // Change dark square colour:
    Color newColour = JColorChooser.showDialog(this, "Select Dark Square Colour", new Color(127, 158, 195));
    if (newColour != null) {
        this.darkSquareColour = newColour;
        this.updateGUI();
    }
}

private void textColMenuItemActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
    // Change colour of all JLabels:
    Color newColour = JColorChooser.showDialog(this, "Select Text Colour", Color.BLACK);
    if (newColour != null) {
        this.jLabel1.setForeground(newColour);
        this.jLabel2.setForeground(newColour);
        this.jLabel3.setForeground(newColour);
        this.jLabel4.setForeground(newColour);
        this.jLabel5.setForeground(newColour);
        this.jLabel6.setForeground(newColour);
        this.jLabel7.setForeground(newColour);
        this.jLabel8.setForeground(newColour);
        this.jLabel9.setForeground(newColour);
        this.jLabel10.setForeground(newColour);
        this.jLabel11.setForeground(newColour);
        this.jLabel12.setForeground(newColour);
        this.jLabel13.setForeground(newColour);
        this.jLabel14.setForeground(newColour);
        this.jLabel15.setForeground(newColour);
        this.jLabel16.setForeground(newColour);
        this.jLabel17.setForeground(newColour);
        this.jLabel18.setForeground(newColour);
        this.jLabel19.setForeground(newColour);
        this.jLabel20.setForeground(newColour);
        this.jLabel21.setForeground(newColour);
        this.jLabel22.setForeground(newColour);
        this.jLabel23.setForeground(newColour);
        this.jLabel24.setForeground(newColour);
        this.jLabel25.setForeground(newColour);
        this.jLabel26.setForeground(newColour);
        this.jLabel27.setForeground(newColour);
        this.jLabel28.setForeground(newColour);
        this.jLabel29.setForeground(newColour);
        this.jLabel30.setForeground(newColour);
        this.jLabel31.setForeground(newColour);
        this.jLabel32.setForeground(newColour);
        this.checkLabel.setForeground(newColour);
        this.turnLabel.setForeground(newColour);
    }
}
```

```
private void selectedPieceColMenuItemActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
    // Change piece selection colour:
    Color newColour = JColorChooser.showDialog(this, "Select Piece Selection Colour", Color.CYAN);
    if (newColour != null) {
        this.pieceSelectionColour = newColour;
    }
}

private void validPosColMenuItemActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
    // Change valid position colour:
    Color newColour = JColorChooser.showDialog(this, "Select Valid Position Colour", new Color(244, 255, 82));
    if (newColour != null) {
        this.validPositionColour = newColour;
    }
}

private void noMovesColMenuItemActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
    // Change no available moves colour:
    Color newColour = JColorChooser.showDialog(this, "Select No Available Moves Colour", new Color(255, 0, 53));
    if (newColour != null) {
        this.noAvailableMovesColour = newColour;
    }
}

private void prevMoveColMenuItemActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
    // Change previous move colour:
    Color newColour = JColorChooser.showDialog(this, "Select Previous Move Colour", Color.ORANGE);
    if (newColour != null) {
        this.previousMoveColour = newColour;
    }
}

private void openGameMenuItemActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
    JFileChooser fileChooser = new JFileChooser();

    // Shows dialog to user asking them to select a file, returns JFileChooser.APPROVE_OPTION if they have selected one:
    boolean userHasSelectedFile = fileChooser.showOpenDialog(null) == JFileChooser.APPROVE_OPTION;

    if (userHasSelectedFile) {
        File file = fileChooser.getSelectedFile(); // Gets the file the user selected
        try {
            Scanner reader = new Scanner(file);
            if (reader.hasNextLine()) { // If the file has a next line
                String fen = reader.nextLine(); // Get the next line from the file
                this.board.loadGameString(fen); // Load this string into the board
            }
            this.updateGUI();
        } catch (Exception ex) {
            System.err.println(ex);
        }
    }
}

/**
 * @param args the command line arguments
 */
public static void main(String args[]) {

    /* Create and display the form */
    java.awt.EventQueue.invokeLater(new Runnable() {
        public void run() {
            new Chess().setVisible(true);
        }
    });
}
```

```

// Variables declaration - do not modify
private javax.swing.JPanel backgroundPanel;
private javax.swing.JMenuItem bkgdColMenuItem;
private javax.swing.JLabel checkLabel;
private javax.swing.JMenuItem darkSquareColMenuItem;
private javax.swing.JMenu fileMenu;
private javax.swing.JMenu gameMenu;
private javax.swing.JMenu helpMenu;
private javax.swing.JMenuItem helpMenuItem;
private javax.swing.JButton jButtonA1;
private javax.swing.JButton jButtonA2;
private javax.swing.JButton jButtonA3;
private javax.swing.JButton jButtonA4;
private javax.swing.JButton jButtonA5;
private javax.swing.JButton jButtonA6;
private javax.swing.JButton jButtonA7;
private javax.swing.JButton jButtonA8;
private javax.swing.JButton jButtonB1;
private javax.swing.JButton jButtonB2;
private javax.swing.JButton jButtonB3;
private javax.swing.JButton jButtonB4;
private javax.swing.JButton jButtonB5;
private javax.swing.JButton jButtonB6;
private javax.swing.JButton jButtonB7;
private javax.swing.JButton jButtonB8;
private javax.swing.JButton jButtonC1;
private javax.swing.JButton jButtonC2;
private javax.swing.JButton jButtonC3;
private javax.swing.JButton jButtonC4;
private javax.swing.JButton jButtonC5;
private javax.swing.JButton jButtonC6;
private javax.swing.JButton jButtonC7;
private javax.swing.JButton jButtonC8;
private javax.swing.JButton jButtonD1;
private javax.swing.JButton jButtonD2;
private javax.swing.JButton jButtonD3;
private javax.swing.JButton jButtonD4;
private javax.swing.JButton jButtonD5;
private javax.swing.JButton jButtonD6;
private javax.swing.JButton jButtonD7;
private javax.swing.JButton jButtonD8;
private javax.swing.JButton jButtonE1;
private javax.swing.JButton jButtonE2;
private javax.swing.JButton jButtonE3;
private javax.swing.JButton jButtonE4;
private javax.swing.JButton jButtonE5;
private javax.swing.JButton jButtonE6;
private javax.swing.JButton jButtonE7;
private javax.swing.JButton jButtonE8;
private javax.swing.JButton jButtonF1;
private javax.swing.JButton jButtonF2;
private javax.swing.JButton jButtonF3;
private javax.swing.JButton jButtonF4;
private javax.swing.JButton jButtonF5;
private javax.swing.JButton jButtonF6;
private javax.swing.JButton jButtonF7;
private javax.swing.JButton jButtonF8;
private javax.swing.JButton jButtonG1;
private javax.swing.JButton jButtonG2;

```

```

private javax.swing.JButton jButtonG3;
private javax.swing.JButton jButtonG4;
private javax.swing.JButton jButtonG5;
private javax.swing.JButton jButtonG6;
private javax.swing.JButton jButtonG7;
private javax.swing.JButton jButtonG8;
private javax.swing.JButton jButtonH1;
private javax.swing.JButton jButtonH2;
private javax.swing.JButton jButtonH3;
private javax.swing.JButton jButtonH4;
private javax.swing.JButton jButtonH5;
private javax.swing.JButton jButtonH6;
private javax.swing.JButton jButtonH7;
private javax.swing.JButton jButtonH8;
private javax.swing.JLabel jLabel1;
private javax.swing.JLabel jLabel10;
private javax.swing.JLabel jLabel11;
private javax.swing.JLabel jLabel12;
private javax.swing.JLabel jLabel13;
private javax.swing.JLabel jLabel14;
private javax.swing.JLabel jLabel15;
private javax.swing.JLabel jLabel16;
private javax.swing.JLabel jLabel17;
private javax.swing.JLabel jLabel18;
private javax.swing.JLabel jLabel19;
private javax.swing.JLabel jLabel2;
private javax.swing.JLabel jLabel20;
private javax.swing.JLabel jLabel21;
private javax.swing.JLabel jLabel22;
private javax.swing.JLabel jLabel23;
private javax.swing.JLabel jLabel24;
private javax.swing.JLabel jLabel25;
private javax.swing.JLabel jLabel26;
private javax.swing.JLabel jLabel27;
private javax.swing.JLabel jLabel28;
private javax.swing.JLabel jLabel29;
private javax.swing.JLabel jLabel3;
private javax.swing.JLabel jLabel30;
private javax.swing.JLabel jLabel31;
private javax.swing.JLabel jLabel32;
private javax.swing.JLabel jLabel4;
private javax.swing.JLabel jLabel5;
private javax.swing.JLabel jLabel6;
private javax.swing.JLabel jLabel7;
private javax.swing.JLabel jLabel8;
private javax.swing.JLabel jLabel9;
private javax.swing.JMenuBar jMenuBar1;
private javax.swing.JPanel jPanel1;
private javax.swing.JMenuItem lightSquareColMenuItem;
private javax.swing.JMenuItem noMovesColMenuItem;
private javax.swing.JMenuItem openGameMenuItem;
private javax.swing.JMenuItem prevMoveColMenuItem;
private javax.swing.JMenuItem saveGameMenuItem;
private javax.swing.JMenuItem selectedPieceColMenuItem;
private javax.swing.JMenuItem surrenderMenuItem;
private javax.swing.JMenuItem textColMenuItem;

```

```

private javax.swing.JMenuBar jMenuBar1;
private javax.swing.JPanel jPanel2;
private javax.swing.JMenuItem lightSquareColMenuItem;
private javax.swing.JMenuItem noMovesColMenuItem;
private javax.swing.JMenuItem openGameMenuItem;
private javax.swing.JMenuItem prevMoveColMenuItem;
private javax.swing.JMenuItem saveGameMenuItem;
private javax.swing.JMenuItem selectedPieceColMenuItem;
private javax.swing.JMenuItem surrenderMenuItem;
private javax.swing.JMenuItem textColMenuItem;
private javax.swing.JLabel turnLabel;
private javax.swing.JMenuItem undoMenuItem;
private javax.swing.JMenuItem validPosColMenuItem;
private javax.swing.JMenu viewMenu;
// End of variables declaration
}

```

ChessBoard.java

```
import java.util.ArrayList;
import javax.swing.JFrame;

public class ChessBoard {

    private Piece[][] board; // null = empty space
    private int currentTurn; // even num = white, odd num = black
    private MoveHistory moveHistory;
    private boolean playingAgainstComputer;

    private final int[][] pawnEvalBonus = {
        {-900, -900, -900, -900, -900, -900, -900},
        {-50, -50, -50, -50, -50, -50, -50},
        {-10, -10, -20, -30, -30, -20, -10, -10},
        {-5, -5, -10, -25, -25, -10, -5, -5},
        {0, 0, 0, -20, -20, 0, 0, 0},
        {-5, 5, 10, 0, 0, 10, 5, -5},
        {-5, -10, -10, 20, 20, -10, -10, -5},
        {0, 0, 0, 0, 0, 0, 0, 0}
    };

    private final int[][] rookEvalBonus = {
        {0, 0, 0, 0, 0, 0, 0, 0},
        {-5, -10, -10, -10, -10, -10, -10, -5},
        {5, 0, 0, 0, 0, 0, 0, 5},
        {5, 0, 0, 0, 0, 0, 0, 5},
        {5, 0, 0, 0, 0, 0, 0, 5},
        {5, 0, 0, 0, 0, 0, 0, 5},
        {5, 0, 0, 0, 0, 0, 0, 5},
        {0, 0, 0, -5, -5, 0, 0, 0}
    };

    private final int[][] knightEvalBonus = {
        {50, 40, 30, 30, 30, 30, 40, 50},
        {40, 20, 0, 0, 0, 0, 20, 40},
        {30, 0, -10, -15, -15, -10, 0, 30},
        {30, -5, -15, -20, -20, -15, -5, 30},
        {30, 0, -15, -20, -20, -15, 0, 30},
        {30, -5, -10, -15, -15, -10, -5, 30},
        {40, 20, 0, -5, -5, 0, 20, 40},
        {50, 40, 30, 30, 30, 30, 40, 50}
    };

    private final int[][] bishopEvalBonus = {
        {20, 10, 10, 10, 10, 10, 10, 20},
        {10, 0, 0, 0, 0, 0, 0, 10},
        {10, 0, -5, -10, -10, 5, 0, 10},
        {10, -5, -5, -10, -10, -5, -5, 10},
        {10, 0, -10, -10, -10, -10, 0, 10},
        {10, -10, -10, -10, -10, -10, -10, 10},
        {10, -5, 0, 0, 0, 0, -5, 10},
        {20, 10, 10, 10, 10, 10, 10, 20}
    };

    private final int[][] queenEvalBonus = {
        {20, 10, 10, 5, 5, 10, 10, 20},
        {10, 0, 0, 0, 0, 0, 0, 10},
        {10, 0, -5, -5, -5, -5, 0, 10},
        {5, 0, -5, -5, -5, -5, 0, 5},
        {0, 0, -5, -5, -5, -5, 0, 5},
        {10, -5, -5, -5, -5, -5, 0, 10},
        {10, 0, -5, 0, 0, 0, 0, 10},
        {20, 10, 10, 5, 5, 10, 10, 20}
    };

    private final int[][] kingEvalBonus = {
        {30, 40, 40, 50, 50, 40, 40, 30},
        {30, 40, 40, 50, 50, 40, 40, 30},
        {30, 40, 40, 50, 50, 40, 40, 30},
        {30, 40, 40, 50, 50, 40, 40, 30},
        {20, 30, 30, 40, 40, 30, 30, 20},
        {10, 20, 20, 20, 20, 20, 20, 10},
        {-20, -20, 0, 0, 0, 0, -20, -20},
        {-20, -30, -10, 0, 0, -10, -30, -20}
    };
}
```

```

public ChessBoard() {
    // Setting initial layout of pieces on board:
    this.board = new Piece[][]{
        {new Rook(1), new Knight(1), new Bishop(1), new Queen(1), new King(1), new Bishop(1), new Knight(1), new Rook(1)},
        {new Pawn(1), new Pawn(1)},
        {null, null, null, null, null, null, null, null},
        {new Pawn(0), new Pawn(0)},
        {new Rook(0), new Knight(0), new Bishop(0), new Queen(0), new King(0), new Bishop(0), new Knight(0), new Rook(0)}
    };

    this.currentTurn = 0;
    this.playingAgainstComputer = false;
    this.moveHistory = new MoveHistory();
}

// Displays window asking user whether they are playing against computer
public void selectPlayer(JFrame parent, boolean modal) {
    SelectPlayer selectPlayer = new SelectPlayer(parent, modal);
    this.playingAgainstComputer = selectPlayer.showDialog();
}

// Returns 8x8 boolean array for a particular piece, true = piece can move there
public boolean[][] getValidEndPositions(int pos) {
    boolean[][] validPositions = new boolean[8][8];

    for (int i = 0; i < 64; i++) {
        // Generates move string
        String move = "";
        if (pos < 10) {
            move += "0";
        }
        move += pos;

        if (i < 10) {
            move += "0";
        }
        move += i;

        if (i != pos && this.board[pos / 8][pos % 8].isValidMove(move, this.board)) {
            // Makes the move:

            int startPos = Integer.valueOf(move.substring(0, 2));
            int endPos = Integer.valueOf(move.substring(2, 4));

            if (this.board[endPos / 8][endPos % 8] == null) {
                move += "0";
            } else {
                move += this.board[endPos / 8][endPos % 8].getType();
            }

            this.moveHistory.add(move);
            this.currentTurn++;

            this.board[endPos / 8][endPos % 8] = this.board[startPos / 8][startPos % 8];
            this.board[startPos / 8][startPos % 8] = null;

            // Only valid if move didn't put the player into check
            if (!this.isInCheck((this.currentTurn - 1) % 2)) {
                validPositions[i / 8][i % 8] = true;
            }
            this.undoMove();
        }
    }
}

return validPositions;
}

```

```
// Applies given move to the board
public void makeMove(String move) {
    int startPos = Integer.valueOf(move.substring(0, 2));
    int endPos = Integer.valueOf(move.substring(2, 4));

    if (this.board[startPos / 8][startPos % 8] != null && this.board[startPos / 8][startPos % 8].isValidMove(move, this.board)) {
        if (this.board[endPos / 8][endPos % 8] == null) {
            move += "0";
        } else {
            move += this.board[endPos / 8][endPos % 8].getType();
        }
    }

    int colour = this.board[startPos / 8][startPos % 8].getColour();
    boolean pawnPromotion = this.board[startPos / 8][startPos % 8].getType() == 1
        && ((endPos / 8 == 0 && colour == 0) || (endPos / 8 == 7 && colour == 1));

    if (pawnPromotion) {
        move += "p";
    }

    this.moveHistory.add(move);
    this.currentTurn++;

    if (pawnPromotion) {
        this.board[endPos / 8][endPos % 8] = new Queen(colour);
    } else {
        this.board[endPos / 8][endPos % 8] = this.board[startPos / 8][startPos % 8];
    }
    this.board[startPos / 8][startPos % 8] = null;

    if (this.isInCheck((this.currentTurn - 1) % 2)) {
        this.undoMove();
    }
}
```

```
// Makes move without promoting pawn for use by AI
private void makeTempMove(String move) {
    int startPos = Integer.valueOf(move.substring(0, 2));
    int endPos = Integer.valueOf(move.substring(2, 4));

    if (this.board[startPos / 8][startPos % 8] != null && this.board[startPos / 8][startPos % 8].isValidMove(move, this.board)) {
        if (this.board[endPos / 8][endPos % 8] == null) {
            move += "0";
        } else {
            move += this.board[endPos / 8][endPos % 8].getType();
        }
    }

    this.moveHistory.add(move);
    this.currentTurn++;

    this.board[endPos / 8][endPos % 8] = this.board[startPos / 8][startPos % 8];
    this.board[startPos / 8][startPos % 8] = null;

    if (this.isInCheck((this.currentTurn - 1) % 2)) {
        this.undoMove();
    }
}
```

```
// Undoes previous move (if there is one)
public void undoMove() {
    if (this.moveHistory.getSize() > 0) {
        String move = this.moveHistory.pop();
        this.currentTurn--;

        int startPos = Integer.valueOf(move.substring(0, 2));
        int endPos = Integer.valueOf(move.substring(2, 4));
        int typeOfCapturedPiece = Integer.valueOf(move.substring(4, 5));
        Piece piece = null;

        switch (typeOfCapturedPiece) {
            case 1:
                piece = new Pawn((this.currentTurn - 1) % 2);
                break;
            case 2:
                piece = new Rook((this.currentTurn - 1) % 2);
                break;
            case 3:
                piece = new Knight((this.currentTurn - 1) % 2);
                break;
            case 4:
                piece = new Bishop((this.currentTurn - 1) % 2);
                break;
            case 5:
                piece = new Queen((this.currentTurn - 1) % 2);
                break;
            default:
                break;
        }

        if (move.length() > 5 && move.charAt(5) == 'p') {
            this.board[startPos / 8][startPos % 8] = new Pawn(this.currentTurn % 2);
        } else {
            this.board[startPos / 8][startPos % 8] = this.board[endPos / 8][endPos % 8];
        }

        this.board[endPos / 8][endPos % 8] = piece;
    }
}
```

```
// Determines and makes optimal move
public void makeComputerMove() {
    String move = this.getAIBestMove(this.board, 3);

    int startPos = Integer.valueOf(move.substring(0, 2));
    int endPos = Integer.valueOf(move.substring(2, 4));
    int colour = this.board[startPos / 8][startPos % 8].getColour();
    boolean pawnPromotion = this.board[startPos / 8][startPos % 8].getType() == 1
        && ((endPos / 8 == 0 && colour == 0) || (endPos / 8 == 7 && colour == 1));

    if (pawnPromotion) {
        move += "p";
    }

    this.makeMove(move);
}
```

```
// Returns an int representing value of the given board
private int evaluate(Piece[][] board) {
    int eval = 0;
    int score;

    for (int i = 0; i < 8; i++) {
        for (int j = 0; j < 8; j++) {
            if (this.board[i][j] != null) {
                switch (this.board[i][j].getType()) {
                    // is a pawn
                    case 1:
                        score = 100;
                        if (this.board[i][j].getColour() == 0) {
                            score += this.pawnEvalBonus[i][j];
                        } else {
                            score -= this.pawnEvalBonus[7 - i][j];
                        }
                        break;
                    // is a rook
                    case 2:
                        score = 500;
                        if (this.board[i][j].getColour() == 0) {
                            score += this.rookEvalBonus[i][j];
                        } else {
                            score -= this.rookEvalBonus[7 - i][j];
                        }
                        break;
                    // is a knight
                    case 3:
                        score = 300;
                        if (this.board[i][j].getColour() == 0) {
                            score += this.knightEvalBonus[i][j];
                        } else {
                            score -= this.knightEvalBonus[7 - i][j];
                        }
                        break;
                    // is a bishop
                    case 4:
                        score = 300;
                        if (this.board[i][j].getColour() == 0) {
                            score += this.bishopEvalBonus[i][j];
                        } else {
                            score -= this.bishopEvalBonus[7 - i][j];
                        }
                        break;
                }
            }
        }
    }
}
```

```
        case 5:
            score = 900;
            if (this.board[i][j].getColour() == 0) {
                score += this.queenEvalBonus[i][j];
            } else {
                score -= this.queenEvalBonus[7 - i][j];
            }
            break;
        // is a king
        default:
            score = 20000;
            if (this.board[i][j].getColour() == 0) {
                score += this.kingEvalBonus[i][j];
            } else {
                score -= this.kingEvalBonus[7 - i][j];
            }
            break;
    }

    if (this.board[i][j].getColour() == 0) { // if piece is white
        score *= -1;
    }

    eval += score;
}
}
}

return eval;
}
```

```
// Used to determine best move for AI to make
private String getAIBestMove(Piece[][] board, int depth) {
    int maxEval = Integer.MIN_VALUE;
    String maxEvalMove = "";

    ArrayList<String> computerMoves = this.getAllMovesForColour(1);

    for (int i = 0; i < computerMoves.size(); i++) {
        this.makeTempMove(computerMoves.get(i));
        int eval = this.minimax(board, depth - 1, false, Integer.MIN_VALUE, Integer.MAX_VALUE);
        this.undoMove();
        if (eval > maxEval) {
            maxEval = eval;
            maxEvalMove = computerMoves.get(i);
        }
    }

    return maxEvalMove;
}
```

```
// Looks ahead at moves and determines best possible evaluation
private int minimax(Piece[][] board, int depth, boolean isMaximisingPlayer, int alpha, int beta) {
    if (depth == 0 || this.currentPlayerHasNoLegalmoves()) {
        return this.evaluate(board);
    }

    if (isMaximisingPlayer) {
        int maxEval = Integer.MIN_VALUE;

        ArrayList<String> computerMoves = this.getAllMovesForColour(1);

        for (int i = 0; i < computerMoves.size(); i++) {
            this.makeTempMove(computerMoves.get(i));
            int eval = this.minimax(this.board.clone(), depth - 1, false, alpha, beta);
            if (eval > maxEval) {
                maxEval = eval;
            }
            this.undoMove();
            if (alpha < eval) {
                alpha = eval;
            }
            if (beta <= alpha) {
                break;
            }
        }

        return maxEval;
    } else {
        int minEval = Integer.MAX_VALUE;

        ArrayList<String> computerMoves = this.getAllMovesForColour(0);

        for (int i = 0; i < computerMoves.size(); i++) {
            this.makeTempMove(computerMoves.get(i));
            int eval = this.minimax(this.board.clone(), depth - 1, true, alpha, beta);
            if (eval < minEval) {
                minEval = eval;
            }
            this.undoMove();
            if (beta > eval) {
                beta = eval;
            }
            if (beta <= alpha) {
                break;
            }
        }

        return minEval;
    }
}
```

```

// Returns true if the given coloured king is under attack
public boolean isInCheck(int colour) {
    int kingPos = this.determineKingPosition(colour);

    if (kingPos == -1) {
        return false;
    }

    // Checking for pawns
    if (colour == 0) { // if white
        // Black pawn attacking white king from top left
        if ((kingPos / 8) - 1 >= 0 && (kingPos % 8) - 1 >= 0 && this.board[(kingPos / 8) - 1][(kingPos % 8) - 1] != null) {
            if (this.board[(kingPos / 8) - 1][(kingPos % 8) - 1].getType() == 1 && this.board[(kingPos / 8) - 1][(kingPos % 8) - 1].getColour() != colour) {
                return true;
            }
        }
        // Black pawn attacking white king from top right
        if ((kingPos / 8) - 1 >= 0 && (kingPos % 8) + 1 < 8 && this.board[(kingPos / 8) - 1][(kingPos % 8) + 1] != null) {
            if (this.board[(kingPos / 8) - 1][(kingPos % 8) + 1].getType() == 1 && this.board[(kingPos / 8) - 1][(kingPos % 8) + 1].getColour() != colour) {
                return true;
            }
        }
    } else { // if black
        // White pawn attacking black king from bottom left
        if ((kingPos / 8) + 1 < 8 && (kingPos % 8) - 1 >= 0 && this.board[(kingPos / 8) + 1][(kingPos % 8) - 1] != null) {
            if (this.board[(kingPos / 8) + 1][(kingPos % 8) - 1].getType() == 1 && this.board[(kingPos / 8) + 1][(kingPos % 8) - 1].getColour() != colour) {
                return true;
            }
        }
        // White pawn attacking black king from bottom right
        if ((kingPos / 8) + 1 >= 0 && (kingPos % 8) + 1 < 8 && this.board[(kingPos / 8) + 1][(kingPos % 8) + 1] != null) {
            if (this.board[(kingPos / 8) + 1][(kingPos % 8) + 1].getType() == 1 && this.board[(kingPos / 8) + 1][(kingPos % 8) + 1].getColour() != colour) {
                return true;
            }
        }
    }

    //Checking for kings
    // Top left
    if ((kingPos / 8) - 1 >= 0 && (kingPos % 8) - 1 >= 0 && this.board[(kingPos / 8) - 1][(kingPos % 8) - 1] != null) {
        if (this.board[(kingPos / 8) - 1][(kingPos % 8) - 1].getType() == 6 && this.board[(kingPos / 8) - 1][(kingPos % 8) - 1].getColour() != colour) {
            return true;
        }
    }
    // Above
    if ((kingPos / 8) - 1 >= 0 && this.board[(kingPos / 8) - 1][(kingPos % 8)] != null) {
        if (this.board[(kingPos / 8) - 1][(kingPos % 8)].getType() == 6 && this.board[(kingPos / 8) - 1][(kingPos % 8)].getColour() != colour) {
            return true;
        }
    }
    // Top right
    if ((kingPos / 8) - 1 >= 0 && (kingPos % 8) + 1 < 8 && this.board[(kingPos / 8) - 1][(kingPos % 8) + 1] != null) {
        if (this.board[(kingPos / 8) - 1][(kingPos % 8) + 1].getType() == 6 && this.board[(kingPos / 8) - 1][(kingPos % 8) + 1].getColour() != colour) {
            return true;
        }
    }
    // Right
    if ((kingPos % 8) + 1 < 8 && this.board[(kingPos / 8)][(kingPos % 8) + 1] != null) {
        if (this.board[(kingPos / 8)][(kingPos % 8) + 1].getType() == 6 && this.board[(kingPos / 8)][(kingPos % 8) + 1].getColour() != colour) {
            return true;
        }
    }
    // Bottom right
    if ((kingPos / 8) + 1 < 8 && (kingPos % 8) + 1 < 8 && this.board[(kingPos / 8) + 1][(kingPos % 8) + 1] != null) {
        if (this.board[(kingPos / 8) + 1][(kingPos % 8) + 1].getType() == 6 && this.board[(kingPos / 8) + 1][(kingPos % 8) + 1].getColour() != colour) {
            return true;
        }
    }
    // Below
    if ((kingPos / 8) + 1 < 8 && this.board[(kingPos / 8) + 1][(kingPos % 8)] != null) {
        if (this.board[(kingPos / 8) + 1][(kingPos % 8)].getType() == 6 && this.board[(kingPos / 8) + 1][(kingPos % 8)].getColour() != colour) {
            return true;
        }
    }
    // Bottom left
    if ((kingPos / 8) + 1 < 8 && (kingPos % 8) - 1 >= 0 && this.board[(kingPos / 8) + 1][(kingPos % 8) - 1] != null) {
        if (this.board[(kingPos / 8) + 1][(kingPos % 8) - 1].getType() == 6 && this.board[(kingPos / 8) + 1][(kingPos % 8) - 1].getColour() != colour) {
            return true;
        }
    }
    // Left
    if ((kingPos % 8) - 1 >= 0 && this.board[(kingPos / 8)][(kingPos % 8) - 1] != null) {
        if (this.board[(kingPos / 8)][(kingPos % 8) - 1].getType() == 6 && this.board[(kingPos / 8)][(kingPos % 8) - 1].getColour() != colour) {
            return true;
        }
    }
}

```

```

// Checking for knights
// One right, two up
if ((kingPos / 8) - 2 >= 0 && (kingPos % 8) + 1 < 8 && this.board[(kingPos / 8) - 2][(kingPos % 8) + 1] != null) {
    if (this.board[(kingPos / 8) - 2][(kingPos % 8) + 1].getType() == 3 && this.board[(kingPos / 8) - 2][(kingPos % 8) + 1].getColour() != colour) {
        return true;
    }
}
// Two right, one up
if ((kingPos / 8) - 1 >= 0 && (kingPos % 8) + 2 < 8 && this.board[(kingPos / 8) - 1][(kingPos % 8) + 2] != null) {
    if (this.board[(kingPos / 8) - 1][(kingPos % 8) + 2].getType() == 3 && this.board[(kingPos / 8) - 1][(kingPos % 8) + 2].getColour() != colour) {
        return true;
    }
}
// Two right, one down
if ((kingPos / 8) + 1 < 8 && (kingPos % 8) + 2 < 8 && this.board[(kingPos / 8) + 1][(kingPos % 8) + 2] != null) {
    if (this.board[(kingPos / 8) + 1][(kingPos % 8) + 2].getType() == 3 && this.board[(kingPos / 8) + 1][(kingPos % 8) + 2].getColour() != colour) {
        return true;
    }
}
// One right, two down
if ((kingPos / 8) + 2 < 8 && (kingPos % 8) + 1 < 8 && this.board[(kingPos / 8) + 2][(kingPos % 8) + 1] != null) {
    if (this.board[(kingPos / 8) + 2][(kingPos % 8) + 1].getType() == 3 && this.board[(kingPos / 8) + 2][(kingPos % 8) + 1].getColour() != colour) {
        return true;
    }
}
// One left, two down
if ((kingPos / 8) + 2 < 8 && (kingPos % 8) - 1 >= 0 && this.board[(kingPos / 8) + 2][(kingPos % 8) - 1] != null) {
    if (this.board[(kingPos / 8) + 2][(kingPos % 8) - 1].getType() == 3 && this.board[(kingPos / 8) + 2][(kingPos % 8) - 1].getColour() != colour) {
        return true;
    }
}
// Two left, one down
if ((kingPos / 8) + 1 < 8 && (kingPos % 8) - 2 >= 0 && this.board[(kingPos / 8) + 1][(kingPos % 8) - 2] != null) {
    if (this.board[(kingPos / 8) + 1][(kingPos % 8) - 2].getType() == 3 && this.board[(kingPos / 8) + 1][(kingPos % 8) - 2].getColour() != colour) {
        return true;
    }
}
// Two left, one up
if ((kingPos / 8) - 1 >= 0 && (kingPos % 8) - 2 >= 0 && this.board[(kingPos / 8) - 1][(kingPos % 8) - 2] != null) {
    if (this.board[(kingPos / 8) - 1][(kingPos % 8) - 2].getType() == 3 && this.board[(kingPos / 8) - 1][(kingPos % 8) - 2].getColour() != colour) {
        return true;
    }
}
// One left, two up
if ((kingPos / 8) - 2 >= 0 && (kingPos % 8) - 1 >= 0 && this.board[(kingPos / 8) - 2][(kingPos % 8) - 1] != null) {
    if (this.board[(kingPos / 8) - 2][(kingPos % 8) - 1].getType() == 3 && this.board[(kingPos / 8) - 2][(kingPos % 8) - 1].getColour() != colour) {
        return true;
    }
}

// Checking for rooks or queens
int col;
int row;
// Left
col = kingPos % 8;
while (col - 1 >= 0 && this.board[KingPos / 8][col - 1] == null) {
    col--;
}
if (col > 0 && (this.board[KingPos / 8][col - 1].getType() == 2 || this.board[KingPos / 8][col - 1].getType() == 5)
    && this.board[KingPos / 8][col - 1].getColour() != colour) {
    return true;
}
// Right
col = KingPos % 8;
while (col + 1 < 8 && this.board[KingPos / 8][col + 1] == null) {
    col++;
}
if (col < 7 && (this.board[KingPos / 8][col + 1].getType() == 2 || this.board[KingPos / 8][col + 1].getType() == 5)
    && this.board[KingPos / 8][col + 1].getColour() != colour) {
    return true;
}
// Up
row = KingPos / 8;
while (row - 1 >= 0 && this.board[row - 1][KingPos % 8] == null) {
    row--;
}
if (row > 0 && (this.board[row - 1][KingPos % 8].getType() == 2 || this.board[row - 1][KingPos % 8].getType() == 5)
    && this.board[row - 1][KingPos % 8].getColour() != colour) {
    return true;
}
// Down
row = KingPos / 8;
while (row + 1 < 8 && this.board[row + 1][KingPos % 8] == null) {
    row++;
}
if (row < 7 && (this.board[row + 1][KingPos % 8].getType() == 2 || this.board[row + 1][KingPos % 8].getType() == 5)
    && this.board[row + 1][KingPos % 8].getColour() != colour) {
    return true;
}

```

```

// Checking for bishops or queens
// North west
col = kingPos % 8;
row = kingPos / 8;
while (col - 1 >= 0 && row - 1 >= 0 && this.board[row - 1][col - 1] == null) {
    col--;
    row--;
}
if (col > 0 && row > 0 && (this.board[row - 1][col - 1].getType() == 4 || this.board[row - 1][col - 1].getType() == 5)
    && this.board[row - 1][col - 1].getColour() != colour) {
    return true;
}
// North east
col = kingPos % 8;
row = kingPos / 8;
while (col + 1 < 8 && row - 1 >= 0 && this.board[row - 1][col + 1] == null) {
    col++;
    row--;
}
if (col < 7 && row > 0 && (this.board[row - 1][col + 1].getType() == 4 || this.board[row - 1][col + 1].getType() == 5)
    && this.board[row - 1][col + 1].getColour() != colour) {
    return true;
}
// South east
col = kingPos % 8;
row = kingPos / 8;
while (col + 1 < 8 && row + 1 < 8 && this.board[row + 1][col + 1] == null) {
    col++;
    row++;
}
if (col < 7 && row < 7 && (this.board[row + 1][col + 1].getType() == 4 || this.board[row + 1][col + 1].getType() == 5)
    && this.board[row + 1][col + 1].getColour() != colour) {
    return true;
}
// South west
col = kingPos % 8;
row = kingPos / 8;
while (col - 1 >= 0 && row + 1 < 8 && this.board[row + 1][col - 1] == null) {
    col--;
    row++;
}

return col > 0 && row < 7 && (this.board[row + 1][col - 1].getType() == 4 || this.board[row + 1][col - 1].getType() == 5)
    && this.board[row + 1][col - 1].getColour() != colour;
}

```

```

// Returns the position (0-63) of the given coloured king
private int determineKingPosition(int colour) {
    for (int i = 0; i < this.board.length; i++) {
        for (int j = 0; j < this.board[i].length; j++) {
            if (this.board[i][j] != null && this.board[i][j].getColour() == colour
                && this.board[i][j].getType() == 6) {
                return i * 8 + j;
            }
        }
    }
    return -1;
}

// Returns true if the given colour's king is under attack and they have no legal moves
public boolean isInCheckMate(int colour) {

    for (int i = 0; i < 8; i++) {
        for (int j = 0; j < 8; j++) {
            if (this.board[i][j] != null && this.board[i][j].getColour() == colour) {
                ArrayList<String> moves = this.getAllMovesForPiece((i * 8) + j);

                if (!moves.isEmpty()) {
                    return false;
                }
            }
        }
    }

    return true;
}

// Returns true if the current player has no possible moves
public boolean currentPlayerHasNoLegalmoves() {
    return this.getAllMovesForColour(this.currentTurn % 2).isEmpty();
}

// Returns ArrayList of strings representing all possible moves for given colour
private ArrayList<String> getAllMovesForColour(int colour) {
    ArrayList<String> moves = new ArrayList<>();

    for (int i = 0; i < 8; i++) {
        for (int j = 0; j < 8; j++) {
            if (this.board[i][j] != null && this.board[i][j].getColour() == colour) {
                moves = this.combineLists(moves, this.getAllMovesForPiece(i * 8 + j));
            }
        }
    }

    return moves;
}

```

```

// Returns ArrayList of strings representing all possible moves for given piece
private ArrayList<String> getAllMovesForPiece(int pos) {
    ArrayList<String> moves = new ArrayList<>();

    for (int i = 0; i < 64; i++) {
        String move = "";
        if (pos < 10) {
            move += "0";
        }
        move += pos;

        if (i < 10) {
            move += "0";
        }
        move += i;

        if (i != pos && this.board[pos / 8][pos % 8].isValidMove(move, this.board)) {
            int startPos = Integer.valueOf(move.substring(0, 2));
            int endPos = Integer.valueOf(move.substring(2, 4));

            if (this.board[endPos / 8][endPos % 8] == null) {
                move += "0";
            } else {
                move += this.board[endPos / 8][endPos % 8].getType();
            }

            this.moveHistory.add(move);
            this.currentTurn++;

            this.board[endPos / 8][endPos % 8] = this.board[startPos / 8][startPos % 8];
            this.board[startPos / 8][startPos % 8] = null;

            if (!this.isInCheck((this.currentTurn - 1) % 2)) {
                moves.add(move);
            }
            this.undoMove();
        }
    }

    return moves;
}

// Takes in two ArrayLists of strings, returns a single ArrayList containing all elements of both
private ArrayList<String> combineLists(ArrayList<String> a, ArrayList<String> b) {
    if (a.size() < b.size()) {
        for (int i = 0; i < a.size(); i++) {
            b.add(a.get(i));
        }
        return b;
    } else {
        for (int i = 0; i < b.size(); i++) {
            a.add(b.get(i));
        }
        return a;
    }
}

```

```
// Returns a string representing the layout of the board
public String getGameString() {
    String game = "";

    int blank; // Used to count number of blank spaces in a row

    for (int i = 0; i < 8; i++) {
        blank = 0;
        for (int j = 0; j < 8; j++) {
            if (this.board[i][j] == null) {
                blank++;
            } else {
                if (blank != 0) {
                    game += blank;
                    blank = 0;
                }
                switch (this.board[i][j].getType()) {
                    // Capital = white, lower-case = black
                    // is a pawn
                    case 1:
                        game += this.board[i][j].getColour() == 0 ? "P" : "p";
                        break;
                    // is a rook
                    case 2:
                        game += this.board[i][j].getColour() == 0 ? "R" : "r";
                        break;
                    // is a knight
                    case 3:
                        game += this.board[i][j].getColour() == 0 ? "N" : "n";
                        break;
                    // is a bishop
                    case 4:
                        game += this.board[i][j].getColour() == 0 ? "B" : "b";
                        break;
                    // is a queen
                    case 5:
                        game += this.board[i][j].getColour() == 0 ? "Q" : "q";
                        break;
                    // is a king
                    case 6:
                        game += this.board[i][j].getColour() == 0 ? "K" : "k";
                        break;
                }
            }
        }
        if (blank != 0) {
            game += blank;
        }
        if (i != 7) {
            game += "/";
        }
    }

    game += this.currentTurn % 2 == 0 ? " w" : " b"; // If current turn is even - i.e. white, append " w"

    return game;
}
```

```

// Takes a string representing the layout of the board and sets the board to this layout
public void loadGameString(String str) {
    if (str.matches("[rnbqkpRNBQKPl-8]{1,8}\\\\/){7}([rnbqkpRNBQKPl-8]{1,8}) [wb]")) { // Checks string is in correct format
        String[] parts = str.split(" ");
        String[] rows = parts[0].split("/");
        for (int i = 0; i < rows.length; i++) {
            for (int j = 0, col = 0; j < rows[i].length(); j++) {
                switch (rows[i].charAt(j)) {
                    case 'p':
                        this.board[i][col] = new Pawn(1);
                        col++;
                        break;
                    case 'P':
                        this.board[i][col] = new Pawn(0);
                        col++;
                        break;
                    case 'r':
                        this.board[i][col] = new Rook(1);
                        col++;
                        break;
                    case 'R':
                        this.board[i][col] = new Rook(0);
                        col++;
                        break;
                    case 'n':
                        this.board[i][col] = new Knight(1);
                        col++;
                        break;
                    case 'N':
                        this.board[i][col] = new Knight(0);
                        col++;
                        break;
                    case 'b':
                        this.board[i][col] = new Bishop(1);
                        col++;
                        break;
                    case 'B':
                        this.board[i][col] = new Bishop(0);
                        col++;
                        break;
                    case 'q':
                        this.board[i][col] = new Queen(1);
                        col++;
                        break;
                    case 'Q':
                        this.board[i][col] = new Queen(0);
                        col++;
                        break;
                    case 'k':
                        this.board[i][col] = new King(1);
                        col++;
                        break;
                    case 'K':
                        this.board[i][col] = new King(0);
                        col++;
                        break;
                    default:
                        int num = Integer.valueOf(rows[i].substring(j, j + 1));
                        for (int k = 0; k < num; k++) {
                            this.board[i][col] = null;
                            col++;
                        }
                }
            }
        }
        this.moveHistory = new MoveHistory();
        this.currentTurn = parts[1].equals("w") ? 0 : 1;
        if(this.currentTurn % 2 == 1 && this.playingAgainstComputer) {
            this.makeComputerMove();
        }
    }
}

```

```
// Returns 1 if piece is black, 0 if white, -1 if none
public int getColourOfPieceAtPosition(int pos) {
    if(this.board[pos / 8][pos % 8] != null) {
        return this.board[pos / 8][pos % 8].getColour();
    } else {
        return -1;
    }
}

// Returns true if there is a piece, false otherwise
public boolean hasPieceAtPosition(int pos) {
    return this.board[pos / 8][pos % 8] != null;
}

// Returns the board as a 2D array of type Piece
public Piece[][] getBoard() {
    return this.board;
}

// Returns the current turn
public int getCurrentTurn() {
    return this.currentTurn;
}

// Returns true if user is playing against computer
public boolean isPlayingAgainstComputer() {
    return this.playingAgainstComputer;
}

// Returns string representing last move made (or empty string if none have been made)
public String getLastMove() {
    if (this.moveHistory.getSize() > 0) {
        return this.moveHistory.peek();
    } else {
        return "";
    }
}

}
```

Help.java

```
public class Help extends javax.swing.JDialog {

    public Help(java.awt.Frame parent, boolean modal) {
        super(parent, modal);
        initComponents();
    }

    /**
     * This method is called from within the constructor to initialize the form.
     * WARNING: Do NOT modify this code. The content of this method is always
     * regenerated by the Form Editor.
     */
    @SuppressWarnings("unchecked")
    // <editor-fold defaultstate="collapsed" desc="Generated Code">
    private void initComponents() {

        jLabel1 = new javax.swing.JLabel();

        setDefaultCloseOperation(javax.swing.WindowConstants.DISPOSE_ON_CLOSE);
        setTitle("Help");
        setIconImage(null);
        setMinimumSize(new java.awt.Dimension(770, 450));
        setSize(new java.awt.Dimension(764, 488));
        getContentPane().setLayout(null);

        jLabel1.setIcon(new javax.swing.ImageIcon(getClass().getResource("/help-window.png"))); // NOI18N
        getContentPane().add(jLabel1);
        jLabel1.setBounds(0, 0, 764, 430);

        pack();
    } // </editor-fold>

    /**
     * @param args the command line arguments
     */
    public static void main(String args[]) {
        /* Set the Nimbus look and feel */
        Look and feel setting code (optional)

        /* Create and display the dialog */
        java.awt.EventQueue.invokeLater(new Runnable() { ...12 lines ... });
    }

    // Variables declaration - do not modify
    private javax.swing.JLabel jLabel1;
    // End of variables declaration
}
```

King.java

```
public class King extends Piece {

    public King(int colour) {
        super(6, colour, colour == 0 ? "/white-king.png" : "/black-king.png");
    }

    // Determines if the move is pseudo-legal
    @Override
    public boolean isValidMove(String move, Piece[][] board) {
        // Gets start position and end position as ints from the move string
        int startPos = Integer.valueOf(move.substring(0, 2));
        int endPos = Integer.valueOf(move.substring(2, 4));

        boolean capturingPieceOfDifferentColour = (board[endPos / 8][endPos % 8] != null)
            && (board[startPos / 8][startPos % 8].getColour() != board[endPos / 8][endPos % 8].getColour());
        boolean movingToEmptySquare = board[endPos / 8][endPos % 8] == null;

        boolean notMoreThanOneRowAway = Math.abs(endPos / 8 - startPos / 8) <= 1;
        boolean notMoreThanOneColumnAway = Math.abs(endPos % 8 - startPos % 8) <= 1;

        return (capturingPieceOfDifferentColour || movingToEmptySquare) && (notMoreThanOneColumnAway && notMoreThanOneRowAway);
    }

    @Override
    public String toString() {
        if (this.getColour() == 0) {
            return "♔";
        } else {
            return "♚";
        }
    }
}
```

Knight.java

```
public class Knight extends Piece {

    public Knight(int colour) {
        super(3, colour, colour == 0 ? "/white-knight.png" : "/black-knight.png");
    }

    // Determines if the move is pseudo-legal
    @Override
    public boolean isValidMove(String move, Piece[][] board) {
        // Gets start position and end position as ints from the move string
        int startPos = Integer.valueOf(move.substring(0, 2));
        int endPos = Integer.valueOf(move.substring(2, 4));

        boolean capturingPieceOfDifferentColour = (board[endPos / 8][endPos % 8] != null)
            && (board[startPos / 8][startPos % 8].getColour() != board[endPos / 8][endPos % 8].getColour());
        boolean movingToEmptySquare = board[endPos / 8][endPos % 8] == null;

        boolean twoColumnsAway = Math.abs(endPos % 8 - startPos % 8) == 2;
        boolean oneRowAway = Math.abs(endPos / 8 - startPos / 8) == 1;

        boolean twoRowsAway = Math.abs(endPos / 8 - startPos / 8) == 2;
        boolean oneColumnAway = Math.abs(endPos % 8 - startPos % 8) == 1;

        return (capturingPieceOfDifferentColour || movingToEmptySquare)
            && (twoRowsAway && oneColumnAway || twoColumnsAway && oneRowAway);
    }

    @Override
    public String toString() {
        if (this.getColour() == 0) {
            return "♞";
        } else {
            return "♘";
        }
    }
}
```

MoveHistory.java

```
import java.util.ArrayList;

public class MoveHistory {

    private ArrayList<String> moves;

    public MoveHistory() {
        this.moves = new ArrayList<>();
    }

    // Adds move to end of list
    public void add(String move) {
        this.moves.add(move);
    }

    // Removes and returns move at end of list
    public String pop() {
        if(this.moves.size() > 0) {
            String move = this.moves.get(this.moves.size() - 1);
            this.moves.remove(this.moves.size() - 1);
            return move;
        }
        return "";
    }

    // Returns move at end of list
    public String peek() {
        if(this.moves.size() > 0) {
            return this.moves.get(this.moves.size() - 1);
        }
        return "";
    }

    // Returns number of moves in list
    public int getSize() {
        return this.moves.size();
    }

}
```

Pawn.java

```
public class Pawn extends Piece {

    public Pawn(int colour) {
        super(1, colour, colour == 0 ? "/white-pawn.png" : "/black-pawn.png");
    }

    // Determines if the move is pseudo-legal
    @Override
    public boolean isValidMove(String move, Piece[][] board) {
        // Gets start position and end position as ints from the move string
        int startPos = Integer.valueOf(move.substring(0, 2));
        int endPos = Integer.valueOf(move.substring(2, 4));

        boolean capturingPieceOfDifferentColour = (board[endPos / 8][endPos % 8] != null)
            && (board[startPos / 8][startPos % 8].getColour() != board[endPos / 8][endPos % 8].getColour());
        boolean movingToEmptySquare = board[endPos / 8][endPos % 8] == null;

        boolean isWhite = this.getColour() == 0;

        boolean hasNotMoved = (isWhite && startPos / 8 == 6) || (!isWhite && startPos / 8 == 1);

        boolean oneRowAway = Math.abs(endPos / 8 - startPos / 8) == 1;
        boolean twoRowsAway = Math.abs(endPos / 8 - startPos / 8) == 2;
        boolean inSameColumn = endPos % 8 == startPos % 8;
        boolean oneColumnAway = Math.abs(endPos % 8 - startPos % 8) == 1;

        boolean squareInFrontIsEmpty = (isWhite && (startPos / 8) - 1 >= 0 && board[(startPos / 8) - 1][startPos % 8] == null)
            || (!isWhite && (startPos / 8) + 1 < 8 && board[(startPos / 8) + 1][startPos % 8] == null);

        boolean isMovingForwards = (isWhite && endPos / 8 < startPos / 8) || (!isWhite && endPos / 8 > startPos / 8);

        return (capturingPieceOfDifferentColour && oneColumnAway && oneRowAway || movingToEmptySquare && inSameColumn
            && (oneRowAway || hasNotMoved && twoRowsAway && squareInFrontIsEmpty)) && isMovingForwards;
    }

    @Override
    public String toString() {
        if (this.getColour() == 0) {
            return "\u265f";
        } else {
            return "\u265e";
        }
    }
}
```

Piece.java

```
public abstract class Piece {

    private int type; // 1 = pawn, 2 = rook, 3 = knight, 4 = bishop, 5 = queen, 6 = king
    private int colour; // 0 = white, 1 = black
    private String img; // String containing location of image

    public Piece(int type, int colour, String img) {
        this.type = type;
        this.colour = colour;
        this.img = img;
    }

    public abstract boolean isValidMove(String move, Piece[][] board);

    @Override
    public abstract String toString();

    public int getColour() {
        return this.colour;
    }

    public int getType() {
        return this.type;
    }

    public String getImage() {
        return this.img;
    }

    public void setType(int type) {
        this.type = type;
    }

    public void setColour(int colour) {
        this.colour = colour;
    }

    public void setImage(String img) {
        this.img = img;
    }
}
```

Queen.java

```
public class Queen extends Piece {

    public Queen(int colour) {
        super(5, colour, colour == 0 ? "/white-queen.png" : "/black-queen.png");
    }

    // Determines if the move is pseudo-legal
    @Override
    public boolean isValidMove(String move, Piece[][] board) {
        // Gets start position and end position as ints from the move string
        int startPos = Integer.valueOf(move.substring(0, 2));
        int endPos = Integer.valueOf(move.substring(2, 4));

        boolean capturingPieceOfDifferentColour = (board[endPos / 8][endPos % 8] != null)
            && (board[startPos / 8][startPos % 8].getColour() != board[endPos / 8][endPos % 8].getColour());
        boolean movingToEmptySquare = board[endPos / 8][endPos % 8] == null;

        boolean inSameRow = endPos / 8 == startPos / 8;
        boolean inSameColumn = endPos % 8 == startPos % 8;

        int columnChange = Math.abs(endPos % 8 - startPos % 8);
        int rowChange = Math.abs(endPos / 8 - startPos / 8);

        boolean isDiagonal = columnChange == rowChange;

        if((capturingPieceOfDifferentColour || movingToEmptySquare) && (inSameRow || inSameColumn)) {
            if(inSameRow && endPos % 8 < startPos % 8) { // Moving left
                int col = startPos % 8;
                while(col - 1 >= 0 && board[startPos / 8][col-1] == null) {
                    col--;
                }
                return endPos % 8 >= col || capturingPieceOfDifferentColour && endPos % 8 >= col-1;
            } else if (inSameRow && endPos % 8 > startPos % 8) { // Moving right
                int col = startPos % 8;
                while(col + 1 < 8 && board[startPos / 8][col+1] == null) {
                    col++;
                }
                return endPos % 8 <= col || capturingPieceOfDifferentColour && endPos % 8 <= col+1;
            } else if (inSameColumn && endPos / 8 < startPos / 8) { // Moving up
                int row = startPos / 8;
                while(row - 1 >= 0 && board[row-1][startPos % 8] == null) {
                    row--;
                }
                return endPos / 8 >= row || capturingPieceOfDifferentColour && endPos / 8 >= row-1;
            } else if (inSameColumn && endPos / 8 > startPos / 8) { // Moving down
                int row = startPos / 8;
                while(row + 1 < 8 && board[row+1][startPos % 8] == null) {
                    row++;
                }
                return endPos / 8 <= row || capturingPieceOfDifferentColour && endPos / 8 <= row+1;
            }
        }
    }
}
```

```

        }
    } else if((capturingPieceOfDifferentColour || movingToEmptySquare) && isDiagonal) {
        if(endPos % 8 < startPos % 8 && endPos / 8 < startPos / 8) { // Moving north west
            int col = startPos % 8;
            int row = startPos / 8;
            while(col - 1 >= 0 && row - 1 >= 0 && board[row-1][col-1] == null) {
                col--;
                row--;
            }
            return endPos % 8 >= col && endPos / 8 >= row || capturingPieceOfDifferentColour && endPos % 8 >= col-1 && endPos / 8 >= row-1;
        } else if (endPos % 8 > startPos % 8 && endPos / 8 < startPos / 8) { // Moving north east
            int col = startPos % 8;
            int row = startPos / 8;
            while(col + 1 < 8 && row - 1 >= 0 && board[row-1][col+1] == null) {
                col++;
                row--;
            }
            return endPos % 8 <= col && endPos / 8 >= row || capturingPieceOfDifferentColour && endPos % 8 <= col+1 && endPos / 8 >= row-1;
        } else if (endPos % 8 > startPos % 8 && endPos / 8 > startPos / 8) { // Moving south east
            int col = startPos % 8;
            int row = startPos / 8;
            while(col + 1 < 8 && row + 1 < 8 && board[row+1][col+1] == null) {
                col++;
                row++;
            }
            return endPos % 8 <= col && endPos / 8 <= row || capturingPieceOfDifferentColour && endPos % 8 <= col+1 && endPos / 8 <= row+1;
        } else if (endPos % 8 < startPos % 8 && endPos / 8 > startPos / 8) { // Moving south west
            int col = startPos % 8;
            int row = startPos / 8;
            while(col - 1 >= 0 && row + 1 < 8 && board[row+1][col-1] == null) {
                col--;
                row++;
            }
            return endPos % 8 >= col && endPos / 8 <= row || capturingPieceOfDifferentColour && endPos % 8 >= col-1 && endPos / 8 <= row+1;
        }
    }

    return false;
}

@Override
public String toString() {
    if (this.getColour() == 0) {
        return "■";
    } else {
        return "■";
    }
}
}

```

Rook.java

```
public class Rook extends Piece {

    public Rook(int colour) {
        super(2, colour, colour == 0 ? "/white-rook.png" : "/black-rook.png");
    }

    // Determines if the move is pseudo-legal
    @Override
    public boolean isValidMove(String move, Piece[][] board) {
        // Gets start position and end position as ints from the move string
        int startPos = Integer.valueOf(move.substring(0, 2));
        int endPos = Integer.valueOf(move.substring(2, 4));

        boolean capturingPieceOfDifferentColour = (board[endPos / 8][endPos % 8] != null)
            && (board[startPos / 8][startPos % 8].getColour() != board[endPos / 8][endPos % 8].getColour());
        boolean movingToEmptySquare = board[endPos / 8][endPos % 8] == null;

        boolean inSameRow = endPos / 8 == startPos / 8;
        boolean inSameColumn = endPos % 8 == startPos % 8;

        if((capturingPieceOfDifferentColour || movingToEmptySquare) && (inSameRow || inSameColumn)) {
            if(inSameRow && endPos % 8 < startPos % 8) { // Moving left
                int col = startPos % 8;
                while(col - 1 >= 0 && board[startPos / 8][col-1] == null) {
                    col--;
                }
                return endPos % 8 >= col || capturingPieceOfDifferentColour && endPos % 8 >= col-1;
            } else if (inSameRow && endPos % 8 > startPos % 8) { // Moving right
                int col = startPos % 8;
                while(col + 1 < 8 && board[startPos / 8][col+1] == null) {
                    col++;
                }
                return endPos % 8 <= col || capturingPieceOfDifferentColour && endPos % 8 <= col+1;
            } else if (inSameColumn && endPos / 8 < startPos / 8) { // Moving up
                int row = startPos / 8;
                while(row - 1 >= 0 && board[row-1][startPos % 8] == null) {
                    row--;
                }
                return endPos / 8 >= row || capturingPieceOfDifferentColour && endPos / 8 >= row-1;
            } else if (inSameColumn && endPos / 8 > startPos / 8) { // Moving down
                int row = startPos / 8;
                while(row + 1 < 8 && board[row+1][startPos % 8] == null) {
                    row++;
                }
                return endPos / 8 <= row || capturingPieceOfDifferentColour && endPos / 8 <= row+1;
            }
        }
        return false;
    }

    @Override
    public String toString() {
        if (this.getColour() == 0) {
            return "♜";
        } else {
            return "♜";
        }
    }
}
```

SelectPlayer.java

```
public class SelectPlayer extends javax.swing.JDialog {

    /**
     * Creates new form SelectPlayer
     */
    private boolean playingAgainstComputer;

    public SelectPlayer(java.awt.Frame parent, boolean modal) {
        super(parent, modal);
        initComponents();
        this.playingAgainstComputer = false;
    }

    // Shows window to user
    public boolean showDialog() {
        this.setVisible(true);
        return this.playingAgainstComputer;
    }

    /**
     * This method is called from within the constructor to initialize the form.
     * WARNING: Do NOT modify this code. The content of this method is always
     * regenerated by the Form Editor.
     */
    @SuppressWarnings("unchecked")
    Generated Code

    private void twoPlayerButtonActionPerformed(java.awt.event.ActionEvent evt) {
        // TODO add your handling code here:
        this.playingAgainstComputer = false;
        this.setVisible(false);
        this.dispose();
    }

    private void onePlayerButtonActionPerformed(java.awt.event.ActionEvent evt) {
        // TODO add your handling code here:
        this.playingAgainstComputer = true;
        this.setVisible(false);
        this.dispose();
    }
}
```

```
/**  
 * @param args the command line arguments  
 */  
public static void main(String args[]) {  
    /* Set the Nimbus look and feel */  
    Look and feel setting code (optional)  
  
    /* Create and display the dialog */  
    java.awt.EventQueue.invokeLater(new Runnable() {  
        public void run() {  
            SelectPlayer dialog = new SelectPlayer(new javax.swing.JFrame(), true);  
            dialog.addWindowListener(new java.awt.event.WindowAdapter() {  
                @Override  
                public void windowClosing(java.awt.event.WindowEvent e) {  
                    System.exit(0);  
                }  
            });  
            dialog.setVisible(true);  
        }  
    });  
}  
  
// Variables declaration - do not modify  
private javax.swing.JLabel jLabel1;  
private javax.swing.JLabel jLabel2;  
private javax.swing.JButton onePlayerButton;  
private javax.swing.JButton twoPlayerButton;  
// End of variables declaration  
}
```

ShowWinner.java

```
public class ShowWinner extends javax.swing.JDialog {  
  
    /**  
     * Creates new form ShowWinner  
     */  
  
    public ShowWinner(java.awt.Frame parent, boolean modal) {  
        super(parent, modal);  
        initComponents();  
    }  
  
    // Sets text of JLabel to show correct winner  
    public void setWinner(int colour) {  
        if(colour == 0) {  
            this.winnerLabel.setText("White has won!");  
        } else if (colour == 1) {  
            this.winnerLabel.setText("Black has won!");  
        } else {  
            this.winnerLabel.setText("Stalemate");  
        }  
    }  
  
    /**  
     * This method is called from within the constructor to initialize the form.  
     * WARNING: Do NOT modify this code. The content of this method is always  
     * regenerated by the Form Editor.  
     */  
    @SuppressWarnings("unchecked")  
    Generated Code  
  
    /**  
     * @param args the command line arguments  
     */  
    public static void main(String args[]) {... 38 lines ...}  
  
    // Variables declaration - do not modify  
    private javax.swing.JLabel jLabell;  
    private javax.swing.JLabel winnerLabel;  
    // End of variables declaration  
}
```

Images



To make a move:

- 1) Click the piece you would like to move.
- 2) Click where you would like to move this piece to.

Rules for rooks:

- Can move any number of squares horizontally or vertically.

Rules for bishops:

- Can move any number of squares diagonally.

Rules for kings:

- Can move one square in any direction.

Checkmate:

- If you are in check, and cannot move out of check, you are in checkmate.
- If you are in checkmate, you lose the game.

Rules for pawns:

- Can move one square forwards (or two forwards if it has not moved yet) if that square is empty.
- Can move once diagonally forwards if it is capturing a piece.

Rules for queens:

- Can move any number of squares in any direction.

Rules for knights:

- Can move in an L-shape.

Check:

- If your opponent is attacking your king, you are in check.

