Final Year Project Report

Full Unit – Final Report

Chess with Artificial Intelligence

Alexander Mason

A report submitted in part fulfilment of the degree of

BSc (Hons) in Computer Science

Supervisor: Nery Riquelme Granada

Department of Computer Science
Royal Holloway, University of London

April 23, 2020

Declaration

This report has been prepared on the basis of my own work. Where other published and unpublished source materials have been used, these have been acknowledged.

Word Count: 12,832

Student Name: Alexander Mason

Date of Submission: 23/04/2020

Signature: 

Table of Contents

Abstract

In recent years, Artificial Intelligence has been growing exponentially in the industry, especially in the gaming sector. Google is a prime example of a company who has implemented Artificial Intelligence into the game Chess, which has gone onto beating some of the world's best human chess players. This concept, and how they approached this, was of great interest to me. This is my main motivation for wanting to research Artificial Intelligence and to create my own Chess game which incorporates a basic AI system in the Java programming language.

This project will be beneficial for my career in Computer Science and in the IT industry because it enhances my Java programming skills, which is most important as I am looking to be a software developer, ideally in the Java programming language. Not only this, but this project allows me to gain a better insight into Artificial Intelligence and the links Chess has to Computer Science, as well as implementing an Artificial Intelligence algorithm into my game.

In the first term, my aims of the project were mainly to research the background and rules of Chess, as well as get most of the foundation/base of the game programmed in Java. In the second term, my aims of the project were to complete a GUI implementing the foundation/base I programmed in the previous term, as well as implementing an Artificial Intelligence approach (the Minimax algorithm) as a way of adding player vs computer functionality. Now that both terms have come to end, in this report I will be showing the research I have undertaken into the game's background and the implementation of my research into code for my game, as well as talking about how well I have met my aims, objectives, and planning from my original project plan so far.

Chapter 1: **Introduction**

To introduce, I feel it is a good idea to provide some background information about the game. Here is a shortened abstract on the game's history from my 'History and Rules of Chess' report.

Chess goes back roughly 1500 years, with the game first dating back to Northern India in the 6th century. In its earliest form in the 6th century, the game was known as Chaturanga. There were four different divisions known as the infantry, the cavalry, elephants, and chariotry. Through time, these divisions became the modern pawn, knight, bishop, and rook that we use in today's game. [2]

It then spread to Persia, and subsequently Southern Europe. At roughly 1475- 1500AD was the birth of the modern game that we still know at present day, with the difference being new moves for the queen and bishop compared to earlier versions of the game. Pawns gained the ability to move two squares in their first move. These modern rules were adopted by Italy and Spain. [3] The queen had become the most powerful piece by this time, it was decided that the player on the white side would make the starting move, and the 'Castling' manoeuvre was first introduced. These were the rules that were known in Western Europe when the game had spread/reached there.

During the early 19th century, the rules concerning a stalemate situation were finalised, so that a stalemate is widely recognised as a draw in the modern game. In the earlier versions, such as in chaturanga, the side who caused a stalemate would have been the winner. [4] Although castling was introduced in the 1500s, the rules when castling a rook and king were also finalised and become standardised. As these rules were standardised in Western Europe, this version of the game was sometimes known as Western Chess.

This leads us to the 20th and 21st century, where Computer Science and the IT industry made advances in the theory of Chess. These advances lead to the development of Artificial Intelligence within chess engines (a program with the ability to play the game against a human player), as well as the development of chess databases. In 1996, Deep Blue was the first computer system to use Artificial Intelligence to successfully beat a world championship player in the game. Despite computers having the ability to beat world championship chess players, modern computers cannot "solve" chess due the very large number of possible ways to play the game. A breakthrough in Quantum computing would be needed to even start attempting to solve Chess. [6]

These advancements in the theory and the development of various Chess engines led me to choose Chess with Artificial Intelligence for my project topic. During the first term, I have started to create my own Chess engine in the Java programming language, with the intention of adding Artificial Intelligence concepts to the Chess engine as part of the second term of my project. You will see below the advancements I have made in my own Chess engine as well as my research findings throughout my report.

Chapter 2: **Aims, Objectives and Literature Survey**

2.1 – Aims and Objectives

First Term

For the first term in my project, my aims were to research the background and rules of Chess to gain a better understanding of the game so that I can program it. My aims in terms of programming were to have four different 'phases' of development for the game. Below is a list of each of my aims, a description for each aim, and what objectives I have to complete that aim for the first term of my project:

1) **Initialisation/Pre-Conditions**

- Aim: Setting up a class used to initialise the game when it starts. It will store pre-conditions such as the starting positions for each chess piece, initialising variables, creating new instances of classes, and setting tile occupation statuses so that only tiles containing the chess pieces in their starting positions are marked as "Occupied" to begin with.

- Objective: I will think about what data structures and variables would be most suitable to store my pre-condition and initialisation data. Not only this, I will identify each tile with a tile co-ordinate as a clear way of setting piece positions.

2) **Report on the History and Rules of Chess**

- Aim: To research the background and rules of the game so that I can program the game accurately to follow the rules of Chess.

- Objective: I am going to research the history and rules of the game to gain useful insight into its background and the conditions I will need to know in order to program the game.

3) **Properties/Validation**

- Aim: The second 'phase' of my game is to create a class for each of the different types of Chess pieces. These classes will contain code to validate what is a legal move for that piece so that the move is in line with the rules of Chess.

- Objective: I am going to use my research findings in the History and Rules of Chess report to translate the rules for a valid move for each piece type into code so that my game is an accurate representation of Chess.

4) **Square Occupation**

- Aim: The third 'phase' of my game is sorting the square/tile occupation so that I can return statuses of each tile and use them as a way of knowing if additional validation is needed to check whether the tile contains an opponent piece or a piece of the own player's, assuming the status is "Occupied".

- Objective: I am going to use tile occupation statuses as a way of knowing if a tile is occupied or not. I propose to use a data structure which takes a key (such as a tile co-ordinate) and maps that to a value. This value will be the tile occupation status and could be either “Empty” or “Occupied”.

5) Coding Progression Report

- Aim: The purpose of this report is to document the coding development and progression of the game since I started the Initialisation/Pre-Conditions stage.

- Objective: I will be looking at my project diary and the code I have written to effectively demonstrate the progression of my game since the first task.

6) GUI

- Aim: Creating a GUI to give users a visual output of the game rather than the textual output used for developing the game when checking the current game state.

- Objective: To create a GUI, I will look at other examples of chess game GUIs online for inspiration on how I want my game’s aesthetics to look. I want to go for a modern looking GUI so will be looking at similar examples. I will use the tool Scene Builder to create the GUI and link my code to the GUI in Eclipse.

  Note: This aim was completed as part of the second term despite originally being planned for the first term

Second Term

In the second term, my aims were mostly about getting a game loop created so that my game is playable. I would do this by linking my game logic to a GUI, which I wasn’t able to finish in term 1. Therefore, my other aims were to create the GUI this term and add a 1 player mode to the GUI which would take my implementation of the minimax Artificial Intelligence algorithm so the player can play against the computer.

1) GUI

- Aim: Creating a GUI to give users a visual output of the game rather than the textual output used for developing the game when checking the current game state.

- Objective: To create a GUI, I will look at other examples of chess game GUIs online for inspiration on how I want my game’s aesthetics to look. I want to go for a modern looking GUI so will be looking at similar examples. I will use the tool Scene Builder to create the GUI and link my code to the GUI in Eclipse.

2) Two Players/Game Loop

- Aim: Adding a game loop to the game so that all classes interact with each other and is in a playable form between two players.

- Objective: I am going to use the data structures and validation methods from each class in order to program my game loop and a two-player game. I will constantly be checking to

see if the current player has taken their turn in the game loop before switching to the opponent player. I will also check for a Boolean checkmate condition, which is the main goal state for the game (where the loop terminates).

3) Report on Artificial Intelligence in Games

- Aim: Write a report on the usage of Artificial Intelligence in Games, as well as more going into more depth on how Artificial Intelligence is linked to Chess.

- Objective: To research Artificial Intelligence in Games, I will be looking at the development of AI in games through time and also look at how Chess computer systems, such as DeepBlue, were developed.

4) Basic AI Approach

- Aim: Implement a base for an AI algorithm, this could be methods allowing the game to scan for possible moves, which can then be used for an AI algorithm as part of the 'Advanced AI' aim.

- Objective: Create the base method by taking the selected piece to scan through all its potential moves, identifying pieces it can capture and/or tiles it can go to. I can then implement an AI algorithm onto this for the Advanced AI objective.

5) Report on Artificial Intelligence Coding Progression

- Aim: Create a report on the coding progression of adding Artificial Intelligence into my game so far.

- Objective: I am going to reflect on the code I have written and discuss how I have progressed with implementing basic AI into my Chess game.

2.2 – Professional Issues

During the progression of my project through the academic year, there have been a few professional issues that did raise my concern. They are mainly related to usability, plagiarism, and licensing.

Usability

Accessibility

I feel one professional issue with my project is the lack of accessibility in relation to the GUI that I have created for my game. One issue is with my Chessboard image used as part of the GUI, where the black tiles are replaced with dark orange tiles and white tiles replaced with light orange tiles. I originally used this image as a way of modernising the classic white and black tile Chessboard. However, with 1 in 12 men and 1 in 200 women being colour blind [12], my interface would be unsuitable for a user who is colour blind because they would have difficulty with distinguishing between one colour and another, especially with an interface using two shades of the same colour.

To resolve this, I feel I could have included a classic or greyscale view where the tiles revert back to white and black, as they would be on a classic Chess board.

Another professional issue with my project is the inability to change the text size in the GUI's output box to a size appropriate to the user and lack of text to speech option. This is a usability issue because it will be hard to read for users, more so those with visual impairments where it is reported that in the UK alone 2 million have a visual impairment and 360,000 in the UK registered as blind or partially sighted [13]. The GUI output box is important when playing the game because it displays to the user any errors or illegal moves they have made so that the user can recover from this error by changing the movement path or way in which they are playing the game. Therefore, it is important that to make my project more accessible that I include a text size feature and text to speech option to rectify these accessibility issues.

Artificial Intelligence and Replacing Humans

These are two separate issues but for my project are both closely related, hence grouping them under the same category. I feel that my implementation of Artificial Intelligence in my game results in replacing humans in the sense that it means you no longer need a 2nd player in order to play the game. Although this could be seen as a positive with the ability to make the game more accessible and flexible, this does have social issues because it takes away the aspect of finding another person to play the game and interact with. Furthermore, with Artificial Intelligence taking the place of a 2nd player, it takes away a huge element of the game where you try and understand the other player's thought process behind a move or try to work out what their next move would be. This is harder with Artificial Intelligence replacing a player, because the AI is all done behind the scenes so it is not clear or predictable how the system will play.

Plagiarism

Correct Citation

Plagiarism, namely in the form of correct citation, is a professional issue which has been a concern during the development/progression of my project, more so when writing reports. This is due to reading a lot of material and using research to back my understanding in each of the reports I have written. Therefore, it is important that instead of taking material with no mention to the original author and claiming it to be my own work, that I cite the original author's work so it is clear where I got my information from and so that I do not claim their research work to be my own. This is something I have done throughout my project to prevent this from potentially being an issue to actually being a professional issue.

Licensing

Copyright

The final professional issue relevant to my project is the potential usage of images which have copyright protection on them. During the creation of my Chess GUI, I needed to source images for each of the chess pieces and the chess board. It was important that I sourced images which were copyright free so that I do not infringe copyright laws. Although I understand that because my project is for educational purposes and copyright protected work could be used under these circumstances, I may want to use my project as an example of the work I have achieved during my time at university which would be beyond the scope of the exceptions to using copyrighted work.

To prevent this professional issue, I ensured the images used in my work were under a Creative Commons license [14], meaning the author of the original work has enabled the free distribution of their work so that other people have the right to share, use, and build upon what they have created [15].

2.3 - Literature Survey

How Computers Play Chess

By David N.L. Levy (Author), Monty Newborn (Contributor), Published 14th April 2009, ISBN: 4871878015

This book was very useful because it provided a detailed history of computer chess and mentioned some of the earlier examples of programs that could win a game of chess. What mostly interested me to read this book, however, was the fact that it said that any reader who wants to write their own chess program will find sufficient information to make a good start. Having read the book, I can agree that it helped to think logically about how I would develop my game.

Unbeknownst to me, it also contained a section on alpha-beta pruning, which is an Artificial Intelligence algorithm approach used to help calculate an 'optimal' move (a move with the best payoff) by reducing the pool of moves the piece could take (pruning) if the move isn't optimal. This is going to be great for term two when I focus more on the Artificial Intelligence aspect of my game.

Game Programming Patterns

By Robert Nystrom, Published 2nd November 2014, ISBN: 0990582906

This is a book that was recommended to me by my advisor for my project. It covered the implementation of different design patterns, and which design pattern would be most suitable depending on what that particular class or program does. The book also covers optimisation, which is useful because I want my game to run fairly optimally, as well as how to write a robust game loop. Although I haven't applied much of what I have learnt from this book in my aims and objectives for term 1, what I have learnt from this book will be demonstrated in term 2 as I am looking to write a game loop in the very near future and implement design patterns.

Killer Game Programming in Java

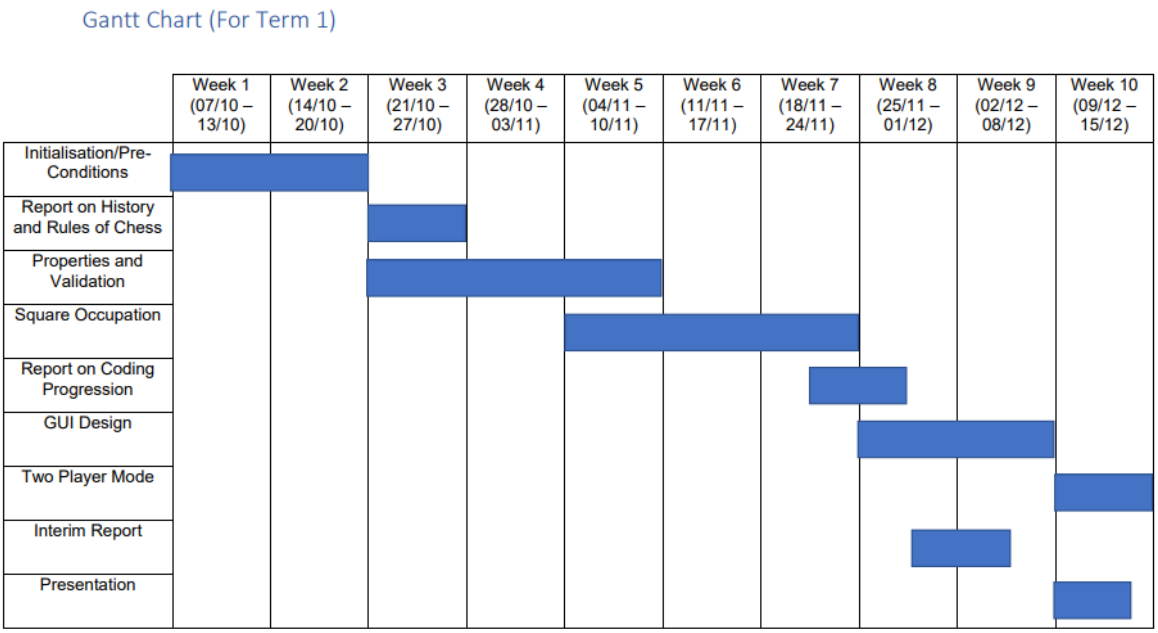
By Andrew Davison, Published 30th May 2005, ISBN: 0596007302

This is the final book I have read. It contains information on 2D Java APIs as well as the fundamentals of 2D graphics. Although interesting, the majority of the book hasn't taught me as much as I would have hoped. This could be because I am already familiar with a 2D Java API named Scene Builder from my Software Engineering module in my second year, which is the API I am going to be using for my GUI.

Chapter 3: **Planning and Timescale**

First Term Plan

In terms of my timeline of tasks, here is the Gantt chart I created showing the timescales for each task for the first term.



I feel I have mostly stuck to the timescale set out in my initial project plan. Towards the beginning of the project I was ahead of my plan, finishing most of my initialisation and pre-conditions a week earlier than anticipated. I started my report on the History of Chess a week later than the dates set out for writing the report because I did not allow myself time to do research before writing the report.

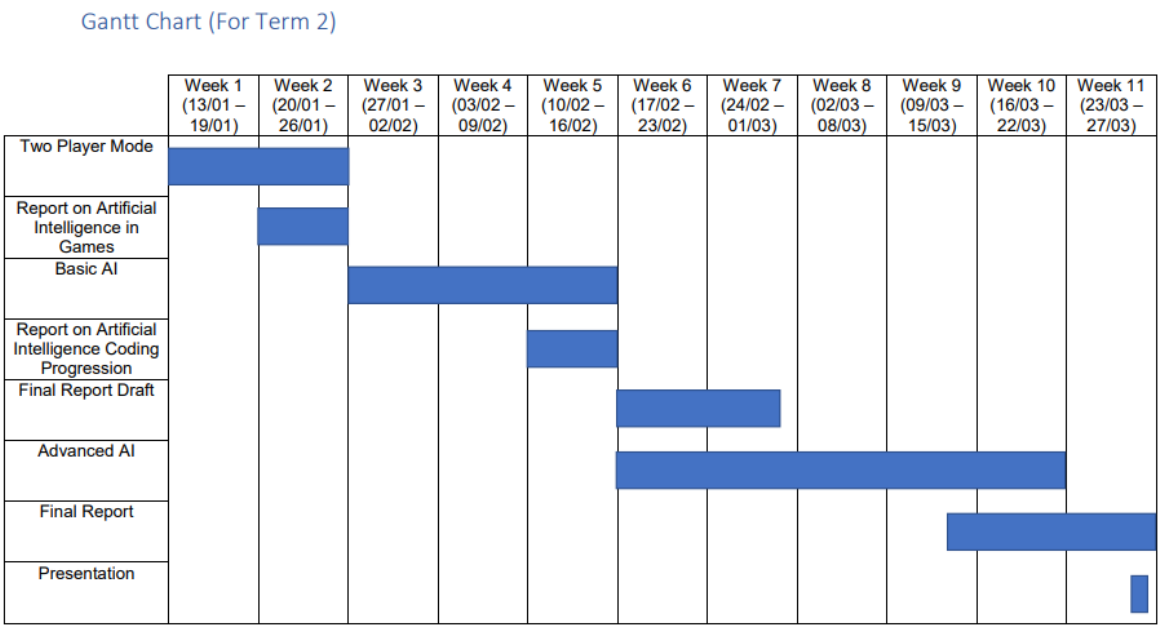
Due to this slight delay, I was able to start my next task (Properties/Validation) on 4th November, which is almost 2 weeks behind the plan. However, I completed the Square Occupation task early as part of the Initialisation/Pre-Conditions coding as I realised that the square occupation will be necessary **before** writing validation and properties for each piece type. This gave me the time needed for the Properties/Validation..

I started my Coding Progression report on 30th November and finished the report on 5th December. The majority of the contents in the Coding Progression report features in the 'Summary of Completed Work' heading.

I finally finished the Properties/Validation task on 2nd December, roughly a week behind schedule. This is due to the validation actually taking longer than first anticipated, as there is a lot that goes into what counts as a valid move for each piece and I slightly underestimated its complexity, but also due to a few bugs I discovered towards the end of the task which took a while to recognise and fix appropriately. I decided not to do my GUI until next term to focus on the technical side/code and my reports over a GUI, as this is a lower priority. Despite this, I still think that I have mostly stuck to the plan (+/- 1 week) throughout the majority of the project in the first term.

Second Term Plan

This is the Gantt chart I created in my project plan for the tasks I would like to complete in the second term.



In the second term, the actual ordering and timescale of tasks has been a lot different. During the first month of the term, I was working on separate classes for Stalemate and Checkmate. These classes would test for there being a checkmate or stalemate in each game state.

This task was much more complex than originally planned as it took a long time to develop and get working properly (this is explained in more depth in the 'Project Difficulties' section). When I was testing the Stalemate and Checkmate conditions independently of each other, I found a lot of bugs with the code written from term 1 which would not have been picked up from the Junit tests for the classes affected due to the bugs only becoming known during the integration of the game loop and that class. This led to further delay fixing these bugs so that I can integrate my game logic with the loop.

It wasn't until the end of February when I had the GUI developed, mostly completed and integrated with the game logic. From this point onwards the rest of the project was fairly simple because the harder parts involving the game's logic were now complete.

At this stage in the project, it was apparent as to how inefficient the code was running so it was important that I refactored it in such a way that reduces the amount of iterations and checking it does in each game state. To solve this, I created a LegalMoves class which handles not only calculating the legal moves in each game state, but which moves are illegal (puts King in danger or moves that do not remove a threatening piece), and whether we have either a checkmate or stalemate. This class handles the responsibility of what would have been 4 separate classes into 1.

This was in progress from the end of February till mid-April and took a lot of re-working and re-imagining to get the game into a much more playable, efficient and user-friendly state in comparison to how it was before (slow, buggy, full of not so useful output in the console and no GUI).

Towards the end of February, I wrote my Artificial Intelligence in Games report. This is featured in the Summary of Completed Work section below. The end of February also saw the beginning of my Final Project report draft, with the final version being revised based on the feedback given to me to my advisor and complete by the deadline on the 27th March.

Finally, on 23rd April, my game (in Two Player mode) and final project report is complete.

Chapter 4: **Self-Evaluation**

How did the project go?

Overall, I feel that the project went well. The first term went really well, with the majority of the game logic complete by the end of term, most of which I didn't have any real challenges/difficulties with to cause a hinderance to the progression of the project. The second term didn't go as well as the first term, due to the game terminating conditions taking up a lot more time than anticipated, but this kept being worked on until it the problem was resolved despite pushing other tasks back a bit. Pushing other tasks back to complete the loop was deemed necessary as it is crucial to the functionality of the game. Once this problem was resolved, the rest of the term went well with the overall project still being delivered by the deadline.

Where next?

I would like to use the project as a portfolio or example piece of work from what I have done during my time at university. I would also like to build on my project after university to complete tasks which I may not have been able to do due to time constraints as well as improve my game by optimising it, such as finding ways to improve the code so that it performs better and is less taxing on system resources.

As the AI algorithm aspect of the project wasn't fully completed, I would also still like to add this with the understanding I have gained through my research and attempts so far.

What did you do right/wrong?

I feel I was right to have a very code based first term because I think that if I did any less then it would have made things harder for the second term. In some ways I also think I was wrong to not do more coding in term 1 because if I had at least one of the two conditions for the game loop complete before term 2, this would have made things go more to plan/smoothly. This wasn't really possible however as I feel I did as much as I could have done in the first term.

I think I was wrong to not have as much written work for my project because about 80% of my work is all about implementation and coding, which although shows my understanding of my project, more written work showing this would have been just as good if not better.

I also think I was wrong with my estimations of the project. In hindsight, I feel I underestimated the complexity of developing a Chess engine from scratch (with no libraries) and implementing AI algorithm on top of this. At the end of the project, the entire Chess engine was realistic, but I massively underestimated the complexity of adding AI onto my game and found it to not only be difficult but running out of time for even with the project extension.

What have you learnt about doing a project?

The biggest thing I have learnt is that despite creating initial timescales for tasks before starting the project, that tasks rarely go to plan because the complexity of the task is hard to judge and is usually only known once it is actually being started or completed. It is also far too easy to give tasks names relating to what you want to achieve but those tasks may actually be unnecessary or two supposedly different tasks could actually be one task or related to/dependent on each other. An

example of this was Two Player Mode and Game Loop in my case. Although they seemed like two different tasks at the beginning, they ended up being the same because the two-player mode came as part of initialising the game, and then constantly updating game states after this whenever a move was made.

In terms of research, I have learnt a lot about Artificial Intelligence and algorithms used in games. This is something I didn't have much background knowledge on before so throughout my project I have definitely learnt a lot more about AI and had the opportunity to hone my Java skills as the code aspect of the project was 100% based in the Java language. Although I didn't get to fully implement my AI algorithm as I originally hoped, the knowledge I have gained about AI has still been useful and what I have learnt will be taken with me. I believe this knowledge to prove useful in my professional/working career after university.

Chapter 5: **Running Software**

My project was compiled with JDK 13.0.2 (Released 14th January 2020) and also uses JavaFX SDK 13.0.2. As this is the latest version at the time of writing this report, you will need this JDK version to run my project, unless you wish to re-compile the project using an earlier version.

To open the project, ideally you should be running Eclipse 2019-12 as this is the version of the workspace used for the project. Eclipse warns that using an older version of the product for the workspace could be incompatible and potentially cause unexpected behaviour or data loss.

To actually run the software itself, you need to import the project folder into Eclipse (File > Import > Existing Projects into Workspace). Once imported, you need to navigate from the project folder to the Main class (FinalYearProject > src > application > Main.java). Right click on Main.java and select the 'Run As…' option, then select 'Java Application'. This will launch the application and display the Chess GUI for you to try out yourself.

Chapter 6: **Project Difficulties and Potential Improvements**

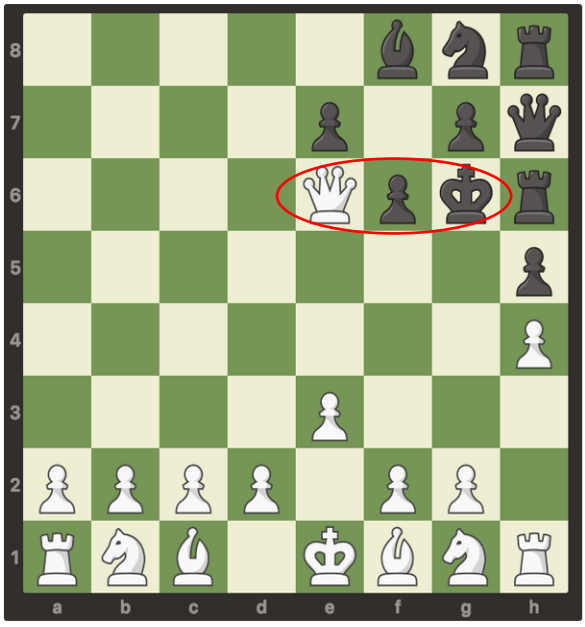
The biggest difficulty in my project was writing the Minimax algorithm and implementing this with the Chess engine I have also created. With the vast number of bugs and refactoring required in term 2, which also took a lot of time during the extension, this did not leave a lot of time to develop the algorithm to have a one-player mode. Not only did I face issues with time constraints, but I also underestimated the complexity of writing the algorithm and integrating this to my existing code base, controller, and GUI.

Due to this, I instead used this last bit of time ensure that my game was fully functional and tested in two-player mode so that I at least have a finished Chess engine and the game is playable without any known bugs, which seems to be the case.

I did, however, manage to get a vital part of the algorithm done, which is to create a set of moves containing only legal moves (a move that will not put the player's own king in danger and enable the player to block or capture threatening pieces of the opponent player). This set of moves was going to form part of calculating the best move in the algorithm.

Regardless of the implementation of the algorithm, the knowledge I have gained from my coding attempts and the research I have undertaken have proved valuable, so my plan for the future is to revisit this after university to get Artificial Intelligence implemented to my game.

The second difficulty in my project was writing the Stalemate condition. This condition is one of two conditions used to determine when the game should terminate. It was difficult to write this particular condition because it is hard to specify when a move a is legal or not. An example of this is having a White Queen, Black Pawn and Black King in a row (circled red). While the Black Pawn moving down to 5F counts as legal move for the Pawn, it isn't overall a legal move because it puts the player's own King in danger of the White Queen. I found it very hard to be able to tell the game that an otherwise legal move is illegal in a case like this.



After numerous approaches and spending a lot of time to get this working, I did manage to get the Stalemate condition working correctly as part of the methods in my LegalMoves class.

A general improvement I'd like to make is to try and make the game more optimal. By this, I mean I'd like to try and further remove the amount of iterations or checking it needs to do in order to calculate a set of legal moves. This is because when I do go and implement the Minimax algorithm again in the future, the algorithm itself is rather inefficient so making the existing code base as efficient as possible will reduce the execution time and computing resources in the game.

Chapter 7: **Summary of Completed Work**

7.1 Reports

7.1.1 History and Rules of Chess Report (Term 1)

This report is based on researching the background/history of the game. It also features my findings from my research on the rules of the game as well as some interesting facts relating to the game and its history. Below are the sections from my report where I carried out research and reported my findings, note that this excludes my history section as this is mostly covered in the introduction. For the full report, please see [A1] in my Appendix.

Facts About the Game and Its History

1) The most amount of moves you can make in the longest game of chess is 5,949 moves. [7][8]

2) The word "Checkmate" comes from the Arabic word "shah mat", meaning "The king is dead" in English. [7][8]

3) The pawn's ability to move two steps instead of one in its first go was first introduced in Spain in 1280. [7][8]

4) The longest time the World Chess Champion title was held for was 26 years and 337 days by a German man named Dr Emanuel Lasker. [7][8]

5) The chess board we know today first appeared in Europe in 1090. [7][8]

6) The second book which was ever printed in the English Language was about how to play Chess. [7][8]

7) Alan Turing developed the first computer program which was able to play chess in 1951. No computers were powerful enough to process the program, so Turing could only test it by doing calculations himself and playing in accordance to the results of each move. Each move took about several minutes. [7][8]

8) The first time a computer beat an international expert in the game was in November 1988, in Long Beach, California. The computer was named Deep Thought (This is not to be confused with Deep Blue which was the first computer system to beat a world champion in the game). [7][8]

9) The number of possible moves when playing the first four moves (for both sides) in the game is 318,979,564,000. [7]

10) Approximately 600,000,000 people know how to play chess across the world. [8]

Basic Rules of Each Chess Piece

Pawns:

- Can move forward one square if the square is unoccupied.
- A pawn can move forwards two squares if that pawn has not moved before and both squares in front are unoccupied.
- If the squares diagonally in front of the pawn contain an opponent piece, the pawn can move one square diagonally to capture that piece.

Rooks:

- Can move any number of squares vertically or horizontally, as long as the squares are vacant.

Bishops:

- Can move any number of squares diagonally, as long as the squares are vacant.

Knights:

- Can move in an L shape in any direction. It can move two squares forwards, backwards, left or right, then one square to the left or right. Alternatively, it can move one square forwards, backwards, left or right, then two squares to the left or right.
- Knights can jump over pieces in its intended path, so the squares in its path do not have to be vacant.

Queen:

- Can move in any number of squares in all directions (horizontally, vertically, or diagonally), as long as the squares are vacant.

King:

- Can move one square at a time in all directions (horizontally, vertically, or diagonally), as long as the square is vacant. [9]

Chess Manoeuvres

Castling: Castling is the only move where more than one piece can move in a single turn. When castling, the king will move two squares in the direction of the rook the king wants to castle with, the rook then moves over the king into the square the king passed.

There are two conditions which must be met before castling:

- Both the king and rook must not have moved from their original positions
- There cannot be any pieces between the king and rook when castling

En Passant: This is a move which occurs straight after a pawn (which is going to be captured) moves two squares from its starting position. To make an En Passant capture, it must be made in the next turn as you cannot do so afterwards/later on.

These are the conditions for En Passant:

- The capturing pawn must be on the 5th row/rank
- The pawn being captured must be on the same row and has just moved two squares in one move
- The capture must be made in the move after the opponent pawn moves two squares, otherwise the player cannot capture the opponent's pawn en passant.

Pawn Promotion: This is when a pawn reaches the end of the board on the opponent's side. The pawn can be promoted to a queen, rook, bishop or knight, and is entirely down to the player's choice. The piece the pawn is being promoted to does not need to be a previously captured piece. [9]

Check

Check is when it is possible for the opponent to attack/capture the player's king in their next turn. Sometimes a player cannot move one of their pieces as it would put their own king into check for the opponent player, and it is illegal for the player to make a move that would put their king in check.

In informal games, it is known as common practice for the opponent to announce when they've put the player's king in check. In competitive/formal games, however, the opponent does not have to announce that the player's king is in check.

There are a few possible ways to get the king out of check:

- Moving the king to a square that cannot be moved into from an opponent player's piece in the opponent's next turn.
- Capture the opponent's piece (either with the king itself or another piece).
- Block the opponent's path to the king by placing another piece in its way.

If it is not possible for the player to get their king out of check, the king is checkmated, which means the game is now over (see 'Winning Conditions'). [10]

Winning Conditions

Checkmate

If a player's king is in check and they cannot legally get their king out of check, the king is "checkmated", meaning the opponent can capture their king and the game ends. The king is not actually removed or captured because a situation where the king is checkmated is sufficient to end the game without making a further move.

Resigning

A player can resign at any point during the game, which means their opponent would win. A player would usually resign when they think that they are highly likely to lose the game and would be pointless continuing. They can indicate their resignation by simply saying they resign from the game.

Draw

A draw means there is no conclusive winner or loser in a game. It can occur in the following conditions:

1) In a stalemate, which is when the player is not in check but cannot make a legal move.

2) When it is not possible to checkmate for both the player and their opposition with any legal move. This can happen when there are insufficient pieces, or the existing pieces are unable to make a move to put a king in checkmate. These scenarios can occur with the following combination of pieces:
    - King against King
    - King against King and Bishop
    - King against King and Knight
    - King and Bishop against King and Bishop (where both Bishops are squares of the same colour)

3) When both players agree to draw when one of the players offers to draw.

Time Control

This can happen in a competitive game where a player under time control can run out of their allocated/specified time limit to make a move, thus ending in a loss for the player and win for their opponent. [11]

7.1.2 Coding Progression Report (Term 1)

This report covers how my code has progressed since September to the beginning of December during the first term. It goes into detail by describing each method and how it works. As the report is very long, I have included an abstract from that report describing my validation for a Pawn under my 'Validation' heading. The full report is [A2] in the Appendix.

Pawns

According to the rules for a Pawn; "*Pawns can move forward one square, if that square is unoccupied. If it has not yet moved, the pawn has the option of moving two squares forward provided both squares in front of the pawn are unoccupied. A pawn cannot move backward. Pawns are the only pieces that capture differently from how they move. They can capture an enemy piece on either of the two spaces adjacent to the space in front of them (i.e., the two squares diagonally in front of them) but cannot move to these spaces if they are vacant."* [1]

Therefore, in my code for the movePawn method in my Pawn class, my first condition (surrounded in a red square below) uses a Boolean method firstMove to check if the particular pawn passed to the method has moved before or not. This firstMove method scans through an ArrayList which contains a list of pawns that get added to the list once they have moved. If the pawn isn't in the list, the pawn hasn't moved yet and can take advantage of moving two squares forward.

The first condition also checks the Pawn is moving straight forward (staying in the same column) and is moving either one or two spaces forward (as the pawn doesn't have to move two spaces forward in its first go).

If the first condition isn't met, the program goes to the second condition/else if statement (surrounded in a green square below). This condition checks to see if the Pawn simply wants to move forward a tile, by seeing that the pawn is moving forward/staying in the same column, but there is a difference of 1 from its origin tile to its destination tile. It can be safely assumed that if the program goes into this condition that the pawn has already moved before, otherwise it would have executed the code in the first condition.

In the third condition (surrounded in a blue square below), the program checks to see if the Pawn is moving diagonally one space either North West or North East. This would indicate an offensive move by a Pawn to capture an opponent player's piece. If none of the above conditions are met, we must have an illegal move because the moves valid with a Pawn piece type are coded and have already been checked against.

```
public void movePawn(String player, String selectedPiece, String toTile)
    throws InvalidPieceException, InvalidPlayerException {

  String fromTile = board.map.getTile(selectedPiece);
  char fromRow = fromTile.charAt(0);
  char fromColumn = fromTile.charAt(1);
  char toRow = toTile.charAt(0);
  char toColumn = toTile.charAt(1);

  int fromRowNo = Integer.parseInt(String.valueOf(fromRow));
  int toRowNo = Integer.parseInt(String.valueOf(toRow));

  /* If statement checking if it is the pawn's first move and if so, the pawn is being moved one
   * or two spaces forward.
   */
  if (firstMove(selectedPiece) && (toRowNo == fromRowNo + 1 || toRowNo == fromRowNo + 2)
      && fromColumn == toColumn) {
    setPos(player, selectedPiece, toTile);
    movedPawns.add(selectedPiece);

  /* Else if statement for checking if the player wants to move the pawn forward 1 tile, assuming
   * its not the pawn's first move based on the previous if statement checking this */
  } else if ((toRowNo == fromRowNo + 1) && fromColumn == toColumn) {
    setPos(player, selectedPiece, toTile);

  /* Checking for moving the pawn forward diagonally and taking an opponent piece.
     It gets the column letter by using (int) so we get the ASCII value of the char.
     If the ASCII value is +/- 1 (a column to left or right of current pawn), the row is + 1
     (in front of the current pawn), and the target tile is occupied with an opponent piece,
     we have a valid move to capture an opponent piece. */
  } else if ((toRowNo == fromRowNo + 1) && (int)fromColumn == (int)toColumn + 1
      || (toRowNo == fromRowNo + 1) && (int)fromColumn == (int)toColumn - 1) {
    setPos(player, selectedPiece, toTile);

  } else {
    System.out.println("Illegal Move. Please move your piece in accordance to the game's rules");
  }

}
```

7.1.3 Artificial Intelligence in Games Report (Term 2)

What is Artificial Intelligence?

To summarise, Artificial Intelligence is a field in Computer Science which demonstrates the intelligence of machines/computer systems. Not only this, but AI is also used to describe machines which can learn or study based on actions of a human player (more commonly known as machine learning but is often used in conjunction with Artificial Intelligence), and can be seen as machines which mimic normal/cognitive human functions [16].

The basic goals of AI include:

- Reasoning: The ability for a computer system or program to automatically and fully reason.
- Knowledge Representation: The ability for a computer system or program to use knowledge about the world or state it is in which the computer can utilise to perform/complete tasks.
- Planning: The ability for a system to visualise/plan a set of actions it can take.
- Learning: The ability for the system to learn or add new information to its knowledgebase which it can then utilise to complete future tasks.

- Natural Language Processing: The ability for a computer system to interact with humans in their natural language, and how the computer can interpret and process the natural language into one it can understand itself.
- Perception: The ability for a computer system or program to gain useful information through the usage of hardware as a method of examining its surroundings and using this information to respond to the environment appropriately.

History of Artificial Intelligence in Games

Artificial Intelligence in video games is a different aspect/subdivision of AI in general and from the summary described above. This is due to AI in video games having a different goal, which is to enhance the game experience as opposed to the academic side of AI, which is focused on the traditional goals and machine learning or decision making.

In the early 1950s, the implementation of AI has been a huge area of research since its inception at this time period. The earliest examples of AI being used in a computer game is the game Nim, which was made in 1951 and published in 1952.

In 1951, Christopher Strachey wrote a checkers program and Dietrich Prinz wrote a Chess program both using the Ferranti Mark 1 machine belonging to the University of Manchester. These were other examples of the earliest computer programs to ever be written. Developed during the middle of the 1950s to early 1960s, Arthur Samuel's checkers program was the first to have enough skill to take on a highly regarded amateur in the game.

In the 1960s and 1970s, well-known games such as Spacewar!, Pong, and Gotcha all implemented discrete logic and involved two player game modes, meaning there wasn't an AI implementation/one player mode.

The first one player mode games started to emerge in the 1970s, with notable games including Taito, Qwak, and Pursuit. There were also text-based (normally based in the command line) games which had enemies, the movements of which were based on coded patterns. Microprocessors were used in these games to allow more randomisation over the movement patterns as a way of making them less predictable. Two of these text-based games included Hunt the Wumpus and Star Trek.

Arguably, AI exponentially grew in popularity and development during the video arcade game era. This was mainly down to the success of Space Invaders (released in 1978) which had various difficulty levels, randomised movement patterns, and game events controlled by hash functions which used the player's input. Pac-Man (released in 1980) also used Artificial Intelligence-controlled movement patterns for the enemies in the game, each of which having different traits to vary the gameplay.

In the 1990s, the emergence of new game genres and sub-genres used Artificial Intelligence in a more complex manner. For example, games using real time strategy used AI with multiple objects, gave the system incomplete information, pathfinding problems, and forcing the system to make decisions in real-time. In 1997, thanks to the progression of Artificial Intelligence and the building onto the early checkers and chess programs made in the 50s and 60s, IBM's Deep Blue computer

managed to beat Garry Kasparov (a chess grandmaster and former world chess champion) in the game.

In modern day games, existing techniques and methodologies are still heavily used as a way to control the actions of non-player characters (often abbreviated to NPCs). Most commonly used methodologies today are pathfinding and decision trees to guide the NPC. Artificial Intelligence is also used in processes not usually visible to the user, examples of which are data mining and procedural generation [17].

What Makes Good and Bad AI?

This is a short section in my report covering what is deemed good and bad AI in video games based on the opinions of those who play video games or develop them. In short, good Artificial Intelligence implementations are ones that are realistic and are fairly smart, but also not programmed perfectly in the sense that they can't be beaten. It is important that the player feels they can still beat or get a sense of achievability from the game they are playing which includes a form of AI.

Bad AI in video games is said to be implementations which use Machine Learning. This is because the AI constantly becomes better as the player plays the game, it would be hard for developers to balance the AI so that it is still fun to play against. On a thread discussing what makes AI bad in games, one of the responses were; "The goal of machine learning AI would be to beat the player, while the goal of the game designer is to make the AI fun to beat (challenging but not overly so)". Therefore, this backs up that it would be hard to balance as both concepts are conflicting.

Another example of bad AI implementations in video games include those that have the same movement patterns. This is because humans are good at recognising patterns and after a while the AI becomes predictable which can make the game easy and lose its appeal, especially when AI is used as a way to improve the overall game experience. This means that it is good to have an element of randomisation (RNG) in Artificial Intelligence approaches so that the AI's patterns aren't completely predictable. This gives a more realistic feel to the AI in the game, and in the case of enemies, means that they still pose a challenge (at least to some extent) and remain to have an element of unpredictability/ randomisation in what they do and their movement patterns [18].

AI Algorithms

Monte Carlo Tree Search

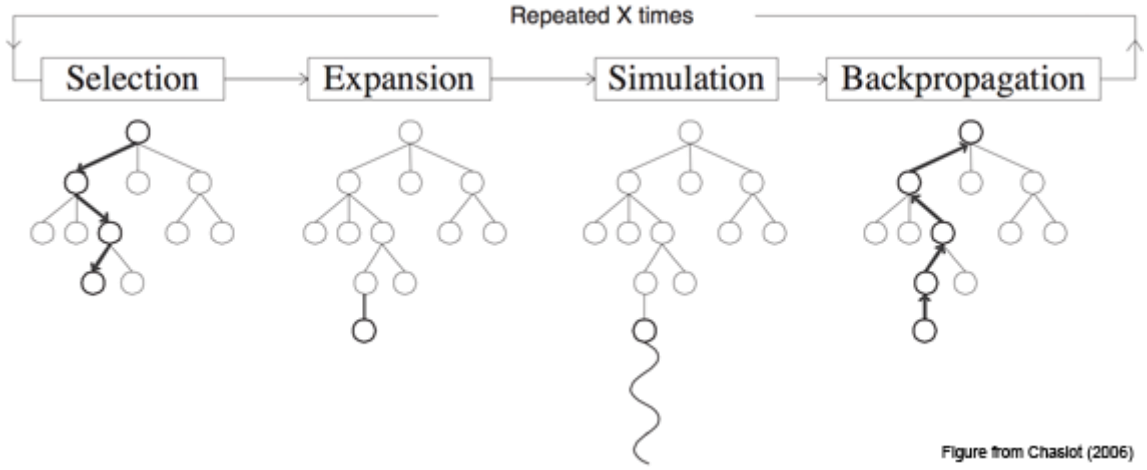
The Monte Carlo algorithm is used in computer systems as a way to determine the behaviour of another system, in this case a game implementing Artificial Intelligence. It mainly uses the element

of randomness and statistics as a way to determine the outcome. The algorithm does not use fixed inputs and instead uses numbers to set the probability of a certain event happening in the outcome after the algorithm is run [19].

Monte Carlo has been applied to Artificial Intelligence for games by developing it into the Monte Carlo tree search. This tree search is used to search for what is deemed the best move/outcome in a game.

The tree search has these four main steps:

1) Starting at the root node, recursively select the most optimal child nodes until a leaf node is reached.
2) If the leaf node is not a terminal node (meaning a node which doesn't end the game), then create a child or children nodes and select one of them.
3) Simulate what would happen in the game (playout) until a result is achieved.
4) Update the tree/move sequence with the new child node with the simulation result (known as backpropagation) [20].



It is important to note that each of nodes in the tree should contain two pieces of information. These are an estimated value (payoff) based on the results of the simulation, and the amount of times the node has been visited.

Monte Carlo Tree Search – Benefits

Aheuristic – The tree search does not need any form of strategic or tactical knowledge about the game it is used in to be able to make rational decisions. The only information the search does need of the game it is being implemented in, is what counts as a legal move and the terminating/goal states for the game. If you were to write a single implementation of the Monte Carlo tree search, you would be able to reuse that implementation into many other games with very few modifications needed.

Asymmetric – The tree search can asymmetrically grow the tree (its search space). It uses the algorithm to expand the optimal nodes much more often than the not so optimal nodes, therefore its searches are usually in the relevant parts of the tree.

This means that the Monte Carlo tree search can be used in games with large amounts of outcomes. A large combination of outcomes can be problematic or not as efficient if using other forms of tree searches such as breadth-based or depth-based tree searches. Therefore, due to how adaptive Monte Carlo is, it will find the moves that appear most optimal even if it takes a while to find and focus its searching efforts in that part of the tree.

Availability – One of the major benefits to the algorithm is that it can be stopped at any time to return the best estimate it has currently come up with. The tree that has been built so far can either be started over or kept for future use [20].

Monte Carlo Tree Search - Drawbacks

Playing Strength – This can be seen as a major drawback as the Monte Carlo tree search can sometimes fail to find the most reasonable move, even for games which aren't that complex, within a reasonable amount of time. This is due to the size of the tree and that the search of the tree may not have visited all the nodes enough times to be able to give the best estimates.

Speed – Another major drawback with the Monte Carlo tree search is related to 'Playing Strength' listed above but presents speed as another drawback. The search can take numerous iterations to come up with the best solution, and as a result of this, it wouldn't make the game very efficient and would make it harder to optimise. In the Chess game I am producing where there are lots of outcomes, a search which does not require numerous iterations would be more optimal/efficient on both the system's resources and actual compute time on the system running the game [20].

Both these drawbacks can be improved on so that the search's performance is better. An approach known as **domain knowledge** can be used as a method to filter out implausible/unrealistic moves so that the moves made by the search are more similar to those of a human player. The benefits of this are fewer iterations, resulting in faster compute time and more realistic playoffs.

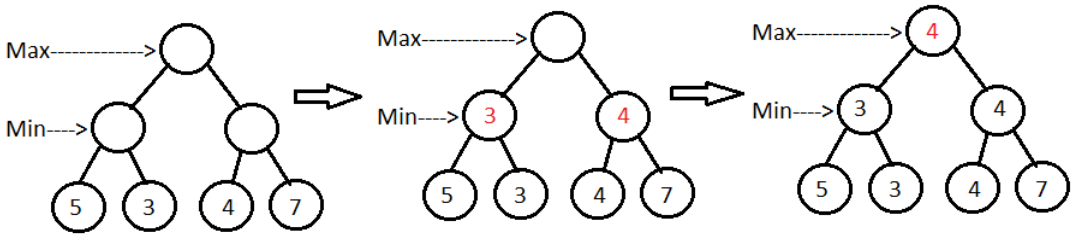
Minimax

Minimax is a decision-based Artificial Intelligence algorithm where one player attempts to achieve the maximum payoff/result against the opponent player, who is trying to minimise the payoff by countering the move (normally the computer). IBM's Deep Blue computer (made in 1997) was using the Minimax algorithm to defeat Garry Kasparov, who was a chess grandmaster and former world champion in the game.

In terms of the algorithm, it is assumed that the goal for the player is to play optimally in order to win the game, whereas the goal of the computer is to play strategically and minimise the payoff of the human player's move. The algorithm does this by iterating through a search tree and selecting the nodes with the lower evaluation scores (lowest payoffs). The algorithm can be designed in such a way so that it is smart, by having the ability to look at potential moves the opponent can make in response to the player's move [21].

The tree search has these three main steps:

1) Traverse the tree using a depth-first search, meaning the tree looks at the lowest level nodes known as the leaf nodes. As it is the min player's turn to choose a move in the tree, the evaluation function looks at the values of the two nodes and stores the lower value in the parent node.
2) Recursively store the scores from the leaf or children nodes (whether it is a leaf of child node depends on the level of the tree we are traversing) to their parent nodes until reaching root. When it is the min player's turn, keep selecting the leaf or child node with the lowest score. When it is the max player's turn, keep selecting the leaf or child node with the maximum score.
3) Once the root node is reached, the higher of the two values is selected and the move with the higher payoff is made [22].



Minimax Tree Search – Benefits

Decision Making – It was one of the first algorithms to make decision making based on evaluating pay off scores as a way to determine the computer's next move. It is widely used in computer board games such as Chess, but due to its drawback being a major one, it is better used in the conjunction with the alpha-beta pruning algorithm.

Minimax Tree Search - Drawbacks

Speed – One of the biggest drawbacks for the Minimax tree search is that it can be time consuming for the algorithm to compute which move it will make if there are a lot of branches (possible outcomes) in the game it is being implemented in. As the algorithm would be implemented in Chess, the complexity, compute time and branching factor of the algorithm would be too high. As it can be very time consuming to build a game tree for all possible outcomes or compute the highest payoff move for a game with a high branching factor, the biggest improvement that can be

made to the Minimax algorithm is pruning. The pruning process is used in the Alpha-beta pruning algorithm, which is an optimised variant of the Minimax algorithm.

Alpha-Beta Pruning

Alpha-Beta pruning is a variant of the Minimax algorithm. It is used to optimise the compute time and overall efficiency of the Minimax algorithm by allowing the search to be faster. Alpha-Beta pruning works by cutting off branches (prune them) which do not need to be searched in future iterations of the tree because there already exists a better/more optimal move. Therefore, the algorithm does not need to take that branch of the tree into account after the branch is pruned. Alpha-Beta pruning gets its name by using alpha and beta parameters, which are additional to the parameters needed for the minimax algorithm.

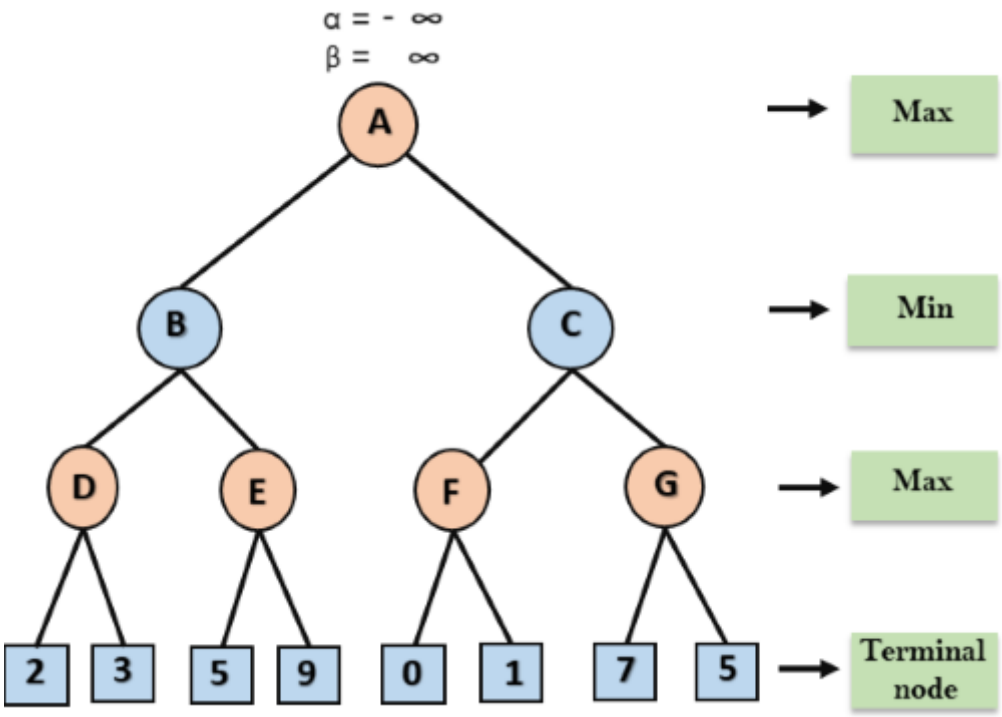
Alpha: The best value that the maximiser can guarantee at any point in its path so far.

Beta: The best value that the minimiser can guarantee at any point in its path so far.

The main pre-requisites for alpha-beta pruning are that the max/maximiser player will only update the value of alpha, the min/minimiser player will only update the value of beta, while backtracking the tree the nodes will be passed to upper/parent nodes instead of alpha and beta values, and we will only pass alpha and beta values to the child nodes [23].

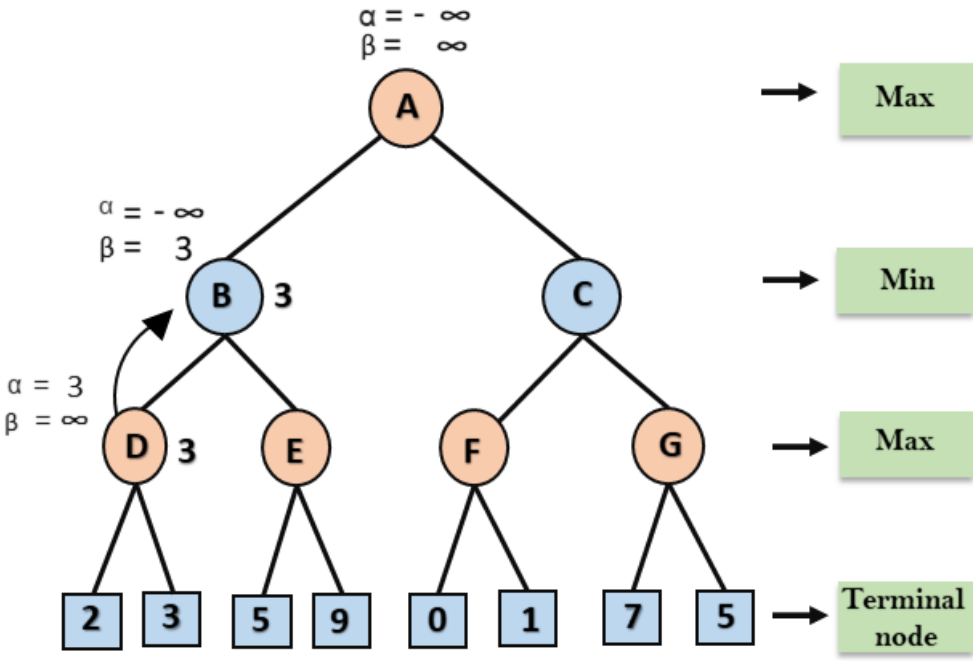
How Does Alpha-Beta Pruning Work?

1) Max player starts from node A where alpha = -∞ and beta = +∞. These alpha and beta values are passed down to node B, and then to child node D.

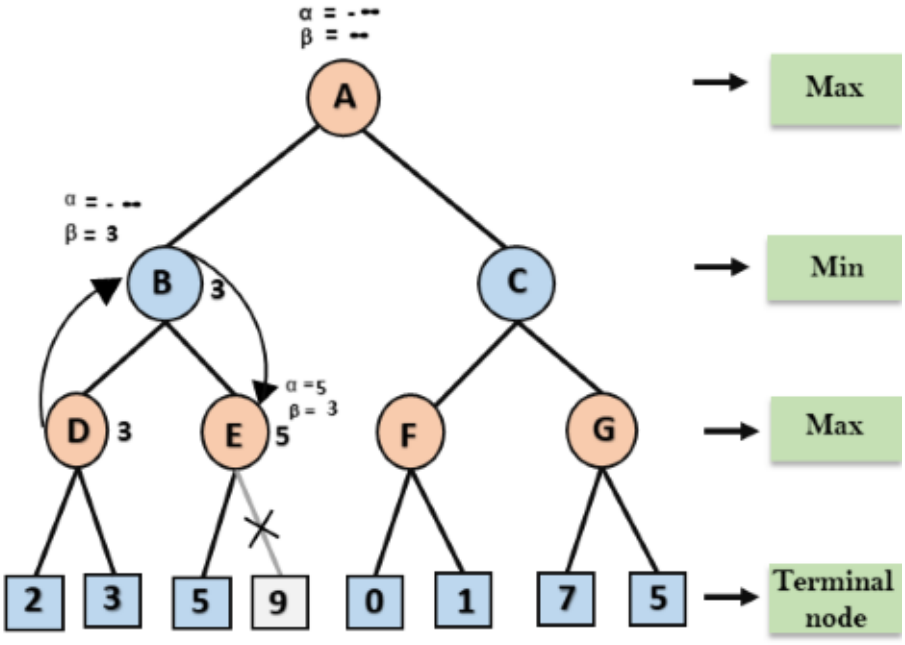


2) At node D, it is Max player's turn so the value of alpha will be compared with 2 and 3. As 3 is the higher value, the value of alpha and node D's value will both be set to 3.

3) The algorithm backtracks to node B where it is Min player's turn. Beta will compare with the available subsequent nodes value, so for now beta is set to 3 and alpha is set to -∞ again.

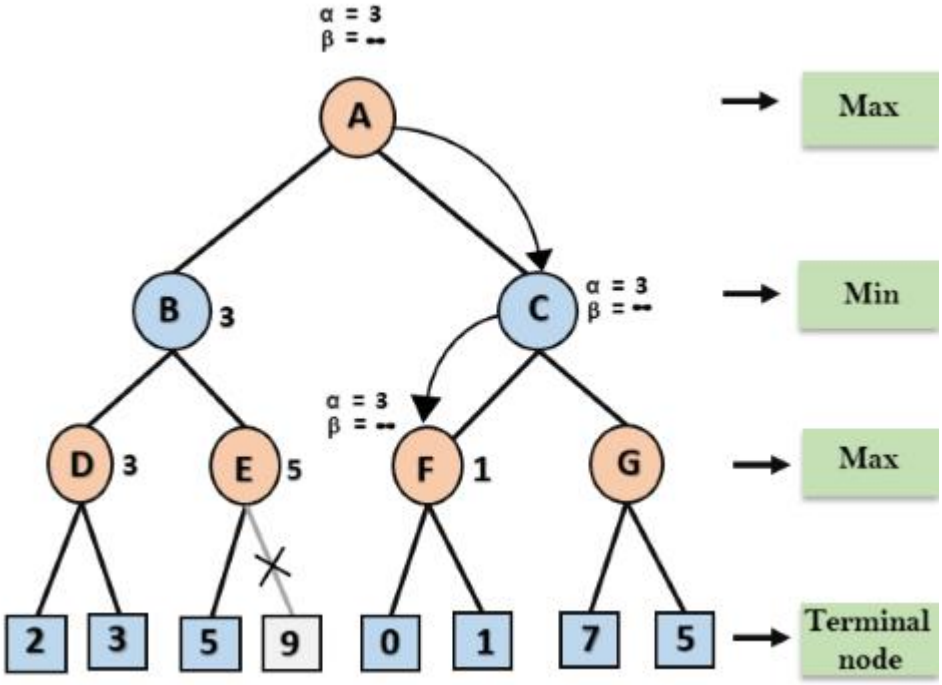


4) The algorithm goes to node E, it is Max player's turn again. The current value of alpha will be compared with 5 (so max(-∞, 5) = 5). Therefore, at node E, alpha = 5 and beta = 3.



5) The algorithm backtracks from node B to node A. At node A, the value of alpha is changed to the maximum available value which is 3 as max (-∞, 3) = 3 and beta = +∞. These values get passed to node C, and then node F.

6) At node F, the value of alpha once again gets compared with the terminal nodes. The left child is 0 so max(3, 0) = 3, and the right child is 1 so max(3, 1) = 3. As both remain 3, alpha remains the same but node F's value is set to 1 as it's the highest of 0 and 1.



7) Node F returns the value 1 to node C. At C the current values are alpha = 3 and beta = +∞. The value of beta will be changed as it is Min's turn. It will compare with 1 so min(∞, 1)= 1. At C, we have alpha = 3 and beta = 1. It satisfies the condition that alpha is larger than or equal to beta. The next child of C is G, which is pruned because C contains a beta smaller than alpha/alpha larger than or equal to beta.

8) Node C returns the value 1 to node A, and Node B is returning the value 3. As it is Max's turn, the best value for node A would be 3, as max(3, 1) = 3. This concludes that the optimal value for Max is 3 [23].

Alpha-Beta Pruning – Benefits

Efficiency: The algorithm can reduce the number of nodes it needs to visit (via pruning) so that the optimal move is found in a lower compute time. If the algorithm uses good ordering (explained in

drawbacks below), it can be twice as fast as a Minimax algorithm without any sort of alpha-beta pruning implementation.

Alpha-Beta Pruning – Drawbacks

Ordering - The efficiency/effectiveness of the implementation of alpha-beta pruning depends on the ordering of which nodes are examined. If the algorithm does not prune any of leaves or branches in the tree, it is in worst ordering. It basically means the algorithm is the same as the Minimax algorithm with this ordering and is very slow.

For an efficient implementation, the algorithm should be using ideal ordering. Ideal ordering occurs when lots of pruning happens on the leaves and branches in the tree. The best moves are found in the left side of the tree. Depth first search is used hence the searching from the bottom left of the tree to until it recursively gets to the leaf nodes on the right-hand side of the tree.

To find good ordering:

- Occur the best move from the shallowest/leaf nodes
- Order nodes in such a way that the best nodes are examined first
- Make use of domain knowledge to find the best move. For example, for Chess the priority could be captures first, then threats, then forward moves, and then backward moves.
- Record the states as there is a possibility that states may repeat [23].

7.2 Implementation

Data Structures

```
public class TreeMaps {

  private TreeMap<String, String> piecePos;
  private TreeMap<String, String> tileOccupation;
  public ArrayList<String> capturedPieces;
  public boolean valueLock = false;

  /**
   * This is a constructor for creating/initialising the TreeMaps I will need
   * for storing my Chess game data.
   */
  public TreeMaps() {
    piecePos = new TreeMap<String, String>();
    tileOccupation = new TreeMap<String, String>();
    capturedPieces = new ArrayList<String>();
  }
```

PiecePos TreeMap

This TreeMap takes a tile co-ordinate (e.g. 1A) and maps it to a chess piece. This means that the chess piece is currently in that tile. When a chess piece is captured, its position becomes null so that it no longer exists (no longer mapped to a tile co-ordinate) on the chess board.

TileOccupation TreeMap

This TreeMap takes a tile co-ordinate (e.g. 1A) and maps it to a value which is the tile's/square's occupation status. A tile can either be "Occupied" or "Empty". This correlates to if a piece is inside the tile or not. For example, if a piece is inside a given tile, the status will be "Occupied". If isn't a piece in the given tile, its status is "Empty".

CapturedPieces ArrayList

This is a smaller data structure used for simply storing the pieces that have been captured by either of the two players. This way I can store the pieces which are no longer on the Chess board, but still exists as a captured piece.

Validation

There are validation methods inside each Piece type class, therefore, there are six validation methods for six different piece types in Chess. Each validation method screenshot below contains comments to help understand the purpose/thought process for the components of the method.

Move Pawn Validation (1 of 1)

```
public void movePawn(String player, String selectedPiece, String toTile)
    throws InvalidPieceException, InvalidPlayerException {

  String fromTile = board.map.getTile(selectedPiece);
  char fromRow = fromTile.charAt(0);
  char fromColumn = fromTile.charAt(1);
  char toRow = toTile.charAt(0);
  char toColumn = toTile.charAt(1);

  int fromRowNo = Integer.parseInt(String.valueOf(fromRow));
  int toRowNo = Integer.parseInt(String.valueOf(toRow));

  /* If statement checking if it is the pawn's first move and if so, the pawn is being moved one
   * or two spaces forward.
   */
  if (firstMove(selectedPiece) && (toRowNo == fromRowNo + 1 || toRowNo == fromRowNo + 2)
      && fromColumn == toColumn) {
    setPos(player, selectedPiece, toTile);
    movedPawns.add(selectedPiece);

  /* Else if statement for checking if the player wants to move the pawn forward 1 tile, assuming
   * its not the pawn's first move based on the previous if statement checking this */
  } else if ((toRowNo == fromRowNo + 1) && fromColumn == toColumn) {
    setPos(player, selectedPiece, toTile);

  /* Checking for moving the pawn forward diagonally and taking an opponent piece.
     It gets the column letter by using (int) so we get the ASCII value of the char.
     If the ASCII value is +/- 1 (a column to left or right of current pawn), the row is + 1
     (in front of the current pawn), and the target tile is occupied with an opponent piece,
     we have a valid move to capture an opponent piece. */
  } else if ((toRowNo == fromRowNo + 1) && (int)fromColumn == (int)toColumn + 1
      || (toRowNo == fromRowNo + 1) && (int)fromColumn == (int)toColumn - 1) {
    setPos(player, selectedPiece, toTile);

  } else {
    System.out.println("Illegal Move. Please move your piece in accordance to the game's rules");
  }

}
```

Move Rook Validation (1 of 2)

```
/**
 * This method is responsible for validating a move when the player has selected a Rook piece.
 * If all validation conditions are met, the method will then call a set position method to move/
 * update the selected Rook's position on the board.
 *
 * @param player Takes the current player (either "White" or "Black") so this can passed to the
 *       sub-method 'setPos' later on to set the Rook's position.
 * @param selectedRook Takes the selected Rook piece to validate and move that piece.
 * @param toTile The tile the selected Rook intends to move to.
 * @throws InvalidPlayerException if the player passed to the method is not "White" or "Black".
 * @throws InvalidPieceException if an invalid piece is passed to the method.
 */
public void moveRook(String player, String selectedRook, String toTile)
    throws InvalidPieceException, InvalidPlayerException {

  String fromTile = board.map.getTile(selectedRook);
  int fromRow = Integer.parseInt(String.valueOf(fromTile.charAt(0)));
  char fromColumn = fromTile.charAt(1);
  int toRow = Integer.parseInt(String.valueOf(toTile.charAt(0)));
  char toColumn = toTile.charAt(1);

  /* MOVING LEFT. Where the row number remains the same, but the fromColumn is greater than the
   * toColumn. For example, moving from 8H (where H in ASCII is 72) to 8A (where A in ASCII is
   * 65).
   */
  if (fromRow == toRow && (int)fromColumn > (int)toColumn) {
    /* CHECKING FOR ANY PIECES IN ROOK'S MOVEMENT PATH (EXCLUDING DESTINATION TILE)
    I search each tile to the left of the current one to see if it contains any pieces, until
    reaching one tile before the Rook's destination tile */
```

```
  /*
   * MOVING UP. Where the column letter (as an ASCII value) remains the same, but the fromRow
   * number is less than the toRow number. For example, moving from 1A (where 1 is the row) to
   * 3A.
   */
  } else if ((int)fromColumn == (int)toColumn && (fromRow < toRow)) {
    /* CHECKING FOR ANY PIECES IN THE ROOK'S MOVEMENT PATH (EXCLUDING DESTINATION TILE)
    I search each tile above the current tile to see if it contains any pieces (player or
    opponent), until reaching one tile before the Rook's destination tile. */
    System.out.println("----- Moving Up -----");
    for (int i = fromRow + 1; i < toRow; i++) {
      System.out.println("The tile is: " + i + fromColumn);
      if (board.map.getPieceOrOccupation("tileOccupation", i + String.valueOf(fromColumn))
          == "Occupied" && piece.isOpponentPiece(player, board.map.getPieceOrOccupation(
          "piecePos", i + String.valueOf(fromColumn))) == false) {
        System.out.println("Illegal Move. There is a piece in your Rook's movement path "
            + "(Tile: " + i + fromColumn + ")");
        pieceInPath++;
      }
    }
    setPos(player, selectedRook, toTile);
  /*
   * MOVING DOWN. Where the column letter (as an ASCII value) remains the same, but the fromRow
   * number is greater than the toRow number. For example, moving from 8A (where 8 is the row) to
   * 4A.
   */
  } else if ((int)fromColumn == (int)toColumn && (fromRow > toRow)) {
    /* CHECKING FOR ANY PIECES IN THE ROOK'S MOVEMENT PATH (EXCLUDING DESTINATION TILE)
    I search each tile below the current tile to see if it contains any pieces (player or
    opponent), until reaching one tile before the Rook's destination tile. */
    System.out.println("----- Moving Down -----");
    for (int i = fromRow - 1; i > toRow; i--) {
      System.out.println("The tile is: " + i + fromColumn);
      if (board.map.getPieceOrOccupation("tileOccupation", i + String.valueOf(fromColumn))
          == "Occupied" && piece.isOpponentPiece(player, board.map.getPieceOrOccupation(
          "piecePos", i + String.valueOf(fromColumn))) == false) {
        System.out.println("Illegal Move. There is a piece in your Rook's movement path "
            + "(Tile: " + i + fromColumn + ")");
        pieceInPath++;
      }

    }
    setPos(player, selectedRook, toTile);

  /*
   * If we do not have a horizontal or vertical move, the move must be an illegal move.
   */
  } else {
    System.out.println("Illegal Move. Please move your piece in accordance to the game's rules");
  }
}
```

Move Rook Validation
(2 of 2)

```java
/**
 * This method is responsible for validating and then moving a Bishop piece.
 *
 * @param player Takes the current player (Either "White" or "Black").
 * @param selectedBishop Takes the selected Bishop piece.
 * @param toTile Takes the destination tile the player intends to move the Bishop to.
 * @throws InvalidPlayerException If an invalid player is passed to the method.
 * @throws InvalidPieceException If an invalid piece is passed to the method.
 */
public void moveBishop(String player, String selectedBishop, String toTile)
    throws InvalidPlayerException, InvalidPieceException {

  String fromTile = board.map.getTile(selectedBishop);

  int fromRow = Integer.parseInt(String.valueOf(fromTile.charAt(0)));
  char fromColumn = fromTile.charAt(1);
  int toRow = Integer.parseInt(String.valueOf(toTile.charAt(0)));
  char toColumn = toTile.charAt(1);

  /* MOVING NORTH EAST DIAGONALLY. Where the toColumn letter (as an ASCII value) is greater than
   * the fromColumn letter, and the toRow number is greater than the fromRow number. For example,
   * moving from 1D (where 1 is the row and D is the column) to 3F.
   */
  if (((int)toColumn > (int)fromColumn) && (toRow > fromRow)) {
    System.out.println("----- Moving North East -----");

    /* Ensuring that the diagonal move is valid by checking that the Bishop
     * is moving the same number of spaces in both directions, therefore we
     * don't get a North East move which could be 2 spaces up and 1 space
     * right for example.
     */
    int colDifference = (int)toColumn - (int)fromColumn;
    int rowDifference = toRow - fromRow;
    int j = (int)fromColumn;

    if (colDifference == rowDifference) {

      for (int i = fromRow + 1; i < toRow; i++) {
        ++j; // Also increments column to move East as part of moving diagonally North East
        if (board.map.getPieceOrOccupation("tileOccupation", i + String.valueOf((char)j))
            == "Occupied" && piece.isOpponentPiece(player, board.map.getPieceOrOccupation(
            "piecePos", i + String.valueOf((char)j))) == false) {
          System.out.println("Illegal Move. There is a piece in your Bishop's movement path "
              + "(Tile: " + i + (char)j + ")");
          pieceInPath++;
        }
      }

      setPos(player, selectedBishop, toTile);

    } else {
      System.out.println("Illegal Move. Please move your Bishop in accordance to the game rules");
    }
```

Move Bishop Validation
(1 of 3)

```
/* MOVING SOUTH EAST DIAGONALLY. Where the toColumn letter (as an ASCII value) is greater than
 * the fromColumn letter, and the fromRow number is greater than the toRow number. For example,
 * moving from 8C (where 8 is the row and C is the column) to 6E.
 */
} else if (((int)toColumn > (int)fromColumn) && (toRow < fromRow)) {
  System.out.println("----- Moving South East -----");

  /* Ensuring that the diagonal move is valid by checking that the Bishop
   * is moving the same number of spaces in both directions, therefore we
   * don't get a South East move which could be 2 spaces down and 1 space
   * right for example.
   */
  int colDifference = (int)toColumn - (int)fromColumn;
  int rowDifference = fromRow - toRow;
  int j = (int)fromColumn;

  if (colDifference == rowDifference) {
    for (int i = fromRow - 1; i > toRow; i--) {
      System.out.println("i is: " + i);
      ++j; // Also increments column to move East as part of moving South East diagonally.
      System.out.println("j is " + (char)j);
      if (board.map.getPieceOrOccupation("tileOccupation", i + String.valueOf((char)j))
          == "Occupied" && piece.isOpponentPiece(player, board.map.getPieceOrOccupation(
          "piecePos", i + String.valueOf((char)j))) == false) {
        System.out.println("Illegal Move. There is a piece in your Bishop's movement path "
            + "(Tile: " + i + (char)j + ")");
        pieceInPath++;
      }
    }

    setPos(player, selectedBishop, toTile);

  } else {
    System.out.println("Illegal Move. Please move your Bishop in accordance to the game rules");
  }
```

Move Bishop Validation
(2 of 3)

```
/* MOVING SOUTH WEST DIAGONALLY. Where the fromColumn letter (as an ASCII value) is greater than
 * the toColumn letter, and the fromRow number is greater than the toRow number. For example,
 * moving from 8D (where 8 is the row and D is the column) to 6B.
 */
} else if (((int)toColumn < (int)fromColumn) && (toRow < fromRow)) {
  System.out.println("----- Moving South West -----");

  /* Ensuring that the diagonal move is valid by checking that the Bishop
   * is moving the same number of spaces in both directions, therefore we
   * don't get a South West move which could be 2 spaces left and 1 space
   * down for example.
   *
   */
  int colDifference = (int)fromColumn - (int)toColumn;
  int rowDifference = fromRow - toRow;
  int j = (int)fromColumn;

  if (colDifference == rowDifference) {
    for (int i = fromRow - 1; i > toRow; i--) {
      --j; // Also decrements column to move West as part of moving diagonally South West.
      if (board.map.getPieceOrOccupation("tileOccupation", i + String.valueOf((char)j))
          == "Occupied" && piece.isOpponentPiece(player, board.map.getPieceOrOccupation(
          "piecePos", i + String.valueOf((char)j))) == false) {
        System.out.println("Illegal Move. There is a piece in your Bishop's movement path "
            + "(Tile: " + i + (char)j + ")");
        pieceInPath++;
      }
    }

    setPos(player, selectedBishop, toTile);

  } else {
    System.out.println("Illegal Move. Please move your Bishop in accordance to the game rules");
  }
```

```
/* MOVING NORTH WEST DIAGONALLY. Where the fromColumn letter (as an ASCII value) is greater than
 * the toColumn letter, and the toRow number is greater than the fromRow number. For example,
 * moving from 1D (where 1 is the row and D is the column) to 3B.
 */
} else if (((int)toColumn < (int)fromColumn) && (toRow > fromRow)) {
  System.out.println("----- Moving North West -----");

  /* Ensuring that the diagonal move is valid by checking that the Bishop
   * is moving the same number of spaces in both directions, therefore we
   * don't get a North West move which could be 2 spaces up and 1 space left,
   * for example.
   */
  int colDifference = (int)fromColumn - (int)toColumn;
  int rowDifference = toRow - fromRow;
  int j = (int)fromColumn;

  if (colDifference == rowDifference) {
    for (int i = fromRow + 1; i < toRow; i++) {
      --j; // Also decrements column to move West as part of moving diagonally North West.
      if (board.map.getPieceOrOccupation("tileOccupation", i + String.valueOf((char)j))
          == "Occupied" && piece.isOpponentPiece(player, board.map.getPieceOrOccupation(
          "piecePos", i + String.valueOf((char)j))) == false) {
        System.out.println("Illegal Move. There is a piece in your Bishop's movement path "
            + "(Tile: " + i + (char)j + ")");
        pieceInPath++;
      }
    }

    setPos(player, selectedBishop, toTile);

  } else {
    System.out.println("Illegal Move. Please move your Bishop in accordance to the game rules");
  }

} else {
  System.out.println("Illegal Move. Please move your Bishop in accordance to the game's rules");
}
```

Move Bishop Validation (3 of 3)

```
/**
 * This method is responsible for validating and then moving a Knight
 * chess piece, in accordance to the game's rules.
 *
 * @param player Takes the current player (Either "White" or "Black").
 * @param selectedKnight Takes the selected Knight piece.
 * @param toTile The destination tile that the Knight intends to move to.
 * @throws InvalidPlayerException If the player is not "White" or "Black".
 * @throws InvalidPieceException If an invalid piece is passed to the method.
 */
public void moveKnight(String player, String selectedKnight, String toTile)
    throws InvalidPieceException, InvalidPlayerException {

  String fromTile = board.map.getTile(selectedKnight);

  int fromRow = Integer.parseInt(String.valueOf(fromTile.charAt(0)));
  char fromColumn = fromTile.charAt(1);
  int toRow = Integer.parseInt(String.valueOf(toTile.charAt(0)));
  char toColumn = toTile.charAt(1);

  // MOVING UP 2 SPACES AND 1 SPACE LEFT
  if ((toRow == fromRow + 2) && (toColumn == fromColumn - 1)) {
    System.out.println("----- Moving 2 Spaces Up and 1 Space Left -----");
    setPos(player, selectedKnight, toTile);
```

Move Knight Validation (1 of 2)

Move Knight Validation (2 of 2)

```
// MOVING UP 2 SPACES AND 1 SPACE RIGHT
} else if ((toRow == fromRow + 2) && (toColumn == fromColumn + 1)) {
  System.out.println("----- Moving 2 Spaces Up and 1 Space Right -----");
  setPos(player, selectedKnight, toTile);

// MOVING RIGHT 2 SPACES AND 1 SPACE UP
} else if ((toColumn == fromColumn + 2) && (toRow == fromRow + 1)) {
  System.out.println("----- Moving 2 Spaces Right and 1 Space Up -----");
  setPos(player, selectedKnight, toTile);

// MOVING RIGHT 2 SPACES AND 1 SPACE DOWN
} else if ((toColumn == fromColumn + 2) && (toRow == fromRow - 1)) {
  System.out.println("----- Moving 2 Spaces Right and 1 Space Down -----");
  setPos(player, selectedKnight, toTile);

// MOVING DOWN 2 SPACES AND 1 SPACE LEFT
} else if ((toRow == fromRow - 2) && (toColumn == fromColumn - 1)) {
  System.out.println("----- Moving 2 Spaces Down and 1 Space Left -----");
  setPos(player, selectedKnight, toTile);

// MOVING DOWN 2 SPACES AND 1 SPACE RIGHT
} else if ((toRow == fromRow - 2) && (toColumn == fromColumn + 1)) {
  System.out.println("----- Moving 2 Spaces Down and 1 Space Right -----");
  setPos(player, selectedKnight, toTile);

// MOVING LEFT 2 SPACES AND 1 SPACE UP
} else if ((toColumn == fromColumn - 2) && (toRow == fromRow + 1)) {
  System.out.println("----- Moving 2 Spaces Left and 1 Space Up -----");
  setPos(player, selectedKnight, toTile);

// MOVING LEFT 2 SPACES AND 1 SPACE DOWN
} else if ((toColumn == fromColumn - 2) && (toRow == fromRow - 1)) {
  System.out.println("----- Moving 2 Spaces Left and 1 Space Down -----");
  setPos(player, selectedKnight, toTile);

/* If we have none of the above combinations which are valid moves for a Knight, print "Illegal
 * Move". */
} else {
  System.out.println("Illegal Move. Please move your Knight in accordance to the game's rules");
}
```

Move Queen Validation (1 of 5)

```
/**
 * This method is responsible for validating a move with a Queen piece.
 *
 * @param player The current player (Either "White" or "Black").
 * @param selectedPiece The selected queen.
 * @param toTile The destination tile the player intends the Queen to move to.
 * @throws InvalidPieceException If we receive an invalid piece (doesn't contain a type).
 * @throws InvalidPlayerException  If the player isn't "White" or "Black"
 */
public void moveQueen(String player, String selectedPiece, String toTile)
    throws InvalidPlayerException, InvalidPieceException {
  /* Queen can move any direction, any spaces */
  String fromTile = board.map.getTile(selectedPiece);

  int fromRow = Integer.parseInt(String.valueOf(fromTile.charAt(0)));
  char fromColumn = fromTile.charAt(1);
  int toRow = Integer.parseInt(String.valueOf(toTile.charAt(0)));
  char toColumn = toTile.charAt(1);
```

```
/* MOVING LEFT. Where the row number remains the same, but the fromColumn is greater than the
 * toColumn. For example, moving from 8H (where H in ASCII is 72) to 8A (where A in ASCII is
 * 65).
 */
if (fromRow == toRow && (int)fromColumn > (int)toColumn) {
  /* CHECKING FOR ANY PIECES IN QUEEN'S MOVEMENT PATH (EXCLUDING DESTINATION TILE)
  I search each tile to the left of the current one to see if it contains any pieces, until
  reaching one tile before the Queen's destination tile */
  System.out.println("----- Moving Left  -----");
  for (int i = (int)fromColumn - 1; i > (int)toColumn; i--) {
    // If a tile in the Queen's path is not null (occupied by a piece), move is invalid.
    System.out.println("The tile is: " + toRow + String.valueOf((char)i));

    if (board.map.getPieceOrOccupation("tileOccupation", toRow + String.valueOf((char)i))
        == "Occupied" && piece.isOpponentPiece(player, board.map.getPieceOrOccupation(
        "piecePos", toRow + String.valueOf((char)i))) == false) {
      System.out.println("Illegal Move. There is a piece in your Queen's movement path "
          + "(Tile: " + toRow + String.valueOf((char)i) + ")");
      pieceInPath++;
    }
  }
  setPos(player, selectedPiece, toTile);

/*
 * MOVING RIGHT. Where the row number remains the same, but the toColumn number is greater than
 * thefromColumn number. For example, moving from 3B (where B in ASCII is 66) to 3E (where E in
 * ASCII is 69).
 */
} else if (fromRow == toRow && (int)toColumn > (int)fromColumn) {
  /* CHECKING FOR ANY PIECES IN THE QUEEN'S MOVEMENT PATH (EXCLUDING DESTINATION TILE)
  I search each tile to the right of the current tile to see if it contains any pieces (player
  or opponent), until reaching one tile before the Queen's destination tile */
  System.out.println("----- Moving Right -----");
  for (int i = (int)fromColumn + 1; i < (int)toColumn; i++) {
    // If a tile in the Queen's path is not null (occupied by a piece, move is invalid.
    System.out.println("The tile is: " + toRow + String.valueOf((char)i));
    if (board.map.getPieceOrOccupation("tileOccupation", toRow + String.valueOf((char)i))
        == "Occupied" && piece.isOpponentPiece(player, board.map.getPieceOrOccupation(
        "piecePos", toRow + String.valueOf((char)i))) == false) {
      System.out.println("Illegal Move. There is a piece in your Queen's movement path "
          + "(Tile: " + toRow + String.valueOf((char)i) + ")");
      pieceInPath++;
    }
  }
  setPos(player, selectedPiece, toTile);
```

Move Queen Validation (2 of 5)

```
  /*
   * MOVING UP. Where the column letter (as an ASCII value) remains the same, but the fromRow
   * number is less than the toRow number. For example, moving from 1A (where 1 is the row) to
   * 3A.
   */
  } else if ((int)fromColumn == (int)toColumn && (fromRow < toRow)) {
    /* CHECKING FOR ANY PIECES IN THE QUEEN'S MOVEMENT PATH (EXCLUDING DESTINATION TILE)
    I search each tile above the current tile to see if it contains any pieces (player or
    opponent), until reaching one tile before the Queen's destination tile. */
    System.out.println("----- Moving Up -----");
    for (int i = fromRow + 1; i < toRow; i++) {
      System.out.println("The tile is: " + i + fromColumn);
      if (board.map.getPieceOrOccupation("tileOccupation", "" + i + fromColumn) == "Occupied"
          && piece.isOpponentPiece(player, board.map.getPieceOrOccupation("piecePos", "" + i
          + fromColumn)) == false) {
        System.out.println("Illegal Move. There is a piece in your Queen's movement path "
            + "(Tile: " + i + fromColumn + ")");
        pieceInPath++;
      }
    }
    setPos(player, selectedPiece, toTile);
```

```
/*
 * MOVING DOWN. Where the column letter (as an ASCII value) remains the same, but the fromRow
 * number is greater than the toRow number. For example, moving from 8A (where 8 is the row) to
 * 4A.
 */
} else if ((int)fromColumn == (int)toColumn && (fromRow > toRow)) {
  /* CHECKING FOR ANY PIECES IN THE QUEEN'S MOVEMENT PATH (EXCLUDING DESTINATION TILE)
  I search each tile below the current tile to see if it contains any pieces (player or
  opponent), until reaching one tile before the Queen's destination tile. */
  System.out.println("----- Moving Down -----");
  for (int i = fromRow - 1; i > toRow; i--) {
    System.out.println("The tile is: " + i + fromColumn);
    if (board.map.getPieceOrOccupation("tileOccupation", "" + i + fromColumn) == "Occupied"
        && piece.isOpponentPiece(player, board.map.getPieceOrOccupation("piecePos", "" + i
        + fromColumn)) == false) {
      System.out.println("Illegal Move. There is a piece in your Queen's movement path "
          + "(Tile: " + i + fromColumn + ")");
      pieceInPath++;
    }
  }
  setPos(player, selectedPiece, toTile);

/* MOVING NORTH EAST DIAGONALLY. Where the toColumn letter (as an ASCII value) is greater than
 * the fromColumn letter, and the toRow number is greater than the fromRow number. For example,
 * moving from 1D (where 1 is the row and D is the column) to 3F.
 */
} else if (((int)toColumn > (int)fromColumn) && (toRow > fromRow)) {
  System.out.println("----- Moving North East -----");

  /* Ensuring that the diagonal move is valid by checking that the Queen
   * is moving the same number of spaces in both directions, therefore we
   * don't get a North East move which could be 2 spaces up and 1 space
   * right for example.
   */
  int colDifference = (int)toColumn - (int)fromColumn;
  int rowDifference = toRow - fromRow;
  int j = (int)fromColumn;

  if (colDifference == rowDifference) {

    for (int i = fromRow + 1; i < toRow; i++) {
      ++j; // Also increments column to move East as part of moving diagonally North East
      if (board.map.getPieceOrOccupation("tileOccupation", i + String.valueOf((char)j))
          == "Occupied" && piece.isOpponentPiece(player, board.map.getPieceOrOccupation(
          "piecePos", i + String.valueOf((char)j))) == false) {
        System.out.println("Illegal Move. There is a piece in your Queen's movement path "
            + "(Tile: " + i + (char)j + ")");
        pieceInPath++;
      }
    }
    setPos(player, selectedPiece, toTile);
  } else {
    System.out.println("Illegal Move. Please move your Queen in accordance to the game rules");
  }
```

Move Queen Validation (3 of 5)

```
/* MOVING SOUTH EAST DIAGONALLY. Where the toColumn letter (as an ASCII value) is greater than
 * the fromColumn letter, and the fromRow number is greater than the toRow number. For example,
 * moving from 8D (where 8 is the row and D is the column) to 6F.
 */
} else if (((int)toColumn > (int)fromColumn) && (toRow < fromRow)) {
  System.out.println("----- Moving South East -----");

  /* Ensuring that the diagonal move is valid by checking that the Queen
   * is moving the same number of spaces in both directions, therefore we
   * don't get a South East move which could be 2 spaces down and 1 space
   * right for example.
   */
  int colDifference = (int)toColumn - (int)fromColumn;
  int rowDifference = fromRow - toRow;
  int j = (int)fromColumn;

  if (colDifference == rowDifference) {
    for (int i = fromRow - 1; i > toRow; i--) {
      ++j; // Also increments column to move East as part of moving diagonally South East
      if (board.map.getPieceOrOccupation("tileOccupation", i + String.valueOf((char)j))
          == "Occupied" && piece.isOpponentPiece(player, board.map.getPieceOrOccupation(
          "piecePos", i + String.valueOf((char)j))) == false) {
        System.out.println("Illegal Move. There is a piece in your Queen's movement path "
            + "(Tile: " + i + (char)j + ")");
        pieceInPath++;
      }
    }
    setPos(player, selectedPiece, toTile);
  } else {
    System.out.println("Illegal Move. Please move your Queen in accordance to the game rules");
  }
```

Move Queen Validation (4 of 5)

```
/* MOVING SOUTH WEST DIAGONALLY. Where the fromColumn letter (as an ASCII value) is greater than
 * the toColumn letter, and the fromRow number is greater than the toRow number. For example,
 * moving from 8D (where 8 is the row and D is the column) to 6B.
 */
} else if (((int)toColumn < (int)fromColumn) && (toRow < fromRow)) {
  System.out.println("----- Moving South West -----");

  /* Ensuring that the diagonal move is valid by checking that the Queen
   * is moving the same number of spaces in both directions, therefore we
   * don't get a South West move which could be 2 spaces left and 1 space
   * down for example.
   *
   */
  int colDifference = (int)fromColumn - (int)toColumn;
  int rowDifference = fromRow - toRow;
  int j = (int)fromColumn;

  if (colDifference == rowDifference) {
    for (int i = fromRow - 1; i > toRow; i--) {
      --j; // Also decrements column to move West as part of moving diagonally South West.
      if (board.map.getPieceOrOccupation("tileOccupation", i + String.valueOf((char)j))
          == "Occupied" && piece.isOpponentPiece(player, board.map.getPieceOrOccupation(
          "piecePos", i + String.valueOf((char)j))) == false) {
        System.out.println("Illegal Move. There is a piece in your Queen's movement path "
            + "(Tile: " + i + (char)j + ")");
        pieceInPath++;
      }
    }
    setPos(player, selectedPiece, toTile);
  } else {
    System.out.println("Illegal Move. Please move your Queen in accordance to the game rules");
  }
```

```
  /* MOVING NORTH WEST DIAGONALLY. Where the fromColumn letter (as an ASCII value) is greater than
   * the toColumn letter, and the toRow number is greater than the fromRow number. For example,
   * moving from 1D (where 1 is the row and D is the column) to 3B.
   */
  } else if (((int)toColumn < (int)fromColumn) && (toRow > fromRow)) {
    System.out.println("----- Moving North West -----");

    /* Ensuring that the diagonal move is valid by checking that the Queen
     * is moving the same number of spaces in both directions, therefore we
     * don't get a North West move which could be 2 spaces up and 1 space left,
     * for example.
     */
    int colDifference = (int)fromColumn - (int)toColumn;
    int rowDifference = toRow - fromRow;
    int j = (int)fromColumn;

    if (colDifference == rowDifference) {
      for (int i = fromRow + 1; i < toRow; i++) {
        --j; // Also decrements column to move West as part of moving diagonally North West.
        if (board.map.getPieceOrOccupation("tileOccupation", i + String.valueOf((char)j))
            == "Occupied" && piece.isOpponentPiece(player, board.map.getPieceOrOccupation(
            "piecePos", i + String.valueOf((char)j))) == false) {
          System.out.println("Illegal Move. There is a piece in your Queen's movement path "
              + "(Tile: " + i + (char)j + ")");
          pieceInPath++;
        }
      }
      setPos(player, selectedPiece, toTile);
    } else {
      System.out.println("Illegal Move. Please move your Queen in accordance to the game rules");
    }

  } else {
    System.out.println("Illegal Move. Please move your Queen in accordance to the game's rules");
  }
}
```

Move Queen Validation (5 of 5)

```
/**
 * This method is responsible for validating and then moving a King piece.
 *
 * @param player Takes the current player (either "White" or "Black").
 * @param selectedPiece Takes the selected King piece.
 * @param toTile Takes the destination tile the piece intends to move to.
 * @throws InvalidPlayerException If the player passed is not "White" or "Black".
 * @throws InvalidPieceException If we have an invalid piece.
 */
public void moveKing(String player, String selectedPiece, String toTile)
    throws InvalidPieceException, InvalidPlayerException {
  String fromTile = board.map.getTile(selectedPiece);

  int fromRow = Character.getNumericValue(fromTile.charAt(0));
  int fromColumn = Character.getNumericValue(fromTile.charAt(1));
  int toRow = Character.getNumericValue(toTile.charAt(0));
  int toColumn = Character.getNumericValue(toTile.charAt(1));
  // 1A, 1 is row, A is column

  // MOVING LEFT
  if (fromRow == toRow && (toColumn == fromColumn - 1)) {
    System.out.println("Lands in Left");
    setPos(player, selectedPiece, toTile);
```

Move King Validation (1 of 2)

```
// MOVING RIGHT
} else if (fromRow == toRow && (toColumn == fromColumn + 1)) {
  System.out.println("Lands in Right");
  setPos(player, selectedPiece, toTile);

// MOVING UP
} else if (fromColumn == toColumn && (toRow == fromRow + 1)) {
  System.out.println("Lands in Up");
  setPos(player, selectedPiece, toTile);

// MOVING DOWN
} else if (fromColumn == toColumn && (toRow == fromRow - 1)) {
  System.out.println("Lands in Down");
  setPos(player, selectedPiece, toTile);

// MOVING NORTH EAST DIAGONALLY
} else if ((toColumn == fromColumn + 1) && (toRow == fromRow + 1)) {
  System.out.println("Lands in North East");
  setPos(player, selectedPiece, toTile);

// MOVING SOUTH EAST DIAGONALLY
} else if ((toColumn == fromColumn + 1) && (toRow == fromRow - 1)) {
  System.out.println("Lands in South East");
  setPos(player, selectedPiece, toTile);

// MOVING SOUTH WEST DIAGONALLY
} else if ((toColumn == fromColumn - 1) && (toRow == fromRow - 1)) {
  System.out.println("Lands in South West");
  setPos(player, selectedPiece, toTile);

// MOVING NORTH WEST DIAGONALLY
} else if ((toColumn == fromColumn - 1) && (toRow == fromRow + 1)) {
  System.out.println("Lands in North West");
  setPos(player, selectedPiece, toTile);

} else {
  System.out.println("Illegal Move. Please move your King in accordance to the game's rules");
}
```

Move King Validation (2 of 2)

Setting Positions

The final main component for each of the six piece types is the set position method. This set position method is called after validation is completed in the piece's corresponding 'move' method.

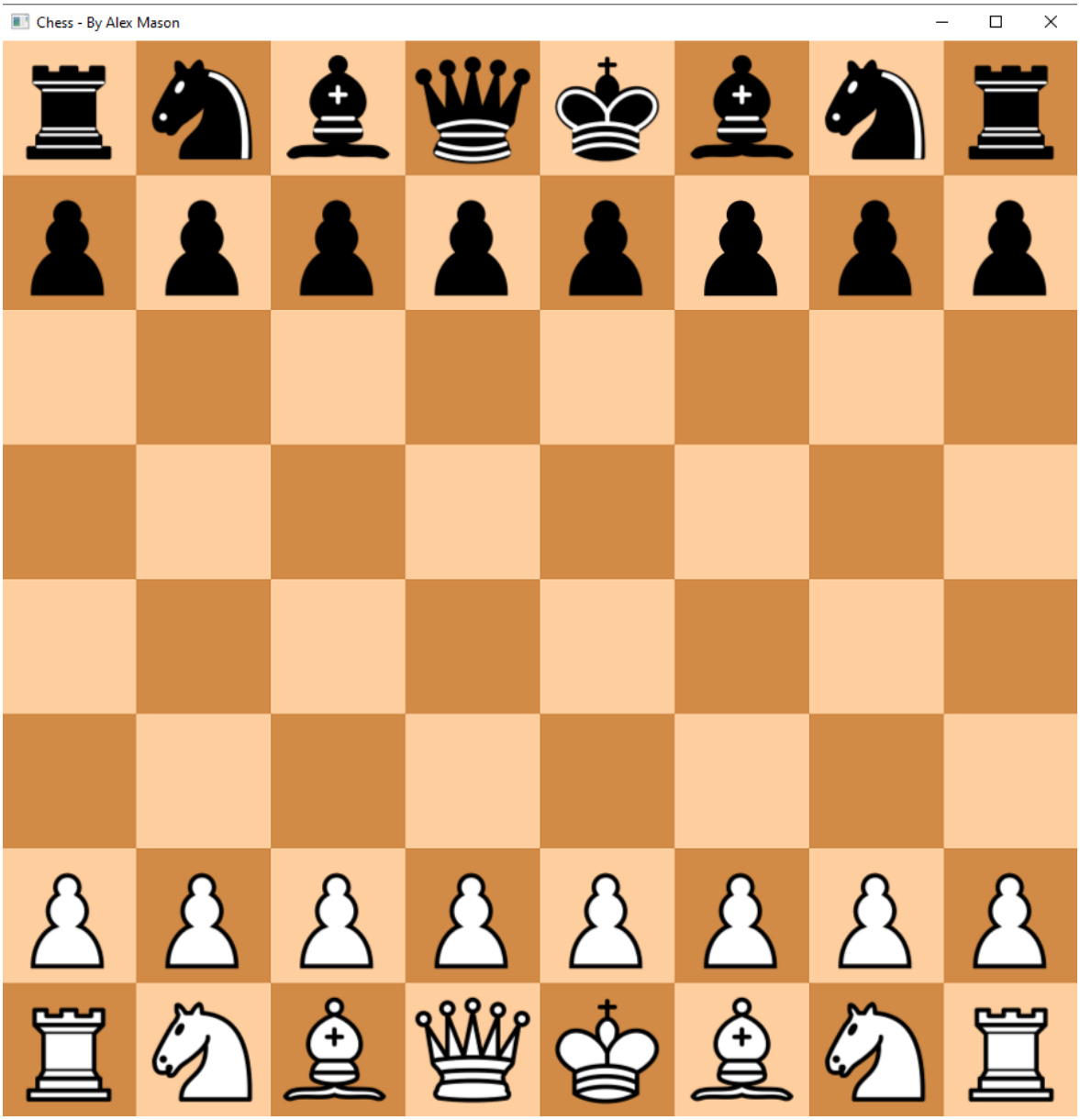
The method performs a final validation check to see what piece is in the destination tile that the player is trying to move their piece into. This is to eliminate the chance of a player trying to move a piece into a tile which is already occupied with another one of their pieces. It also means that if the player is moving their piece into a tile containing an opponent piece, then the player can capture this piece and move there.

Once this check is completed, the position of the piece the player is trying to move is actually set in the TreeMap data structures I have, thus setting the piece's new position on the Chess board.

```
/**
 *  This method is responsible for setting the final position of the Queen to the destination tile
 *  after validation has taken place in the moveQueen method. The setPos method also checks for
 *  the piece in the destination tile, so that if there's an opponent piece in the destination
 *  tile, the Queen can capture it. If there is the player's own piece in the destination tile,
 *  the method will indicate this to the player and not move the Queen.
 *
 * @param player Takes the current player (either "White" or "Black") to validate if the piece
 *       during an capturing move is an opponent piece.
 * @param selectedQueen Takes the selected Queen piece to set it to its destination tile.
 * @param toTile The tile the selected Queen intends to move to (destination tile).
 * @throws InvalidPlayerException if the player passed to the method is not "White" or "Black".
 * @throws InvalidPieceException if an invalid piece is passed to the method.
 */
public void setPos(String player, String selectedQueen, String toTile) throws
    InvalidPieceException, InvalidPlayerException {
  System.out.println("Lands in setPos");
  /*
   * This 'if' statement is checking there are zero pieces in the Queen's movement path,
   * and the destination tile is empty, so that the Queen can move there.
   */
  if (pieceInPath == 0 && board.map.getPieceOrOccupation("tileOccupation", toTile) == "Empty") {
    board.map.setValue("piecePos", toTile, selectedQueen);

  /*
   * This 'else if' is for checking there are zero pieces in the Queen's movement path, and
   * the destination tile contains an opponent piece which the Queen can capture.
   */
  } else if (pieceInPath == 0 && piece.isOpponentPiece(player, board.map.getPieceOrOccupation(
      "piecePos", toTile)) == true) {
    board.map.capturePiece(board.map.getPieceOrOccupation("piecePos", toTile));
    board.map.setValue("piecePos", toTile, selectedQueen);

  /*
   * This 'else if' assumes the current player's piece is in the destination tile, as the tile is
   * occupied but not with an opponent piece)
   */
  } else if (piece.isOpponentPiece(player, selectedQueen) == false
      && board.map.getPieceOrOccupation("tileOccupation", toTile) == "Occupied") {
    System.out.println("Illegal Move. You cannot move your Queen to a tile containing your own "
        + "piece");
  }

  pieceInPath = 0; // Reset counter to avoid potential conflicts later

  /* If none of the above conditions are met, do nothing. No error message is needed since this is
  dealt with in the 'moveQueen' method */
}
```

GUI

This is the GUI I have created for my game. These were sourced from Wikimedia Commons, which means they are under a commons license and are copyright free. The GUI has the modern approach/feel I was intending on going for. I feel the Black and White player pieces contrast well with the dark orange and light orange tiles, with the intention that the colours in the interface are easy on the eye for the player/user.



Controller Class

Below is some of the code from the Controller class. This links the GUI to the code I have written for my Chess engine. The full source code can be viewed in my GitHub repository.

```
/**
 * This method checks to see what piece and tile is clicked based on X and Y co-ords.
 *
 * @param event The current MouseEvent event handler.
 * @throws InvalidPieceException If the piece is not a valid piece type.
 * @throws InvalidPlayerException If the player is not "White" or "Black".
 */

@FXML
public void pieceClicked(@SuppressWarnings("exports") MouseEvent event)
     throws InvalidPieceException, InvalidPlayerException {
  String id = event.getPickResult().getIntersectedNode().getId();
  //System.out.print("Selected ");

  if (id == null) {
    String tile = findTile(event.getX(), event.getY());
    String piece = Board.board.map.getPieceOrOccupation("piecePos", tile);

    if (piece == null || piece.equals("null")) {
      movementPath.add(tile);
      //System.out.print(tile);

      if (movementPath.size() == 2) {
        System.out.println("to move to " + tile);
        validate(movementPath);
      }
    } else if (!piece.equals("null")) {
      movementPath.add(piece);

      if (movementPath.size() == 2) {
        System.out.print("to capture " + piece);
        validate(movementPath);
      } else {
        System.out.println("-------------------------------------------------");
        System.out.print("Selected " + piece + ", ");
      }
    } else {
      movementPath.add(tile);
    }

  } else if (id.contains("white") || id.contains("black")) {
```

Controller Class (1 of 6)

Controller Class (2 of 6)

```
/**
 * This method is responsible for taking two co-ordinates and
 * matching them to a tile on the Chessboard GUI, based on what range
 * these co-ordinates fall into.
 * @param x A co-ordinate from mouse click event.
 * @param y A co-ordinate from mouse click event.
 * @return tile corresponding to co-ordinates.
 */
public static String findTile(double x, double y) {
  if (x >= 0 && x <= 112.5 && y >= 0 && y <= 112.5) {
    return "8A";
  } else if (x >= 112.5 && x <= 225 && y >= 0 && y <= 112.5) {
    return "8B";
  } else if (x >= 225 && x <= 337.5 && y >= 0 && y <= 112.5) {
    return "8C";
  } else if (x >= 337.5 && x <= 450 && y >= 0 && y <= 112.5) {
    return "8D";
  } else if (x >= 450 && x <= 562.5 && y >= 0 && y <= 112.5) {
    return "8E";
  } else if (x >= 562.5 && x <= 675 && y >= 0 && y <= 112.5) {
    return "8F";
  } else if (x >= 675 && x <= 787.5 && y >= 0 && y <= 112.5) {
    return "8G";
  } else if (x >= 787.5 && x <= 900 && y >= 0 && y <= 112.5) {
    return "8H";
  } else if (x >= 0 && x <= 112.5 && y >= 112.5 && y <= 225) {
    return "7A";
  } else if (x >= 112.5 && x <= 225 && y >= 112.5 && y <= 225) {
    return "7B";
  } else if (x >= 225 && x <= 337.5 && y >= 112.5 && y <= 225) {
    return "7C";
  } else if (x >= 337.5 && x <= 450 && y >= 112.5 && y <= 225) {
    return "7D";
  } else if (x >= 450 && x <= 562.5 && y >= 112.5 && y <= 225) {
    return "7E";
  } else if (x >= 562.5 && x <= 675 && y >= 112.5 && y <= 225) {
    return "7F";
  } else if (x >= 675 && x <= 787.5 && y >= 112.5 && y <= 225) {
    return "7G";
  } else if (x >= 787.5 && x <= 900 && y >= 112.5 && y <= 225) {
    return "7H";
  } else if (x >= 0 && x <= 112.5 && y >= 225 && y <= 337.5) {
    return "6A";
  } else if (x >= 112.5 && x <= 225 && y >= 225 && y <= 337.5) {
    return "6B";
  } else if (x >= 225 && x <= 337.5 && y >= 225 && y <= 337.5) {
    return "6C";
  } else if (x >= 337.5 && x <= 450 && y >= 225 && y <= 337.5) {
    return "6D";
  } else if (x >= 450 && x <= 562.5 && y >= 225 && y <= 337.5) {
    return "6E";
```

```java
/**
 * This method validates a pair of moves, whether this a piece to piece, piece to tile,
 * or tile to tile move, depending on where the user clicked on the board.
 * @param path The path/route the user is intending to take.
 * @throws InvalidPlayerException If the player is not "White" or "Black".
 * @throws InvalidPieceException If an invalid piece type is received.
 */
public static void validate(ArrayList<String> path) throws InvalidPieceException,
    InvalidPlayerException {
  if (path.size() == 2) {
    String idOrTile1 = path.get(0);
    String idOrTile2 = path.get(1);
    // Checks for White player piece moving to Black player piece
    if (idOrTile1.contains("White") && idOrTile2.contains("Black")) {
      //System.out.println("Detected White player move to Black player piece");
      Controller.guiPieceToClass(idOrTile1, idOrTile2);
      movementPath.clear();

    // Checks for Black player piece moving to White player piece
    } else if (idOrTile1.contains("Black") && idOrTile2.contains("White")) {
      //System.out.println("Detected Black player move to White player piece");
      Controller.guiPieceToClass(idOrTile1, idOrTile2);
      movementPath.clear();

    // Checks for White player move to another tile
    } else if (idOrTile1.contains("White") && idOrTile2.length() == 2) {
      //System.out.println("Detected White player move to another tile");
      Controller.guiPieceToClass(idOrTile1, idOrTile2);
      movementPath.clear();

    // Checks for Black player move to another tile
    } else if (idOrTile1.contains("Black") && idOrTile2.length() == 2) {
      //System.out.println("Detected Black player move to another tile");
      Controller.guiPieceToClass(idOrTile1, idOrTile2);
      movementPath.clear();

    /*
     * ERROR CHECKING/PREVENTION STARTS HERE
     */
    } else if (idOrTile1.contains("Black") && idOrTile2.contains("Black")) {
      System.out.println("You cannot move your own piece to another tile containing your own "
          + "piece");
      movementPath.clear();

    } else if (idOrTile1.contains("White") && idOrTile2.contains("White")) {
      System.out.println("You cannot move your own piece to another tile containing your own "
          + "piece");
      movementPath.clear();

    } else if (idOrTile1.length() == 2 && idOrTile2.length() == 2) {
      System.out.println("Please select a piece and a destination tile. If you did select a"
          + " piece, please ensure you click directly on the piece");
      movementPath.clear();
```

Controller Class (3 of 6)

```java
/**
 * After taking a piece in the GUI and moving it (after validation in the corresponding piece
 * class), the method below updates the image to the new tile the piece has moved to.
 * @param img Takes the piece as an ImageView object to set new X and Y positions.
 * @param tile Takes the tile to specify the X and Y positions the image should move to.
 */
public static void setImagePos(@SuppressWarnings("exports") ImageView img, String tile) {

  if (tile.equals("8A")) {
    img.setLayoutX(0.25);
    img.setLayoutY(0.25);
  } else if (tile.equals("8B")) {
    img.setLayoutX(112.75);
    img.setLayoutY(0.25);
  } else if (tile.equals("8C")) {
    img.setLayoutX(225.25);
    img.setLayoutY(0.25);
  } else if (tile.equals("8D")) {
    img.setLayoutX(337.75);
    img.setLayoutY(0.25);
  } else if (tile.equals("8E")) {
    img.setLayoutX(450.25);
    img.setLayoutY(0.25);
  } else if (tile.equals("8F")) {
    img.setLayoutX(562.75);
    img.setLayoutY(0.25);
  } else if (tile.equals("8G")) {
    img.setLayoutX(675.25);
    img.setLayoutY(0.25);
  } else if (tile.equals("8H")) {
    img.setLayoutX(787.75);
    img.setLayoutY(0.25);
  } else if (tile.equals("7A")) {
    img.setLayoutX(0.25);
    img.setLayoutY(112.75);
  } else if (tile.equals("7B")) {
    img.setLayoutX(112.75);
    img.setLayoutY(112.75);
  } else if (tile.equals("7C")) {
    img.setLayoutX(225.25);
    img.setLayoutY(112.75);
  } else if (tile.equals("7D")) {
    img.setLayoutX(337.75);
    img.setLayoutY(112.75);
  } else if (tile.equals("7E")) {
    img.setLayoutX(450.25);
    img.setLayoutY(112.75);
  } else if (tile.equals("7F")) {
    img.setLayoutX(562.75);
    img.setLayoutY(112.75);
  } else if (tile.equals("7G")) {
    img.setLayoutX(675.25);
    img.setLayoutY(112.75);
```

Controller Class (4 of 6)

Controller Class (5 of 6)

```
/**
 * This method is responsible for taking pieces/tiles from the GUI and linking
 * them to the rest of the game logic, such as the corresponding piece validation classes.
 * @param piece Takes a piece ID.
 * @param value Takes either a second piece ID or tile co-ord for the first piece to move to.
 * @throws InvalidPlayerException If the player in the piece ID is not White or Black.
 * @throws InvalidPieceException If the piece ID does not contain a valid piece type.
 */
public static void guiPieceToClass(String piece, String value)
    throws InvalidPieceException, InvalidPlayerException {

  Board.board.map.valueLock = false;
  String player = "";
  String oppPlayer = "";
  String toTile = "";
  String kingPos = "";
  boolean tile = false;
  ImageView img = (ImageView) Main.scene.lookup(idToLower(piece));
  ImageView img2 = new ImageView();

  if (piece.contains("White")) {
    player = "White";
    oppPlayer = "Black";
  } else if (piece.contains("Black")) {
    player = "Black";
    oppPlayer = "White";
  }

  // If true, the second value is a tile co-ord. If false, it is a piece.
  if (value.length() == 2) {
    tile = true;
    toTile = value;
  } else {
    tile = false;
    toTile = Board.board.map.getTile(value);
    img2 = (ImageView) Main.scene.lookup(idToLower(value));
  }

  kingPos = Board.board.map.getTile(oppPlayer + "King");
  LegalMoves.legalMoves(player);

  if (piece.contains("Pawn")) {
    if (tile) {
      //pawn.movePawn(player, piece, value);

      if (LegalMoves.legalMoves.contains(value + ", " + piece)) {
        if (LegalMoves.stalemate) {
          System.out.println("Stalemate Detected. It's a Draw!");
        } else if (toTile.equals(kingPos)) {
          System.out.println("Checkmate Detected " + player + " Wins!");
        } else {
          pawn.movePawn(player, piece, value);
          setImagePos(img, value);
```

```java
/**
 * The purpose of this method is to alter a piece ID so that the beginning
 * letter is capitalised so it is uniform/standard with ID validation throughout
 * the rest of the program.
 * @param id Takes a piece ID.
 * @return Returns a string with the beginning letter of the ID capitalised.
 */
public String idToUpper(String id) {
  char firstLetter = id.charAt(0);
  char upperCase = Character.toUpperCase(firstLetter);

  String alteredID = "";

  for (int i = 1; i < id.length(); i++) {
    alteredID += id.charAt(i);
  }

  return "" + upperCase + alteredID;
}

/**
 * The purpose of this method is to alter a piece ID so that the beginning
 * letter is lower-case so that it can be used as an FX ID to reference
 * (update position or delete) the piece on the GUI view.
 * @param id Takes a piece ID.
 * @return Returns a string with the beginning letter of the ID in lower case.
 */
public static String idToLower(String id) {
  char firstLetter = id.charAt(0);
  char lowerCase = Character.toLowerCase(firstLetter);
  String convertToID = "#" + lowerCase;

  for (int i = 1; i < id.length(); i++) {
    convertToID += id.charAt(i);
  }

  //System.out.println("Convert to ID is: " + convertToID);
  return convertToID;
}

public static void main(String[] args) throws InvalidPlayerException, InvalidPieceException {
  new Controller();

}
```

Controller Class (6 of 6)

Legal Moves Class

Below is some of the code from my Legal Moves class. This scans through all possible moves the player can make in their turn and based on the current game state, and then returns a list of the moves that the player can legally make. This would be moves that do not put the player's own King in danger, moves that would normally be safe, and moves that allow a player's piece to capture or block an opponent player's piece that is threatening the capture of the player's king.

```java
/**
 * This method is responsible for adding all legal moves to an ArrayList.
 * The algorithm can then choose which of those moves to take based on values
 * to specify which have a higher payoff than others.
 *
 * @param player Takes the current player (either "White" or "Black").
 * @return An ArrayList of legalMoves intended to be used by the AI algorithm.
 * @throws InvalidPieceException If an invalid piece is received (not matching a piece type).
 * @throws InvalidPlayerException If an invalid player is received (not "White" or "Black").
 */
public static ArrayList<String> legalMoves(String player) throws InvalidPieceException,
      InvalidPlayerException {
  legalMoves = new ArrayList<String>();
  illegalMoves = new ArrayList<String>();
  playerPieces = new ArrayList<String>();
  oppPlayerPieces = new ArrayList<String>();
  whiteReachKing = new ArrayList<String>();
  blackReachKing = new ArrayList<String>();
  whiteReachKingTile = new ArrayList<String>();
  blackReachKingTile = new ArrayList<String>();
  threatenPieceTiles = new ArrayList<String>();
  String oppPlayer = "";
  String toTile = "";
  String prevTile = "";
  Board.board.map.valueLock = true;

  if (player.equals("White")) {
    oppPlayer = "Black";
  } else if (player.equals("Black")) {
    oppPlayer = "White";
  }

  // Creating a list of opponent player's piece
  for (Map.Entry<String, String> entry : Board.board.map.piecePos.entrySet()) {
    String piece = entry.getValue();

    if (piece.contains(player)) {
      playerPieces.add(piece);
    } else {
      oppPlayerPieces.add(piece);
    }
  }

  // Iterate Rows
  for (int i = 1; i <= 8; i++) {
    // Iterate Columns
    for (int j = 65; j <= 72; j++) {
      for (int k = 0; k < playerPieces.size(); k++) {
        String piece = playerPieces.get(k);
        String kingPos = Board.board.map.getTile(player + "King");
        //System.out.println("---- Value is: " + piece + " ----");
        toTile = "" + i + (char)j;
```

Legal Moves Class (1 of 4)

Legal Moves Class (2 of 4)

```
/**
 * Checks to see if an opponent player's piece can reach the player's King or not.
 *
 * @param oppPlayer Takes the opposite player (either "White" or "Black").
 * @return true if oppPlayer can reach the player's king
 * @throws InvalidPlayerException If an invalid player is received (not "White" or "Black").
 * @throws InvalidPieceException If an invalid piece is received (not matching a piece type).
 */
public static boolean reachKing(String oppPlayer, String toTile) throws
      InvalidPieceException, InvalidPlayerException {

  for (int i = 0; i < oppPlayerPieces.size(); i++) {
    String piece = oppPlayerPieces.get(i);
    Board.board.map.valueLock = true;

    if (piece.contains(oppPlayer)) {
      if (piece.contains("Pawn")) {
        Controller.pawn.passedValidation = false;
        /*System.out.println("Opp Player is: " + oppPlayer + ", piece is: " + piece
            + ", to tile is: " + toTile);*/
        Controller.pawn.movePawn(oppPlayer, piece, toTile);

        if (Controller.pawn.passedValidation && Controller.pawn.attackMove) {
          //System.out.println("Lands Pawn Reach King");
          //System.out.println(oppPlayer + " is moving " + piece + " to " + toTile);

          if (oppPlayer.equals("White")) {
            whiteReachKing.add(piece);
            whiteReachKingTile.add(toTile);
            blockThreat("Black", piece, toTile);
          } else {
            blackReachKing.add(piece);
            blackReachKingTile.add(toTile);
            blockThreat("White", piece, toTile);
          }

          return true;
        }

        Controller.pawn.passedValidation = false;
        Controller.pawn.attackMove = false;

      } else if (piece.contains("Rook")) {
        Controller.rook.passedValidation = false;
        Controller.rook.moveRook(oppPlayer, piece, toTile);

        if (Controller.rook.passedValidation) {
          //System.out.println("Lands Rook Reach King");
          //System.out.println(oppPlayer + " is moving " + piece + " to " + toTile);

          if (oppPlayer.equals("White")) {
            whiteReachKing.add(piece);
            whiteReachKingTile.add(toTile);
```

```java
/**
 * This method is responsible for checking if a player's piece can be used to
 * block an opponent player's piece which is threatening the capture of their King.
 * @param player Takes the current player (either "White" or "Black").
 * @param piece Takes an ArrayList of threatening pieces.
 * @param tile Takes an ArrayList of tiles the threatening pieces can move to.
 */
public static void blockThreat(String player, String piece, String tile)
    throws InvalidPieceException, InvalidPlayerException {

  Board.board.map.valueLock = true;

  //threatenPieceTiles = new ArrayList<String>();
  String fromTile = Board.board.map.getTile(piece);
  String kingPos = Board.board.map.getTile(player + "King");
  int fromRow = Integer.parseInt(String.valueOf(fromTile.charAt(0)));
  char fromColumn = fromTile.charAt(1);
  int toRow = Integer.parseInt(String.valueOf(kingPos.charAt(0)));
  char toColumn = kingPos.charAt(1);

  // Left
  if (fromRow == toRow && fromColumn > toColumn) {
    for (int i = fromColumn - 1; i > toColumn; i--) {
      threatenPieceTiles.add("" + toRow + String.valueOf((char)i));
    }

  // Right
  } else if (fromRow == toRow && toColumn > fromColumn) {
    for (int i = fromColumn + 1; i < toColumn; i++) {
      threatenPieceTiles.add("" + toRow + String.valueOf((char)i));
    }

  // Up
  } else if (fromColumn == toColumn && (fromRow < toRow)) {
    for (int i = fromRow + 1; i < toRow; i++) {
      threatenPieceTiles.add("" + i + fromColumn);
    }

  // Down
  } else if (fromColumn == toColumn && (fromRow > toRow)) {
    for (int i = fromRow - 1; i > toRow; i--) {
      threatenPieceTiles.add("" + i + fromColumn);
    }

  // North East
  } else if ((toColumn > fromColumn) && (toRow > fromRow)) {
    int colDifference = toColumn - fromColumn;
    int rowDifference = toRow - fromRow;
    int j = fromColumn;

    if (colDifference == rowDifference) {

      for (int i = fromRow + 1; i < toRow; i++) {
```

Legal Moves Class (3 of 4)

Legal Moves Class (4 of 4)

```
/**
 * This method is responsible for checking if an opponent's piece which can reach
 * the player's King can be captured by another one of the player's pieces.
 * @param player Takes the current player (either "White" or "Black").
 * @param piece Takes the piece to check.
 * @throws InvalidPieceException If an invalid piece is passed (not a piece type).
 * @throws InvalidPlayerException If an invalid player is passed (not "White" or "Black").
 */
public static void captureThreat(String player, String piece)
    throws InvalidPieceException, InvalidPlayerException {
  for (int i = 0; i < playerPieces.size(); i++) {
    if (playerPieces.get(i).contains("Pawn") && !legalMoves.contains(
          Board.board.map.getTile(piece) + ", " + playerPieces.get(i))) {
      Controller.pawn.passedValidation = false;
      Controller.pawn.attackMove = false;
      Controller.pawn.movePawn(player, playerPieces.get(i), Board.board.map.getTile(piece));

      if (Controller.pawn.passedValidation && Controller.pawn.attackMove) {
        legalMoves.add(Board.board.map.getTile(piece) + ", " + playerPieces.get(i));
        Controller.pawn.attackMove = false;
        Controller.pawn.passedValidation = false;
      }

    } else if (playerPieces.get(i).contains("Rook") && !legalMoves.contains(
          Board.board.map.getTile(piece) + ", " + playerPieces.get(i))) {
      Controller.rook.passedValidation = false;
      Controller.rook.moveRook(player, playerPieces.get(i), Board.board.map.getTile(piece));

      if (Controller.rook.passedValidation) {
        legalMoves.add(Board.board.map.getTile(piece) + ", " + playerPieces.get(i));
        Controller.rook.passedValidation = false;
      }

    } else if (playerPieces.get(i).contains("Knight") && !legalMoves.contains(
          Board.board.map.getTile(piece) + ", " + playerPieces.get(i))) {
      Controller.knight.passedValidation = false;
      Controller.knight.moveKnight(player, playerPieces.get(i), Board.board.map.getTile(piece));

      if (Controller.knight.passedValidation) {
        legalMoves.add(Board.board.map.getTile(piece) + ", " + playerPieces.get(i));
        Controller.knight.passedValidation = false;
      }

    } else if (playerPieces.get(i).contains("Bishop") && !legalMoves.contains(
        Board.board.map.getTile(piece) + ", " + playerPieces.get(i))) {
      Controller.bishop.passedValidation = false;
      Controller.bishop.moveBishop(player, playerPieces.get(i), Board.board.map.getTile(piece));

      if (Controller.bishop.passedValidation) {
        legalMoves.add(Board.board.map.getTile(piece) + ", " + playerPieces.get(i));
        Controller.bishop.passedValidation = false;
      }
```

7.3 Software Engineering

In this section I will be referring to my usage of Software Engineering tools and approaches used throughout my project.

Test Driven Development

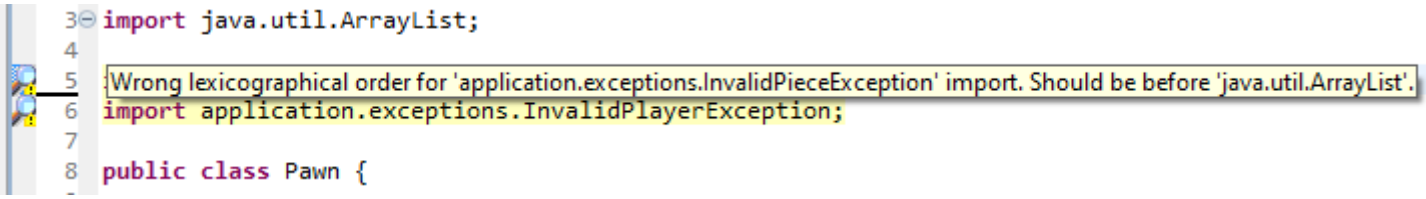
For each method inside every Java class I wrote, I also wrote a corresponding Junit test case to ensure that the method was working correctly and as I intended it to. For methods containing numerous conditional statements, a test case was created for each condition to test each possible scenario the method can execute code for.

Below is an example from the one of many test cases I wrote the Queen class. This particular test case sees if I can move the Queen diagonally North West. I do this by temporarily removing the Pawn blocking her path in tile 2C (as this would prevent the Queen from moving, due to validation checking there are none of the own player's pieces in her path). I then test if she moved to the tile North West of her origin tile, by asserting that the piece inside 3B is the WhiteQueen. The printStatus method call simply prints the tiles and the pieces inside those tiles as a textual representation of the Chess board's current state.

```
/**
 * This tests to see if I can move the Queen diagonally North West.
 */
@Test
void test8() {
  queen.board.map.setValue("piecePos", "2C", "null");
  try {
    System.out.println("Test 8");
    queen.moveQueen("White", "WhiteQueen", "3B");
    queen.board.map.printStatus();
    assertEquals(queen.board.map.getPieceOrOccupation("piecePos", "3B"), "WhiteQueen");
  } catch (InvalidPieceException e) {
    e.printStackTrace();
    fail("Exception Thrown");
  } catch (InvalidPlayerException e) {
    e.printStackTrace();
    fail("Exception Thrown");
  }
}
```

Checkstyle

Throughout my entire project, I have ensured that I used Checkstyle. Checkstyle is a widely used tool that enables me to check that my Java code complies with coding rules and standards. This improves the quality of my code, as it ensures that it is well-structured and eliminates bad code smells in the code I have written.



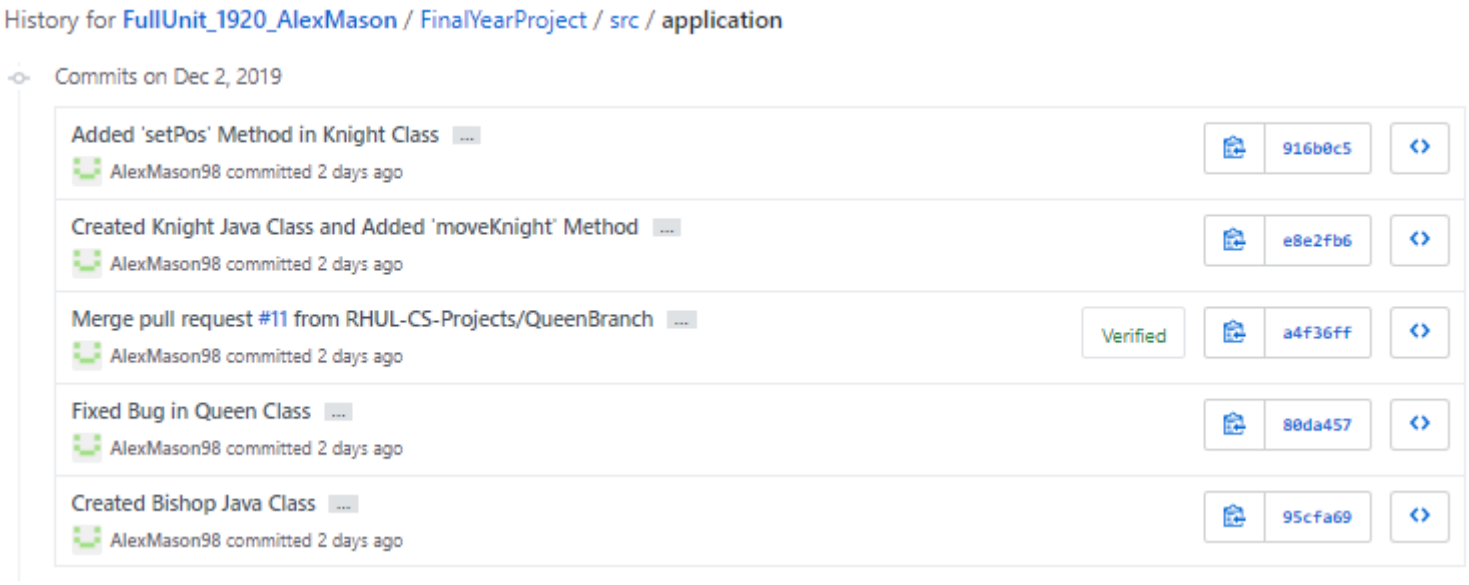
JavaDoc Commenting and General Comments

Another Software Engineering approach I have used is the usage of JavaDoc. JavaDoc is a documentation tool which allows me to document and comment my code. By commenting my code, I increase its readability and reusability so that myself or other programmers can read my comments as a way of understanding what something does or my thought processes when originally writing the code. Below is an example of one of my JavaDoc comments for the 'setPos' method in the Queen class.

```
/**
 *  This method is responsible for setting the final position of the Queen to the destination tile
 *  after validation has taken place in the moveQueen method. The setPos method also checks for
 *  the piece in the destination tile, so that if there's an opponent piece in the destination
 *  tile, the Queen can capture it. If there is the player's own piece in the destination tile,
 *  the method will indicate this to the player and not move the Queen.
 *
 * @param player Takes the current player (either "White" or "Black") to validate if the piece
 *       during an capturing move is an opponent piece.
 * @param selectedQueen Takes the selected Queen piece to set it to its destination tile.
 * @param toTile The tile the selected Queen intends to move to (destination tile).
 * @throws InvalidPlayerException if the player passed to the method is not "White" or "Black".
 * @throws InvalidPieceException if an invalid piece is passed to the method.
 */
public void setPos(String player, String selectedQueen, String toTile) throws
    InvalidPieceException, InvalidPlayerException {
```

Version Control

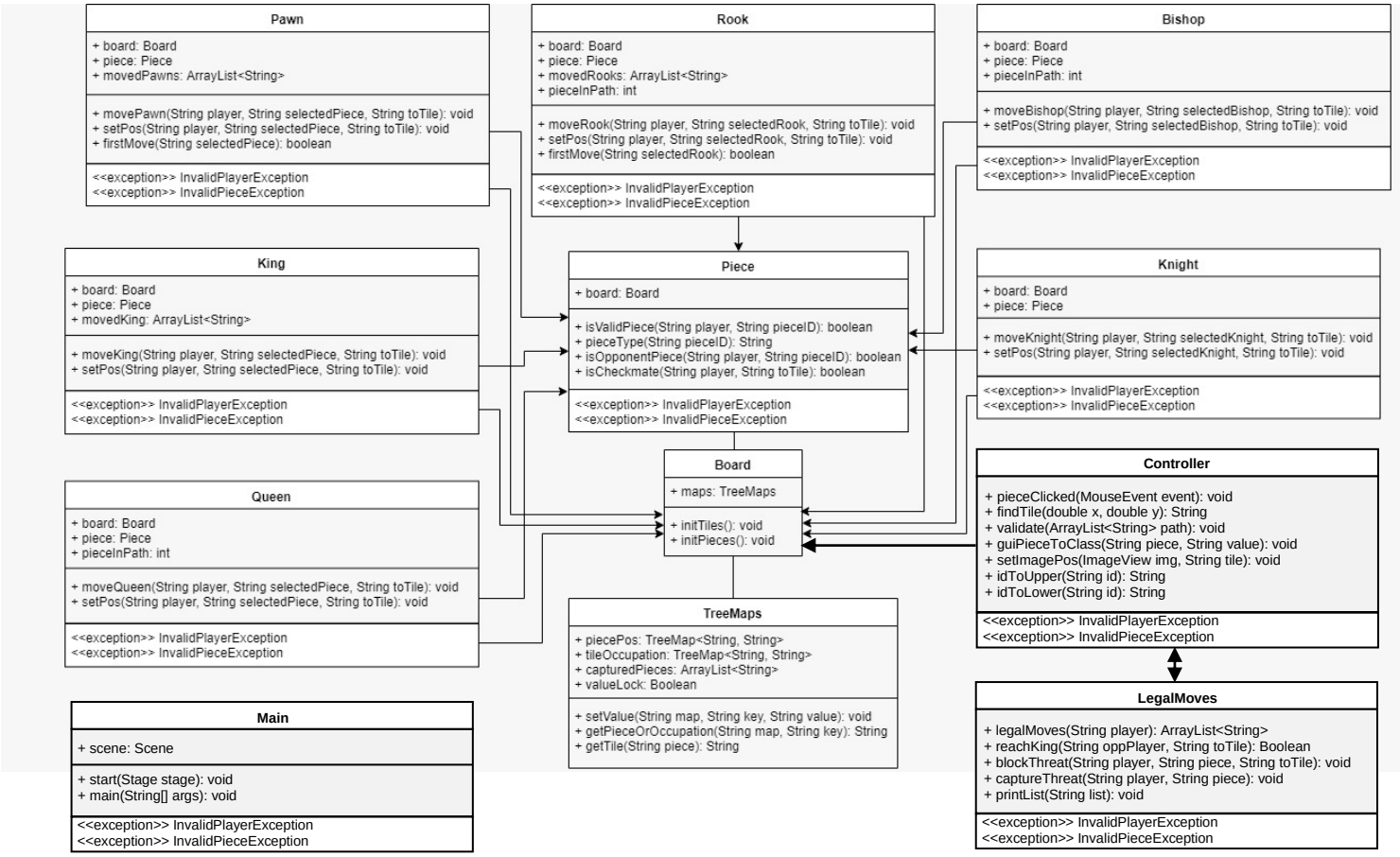
A final Software Engineering approach I have used throughout my project is the usage of version control system, GitHub. GitHub has enabled me to commit (upload) my work to my online repository whenever I have made new changes to my project. It also enables me to view my commit history so that I can see the changes made between one version and another. When I am developing code, I would make a branch to work on that code so that I don't have to upload code I am currently developing to master, as this should only contain code that is completed and tested.



Version Control: Commit History

UML

Below is my UML diagram, this shows the structure of my program at the end of my final year project.



Chapter 8: **Project Diary**

In this section, I will be looking through the dates/entries from both my personal diary and daybook and will be extracting information from entries that are relevant to the aims I had for the first term. These extracts will mention when a particular aim/objective was started and the notes I have relating to the progress of that aim.

Monday 07/10/2019:

- On this date, I created a BoardTest Junit test class and Board Java class. Inside the Board class I wrote a tile initialisation method and a chess piece initialisation method.

Tuesday 08/10/2019:

- On this date, I wrote a getter for the tile occupation, a getter for obtaining the piece from a given tile co-ordinate, a getter for obtaining the tile co-ordinate by a given/passed chess piece, a setter for setting new tile co-ordinates, and a print status method for printing out the TreeMaps.

Thursday 10/10/2019:

- I created a TestPlayer Junit test class and Player Java class. In the Player class I wrote a constructor, a getter for the current player, and a Boolean method for seeing if it's the white player's turn or not. Also created a custom exception named 'InvalidPlayerException'.

Friday 11/10/2019:

- Created an InvalidPieceException. Created a TestPawn Junit test class and Pawn Java class. Added a 'movePawn' method to the Pawn class.

Tuesday 15/10/2019:

- Created a constructor for Board class, which calls its own methods to initialise the game.

Monday 21/10/2019:

- Started research for my History and Rules of Chess report

Monday 28/10/2019:

- Started writing my History and Rules of Chess report

Thursday 31/10/2019:

- Finished writing my History and Rules of Chess report

Monday 04/11/2019:

- Added a set position method in the Pawn class. Added a first move Boolean method to the Pawn class.

Wednesday 06/11/2019:

- Created TestRook Junit test class and Rook Java class. Started working on a move Rook method.

Thursday 07/11/2019:

- Attended meeting with supervisor to get feedback on my History and Rules of Chess report. I will make amendments based on this feedback.

Friday 15/11/2019:

- Carried on with validation code for the move Rook method (checking no pieces in Rook's path).

Sunday 17/11/2019:

- Finished writing code for checking pieces in the Rook's movement path in Rook class.

Monday 18/11/2019:

- Started and finished writing a set position method for setting final position of Rook.

Tuesday 19/11/2019:

- Started revising my History and Rules of Chess report, based on advisor feedback.

Thursday 21/11/2019:

- Attended meeting with supervisor to discuss progress and receive advise on interim report.

Friday 22/11/2019:

- Finished all revisions for History and Rules of Chess report, uploaded to GitHub.

Saturday 23/11/2019:

- Created TestKing Junit test class and King Java class. Added a move king method and set position method.

Sunday 24/11/2019:

- Finished move king method and set position method. Fixed bug found in Rook class. Created TestQueen Junit test class and Queen Java class. Started writing a move queen method.

Monday 25/11/2019:

- Finished moveQueen method. Started and finished a set position method in Queen class.

Thursday 28/11/2019:

- Created TestBishop Junit test class and Bishop Java class. Started writing a move bishop method in the Bishop class.

Saturday 30/11/2019:

- Started writing Coding Progression report.

Sunday 01/12/2019:

- Finished writing move bishop method in Bishop class. Started and finished writing the set position method in the Bishop class.

Monday 02/12/2019:

- Created TestKnight Junit test class and Knight Java class. Started and finished writing both move knight and set position methods in the Knight class.

Thursday 05/12/2019:

- Finished writing Coding Progression report.

Friday 06/12/2019:

- Finished writing Interim report.

25th January 2020:

- Created an IllegalMoves class. This is responsible for checking which moves are illegal when the player is trying to move their piece in the game. This is now redundant to the LegalMoves class which calculates both legal and illegal moves at the same time.

28th January 2020:

- Created a Stalemate class. This class checks to see if either of the players are in a stalemate at the current game state. If so, this would terminate the game. This class is now redundant to the LegalMoves class, which can check if a player is in a stalemate based on if they have any legal moves that they can make or not.

25th February 2020:

- Started writing my Checkmate class. This is responsible for checking to see if the player is in checkmate or not. This class is now redundant to the refactored LegalMoves class which can now search for checkmate.

27th February 2020:

- Created my Chessboard GUI. This contains the game board and the chess pieces to play the game.

27th February 2020:

- Created Main class. This is a JavaFX class which starts my program so that it can be played.

28th February 2020:

- Created a Controller class. This class is responsible for linking the GUI I created to the Chess engine I have coded. It allows the player to move pieces and for this to then be updated in the engine's data structures.

09th March 2020:

- Created LegalMoves class. This is responsible for searching all of the player's pieces to see which tiles it can move to. Not only this, but it checks to see if a move could endanger the player's own King and disallow the move. If the opposition puts the player's King into danger, then the methods inside the class will allow the player's piece to block or capture the threat.

  Note: This class was originally called MinimaxAlgo where the legal moves part was a method within the class. However, due to time constraints in the algorithm's development and the unforeseen complexity to develop the algorithm, this was renamed LegalMoves.

21st April – 23rd April 2020:

- Refactoring code for submission. Involves finishing commenting, fixing outstanding Checkstyle issues, final Junit testing, and removing any unnecessary output from appearing in the console.

23rd April 2020:

- Finishing and updating this report (Final Project report) so that all information is relevant and ready for submission.

Bibliography and Citations

[1] *Deep Blue Chess Computer.* Wikipedia.
https://en.wikipedia.org/wiki/Deep_Blue_(chess_computer)

[2] *History of Chess.* Wikipedia.
https://simple.wikipedia.org/wiki/History_of_chess

[3] Calvo, Ricardo. *"Valencia Spain: The Cradle of European Chess".* 1998.
http://history.chess.free.fr/papers/Calvo%201998.pdf

[4] "*History of the Stalemate Rule".* Wikipedia.
https://en.wikipedia.org/wiki/Stalemate#History_of_the_stalemate_rule

[5] *"The Chess Games of the London Chess Club".* London Chess Club.
https://www.chessgames.com/perl/chessplayer?pid=80740

[6] Answer By '11684'. *Is Chess a Solved Game?.* Chess Stack Exchange, 2016.
https://chess.stackexchange.com/questions/13522/is-chess-a-solved-game

[7] *"Chess Facts". OhFact!*
https://ohfact.com/chess-facts/

[8] *"40 Facts About Chess Most People Don't Know".* TheChessWorld.
https://thechessworld.com/articles/general-information/40-facts-about-chess-most-people-dont-know/

[9] *"Chess Setup and Rules".* Chess Coach Online.
http://www.chesscoachonline.com/chess-articles/chess-rules

[10] *Rules of Chess – Check.* Wikipedia.
https://en.wikipedia.org/wiki/Rules_of_chess#Check

[11] *Rules of Chess – End of the Game.* Wikipedia.
https://en.wikipedia.org/wiki/Rules_of_chess#End_of_the_game

[12] *7 Things Every Designer Needs to Know About Accessibility.* Medium.
https://medium.com/salesforce-ux/7-things-every-designer-needs-to-know-about-accessibility-64f105f0881b

[13] *Vision Loss.* NHS.
https://www.nhs.uk/conditions/vision-loss/

[14] *SVG Chess Pieces.* Wikimedia Commons.
https://commons.wikimedia.org/wiki/Category:SVG_chess_pieces

[15] *Creative Commons License.* Wikipedia.
https://en.wikipedia.org/wiki/Creative_Commons_license

[16] *Artificial Intelligence.* Wikipedia.
https://en.wikipedia.org/wiki/Artificial_intelligence

[17] *Artificial Intelligence in Video Games.* Wikipedia. https://en.wikipedia.org/wiki/Artificial_intelligence_in_video_games

[18] *Why is AI Still So Bad in Games Despite the Increasing Power of Gaming Machines Over the Years.* Reddit. https://www.reddit.com/r/pcgaming/comments/3uml0l/why_is_ai_still_so_bad_in_games_despite_the/

[19] *Monte Carlo Algorithm.* Wikipedia. https://en.wikipedia.org/wiki/Monte_Carlo_algorithm

[20] *Monte Carlo Tree Search. MCTS.ai* https://web.archive.org/web/20151129023043/http://mcts.ai/about/index.html

[21] *Game Theory – The Minimax Algorithm Explained.* Towards Data Science. https://towardsdatascience.com/how-a-chess-playing-computer-thinks-about-its-next-move-8f028bd0e7b1

[22] *Introduction to Minimax Algorithm.* Baeldung. https://www.baeldung.com/java-minimax-algorithm

[23] *Alpha-Beta Pruning.* Javatpoint. https://www.javatpoint.com/ai-alpha-beta-pruning

Appendix

[A1] History and Rules of Chess Report
https://github.com/RHUL-CS-Projects/FullUnit_1920_AlexMason/blob/master/Reports/History%20and%20Rules%20of%20Chess%20Report.docx

[A2] Coding Progression Report
https://github.com/RHUL-CS-Projects/FullUnit_1920_AlexMason/blob/master/Reports/Coding%20Progression%20Report.docx

[A3] Artificial Intelligence in Games Report
https://github.com/RHUL-CS-Projects/FullUnit_1920_AlexMason/blob/master/Reports/Artificial%20Intelligence%20in%20Games%20Report.docx