**UNIVERSITY OF HERTFORDSHIRE**

**School of Computer Science**

**Modular BSc Honours in Computer Science**

**6COM1056 - Software Engineering Project**

**Final Report**

**April 2018**

**CHESS DELTA: A SIMPLE APPROACH TO CHESS AI**

**D. García Molero**

**Supervised by: Dr Daniel Polani**

# Table of Contents

[Table of Contents 2](#_Toc511919437)

[Table of figures 5](#_Toc511919438)

[Abstract 6](#_Toc511919439)

[Acknowledgements 6](#_Toc511919440)

[Important Disclaimer 6](#_Toc511919441)

[Chapter 1: Introduction to the Project 7](#_Toc511919442)

[1.1 Motivation & Context 7](#_Toc511919443)

[1.2 Considerations 7](#_Toc511919444)

[1.3 Objectives 8](#_Toc511919445)

[1.4 Planning 8](#_Toc511919446)

[Chapter 2: Background and Research 9](#_Toc511919447)

[2.1 Chess as a game 9](#_Toc511919448)

[2.1.1 Rules and difficulties 9](#_Toc511919449)

[2.1.2 Past attempts to make a chess AI 9](#_Toc511919450)

[2.2 Technical research 9](#_Toc511919451)

[2.2.1 Algorithms & Computational Techniques 10](#_Toc511919452)

[2.2.1.1 MinMax 10](#_Toc511919453)

[2.2.1.2 NegMax 10](#_Toc511919454)

[2.2.1.3 Alpha Beta Pruning 10](#_Toc511919455)

[2.2.1.4 Transposition Tables 11](#_Toc511919456)

[2.2.1.5 Monte Carlo Tree Search 11](#_Toc511919457)

[2.2.1.6 Neural Networks 11](#_Toc511919458)

[2.2.2 Java Design Patterns 12](#_Toc511919459)

[2.2.2.1 Definition 12](#_Toc511919460)

[2.2.2.2 Advantages and drawbacks 12](#_Toc511919461)

[2.2.3 Software Engineering Practices 12](#_Toc511919462)

[2.2.3.1 Documentation 13](#_Toc511919463)

[2.2.3.2 Coding 13](#_Toc511919464)

[2.2.3.3 Debugging 13](#_Toc511919465)

[2.2.3.4 Testing 13](#_Toc511919466)

[2.2.3.4.1 Unit Tests 13](#_Toc511919467)

[2.2.3.4.2 Alpha Releases and Testers 13](#_Toc511919468)

[2.2.3.4.3 Nim 13](#_Toc511919469)

[2.2.3.4.4 Chess End Games 13](#_Toc511919470)

[2.2.4 Game API 13](#_Toc511919471)

[2.2.4.1 Description 13](#_Toc511919472)

[2.2.4.2 Original API 14](#_Toc511919473)

[2.2.4.3 Features 15](#_Toc511919474)

[Chapter 3: Software Requirements 16](#_Toc511919475)

[3.1 Functional Requirements 16](#_Toc511919476)

[3.2 Non-functional Requirements 16](#_Toc511919477)

[Chapter 4: Feasibility Study and Choices Taken 17](#_Toc511919478)

[4.1 Software choices 17](#_Toc511919479)

[4.1.1 Eclipse JEE 17](#_Toc511919480)

[4.1.2 GitHub 17](#_Toc511919481)

[4.1.2.1 Motivation 17](#_Toc511919482)

[4.1.2.2 Folder structure 17](#_Toc511919483)

[4.1.3 Google Drive 18](#_Toc511919484)

[4.2 Model choices 18](#_Toc511919485)

[4.2.1 New board design 18](#_Toc511919486)

[4.2.2 New piece design 18](#_Toc511919487)

[4.3 Other choices 19](#_Toc511919488)

[4.4 Time and resources estimation 20](#_Toc511919489)

[4.5 Feasibility of the project 20](#_Toc511919490)

[4.6 Testing choices 20](#_Toc511919491)

[4.7 User interface 20](#_Toc511919492)

[Chapter 5: Software Design, Architecture & Diagrams 21](#_Toc511919493)

[5.1 Overview 21](#_Toc511919494)

[5.2 Diagrams 21](#_Toc511919495)

[5.3 Design Patterns Used 21](#_Toc511919496)

[5.3.1 Model-view-controller 21](#_Toc511919497)

[5.3.2 Factory 21](#_Toc511919498)

[5.3.3 Command 21](#_Toc511919499)

[5.3.4 Observer 21](#_Toc511919500)

[Chapter 6: Components & Implementation 22](#_Toc511919501)

[6.1 List of Components 22](#_Toc511919502)

[6.2 Most relevant classes 23](#_Toc511919503)

[6.2.1 ChessAlphaBeta & ChessMinMax 23](#_Toc511919504)

[6.2.2 ChessBoard 23](#_Toc511919505)

[6.2.3 ChessMove 24](#_Toc511919506)

[6.2.4 ChessRules 25](#_Toc511919507)

[6.2.5 ChessPiece 25](#_Toc511919508)

[6.2.6 ChessViews Package 25](#_Toc511919509)

[6.3 User Interface 26](#_Toc511919510)

[Chapter 7: Testing 27](#_Toc511919511)

[7.1 Unit tests 27](#_Toc511919512)

[7.2 toString() debugging 27](#_Toc511919513)

[7.3 Nim 27](#_Toc511919514)

[7.4 Chess end-games 27](#_Toc511919515)

[7.5 Playing against human subjects 27](#_Toc511919516)

[Chapter 8: Project Evaluation 28](#_Toc511919517)

[8.1 Evaluation Overview 28](#_Toc511919518)

[8.2 Strengths of the Project 29](#_Toc511919519)

[8.3 Problems encountered 29](#_Toc511919520)

[8.3.1 Critical problems 29](#_Toc511919521)

[8.3.2 Other problems and solutions 29](#_Toc511919522)

[8.4 Potential improvements 29](#_Toc511919523)

[Chapter 9: Conclusions 30](#_Toc511919524)

[9.1 Conclusions Overview 30](#_Toc511919525)

[9.2 Lessons learnt 30](#_Toc511919526)

[9.3 Should (not) have done 30](#_Toc511919527)

[9.4 Personal opinion 30](#_Toc511919528)

[Chapter 10: Future Work 31](#_Toc511919529)

[10.1 State of the project 31](#_Toc511919530)

[10.2 Future of the game 31](#_Toc511919531)

[10.3 Improvements to existing features 31](#_Toc511919532)

[10.4 Expectations 31](#_Toc511919533)

[Chapter 11: Bibliography 32](#_Toc511919534)

[Chapter 12: Appendices 33](#_Toc511919535)

[Appendix I. MinMax and AlphaBeta code 33](#_Toc511919536)

[Appendix I-a. MinMax 33](#_Toc511919537)

[Appendix I-b. AlphaBeta 36](#_Toc511919538)

[Appendix I-c. Board evaluation heuristics 39](#_Toc511919539)

[Appendix II. ChessBoard code 43](#_Toc511919540)

[Appendix III. ChessMove code 53](#_Toc511919541)

[Appendix IV. ChessRules code 72](#_Toc511919542)

[Appendix V. Chess pieces code 77](#_Toc511919543)

[Appendix V-a. ChessPiece code 78](#_Toc511919544)

[Appendix V-b. Bishop code 78](#_Toc511919545)

[Appendix V-c. King code 79](#_Toc511919546)

[Appendix V-d. Knight code 79](#_Toc511919547)

[Appendix V-e. Pawn code 80](#_Toc511919548)

[Appendix V-f. Queen code 80](#_Toc511919549)

[Appendix V-g. Rook code 81](#_Toc511919550)

[Appendix VI. Chess Views code 81](#_Toc511919551)

[Appendix VI-a. ChessBoardComponent code 82](#_Toc511919552)

[Appendix VI-b. ChessWindowSwingView code 86](#_Toc511919553)

# Table of figures

[Figure 1. One of the first Ataxx matches being played during development of the original game 14](#_Toc511919554)

[Figure 2. Game of Ataxx with random obstacles 15](#_Toc511919555)

[Figure 3. Ataxx being played in the original project 18](#_Toc511919556)

[Figure 4. Chess Delta first implementation of the board and pieces. Pieces were not yet able to move, and each piece was treated as a different player, as can be seen on the Player information board on the right side. 19](#_Toc511919557)

[Figure 5. Gantt Chart Time Estimation 20](#_Toc511919558)

[Figure 6. GitHub contribution graph for the repository 22](#_Toc511919559)

[Figure 7. All classes created for Chess Delta highlighted in blue 23](#_Toc511919560)

[Figure 8. Different exceptions for invalid moves 24](#_Toc511919561)

[Figure 9. Pawn promotion dialog 25](#_Toc511919562)

[Figure 10. The yellow rectangle represents ChessWindowSwingView’s scope, while the red one represents what ChessBoardComponent is in charge of 26](#_Toc511919563)

[Figure 11. Original time and features estimation 28](#_Toc511919564)

# Abstract

The following report aims to describe the steps, decisions and various aspects involved in the development of a *Chess Delta* as part my Undergraduate Thesis in the past few months.

*Chess Delta* is not only a game that you can play with other people in a single computer, but also features several different AI modes.

Every aspect of the project will be discussed in the different sections of this report, including future ideas, design decisions that were made as well as flaws in the project and insights from the author on how these could have been solved or approached in different ways.

The whole project will follow best practices in Software Engineering, such as clean and properly commented code, unit and system tests, version control, documents following the pertinent IEEE standards, etc. CHECK

# Acknowledgements

Firstly, I would like to thank my family and friends for their support and patience with me during these months.

Special thanks to my father, for getting me interested in chess and teaching me how to play at a very young age, and my mother, for supporting me throughout my whole education and final year project.

I am extremely grateful for all the people that believed in me since the beginning and gave me the motivation for a project of this size and level of difficulty.

I would also like to thank Dr Daniel Polani, my project supervisor, for his support and valuable advice throughout the whole process, and Dr Guy Saward, for making my stay at the University of Hertfordshire a truly pleasant experience that taught me a lot both academically and as an individual. CHECK

# Important Disclaimer

It is important to note that although the project here described IS a final version of the software, including all required features to allow subjects to play, as well as a fully functional artificial intelligence engine, some more features might be added for the presentation at my home university (Complutense University of Madrid), since the presentation date at the tribunal (if applicable) will be in July. The potential future features will be discussed in this document as well (See 10.2 Future of the game). CHECK

Chapter 1: Introduction to the Project

# 1.1 Motivation & Context

For a very long time, artificial intelligence has been a prime subject of interest for me. Although this final year project is mainly a Software Engineering project, I also wanted to include Artificial Intelligence and Networking concepts as part of the it. The idea of creating a Chess game was one of the project proposals by Dr Peter Lane at the *Project Planning* module, during semester A. I found it attractive because for a long period of my life, I was an avid chess player.

An additional reason was that during my second year at the *Complutense University of Madrid*, one of the projects required for the *Computer Programming Technology* module involved using a game API (See 2.2.4 Game API) to create different board games by using inheritance in Java. Therefore, my main aim was to reuse the same API, modifying the starting code only where it was needed, since the original API was only designed to support simple games where every piece had the same value to every other. This allowed games like Tic-Tac-Toe, Connect4 or Ataxx to be implemented with ease, given their relatively simple game mechanics and rules. CHECK

# 1.2 Considerations

Chess was the game selected for the game engine, however, it was not the only game contemplated for study and development. Dr Daniel Polani suggested two alternatives to Chess in the early stages of the development.

The first game considered was *Go*, a game original from Asia that is believed to be the oldest board game still in play nowadays[[1]](#footnote-1). The reason why this game was originally considered as a better alternative to Chess, is that *Go* is a much simpler game in terms of representation since every piece is worth the same at the end of the match, and includes way less special moves (See 2.1.1 Rules and difficulties), although it features some of its own that may make the game slightly complicated to compute, such as the Ko or Suicide rules[[2]](#footnote-2). The reason why *Go* was finally dismissed was that the board design differed too much from the original game API that was already working for Chess, and because of the lack of familiarity with this game when compared to Chess. One additional reason is that in *Go*, it is much more difficult to determine the winner and it is often agreed by the players or by resignation.

The second possible alternative was *Focus*. This game was a solid alternative because of the relative ease to compute several positions, as well as only having one kind of pieces per player. The main reason why this game was discarded was, once more, the board design and rules, such as stacking pieces[[3]](#footnote-3). The initial game API offered tools to easily create and represent dimensions boards. While it could be modified, the main aim of the project was not to create and represent the board, but to create an effective artificial intelligence engine to a board game. Familiarity with the game when compared to Chess was also a determining factor. DONE

# 1.3 Objectives

The main objectives for this project are to study and understand the process of creating a complete board game starting from a game API (See 2.2.4 Game API), following software engineering principles such as use of design patterns. The code should also be properly documented. Inheritance and use of interfaces to make the software as reusable and modular are a must as well. It should also include a relatively advanced AI chess engine. As secondary objectives, the game could feature networking and threading concepts, distributed computing or creating a web or mobile app (See 10.2 Future for the full list of potential features that could be included at a later time).

For the full list of requirements, refer to 3.1 Functional Requirements and 3.2 Non-functional Requirements.

# 1.4 Planning

As most final year projects, the project will last for the whole school year (2017-18), however, rather than using the suggested schedule, the development phase will start early – in the first week of November – and will continue for the coming months, until approximately April. This is due to the project size and complexity, that made it quite hard to perform every necessary task if the development started after Christmas as advised. The detailed Gantt Chart containing the full development plan can be found at 4.4 Time and resources estimation.

Chapter 2: Background and Research

# 2.1 Chess as a game

## 2.1.1 Rules and difficulties

Chess as a game has been around for almost two centuries[[4]](#footnote-4) in different versions. However, the most standard version nowadays, dates back to the final decades of the 15th century[[5]](#footnote-5). In the early years of the game, special moves such as castling or pawn opening double moves did not yet exist, but were later introduced into the game, which saw different rules (such as letting either white or black move first) but preserved the essence until the 19th century, when all of the current rules were finally fixated.

Some of these special rules increase the game complexity. Their main purpose is avoiding certain loopholes, such as the possibility of matches that would last for an infinite number of moves. While being greatly useful to enhance the possibilities and strategies in human vs human matches, they do make it a lot harder to develop artificial intelligence engines suitable for this game, as well as making it more complex to evaluate instances of the board, if all special rules are taken into account.

The main rules to take into account are Castling, En-Passant, Pawn Promotion, draw by Stalemate (formerly considered to be a win condition and still argued by some[[6]](#footnote-6)), Fifty-Move rule and Threefold repetition of position.

## 2.1.2 Past attempts to make a chess AI

Chess is notorious for being a very computationally intensive game, having exponential complexity in n (on an board) to find an optimal strategy[[7]](#footnote-7). This means that for a given starting board, there is an estimate of possible different games of chess that could be legally played[[8]](#footnote-8). This number is known as *Shannon Number* and it was an attempt made by *Claude Shannon* to demonstrate that if someone were to create a computer that played all possible chess games until an end in order to make a decision on the move to make next, this computer would never be able to make a move, therefore suggesting that brute-force approaches to creating a chess engine should be abandoned. In the following sections, different approaches to creating a chess AI engine will be explored.

# 2.2 Technical research

Given the complexity above mentioned, creating a chess engine requires a vast knowledge of computational complexity theory, the programming language of choice as well as its libraries and different computational techniques and approaches to game theory. The following sections will cover the research done in each field needed for the development of the game and the base for the decisions and choices taken will be included in the next chapters (See 4.3 Other choices).

## 2.2.1 Algorithms & Computational Techniques

Given that the project’s aim was to explore, develop and experience various approaches to creating an intelligent chess engine, algorithms comprise a vital part of the same. In the following sub-sections, different algorithms suitable for this purpose will be analysed.

### 2.2.1.1 MinMax

MinMax (also known as MiniMax) is a decision rule that can be applied to various fields of knowledge, including statistics and philosophy, but most importantly in this case, for game theory. Its ruled by the MinMax principle which, as described by Michiel Hazewinkel, is: “An optimality principle for a two-person zero-sum game, expressing the tendency of each player to obtain the largest sure pay-off”[[9]](#footnote-9). In other words, the MinMax algorithm is used to select the best possible move to make next, in order for the opponent to be able to gain the least possible potential advantage in the following moves. Trivially, in a two-player game, minimising the opponent’s chances of winning directly increases your chances of winning (or not losing). The main drawbacks for the MinMax algorithm in many cases are that it requires a predefined evaluation function for each board state, as well as the inability to interrupt the search at any given moment, being necessary to establish the depth to be explored beforehand. One more significant disadvantage is the fact that MinMax explores every possible node, rather than being restricted to the most promising ones, which is improved partially by Alpha Beta pruning and much more significantly by Monte Carlo Tree Search. Another restriction for MinMax is that it cannot be applied to games with more than two players, although a generalisation of the MinMax theorem can be used for such cases[[10]](#footnote-10).

In the case of basic chess engines, MinMax is a good option to develop a relatively smart artificial intelligence engine with a limited amount of resources and development time. In general, a more complex but more elegant and powerful solution for this problem can be achieved by using Monte Carlo Tree Search (currently used by some of the strongest game engines, see 2.2.1.5 Monte Carlo Tree Search) or Deep Neural Networks (used by AlphaZero, the strongest game engine up to date, see 2.2.1.6 Neural Networks).

The implementation aspects of MinMax will be discussed on 6.1 List of Components.

### 2.2.1.2 NegMax

Although the principle of NegMax (or NegaMax) is identical to the one applied by MinMax, the main difference resides in the way that it identifies players. Whilst MinMax identifies one player as the *min* player, and the other one as the *max* player, NegMax simply changes the symbol (+, –) of the rating from the previously explored level, making it have almost exactly the same performance and results as MinMax. According to T. Anthony Marsland, “The NegaMax approach is preferred since the programming is simpler [when compared to MinMax]”[[11]](#footnote-11).

### 2.2.1.3 Alpha Beta Pruning

Alpha Beta Pruning is simply an improvement that can be applied to both MinMax and NegMax. Its main objective is to minimise the number of branches explored by not exploring further on those proven to be worse than or equal to at least one other branch, thus saving time by avoiding iterations on branches that are not going to be selected[[12]](#footnote-12). It is a very simple improvement to the algorithm that makes use of two values at each node: α represents the minimum guaranteed rating for the *max* player, whilst β represents the maximum guaranteed rating for the *min* player. If at any given node alpha becomes greater than or equal to beta, it means that the child nodes that would be created from the current one, will never be selected as optimal, thus saving time by omitting them.

### 2.2.1.4 Transposition Tables

POSSIBLY ADD IN THE FUTURE. <https://en.wikipedia.org/wiki/Transposition_table>

### 2.2.1.5 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS from here onwards) is a probabilistic method of evaluation whose aim is analysing the most promising moves by creating a random selection of possible moves and playing the match to and end given a board state. It represents a significant improvement when compared to MinMax and Alpha Beta for several reasons, although if the branching factor is increased sufficiently, the resulting tree has actually been proven to match exactly with the one created by MinMax[[13]](#footnote-13). This accurately represents the main improvement of MCTS respective to MinMax; having a lower branching factor (discarding non-promising branches early), it enables a much deeper search to be made. Some other meaningful advantages are the possibility to interrupt the search and return the best movement found so far at any point of the execution or the fact that MCTS does not require any heuristics except for the rules of the game itself, but no evaluation function whatsoever. These improvements have made this technique quintessential and widely used in all the most advanced game engines created to date. The most remarkable results have been achieved by Google DeepMind’s AlphaGo, AlphaGoZero and AlphaZero[[14]](#footnote-14) by using neural networks (See 2.2.1.6 Neural Networks) to select the most promising branches as early on as possible, thus reducing the branching factor and subsequently number of moves explored significantly.

### 2.2.1.6 Neural Networks

As mentioned while analysing the advantages and drawbacks of the MinMax algorithm, neural networks have been the strongest proven approach to creating a chess engine. They became widely known after testing AlphaZero against the strongest chess engines available at the time[[15]](#footnote-15). The results for said games were heavily in favour of AlphaZero, which after only four hours of playing against itself was able to become the strongest chess player of all times. The main difference between AlphaZero and any other previous engine, is that AlphaZero achieved the results by playing against itself, only knowing the rules of the game and not having access to heuristics or opening tables, whilst traditional chess engines used a more systematic approach to the problem, selecting results based on heuristics tailored by different experts and making use of different opening tables for the initial movements of the game15. The main difference in the concept was that, although AlphaZero also included a Monte Carlo Tree Search as part of the engine, when compared to some of the other stronger engines, it computed a vastly reduced amount of moves per second (in the order of 1/900 approximately15), selecting only the most promising branches by making use of neural networks instead of exploring as many branches as some of the other engines, such as Stockfish15. As of 2018, this has been proved to be the most effective approach to creating a chess engine.

## 2.2.2 Java Design Patterns

The use of design patterns is one of the core features of good practices in Software Engineering, therefore, it needs to be studied to determine the potential advantages and drawbacks of design patterns to later determine whether they should be applied or how to apply them to the project. (See 5.3 Design Patterns Used)

### 2.2.2.1 Definition

Design patterns are not specific pieces of code but rather concepts. These concepts are used to portray practices that should be used to solve common problems in Software Engineering by applying similar solutions across different products with the purpose of improving reusability[[16]](#footnote-16).

### 2.2.2.2 Advantages and drawbacks

Whilst the use of design patterns is widely accepted and encouraged in almost every case, there are also disadvantages[[17]](#footnote-17) to using them, especially the level of complexity that a project can develop if a number design patterns are used. Another good argument against design patterns was made by Peter Norvig who stated that design patterns are not necessary and are currently mainly being used to fix issues that programming languages should not have if properly designed. He also proves that most of these patterns would be simplified or non-existent in other languages such as Lisp[[18]](#footnote-18) because of the way the language naturally addresses these issues without the need to use design patterns. It should also be noted that design patterns were mainly created for Object Oriented languages such as C++ when originally conceived, and some adaptations were later published for Java[[19]](#footnote-19) and other languages.

In general, design patterns try to address certain problems that appear naturally and repeatedly during the software development process and whose solutions — without following said patterns — would cause reusability, maintainability and modularity issues. The main reason why they are implemented is to unify proposed solutions and reduce costs by creating the ability to reuse past solutions, avoiding the repetition of the engineering process multiple times for fairly similar projects that would be costly and present little or no benefit[[20]](#footnote-20).

## 2.2.3 Software Engineering Practices

Software Engineering is defined as “the systematic application of scientific and technological knowledge, methods, and experience to the design, implementation, testing, and documentation of software”[[21]](#footnote-21) by the IEEE. Therefore, it should be no surprise that every software project should follow and include certain standards and features.

### 2.2.3.1 Documentation

Every software project should be well documented to further increase reusability, reduce risks, avoid errors, plan and measure time and financial costs, among other purposes. To adapt to the software life-cycle, the IEEE created a set of standards that should be taken into consideration in every software product development[[22]](#footnote-22). The list of documents described by the IEEE is the following:

* Software quality assurance[[23]](#footnote-23) IEEE Std. 730.
* Software configuration management23 IEEE Std. 828.
* Software test documentation23 IEEE Std. 829.
* Software requirements specification23 IEEE Std. 830.
* Software verification and validation23 IEEE Std. 1012.
* Software design description23 IEEE Std. 1016.
* Software project management23 IEEE Std. 1058.
* Software user documentation23 IEEE Std. 1063.

### 2.2.3.2 Coding

POSSIBLY REMOVE

### 2.2.3.3 Debugging

POSSIBLY REMOVE

### 2.2.3.4 Testing

POSSIBLY REMOVE

#### 2.2.3.4.1 Unit Tests

POSSIBLY REMOVE

#### 2.2.3.4.2 Alpha Releases and Testers

POSSIBLY REMOVE

#### 2.2.3.4.3 Nim

POSSIBLY REMOVE

Only objective research on this game.

#### 2.2.3.4.4 Chess End Games

POSSIBLY REMOVE

## 2.2.4 Game API

As mentioned at the beginning of the document (See 1.3 Objectives), the base for the development of the game will be an existing game API, provided for the coursework at the *Computer Programming Technology module* (*Complutense University of Madrid, year 2015/16*).

### 2.2.4.1 Description

The API was created in the Java language for educational purposes. It included heavy class interdependencies, high modularity and a simple interface. It also included example of possible games that could be created using the API, such as Tic-Tac-Toe or Connect4. The original code for the Chess Delta development, which is not only the code provided by the university but also all the modifications made by myself in the second year of studies, is available at the GitHub repository.

### 2.2.4.2 Original API

The original API enabled students to implement a different game by making use of inheritance in the Java language. An example of this can be found on Figure 1. One of the first Ataxx matches being played during development of the original game.

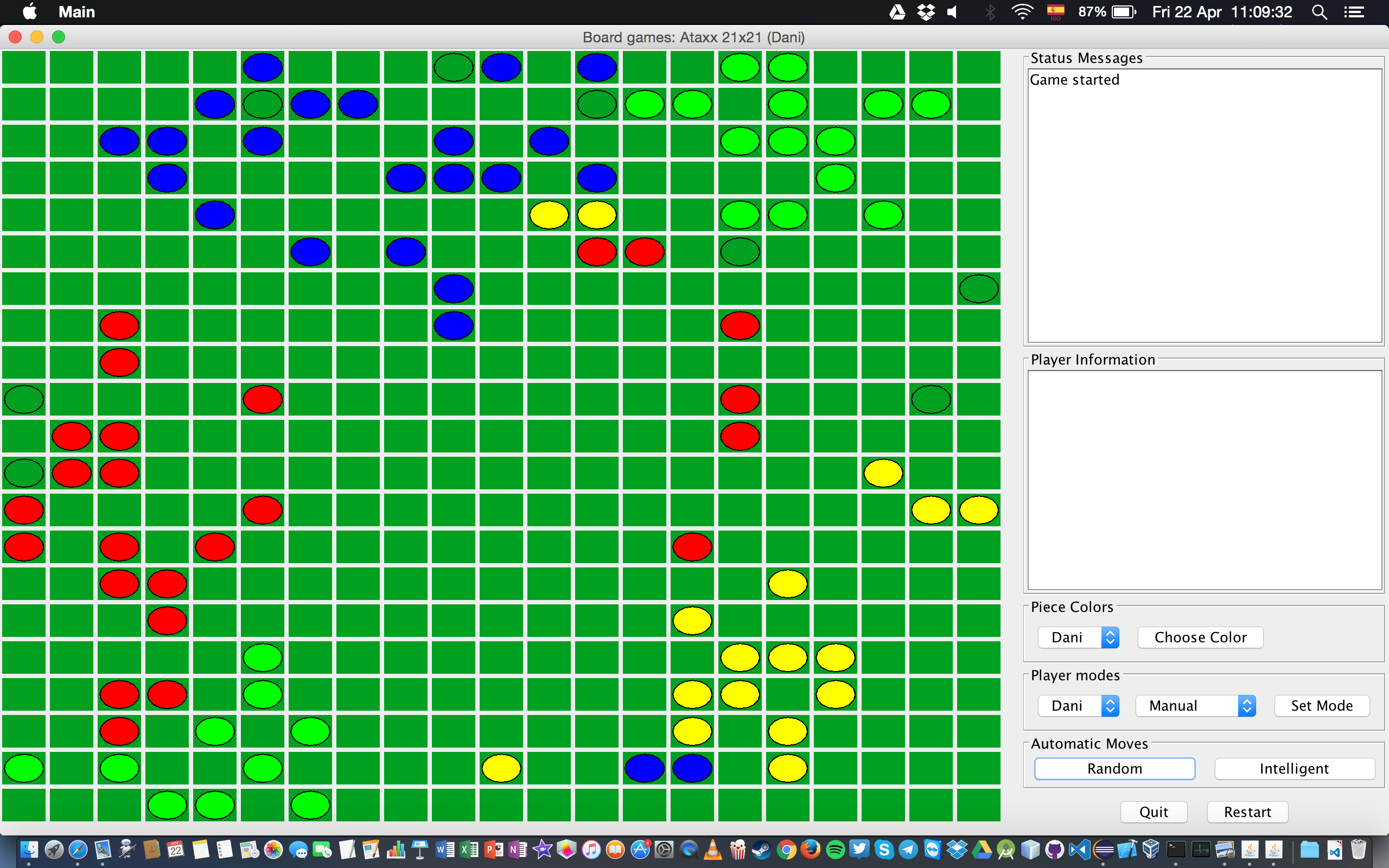


Figure 1. One of the first Ataxx matches being played during development of the original game

While the original game only allowed one type of piece per player, random obstacles were also introduced later for the game Ataxx, as it can be seen on Figure 2. Game of Ataxx with random obstacles.

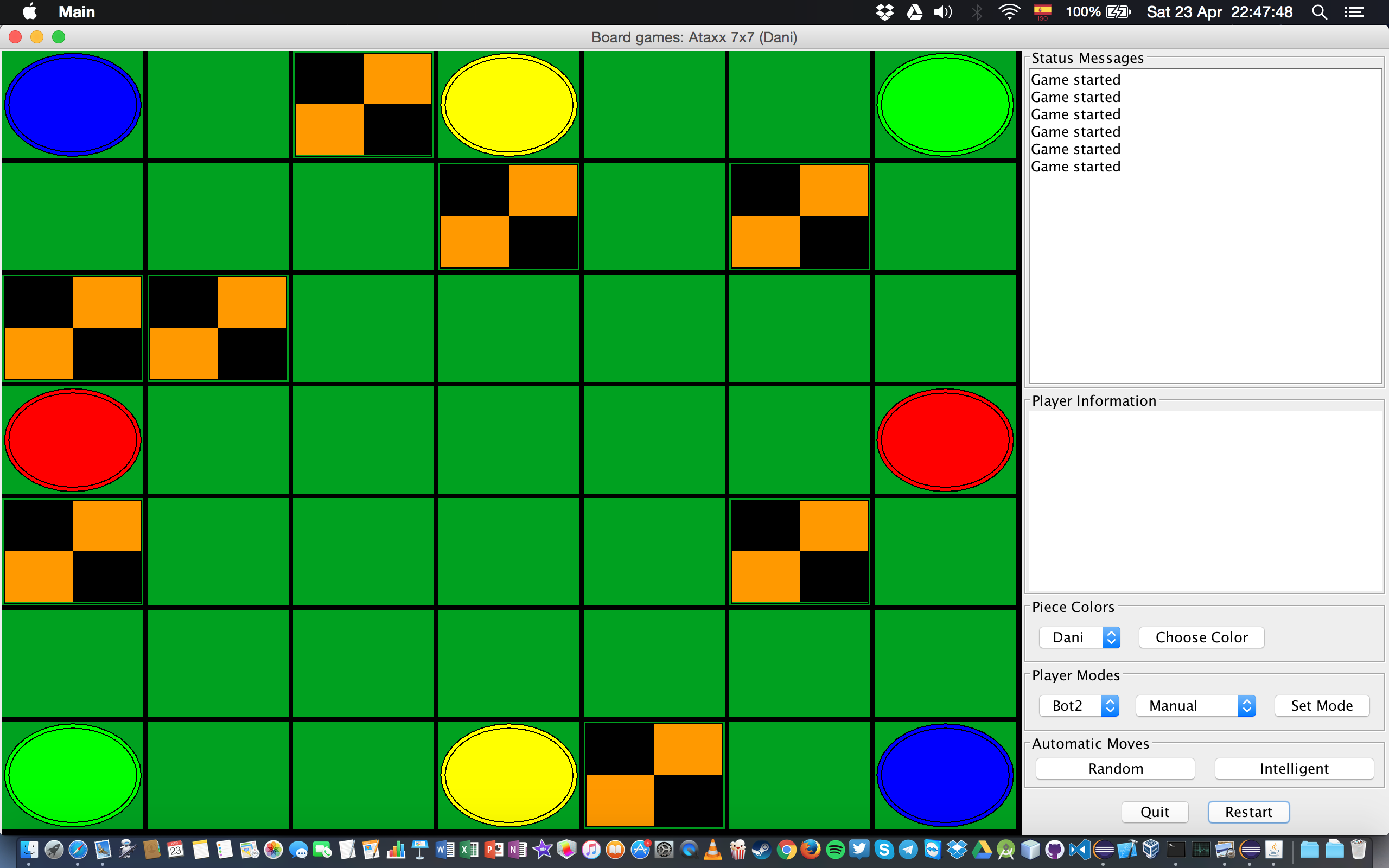


Figure 2. Game of Ataxx with random obstacles

More examples of the original API and the final appearance of the game with different players and player modes can be found under Figure 3. Ataxx being played, in the 4.2.2 New piece design section.

### 2.2.4.3 Features

The original software included all necessary tools to create a suite of board games that could be played on a rectangular board made of similar tiles and where each piece type — displayed as piece colour on the board — represents a player and every piece is equal to every other one regarding movement and attributes as long as they are the same colour. The games should also be turn-based and have at least 2 players.

The API also included tools to implement a MinMax player with possible AlphaBeta pruning options.

Overall, it was properly commented and included all necessary features to generate a solid Javadoc document.

Chapter 3: Software Requirements

# 3.1 Functional Requirements

Functional requirements define the functions a system should include, such as inputs, outputs or behaviours. They shall be defined in various ways and formats as long as they define *what* a system is supposed to accomplish and not *how* it should accomplish it[[24]](#footnote-24).

The functional requirements for this project would be the following:

* Create a fully functional chess game that allows two human subjects to play against each other using the app.
* Include ALL special rules of classical chess, as mentioned in the final part of 2.1.1 Rules and difficulties.
* Recognise all three possible different outcomes of a chess game: *Win, Loss* or *Draw (Both by repetition rules and Stalemate).*
* Check that each move performed by a player is valid and display a message with the reason why the move was not accepted if this happens.
* Allow a single person to play a match against different types of AI players of his choice. The AI players should always follow the rules of the game and have different levels of complexity.
* Enable two AI players to play against each other on the same machine.
* Allow future development of different chess variants using the final state of the project, making the game modular and using interfaces to generalise as often as possible.

# 3.2 Non-functional Requirements

Non-functional requirements are those requirements that do not constitute features of the software itself, but nonetheless, are required characteristics of the same[[25]](#footnote-25), such as constraints that are not related to what should be implemented, but how it should be implemented instead.

The non-functional requirements for this project would be the following:

* The game should be highly usable. The players should only need to know the rules of chess to be able to successfully play the game.
* The app should work adequately on any operating system able to run Java, such as Windows 10, Mac OS X or Linux distributions.
* The game artificial intelligence engine should feature different levels of depth but overall the decision time for a move should fall within reasonable margins to keep the game flowing.

Chapter 4: Feasibility Study and Choices Taken

# 4.1 Software choices

Selecting the adequate software for each task is vital for the proper development of the project. Therefore, this section aims to show the reasons why the following software was used in the project:

## 4.1.1 Eclipse JEE

*Eclipse* is an open-source IDE that has been proven to work in multiple platforms and operating systems. The main reason for the election of this software specifically was familiarity and compatibility, since the development was going to be done entirely in Mac OS X. Some alternatives to *Eclipse* were *IntelliJ* *IDEA* or *NetBeans*, but both were disregarded due to the lack of significant advantages they would bring for this project, especially when compared to the amount of time that would have to be invested on learning how to use them.

## 4.1.2 GitHub

Every project should include a version control system (VCS) to manage changes that happen to all software configuration items according to the IEEE Std. 828[[26]](#footnote-26) — that describes software configuration management (See 2.2.3.1 Documentation). In this case, Git was the selected system for this purpose.

### 4.1.2.1 Motivation

The main reason to include a VCS for this project was the ability to create branches and modify them individually, without influencing the rest of the project and making sure that the changes could be reverted at any time if the result was not the desired one. Whilst this project could not take advantage of the team features of Git since it was an individual one, the version control features of the same were proven to be fairly effective.

### 4.1.2.2 Folder structure

To standardise the structure of the repository and make it as simple as possible to find the software configuration items within, the repository was subdivided into the following structure:

* *doc:* This folder includes documents, guides and resources that could be valuable for writing the report as well as useful material with theoretical information on different parts of the projects, such as MinMax or AlphaBeta[[27]](#footnote-27). Some screenshots from the game have also been included here.
* *img:* It contains the images used to represent the different chess pieces in the game.
* *lib:* Includes the Commons-CLI[[28]](#footnote-28) library, used in the original version of the game API, but not in the final chess game.
* *releases:* It includes some different playable iterations of the game; however, they may not have all final features. They were saved for testing purposes mainly.
* *src:* This folder contains all the source code for the Java classes used in the game on its latest version. The binaries (*bin*) folder is not included in the repository however.

## 4.1.3 Google Drive

In order to minimise risks, measures were taken to assure that no configuration management items were lost during the development. In addition to the Git repository having the changes committed regularly, — every time a relatively significant change was made — the latest version of all files was continuously backed up to the cloud thanks to Google Drive and additionally synchronised with two more backup devices on a daily basis. This way, the chance of any item being lost was remote.

# 4.2 Model choices

While the architecture used was based on the original game engine, there were some modifications made to the structure of certain classes as well as creation of new ones. The most affected elements for this purpose were the board and the pieces.

## 4.2.1 New board design

… Show hard coded values for the board design interface, possibly in 4.7.

## 4.2.2 New piece design

The original API was created to support games that used only one type of piece per player. Players were originally identified by a name and a colour as it is shown in Figure 3. Ataxx being played in the original project.

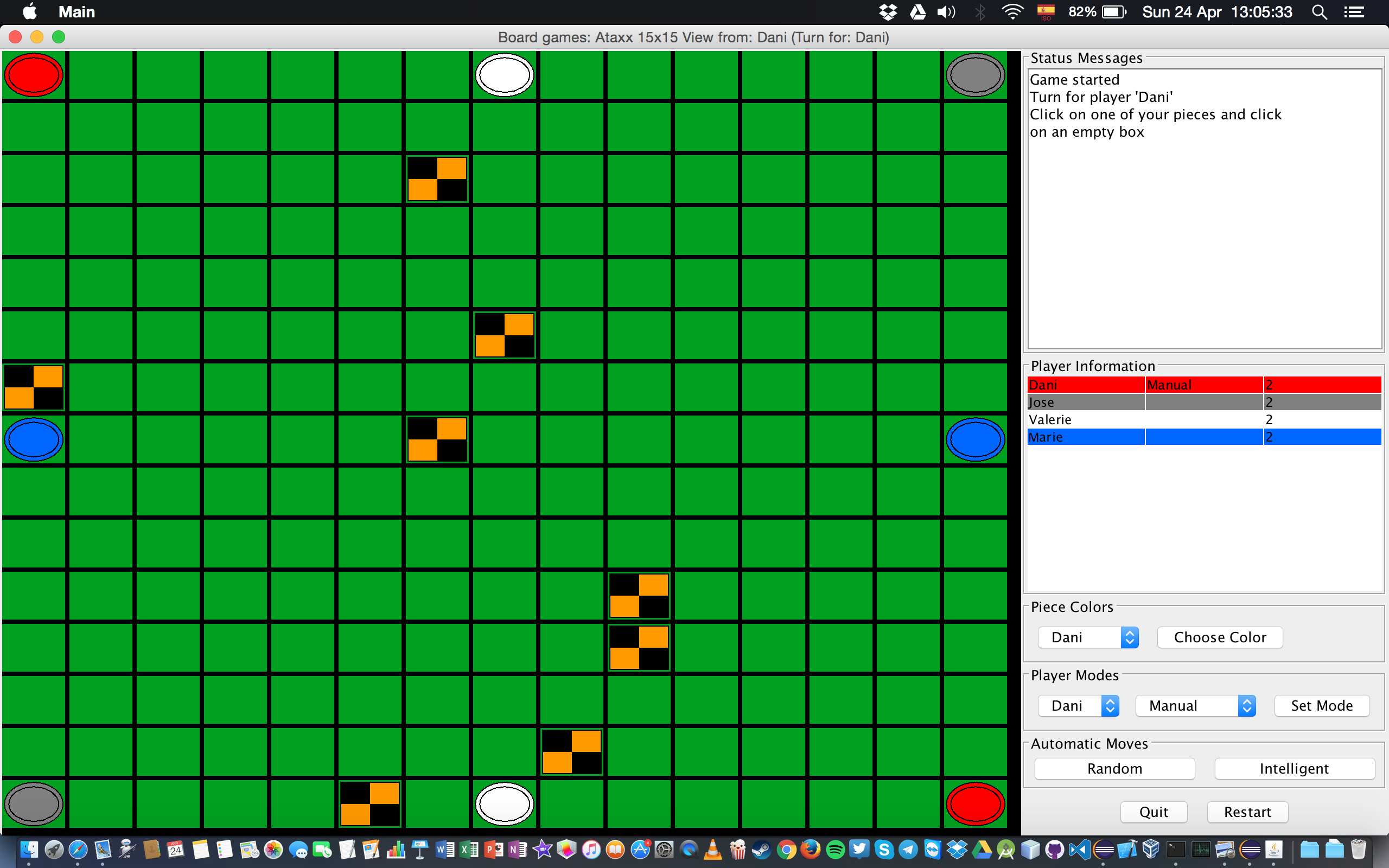


Figure 3. Ataxx being played in the original project

The original games implemented in the API could feature an unlimited number of players, as long as the rules for the game allowed to do so. For example, in Ataxx, dimensions of the board could be modified and so could the number of players, therefore, it was possible to have potentially infinitely many players, as long as the *AtaxxRules* (Which in Chess were represented in the *ChessRules class, see* 6.2.4 ChessRules) class allowed to do so and the board dimensions were large enough to fit all the initial pieces.

Chess however, is a completely different game that requires a set of different pieces per player that are unique and different from each other depending on certain attributes, such as colour, type of piece, ability to castle (for Rooks), to perform an opening move (for Pawns), etc. Additionally, chess only allows two players and each piece must start in a certain predetermined position every game – although chess variations with different starting positions, such as Fischer random chess[[29]](#footnote-29) could of course be easily implemented by slightly modifying the *ChessRules* class.

# 4.3 Programming language choices

The original game API was written in Java, and all the additions done in the second year of my degree used this language as well, therefore, it was a good idea to continue using this language, especially because of the familiarity with the Java language acquired in the development of the original game.

Additionally, since one of the requirements (See 3.1 Functional Requirements) was to create an app that worked in several different operating systems, this was an important factor as to why Java was chosen for the development. An example of the game working in Windows can be seen at Figure 4.

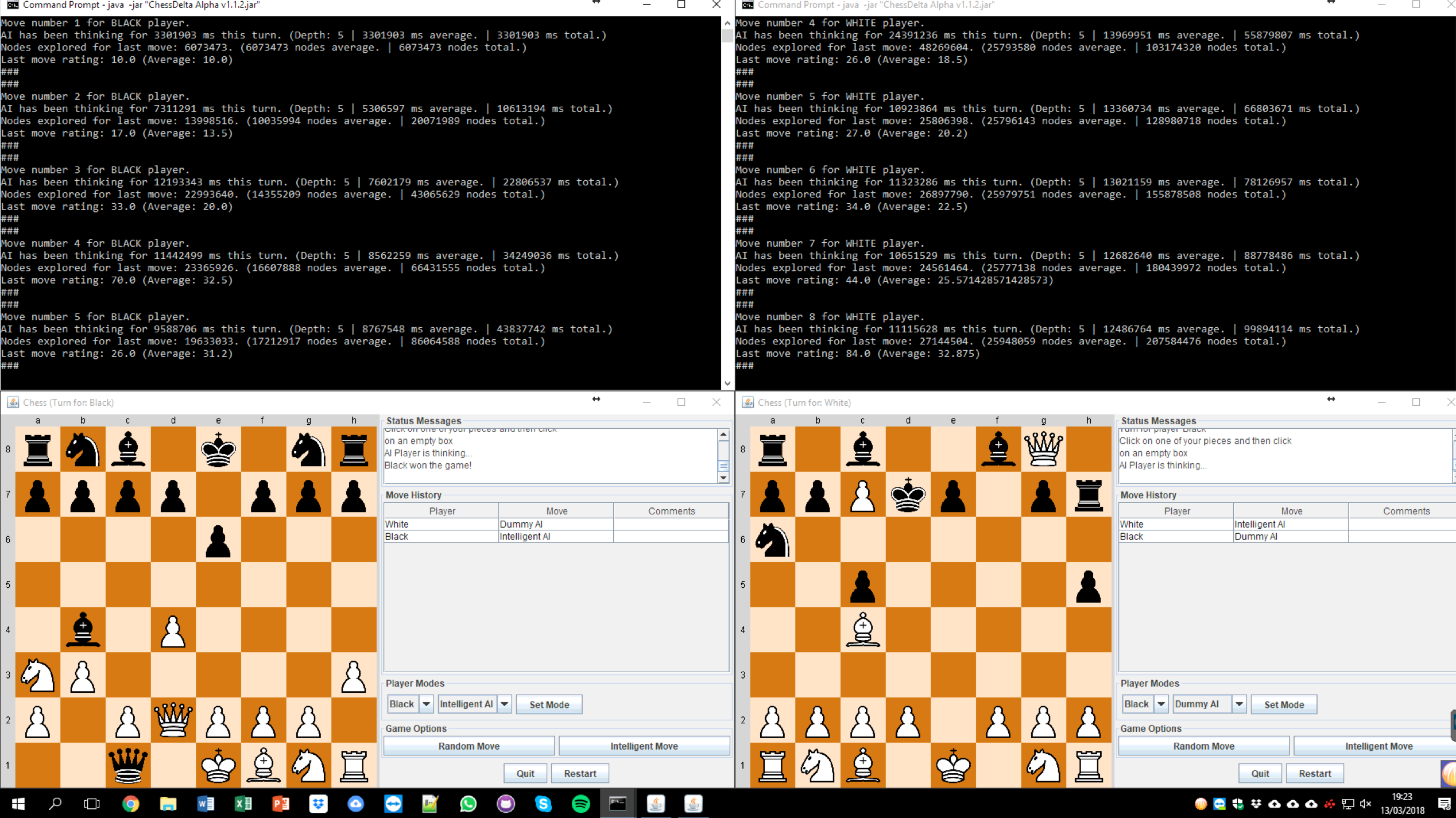


Figure 4. ChessDelta played on Windows 10. MinMax playing a match as white and a match as black simultaneously

# 4.4 Other choices

Over 2,000 variants of chess are estimated to exist[[30]](#footnote-30), however, for this development, the version of chess that is going to be developed is the standard one (See Figure 4), although it must be said that most the variants that can be played on a rectangular board with the regular pieces – such as Chess960, Upside-down Chess or Dunsany’s Chess – can very easily be recreated on Chess Delta by simply modifying the *ChessRules* class. This idea will be extended towards the end of the report at 10.2 Future of the game.



Figure 5. Chess Delta first implementation of the board and pieces. Pieces were not yet able to move, and each piece was treated as a different player, as can be seen on the Player information board on the right side.

# 4.4 Time and resources estimation

Given that the project started in mid-October 2017, the requirements were adjusted to match about six months of development, although the University of Hertfordshire only required the development to start in the second semester of the final year. The final Gantt Chart for the development can be found below on Figure 5. Gantt Chart Time Estimation.

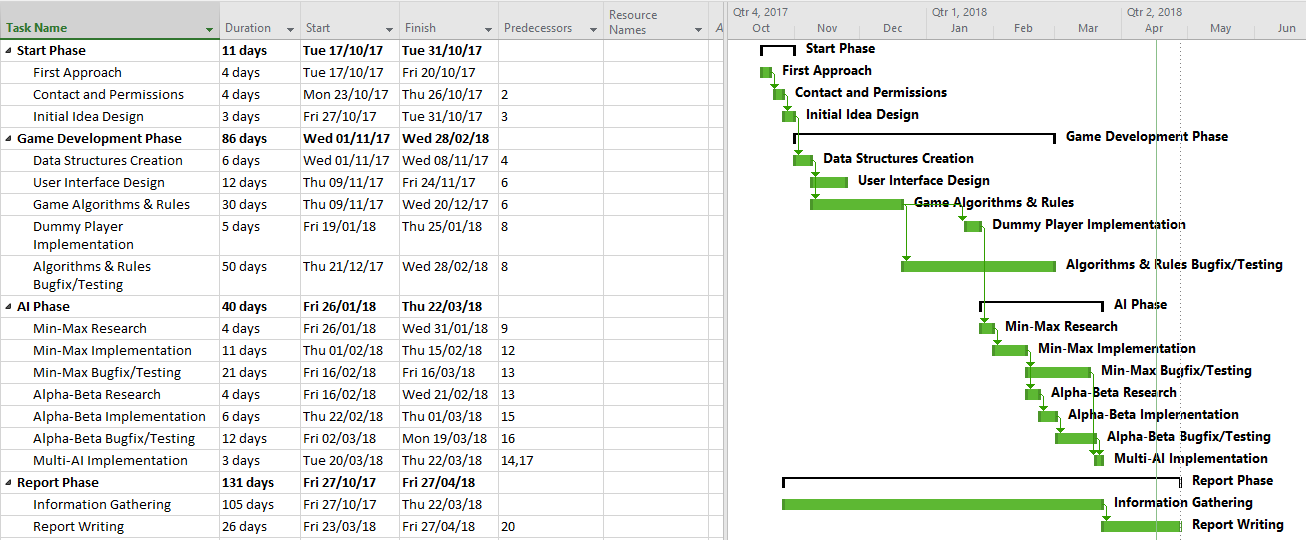


Figure 6. Gantt Chart Time Estimation

This diagram will be contrasted with the initial estimations made under section 8.1 Evaluation Overview.

# 4.5 Feasibility of the project

Although the initial project proposal (See 1.3 Objectives and 8.1 Evaluation Overview) is fairly ambitious, it should be noted that as mentioned before, not all features must be implemented and the development will end when time runs out. This does not mean that the project will be by any means lacklustre, but instead, it is a realisation that the initial objectives are potentially unrealistic and more than sufficient for a BSc thesis, and therefore, a project that comprised only half of them, would still be a very successful project, while also taking into account that the remaining features could be added to the game at a later time.

# 4.6 Testing choices

At the beginning of the project, Dr Daniel Polani suggested that it was a good idea to test MinMax by making it play a simple game called Nim[[31]](#footnote-31). However, after some difficulties in the development and debugging that took longer than expected, it was decided not to implement Nim to save time. Finally, the way MinMax was tested was by performing system tests after several exhaustive unit tests to check that all components of the AI were working as intended and returning the expected values.

By the end of the development, over 100 MinMax and AlphaBeta AI matches had been played without any significant bugs and at least 44 of them have been documented and saved for statistical and game improvement purposes. The statistics and an analysis of them can be found at 8.2 Strengths of the Project.

# 4.7 User interface

The user interface for the game was done programmatically in its entirety and no assistant was used for it. The main reason was that the original game views were made using the Swing library to draw shapes and form a board, however, Chess Delta took this a step further when it was required to create a substantially different board and pieces by making use of images to represent the different pieces. Many parts of the original design were reused but suffered important modifications. More details on the user interface and the implementation of the same can be found at 6.2.6 ChessViews Package and 6.3 User Interface.

Chapter 5: Software Design, Architecture & Diagrams

# 5.1 Overview

Chess Delta is a heavily modularised software that features multitude of design patterns and complies with several design standards and recommended practices in software engineering. It makes use of interfaces to make it as generic as possible to allow new games to be introduced to the game engine easily. While the original games present in the API have been removed to make the project submission as simple as possible, they could be easily added at any given moment by copying the original files to the project workspace.

# 5.2 Diagrams

# 5.3 Design Patterns Used

The original game engine included several design patterns to keep the code organised and no new design patterns were added for the development of Chess Delta. However, understanding how these patterns were applied to the initial API was vital for the implementation of the chess game, since the game API was not initially suitable for such a complex game and some important modifications had to be done to the core of the game. The most relevant design patterns, their implementation and how they affected the project will be described in the following sub-sections.

## 5.3.1 Model-view-controller

…

## 5.3.2 Factory

…

## 5.3.3 Command

…

## 5.3.4 Observer

…

Chapter 6: Components & Implementation

# 6.1 List of Components

Although the final version of the project is composed by about 90 different Java files, most of them are generic classes present in the original version of the game. For the development of the project many of the original files were modified to suit chess. In the following sections the classes and packages that were created from scratch or inheritance from existing ones will be discussed and explained. In total, 27 Java classes were added to the project for the development of Chess Delta, and between the 29th of October 2017 and the 18th of April 2018 a total of 32,189 lines of code have been added and 20,385 lines have been deleted as can be seen from Figure 6. GitHub contribution graph for the repository.

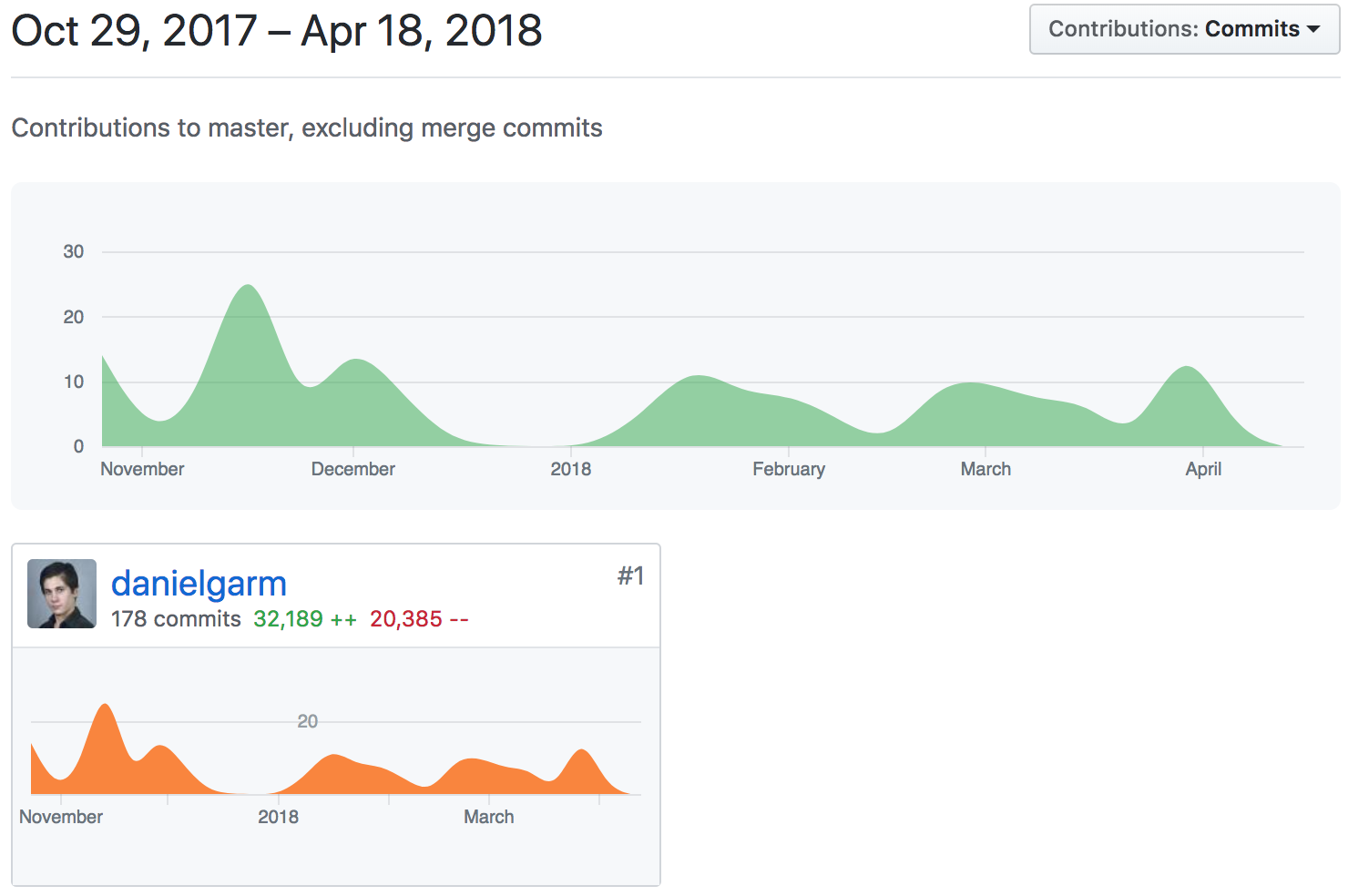


Figure 7. GitHub contribution graph for the repository

The list of all 27 classes can be found on Figure 7. All classes created for Chess Delta highlighted in blue, and a brief explanation of the most relevant and impactful parts of the project can also be found at 6.2 Most relevant classes.



Figure 8. All classes created for Chess Delta highlighted in blue

# 6.2 Most relevant classes

## 6.2.1 ChessAlphaBeta & ChessMinMax

These two classes are the core of the artificial intelligence in the game. They are based on the well-known game theory algorithms researched and discussed on section 2.2.1 Algorithms & Computational Techniques. Although the original package included an implementation of MinMax, this was removed from the package and both algorithms were remade in a way that was as close to my understanding of the concepts as possible, since the original MinMax present in the original game API was not part of the contents of the module where the original code was provided, and I personally never had an understanding of how it worked until I worked on this project.

Therefore, thanks to the research done and the help of my thesis supervisor Dr Daniel Polani, I got a clear understanding of both MinMax and AlphaBeta and created a fully functional and tested implementation of said algorithms.

The implementation of MinMax, AlphaBeta and the heuristics used can be found at Appendix I. MinMax and AlphaBeta code.

## 6.2.2 ChessBoard

*ChessBoard* is arguably one of the most important classes in the whole project. It stores the pieces, the location of each piece on the board and also includes several critical functions of the game, such as detecting whether a King is in Check given a certain board as well as returning a copy of a board when prompted, which is extremely valuable for testing hypothetical moves and scenarios, especially in the case of AI, where the engine needs to think several moves in advance. It also stores copies of the latest states of the board to be able to check for threefold repetition and 50 moves rule (See 2.1.1 Rules and difficulties).

The whole *ChessBoard* class can be found at Appendix II. ChessBoard code.

## 6.2.3 ChessMove

This class is by far the largest in the whole project because of the enormous number of factors that need to be checked to perform a move in Chess, although a possible solution for this is discussed at 10.3 Improvements to existing features.

Together with 6.2.4 ChessRules, *ChessMove* defines the whole set of rules of a game as we understand them. While *ChessRules* contains the higher-level rules, such as where the pieces are placed at the beginning of a match or when a player has won, *ChessMove* defines if a move is valid or not given a certain state of the board. For instance, *ChessMove* makes sure that a player doesn’t perform illegal moves, such as moving a Pawn backwards, trying to capture an ally piece, leaving your King in check after a move or moving a Rook diagonally. This class is also in charge of throwing exceptions to display messages to the player as to why a move has not been performed. This can be seen in Figure 8. Different exceptions for invalid moves.

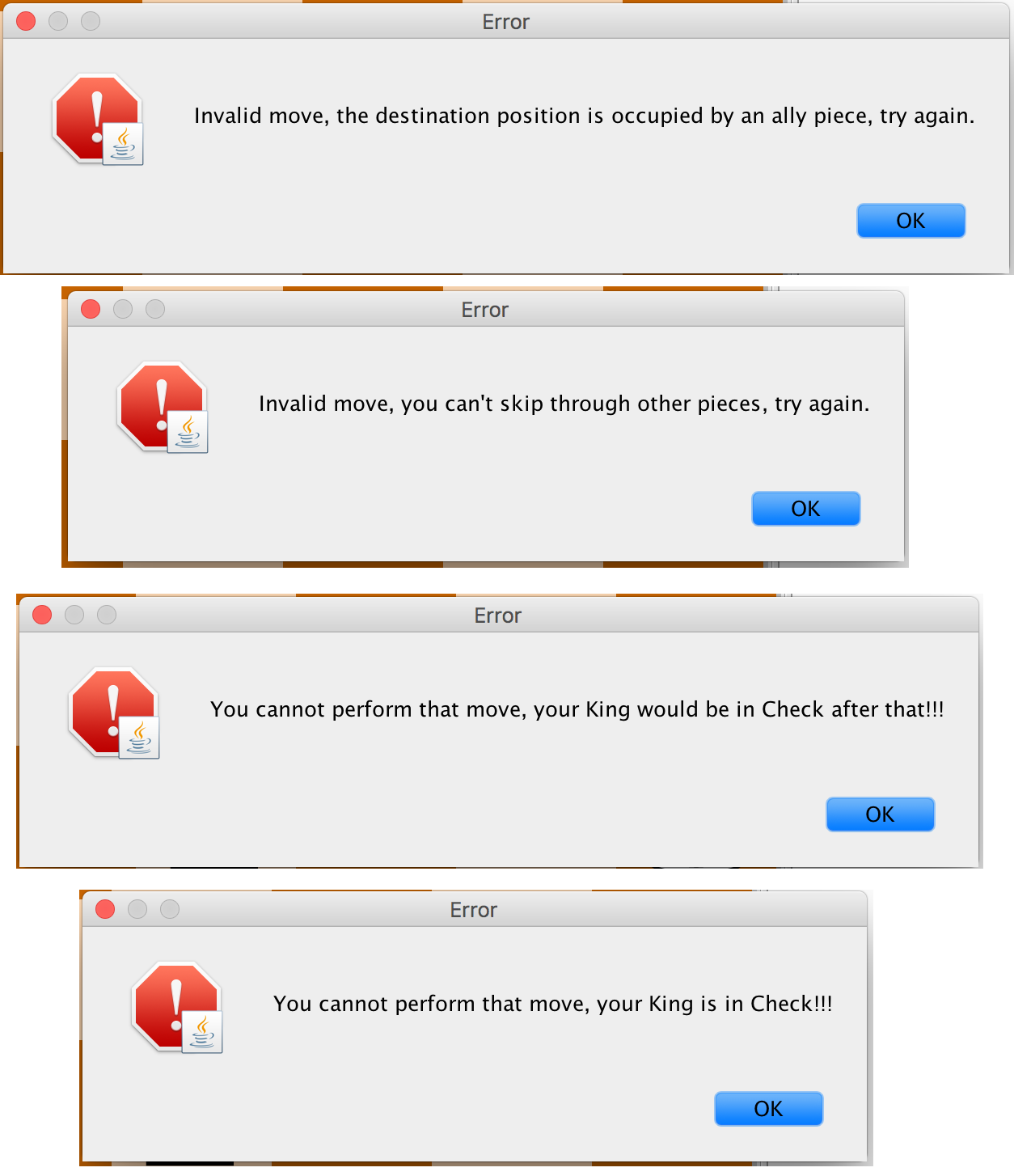


Figure 9. Different exceptions for invalid moves

Additionally, this class must handle Pawn promotion (As seen in Figure 9. Pawn promotion dialog) and also take into account special moves, such as Castling or En-Passant capture of a Pawn.

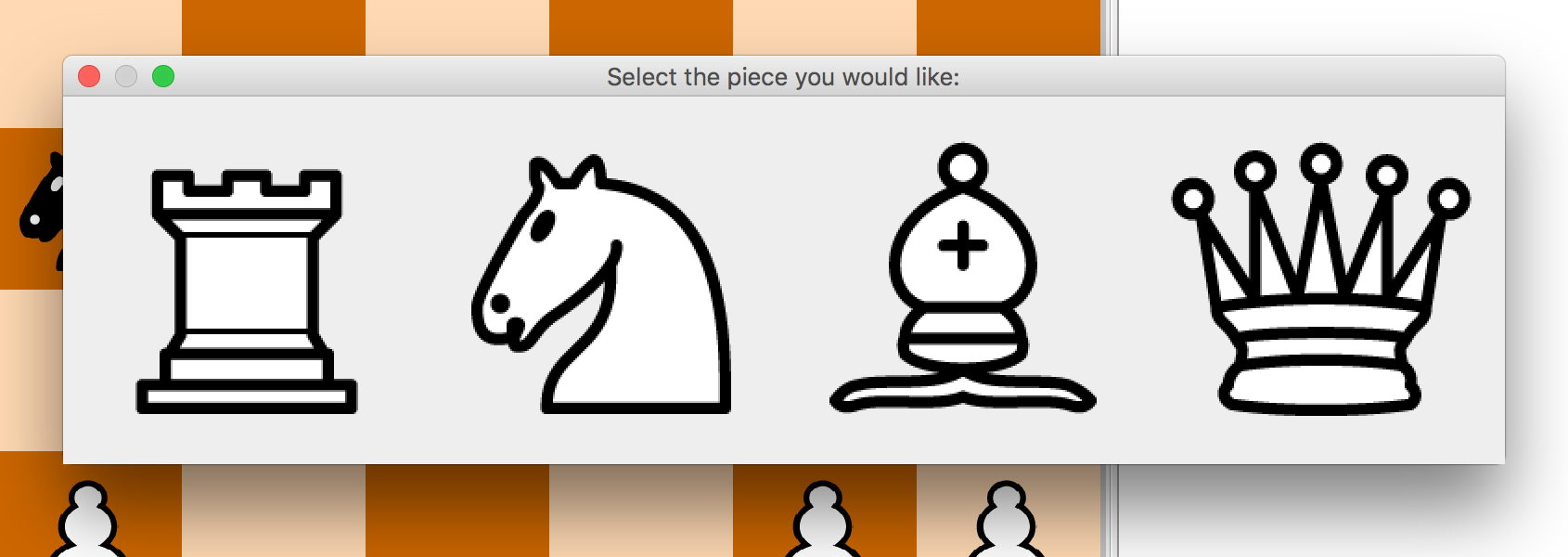


Figure 10. Pawn promotion dialog

The code for this class, can be found at Appendix III. ChessMove code.

## 6.2.4 ChessRules

In contrast with *ChessMove*, this class controls the game flow and serves three purposes mainly:

* Creating the initial board by placing the pieces in the positions they should initially be and determining the initial player, which in classic chess is always the white player. This function can be changed to easily implement chess variants that maintain most of the rules of chess but change the initial positions of the pieces, such as Chess960.
* Returning who the next player to move should be as well as the current state of the game (i.e. White won, Game ended in Draw, Game still in play…).
* Returning a list with all valid moves that a player can perform given a certain state of the board and a player colour.

It is important to note that formerly, the board evaluation heuristics were also part of this class, as it can be seen by the presence of an *evaluate()* function. However, even though this function has not been removed in the final game, it is no longer used and instead, an instance of *ChessBoardEvaluator* should be created for this purpose this is explained in greater depth at 6.2.1 ChessAlphaBeta & ChessMinMax.

The code for *ChessRules* can be found at Appendix IV. ChessRules code.

## 6.2.5 ChessPiece

This class is what defines the pieces used in this game as opposed to the pieces used in any of the games initially present in the API. *ChessPiece* simply extends *Piece* and adds a Boolean attribute called *isWhite*. It defines whether a piece is white or black depending on its value. This wouldn’t be such a big difference from what *Piece* originally was, however, it allows the creation of six other classes that define each one of the pieces used in standard chess, as well as certain important attributes for each of them. For instance, Rooks have an attribute called *canCastle,* that defines whether an instance of a Rook can still castle at a certain point in the game. This attribute is immediately switched to *false* whenever said Rook moves for the first time. The same happens with Pawns – having *En-Passant* and *Opening move* attributes – and Kings having *canCastle* attributes that toggle off whenever they move for the first time.

The code for *ChessPiece, Bishop, King, Knight, Pawn, Queen* and *Rook* is available atAppendix V. Chess pieces code*.*

## 6.2.6 ChessViews Package

One of the most time-consuming parts of creating this game was coding the views for it. Said views are completely hand-coded, without help from any third-party assistant such as NetBeans, mostly because the original views created by me in the second year of the degree for the API were done this way and could be partially reused for chess.

The only thing the games from the original API had in common with chess was that they were played in a rectangular board. However, chess has a completely different board design, composed by light and dark squares, the pieces were images now (See 6.3 User Interface) to represent the different piece types, rather than Swing shapes as they previously were, and the side menus needed to be different as well.

The structure for these classes was similar to the one in the original games, but the functions were mostly remade from scratch so that they would match the needs for a chess game.

The two main classes for this purpose are *ChessWindowSwingView* and *ChessBoardComponent.* Their scope can be more clearly seen at Figure 10.



Figure 11. The yellow rectangle represents ChessWindowSwingView’s scope, while the red one represents what ChessBoardComponent is in charge of

The code for both classes can be found at Appendix VI. Chess Views code. These classes are not as easily readable as the other ones in the game, mostly because of the amount of hard-coded values that needed to be used, nonetheless, the game still adapts to different screen shapes and sizes without issues.

# 6.3 User Interface

Chapter 7: Testing

# 7.1 Unit tests

# 7.2 toString() debugging

# 7.3 Nim

# 7.4 Chess end-games

# 7.5 Playing against human subjects

Chapter 8: Project Evaluation

# 8.1 Evaluation Overview

Although the project has been an overall success, there have been certain issues during the development, especially regarding planning and features to be included in the final version of the project. One notable example of this was the initial estimation for the tasks and allocated timeslots for each of them. As shown below on Figure 11. Original time and features estimation, this was the original feature planning and time estimation for each activity. This has been extracted from my submission to the Assessment number 3 for the *Project Planning* module where a preliminary evaluation and estimation of the project needed to be done. Nonetheless, the actual final features and time allocated for each one of them can be found at Figure 5. Gantt Chart Time Estimation.

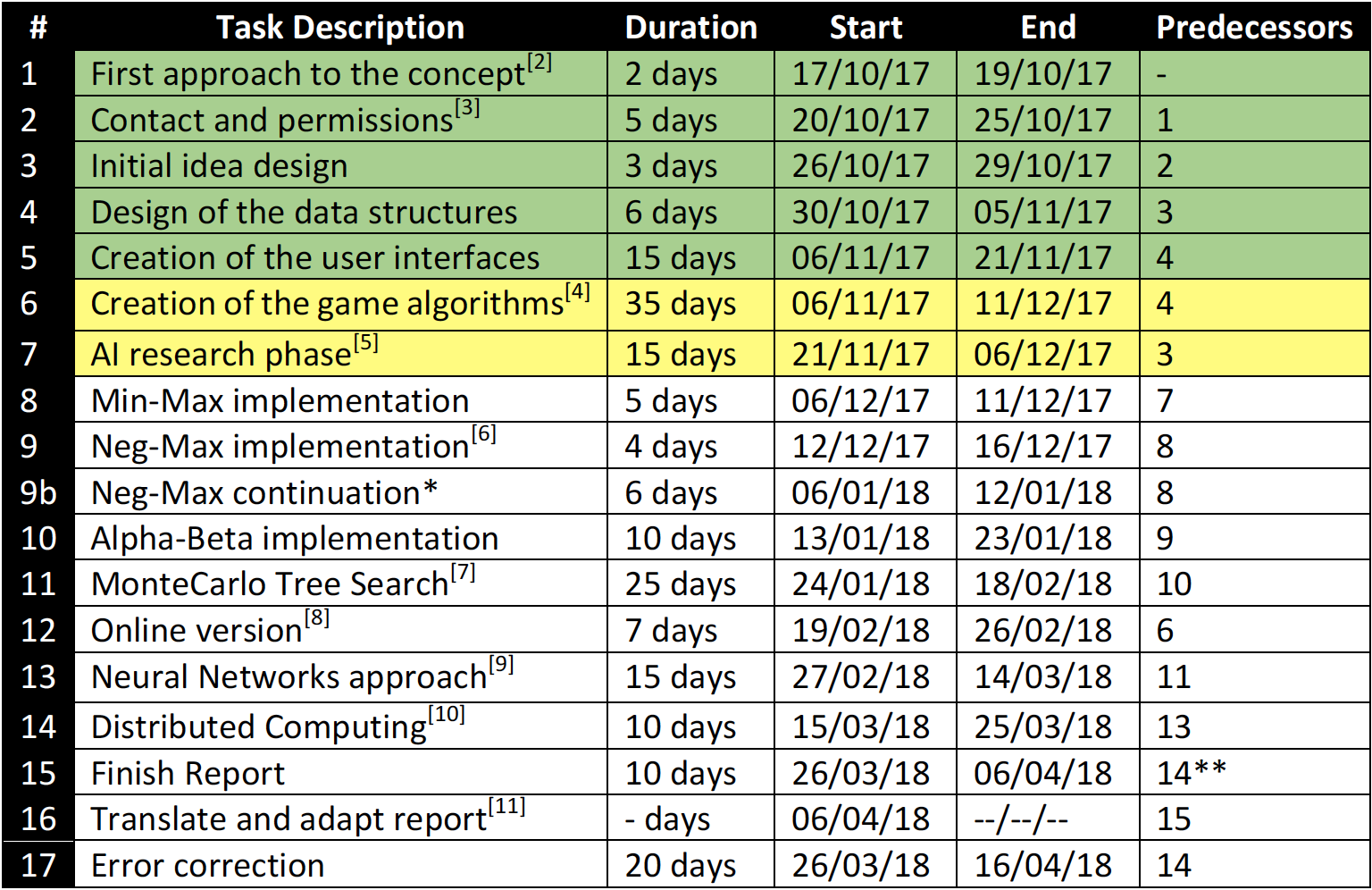


Figure 12. Original time and features estimation

This first time estimation was objectively overly optimistic, especially considering the number of other projects that had to be done during the development of Chess Delta.

The most time-consuming activity while developing the game was proven to be testing and debugging the game, which, as shown above, was missing in the original estimation and was assumed to be part of each task of the development, which was clearly a mistake.

# 8.2 Strengths of the Project

Talk about statistics from Excel. Mentioned at 4.6 Testing choices.

# 8.3 Problems encountered

## 8.3.1 Critical problems

## 8.3.2 Other problems and solutions

Include clicking outside cells causing user to click a different cell.

# 8.4 Potential improvements

Things that could have been done better

Chapter 9: Conclusions

# 9.1 Conclusions Overview

# 9.2 Lessons learnt

# 9.3 Should (not) have done

# 9.4 Personal opinion

Chapter 10: Future Work

# 10.1 State of the project

Why it will continue.

* University (UCM)
* Research on different chess variants
* Keep the API as reusable as it was but with a focus on chess.

# 10.2 Future of the game

Talk about creating chess variants.

# 10.3 Improvements to existing features

Talk about moving move checks inside each piece. (ChessMove to Pawn for instance)

# 10.4 Expectations

Chapter 11: Bibliography

Chapter 12: Appendices

*IMPORTANT: All code found in the appendices can also be found at the GitHub repositories and will be linked for better readability.*

# Appendix I. MinMax and AlphaBeta code

## Appendix I-a. MinMax

<https://github.com/danielgarm/Chess-Delta/blob/master/src/chess/game/mvc/model/chessFiles/chessAI/ChessMinMax.java>

|  |
| --- |
| public class ChessMinMax implements AIAlgorithm {  private static final long serialVersionUID = 556625192621284461L;  private int level; //How hard it is to win against the AI. The higher this number, the more intelligent it becomes.  private BasicChessBoardEvaluator evaluator;  private AIStatistics aiStats; //Instance of the class to create the statistics  private GameRules rules; //Rules of chess  private List<Piece> pieces;  private int minID;  private int maxID;  private ChessAINode bestNode;  private ChessPiece maxPiece;  private ChessPiece minPiece;  private long nodesExplored;    public ChessMinMax() {  this.level = ChessStatic.DEFAULT\_MINMAX\_LEVEL;  this.evaluator = new BasicChessBoardEvaluator();  this.aiStats = new AIStatistics(ChessStatic.DEFAULT\_MINMAX\_LEVEL, "MinMax");  }    public ChessMinMax(int level) {  this.level = level;  this.evaluator = new BasicChessBoardEvaluator();  this.aiStats = new AIStatistics(level, "MinMax");  }    @Override  public GameMove getMove(Piece p, Board board, List<Piece> playersPieces, List<Piece> pieceTypes, GameRules rules) {  this.rules = rules;  this.pieces = playersPieces;  this.bestNode = new ChessAINode();  this.maxID = (((ChessPiece) p).getWhite() ? ChessStatic.WHITE\_ID : ChessStatic.BLACK\_ID); //Takes the max piece ID  this.minID = ((maxID == ChessStatic.WHITE\_ID) ? ChessStatic.BLACK\_ID : ChessStatic.WHITE\_ID); //Takes the min piece ID  this.maxPiece = (ChessPiece) this.pieces.get(this.maxID);  this.minPiece = (ChessPiece) this.pieces.get(this.minID);  this.nodesExplored = 0;  this.aiStats.setColour(((ChessPiece) p).getWhite());  return minMax(board, ChessStatic.STARTING\_DEPTH).getMove(); //Calls minMax for a move!  }    /\*\*  \* Initialises the statistics and calls max for a move. This is always called by the max piece, therefore, the call is always started on max()  \* @param board: Current board  \* @param depth: Maximum depth to be explored  \* @return The best node  \*/  private ChessAINode minMax(Board board, int depth) {  this.aiStats.increaseMoves();    final long startTime = System.currentTimeMillis();  double moveRating = max((ChessBoard) board, depth);  final long thinkingTime = System.currentTimeMillis() - startTime;    this.aiStats.addNodesExplored(this.nodesExplored);  this.aiStats.addRating(moveRating);  this.aiStats.addThinkingTime(thinkingTime);    System.out.println(this.aiStats.getStats());    if(this.bestNode.getMove() == null || this.bestNode == null) {  List<GameMove> validMoves = this.rules.validMoves(board, this.pieces, this.pieces.get(this.maxID));  this.bestNode.changeNode((ChessMove) validMoves.get(Utils.randomInt(validMoves.size())), -1000);  }    /\*ChessBoard newBoard = (ChessBoard) board.copy();  this.bestNode.getMove().executeAICheckedMove(newBoard);  System.out.println(newBoard);  System.out.println("Rating: " + this.bestNode.getRating());\*/ //For debugging purposes    return this.bestNode;  }    private double min(Board board, int depth) {  Pair<State, Piece> gameState = this.rules.updateState(board, pieces, this.minPiece); //Gets the state of the game    this.nodesExplored++;    if(gameState.getFirst().equals(State.Won)) { //If someone won the game  if(gameState.getSecond().equals(this.minPiece)) { //If min player won the game  return 1000;  } else { //If max player won the game  return -1000;  }  } else if(gameState.getFirst().equals(State.Draw)) { //If there is a draw  return 0;  } else if(depth == this.level) { //If the maximum depth has been reached, the board evaluation is simply returned  return this.evaluator.getRating((ChessBoard) board, (ChessPiece) this.pieces.get(this.minID), this.maxPiece);  } else { //In any other case, the game is still in play and the maximum depth has not been reached, therefore, it continues iterating  List<GameMove> validMoves = this.rules.validMoves(board, this.pieces, this.pieces.get(this.minID)); //Get list of all valid possible moves  double lowestInBranch = Double.MAX\_VALUE;  for(GameMove move : validMoves) { //For each valid move...  ChessBoard testBoard = (ChessBoard) board.copy();  ((ChessMove) move).executeAICheckedMove(testBoard); //The move is executed    if(depth == ChessStatic.STARTING\_DEPTH) { //ONLY saves the movements in the case when the depth is 0, moves from a higher depth are not relevant, only the rating is  double currentNodeRating = max(testBoard, depth + 1);  if(currentNodeRating <= lowestInBranch) { //Could be equal in case of loss or win if implemented as infinite rating  lowestInBranch = currentNodeRating; //This is only reached if the current node is the lowest (or equal), therefore it's changed  this.bestNode.changeNode((ChessMove) move, currentNodeRating); //The best node so far is also changed  }  } else {  //If the depth is not 0, there is no need to save the movement, only the rating of it, therefore it is calculated by taking the returned result of max() and comparing it to the lowest in the branch  lowestInBranch = Math.min(max(testBoard, depth + 1), lowestInBranch);  }  }  return lowestInBranch;  }  }    private double max(Board board, int depth) {  Pair<State, Piece> gameState = this.rules.updateState(board, pieces, this.maxPiece);    this.nodesExplored++;    if(gameState.getFirst().equals(State.Won)) {  if(gameState.getSecond().equals(this.maxPiece)) {  return 1000;  } else {  return -1000;  }  } else if(gameState.getFirst().equals(State.Draw)) {  return 0;  } else if(depth == this.level) {  return this.evaluator.getRating((ChessBoard) board, (ChessPiece) this.pieces.get(this.maxID), this.maxPiece);  } else {  List<GameMove> validMoves = this.rules.validMoves(board, this.pieces, this.pieces.get(this.maxID));  double highestInBranch = -Double.MAX\_VALUE;  for(GameMove move : validMoves) {  ChessBoard testBoard = (ChessBoard) board.copy();  ((ChessMove) move).executeAICheckedMove(testBoard);    if(depth == ChessStatic.STARTING\_DEPTH) { //ONLY saves the movements in the case when the depth is 0, moves from a higher depth are not relevant, only the rating is  double currentNodeRating = min(testBoard, depth + 1);  if(currentNodeRating >= highestInBranch) { //Could be equal in case of loss or win if implemented as infinite rating.  highestInBranch = currentNodeRating;  this.bestNode.changeNode((ChessMove) move, currentNodeRating);  }  } else {  highestInBranch = Math.max(min(testBoard, depth + 1), highestInBranch);  }  }  return highestInBranch;  }  }  } |

## Appendix I-b. AlphaBeta

<https://github.com/danielgarm/Chess-Delta/blob/master/src/chess/game/mvc/model/chessFiles/chessAI/ChessAlphaBeta.java>

|  |
| --- |
| public class ChessAlphaBeta implements AIAlgorithm {  private static final long serialVersionUID = 556625192621284461L;  private int level; //How hard it is to win against the AI. The higher this number, the more intelligent it becomes.  private BasicChessBoardEvaluator evaluator;  private AIStatistics aiStats;  private GameRules rules;  private List<Piece> pieces;  private int minID;  private int maxID;  private ChessAINode bestNode;  private ChessPiece maxPiece;  private ChessPiece minPiece;  private long nodesExplored;    public ChessAlphaBeta() {  this.level = ChessStatic.DEFAULT\_MINMAX\_LEVEL;  this.evaluator = new BasicChessBoardEvaluator();  this.aiStats = new AIStatistics(ChessStatic.DEFAULT\_MINMAX\_LEVEL, "AlphaBeta");  }    public ChessAlphaBeta(int level) {  this.level = level;  this.evaluator = new BasicChessBoardEvaluator();  this.aiStats = new AIStatistics(level, "AlphaBeta");  }    @Override  public GameMove getMove(Piece p, Board board, List<Piece> playersPieces, List<Piece> pieceTypes, GameRules rules) {  this.rules = rules;  this.pieces = playersPieces;  this.bestNode = new ChessAINode();  this.maxID = (((ChessPiece) p).getWhite() ? ChessStatic.WHITE\_ID : ChessStatic.BLACK\_ID);  this.minID = ((maxID == ChessStatic.WHITE\_ID) ? ChessStatic.BLACK\_ID : ChessStatic.WHITE\_ID);  this.maxPiece = (ChessPiece) this.pieces.get(this.maxID);  this.minPiece = (ChessPiece) this.pieces.get(this.minID);  this.nodesExplored = 0;  this.aiStats.setColour(((ChessPiece) p).getWhite());  return alphaBeta(board, ChessStatic.STARTING\_DEPTH, -1000, +1000).getMove();  }    private ChessAINode alphaBeta(Board board, int depth, double alpha, double beta) {  this.aiStats.increaseMoves();    final long startTime = System.currentTimeMillis();  double moveRating = max((ChessBoard) board, depth, alpha, beta);  final long thinkingTime = System.currentTimeMillis() - startTime;    this.aiStats.addNodesExplored(this.nodesExplored);  this.aiStats.addRating(moveRating);  this.aiStats.addThinkingTime(thinkingTime);    System.out.println(this.aiStats.getStats());    if(this.bestNode.getMove() == null || this.bestNode == null) {  List<GameMove> validMoves = this.rules.validMoves(board, this.pieces, this.pieces.get(this.maxID));  this.bestNode.changeNode((ChessMove) validMoves.get(Utils.randomInt(validMoves.size())), -1000);  }    /\*ChessBoard newBoard = (ChessBoard) board.copy();  this.bestNode.getMove().executeAICheckedMove(newBoard);  System.out.println(newBoard);  System.out.println("Rating: " + this.bestNode.getRating());\*/    return this.bestNode;  }    /\*\*  \* Alpha is changed in max if the value is higher than Alpha.  \* Beta is changed in min if the value is lower than Beta.  \*  \* Alpha means don't search further if you find a lower value.  \* Beta means don't search further if you find a higher value.  \* Alpha is the minimum guaranteed value for max.  \* Beta is the max guaranteed value for min.  \*/  private double min(Board board, int depth, double alpha, double beta) {  Pair<State, Piece> gameState = this.rules.updateState(board, pieces, this.minPiece);    this.nodesExplored++;    //This part is similar to MinMax  if(gameState.getFirst().equals(State.Won)) {  if(gameState.getSecond().equals(this.minPiece)) {  return 1000;  } else {  return -1000;  }  } else if(gameState.getFirst().equals(State.Draw)) {  return 0;  } else if(depth == this.level) {  return this.evaluator.getRating((ChessBoard) board, (ChessPiece) this.pieces.get(this.minID), this.maxPiece);  } else if(this.bestNode.getRating() < 1000 && this.bestNode.getRating() > -1000) {  List<GameMove> validMoves = this.rules.validMoves(board, this.pieces, this.pieces.get(this.minID));  double lowestInBranch = Double.MAX\_VALUE;  for(GameMove move : validMoves) {  ChessBoard testBoard = (ChessBoard) board.copy();  ((ChessMove) move).executeAICheckedMove(testBoard);    if(depth == ChessStatic.STARTING\_DEPTH) { //ONLY saves the movements in the case when the depth is 0, moves from a higher depth are not relevant, only the rating is  double currentNodeRating = max(testBoard, depth + 1, alpha, beta);  if (currentNodeRating < beta) {  beta = currentNodeRating; //If the value of the current node is higher lower than beta, it becomes the new beta  lowestInBranch = currentNodeRating;  this.bestNode.changeNode((ChessMove) move, currentNodeRating);  }  } else {  double currentNodeRating = max(testBoard, depth + 1, alpha, beta);  if (currentNodeRating < beta) {  beta = currentNodeRating; //If the value of the current node is higher lower than beta, it becomes the new beta  lowestInBranch = currentNodeRating;  }  }  if(alpha >= beta) { //Whenever alpha is equal or higher than beta, there's no need to keep searching, since the explored nodes will never be selected  return lowestInBranch;  }  }  return lowestInBranch;  } else {  return this.bestNode.getRating();  }  }    /\*\*  \* As in MinMax, max is fairly similar to min, with some key differences.  \* @param board  \* @param depth  \* @param alpha  \* @param beta  \* @return  \*/  private double max(Board board, int depth, double alpha, double beta) {  Pair<State, Piece> gameState = this.rules.updateState(board, pieces, this.maxPiece);    this.nodesExplored++;    if(gameState.getFirst().equals(State.Won)) {  if(gameState.getSecond().equals(this.maxPiece)) {  return 1000;  } else {  return -1000;  }  } else if(gameState.getFirst().equals(State.Draw)) {  return 0;  } else if(depth == this.level) {  return this.evaluator.getRating((ChessBoard) board, (ChessPiece) this.pieces.get(this.maxID), this.maxPiece);  } else if(this.bestNode.getRating() < 1000 && this.bestNode.getRating() > -1000) {  List<GameMove> validMoves = this.rules.validMoves(board, this.pieces, this.pieces.get(this.maxID));  double highestInBranch = -Double.MAX\_VALUE;  for(GameMove move : validMoves) {  ChessBoard testBoard = (ChessBoard) board.copy();  ((ChessMove) move).executeAICheckedMove(testBoard);    if(depth == ChessStatic.STARTING\_DEPTH) { //ONLY saves the movements in the case when the depth is 0, moves from a higher depth are not relevant, only the rating is  double currentNodeRating = min(testBoard, depth + 1, alpha, beta);  if(currentNodeRating > alpha) {  alpha = currentNodeRating;  highestInBranch = currentNodeRating;  this.bestNode.changeNode((ChessMove) move, currentNodeRating);  }  } else {  double currentNodeRating = min(testBoard, depth + 1, alpha, beta);  if (currentNodeRating > alpha) {  alpha = currentNodeRating;  highestInBranch = currentNodeRating;  }  }  if(alpha >= beta) {  return highestInBranch;  }  }  return highestInBranch;  } else {  return this.bestNode.getRating();  }  }  } |

## Appendix I-c. Board evaluation heuristics

<https://github.com/danielgarm/Chess-Delta/blob/master/src/chess/game/mvc/model/chessFiles/chessAI/BasicChessBoardEvaluator.java>

|  |
| --- |
| public class BasicChessBoardEvaluator implements ChessBoardEvaluator {    public BasicChessBoardEvaluator() {}    public double getRating(ChessBoard board, ChessPiece currentPiece, ChessPiece maxPiece) {  if(board.checkMovesRulesLimit()) {  return 0;  }    double cumulativeRating = 0;    boolean whiteInCheck = isWhiteInCheck(board);  boolean blackInCheck;    if(!whiteInCheck) {  blackInCheck = isBlackInCheck(board);  } else {  blackInCheck = false; //It's not possible to have both players in check at the same time.  }    if(whiteInCheck) { //If there is a player in check  if(currentPiece.getWhite()) {  cumulativeRating += 10;  } else {  cumulativeRating -= 10;  }  } else if(blackInCheck) {  if(!currentPiece.getWhite()) {  cumulativeRating += 10;  } else {  cumulativeRating -= 10;  }  }    cumulativeRating += rateBoardByPieces(board, maxPiece.getWhite()); //Adds the max player board rating by pieces.    return cumulativeRating;  }    /\*  \* HEURISTICS:  \* Pawn is worth exponentially more more the closer to promoting it is, e.g 2^(rowNumber - 2) --> 5, 6, 7, 8, 13, 28, 45 (Last row means there is a queen)  \* Proportionally, pieces are worth: Queen = 45, Rook = 25, Bishop = 15, Knight = 15, Pawn = 5  \* Those are only base values, since they can be higher if they're in a good position at the board.  \*/  private double rateBoardByPieces(ChessBoard board, boolean isMaxWhite) {  double rating = 0;  for(int rowX = ChessStatic.MIN\_DIM; rowX <= ChessStatic.MAX\_DIM; rowX++) {  for(int colY = ChessStatic.MIN\_DIM; colY <= ChessStatic.MAX\_DIM; colY++) {  if(board.getPosition(rowX, colY) != null) { //If there's a piece  if(((ChessPiece) board.getPosition(rowX, colY)).getWhite() == isMaxWhite) { //If the colour matches the max player...  if(board.getPosition(rowX, colY) instanceof Pawn) {  rating += ratePawn(rowX, colY, ((ChessPiece) board.getPosition(rowX, colY)).getWhite());  } else if(board.getPosition(rowX, colY) instanceof Rook) {  rating += rateRook(rowX, colY, ((ChessPiece) board.getPosition(rowX, colY)).getWhite());  } else if(board.getPosition(rowX, colY) instanceof Knight) {  rating += rateKnight(rowX, colY, ((ChessPiece) board.getPosition(rowX, colY)).getWhite());  } else if(board.getPosition(rowX, colY) instanceof Bishop) {  rating += rateBishop(rowX, colY, ((ChessPiece) board.getPosition(rowX, colY)).getWhite());  } else if(board.getPosition(rowX, colY) instanceof Queen) {  rating += rateQueen(rowX, colY, ((ChessPiece) board.getPosition(rowX, colY)).getWhite());  } else if(board.getPosition(rowX, colY) instanceof King) {  continue; //The King doesn't add any value to the board, since his value would be infinite.  }    //Pieces further away from the sides of the board are usually worth slightly more.  if(colY == 3 || colY == 4) rating += 3;  else if(colY == 2 || colY == 5) rating += 2;  else if(colY == 1 || colY == 6) rating += 1;  } else { //If the colour matches the min player...  if(board.getPosition(rowX, colY) instanceof Pawn) { //If the colour matches the min player...  rating -= ratePawn(rowX, colY, ((ChessPiece) board.getPosition(rowX, colY)).getWhite());  } else if(board.getPosition(rowX, colY) instanceof Rook) {  rating -= rateRook(rowX, colY, ((ChessPiece) board.getPosition(rowX, colY)).getWhite());  } else if(board.getPosition(rowX, colY) instanceof Knight) {  rating -= rateKnight(rowX, colY, ((ChessPiece) board.getPosition(rowX, colY)).getWhite());  } else if(board.getPosition(rowX, colY) instanceof Bishop) {  rating -= rateBishop(rowX, colY, ((ChessPiece) board.getPosition(rowX, colY)).getWhite());  } else if(board.getPosition(rowX, colY) instanceof Queen) {  rating -= rateQueen(rowX, colY, ((ChessPiece) board.getPosition(rowX, colY)).getWhite());  } else if(board.getPosition(rowX, colY) instanceof King) {  continue; //The King doesn't add any value to the board, since his value would be infinite.  }    //Pieces further away from the sides of the board are usually worth slightly more.  if(colY == 3 || colY == 4) rating -= 3;  else if(colY == 2 || colY == 5) rating -= 2;  else if(colY == 1 || colY == 6) rating -= 1;  }  }  }  }    return rating;  }    private double ratePawn(int rowX, int colY, boolean isWhite) {  double rating = 0;  if(isWhite) { //If it's a white pawn...  switch(rowX) {  case 0: rating += 45; break;  case 1: rating += 28; break;  case 2: rating += 13; break;  case 3: rating += 8; break;  case 4: rating += 7; break;  case 5: rating += 6; break;  case 6: rating += 5; break;  case 7: throw new GameError("A white pawn can't be on the bottom row!");  default: throw new GameError("Unexpected row!");  }  } else { //If it's a black pawn...  switch(rowX) {  case 0: throw new GameError("A black pawn can't be on the top row!");  case 1: rating += 5; break;  case 2: rating += 6; break;  case 3: rating += 7; break;  case 4: rating += 8; break;  case 5: rating += 13; break;  case 6: rating += 28; break;  case 7: rating += 45; break;  default: throw new GameError("Unexpected row!");  }  }    return rating;  }    private double rateRook(int rowX, int colY, boolean isWhite) {  double rating = 25;  if(isWhite) {  if(rowX == ChessStatic.MAX\_DIM || rowX == ChessStatic.MAX\_DIM - 1) //If it's white and is on the bottom row it's worth less because it's usually trapped.  rating -= 4;  } else {  if(rowX == ChessStatic.MIN\_DIM || rowX == ChessStatic.MIN\_DIM + 1) //The opposite is true for black pieces  rating -= 4;  }  return rating;  }    private double rateKnight(int rowX, int colY, boolean isWhite) {  double rating = 15;  if(isWhite) {  if(rowX == ChessStatic.MAX\_DIM) //If it's white and is on the bottom row it's worth less because it's usually trapped.  rating -= 3;  } else {  if(rowX == ChessStatic.MIN\_DIM) //The opposite is true for black pieces  rating -= 3;  }  return rating;  }    private double rateBishop(int rowX, int colY, boolean isWhite) {  double rating = 15;  if(isWhite) {  if(rowX == ChessStatic.MAX\_DIM) //If it's white and is on the bottom row it's worth less because it's usually trapped.  rating -= 3;  } else {  if(rowX == ChessStatic.MIN\_DIM) //The opposite is true for black pieces  rating -= 3;  }  return rating;  }    private double rateQueen(int rowX, int colY, boolean isWhite) {  double rating = 45;  if(isWhite) {  if(rowX == ChessStatic.MAX\_DIM) //If it's white and is on the bottom row it's worth less because it's usually trapped.  rating -= 5;  } else {  if(rowX == ChessStatic.MIN\_DIM) //The opposite is true for black pieces  rating -= 5;  }  return rating;  }    /\*\*  \* Returns if the king is in check for the white player!!  \* @return @true if it is, @false otherwise.  \*/  private boolean isWhiteInCheck(ChessBoard board) {  if(board.isKingInCheck(ChessStatic.WHITE))  return true;  else  return false;  }    /\*\*  \* Returns if the king is in check for the black player!!  \* @return @true if it is, @false otherwise.  \*/  private boolean isBlackInCheck(ChessBoard board) {  if(board.isKingInCheck(ChessStatic.BLACK))  return true;  else  return false;  }  } |

# Appendix II. ChessBoard code

<https://github.com/danielgarm/Chess-Delta/blob/master/src/chess/game/mvc/model/chessFiles/ChessBoard.java>

|  |
| --- |
| public class ChessBoard extends BasicBoard {  private static final long serialVersionUID = 1L;    /\*\*  \* The internal representation of the board. Simply a matrix of objects of  \* type {@link Piece}.  \*  \* <p>  \* Representacion interna del tablero: una matriz de objetos de tipo  \* {@link Piece}.  \*/  protected Piece[][] board;  /\*\*  \* Number of columns in the board.  \* <p>  \* Numero de columnas del tablero.  \*/  protected int cols;  /\*\*  \* Number of rows in the board.  \* <p>  \* Numero de filas del tablero.  \*/  protected int rows;  /\*\*  \* Number of moves without moving a pawn or capturing a piece.  \*/  protected int movesWithoutAction;    protected boolean repetitionsRule;    /\*\*  \* Keeps track of the last positions of the board to avoid 3 repeated positions in a match.  \*/  protected HashMap<String, Integer> lastPositions;    /\*\*  \* This constructor constructs a finite rectangular board of a given  \* dimension.  \*  \* <p>  \* Construye un tablero rectangular finito de una dimension determinada.  \*  \* @param rows  \* Number of rows.  \* <p>  \* Numero de filas.  \* @param cols  \* Number of columns.  \* <p>  \* Numero de columnas.  \*/  @SuppressWarnings({ "unchecked", "rawtypes" })  public ChessBoard() {  this.rows = ChessStatic.BOARD\_DIMS;  this.cols = ChessStatic.BOARD\_DIMS;  this.lastPositions = new HashMap();  this.repetitionsRule = false;  this.board = new Piece[rows][cols];  this.movesWithoutAction = 0;  }    @SuppressWarnings({ "unchecked", "rawtypes" })  public ChessBoard(HashMap<String, Integer> positions, boolean repetitions, int noAction) {  this.rows = ChessStatic.BOARD\_DIMS;  this.cols = ChessStatic.BOARD\_DIMS;  this.lastPositions = new HashMap();  this.lastPositions.putAll(positions);  this.repetitionsRule = repetitions;  this.board = new Piece[rows][cols];  this.movesWithoutAction = noAction;  }  @Override  public void setPosition(int row, int col, Piece p) {  if (row < 0 || row >= rows || col < 0 || col >= cols) {  throw new GameError("Trying to access an invalid position (" + row + "," + col + ")");  }  board[row][col] = p;  }  @Override  public ChessPiece getPosition(int row, int col) { //May be worth overriding this for chess pieces.  if (row < 0 || row >= rows || col < 0 || col >= cols) {  throw new GameError("Trying to access an invalid position (" + row + "," + col + ")");  }  return (ChessPiece) board[row][col];  }  @Override  public int getRows() {  return rows;  }  @Override  public int getCols() {  return cols;  }  protected void copyTo(ChessBoard newboard) {  // ask the super class to copy its stuff first.  super.copyTo(newboard);  // copy the actual board  newboard.board = new Piece[rows][cols];  for (int i = 0; i < rows; i++)  for (int j = 0; j < cols; j++)  newboard.board[i][j] = board[i][j];  newboard.cols = cols;  newboard.rows = rows;  }  @Override  public Board copy() {  ChessBoard newboard = new ChessBoard(this.lastPositions, this.repetitionsRule, this.movesWithoutAction);  copyTo(newboard);  return newboard;  }  /\*\*  \* Generates a string that represents the board. The symbols used to print  \* the board are the first characters of the piece identifier.  \*  \* <p>  \* Genera un string que representa el tablero. El simbolo utilizado para  \* cada ficha es el primer caracter de su id.  \*  \* @return A string representation of the board.  \*/  @Override  public String toString() {  //Modified this function to make each board string unique by using both ID characters.  StringBuilder render = new StringBuilder();  int height = getRows();  int width = getCols();  for (int r = 0; r < height; ++r) {  render.append(" +");  for (int c = 0; c < width; ++c)  render.append("----+");  render.append("\n");  render.append("" + (r % 10) + " |");  for (int c = 0; c < width; ++c) {  if (getPosition(r, c) == null) {  render.append(" |");  } else {  render.append(" " + getPosition(r, c).toString() + " |");  }  } // for columns  render.append("\n");  } // for rows  render.append(" +");  for (int c = 0; c < width; ++c)  render.append("----+");  render.append("\n");  render.append(" ");  for (int c = 0; c < width; ++c)  render.append(" " + (c % 10) + " ");  render.append("\n");  return render.toString();  }    //Returns the position of the king from the current player as an 2 digits integer.  //Left digit represents row and right digit represents col. King at (2, 5), would be returned as "25".  //Returns @ChessStatic.UNKNOWN if the king hasn't been found. An exception should be thrown if this happens.  private int findKing(boolean isWhite) {  for(int rowX = ChessStatic.MIN\_DIM; rowX <= ChessStatic.MAX\_DIM; rowX++) {  for(int colY = ChessStatic.MIN\_DIM; colY <= ChessStatic.MAX\_DIM; colY++) {  if(this.getPosition(rowX, colY) != null) {  if(this.getPosition(rowX, colY) instanceof King && (((ChessPiece) this.getPosition(rowX, colY)).getWhite() == isWhite)) {  return rowX \* 10 + colY;  }  }  }  }  return ChessStatic.UNKNOWN;  }  //Checks if the King for the colour passed by parameter is threatened by any piece.  public boolean isKingInCheck(boolean isWhite) {  int kingLocation = findKing(isWhite);  if(kingLocation == ChessStatic.UNKNOWN) {  throw new GameError("Internal error. King not found, this should never happen.");  }  int kingRow = kingLocation / 10;  int kingCol = kingLocation % 10;    //King can't be threatened by the other King. Since a King cannot move to a vulnerable position.  //True means King is in check, False mean King is not in check.  return checkHorizVertThreat(kingRow, kingCol) || checkDiagonalThreat(kingRow, kingCol) ||  checkKnightThreat(kingRow, kingCol) || checkPawnThreat(kingRow, kingCol) ||  checkKingsCollision(kingRow, kingCol);  }    //Checks if the King would move to a position threatened by the enemy King.  private boolean checkKingsCollision(int kingRow, int kingCol) {  for(int rowX = kingRow - 1; rowX <= kingRow + 1; rowX++) {  for(int colY = kingCol - 1; colY <= kingCol + 1; colY++) {  if(rowX >= ChessStatic.MIN\_DIM && rowX <= ChessStatic.MAX\_DIM &&  colY >= ChessStatic.MIN\_DIM && colY <= ChessStatic.MAX\_DIM) { //Can't check outside the board!  if(rowX != kingRow || colY != kingCol) { //To avoid checking the position where the current king is.  if(this.getPosition(rowX, colY) != null) {  if(this.getPosition(rowX, colY) instanceof King) {  return true;  }  }  }  }  }  }  return false;  }    //Checks if the king is threatened from any row or column, horizontally or vertically by either a Queen or a Rook.  private boolean checkHorizVertThreat(int kingRow, int kingCol) {  for(int rowX = kingRow + 1; rowX <= ChessStatic.MAX\_DIM; rowX++) { //South Direction  if(this.getPosition(rowX, kingCol) != null) { //If the position is empty, the loop keeps running.  if(checkHorizVertAttacker(rowX, kingCol, ((ChessPiece) this.getPosition(kingRow, kingCol)).getWhite()))  return true; //If it's not empty and this function above finds an attacker on the explored position, it returns true and breaks every loop.  else  break; //If it's not empty and it finds a piece that isn't a threat or is an ally, it breaks the current loop and starts the next one.  }  }    for(int rowX = kingRow - 1; rowX >= ChessStatic.MIN\_DIM; rowX--) { //North Direction  if(this.getPosition(rowX, kingCol) != null) { //If the position is empty, the loop keeps running.  if(checkHorizVertAttacker(rowX, kingCol, ((ChessPiece) this.getPosition(kingRow, kingCol)).getWhite()))  return true; //If it's not empty and this function above finds an attacker on the explored position, it returns true and breaks every loop.  else  break; //If it's not empty and it finds a piece that isn't a threat or is an ally, it breaks the current loop and starts the next one.  }  }    for(int colY = kingCol + 1; colY <= ChessStatic.MAX\_DIM; colY++) { //East Direction  if(this.getPosition(kingRow, colY) != null) { //If the position is empty, the loop keeps running.  if(checkHorizVertAttacker(kingRow, colY, ((ChessPiece) this.getPosition(kingRow, kingCol)).getWhite()))  return true; //If it's not empty and this function above finds an attacker on the explored position, it returns true and breaks every loop.  else  break; //If it's not empty and it finds a piece that isn't a threat or is an ally, it breaks the current loop and starts the next one.  }  }    for(int colY = kingCol - 1; colY >= ChessStatic.MIN\_DIM; colY--) { //West Direction  if(this.getPosition(kingRow, colY) != null) { //If the position is empty, the loop keeps running.  if(checkHorizVertAttacker(kingRow, colY, ((ChessPiece) this.getPosition(kingRow, kingCol)).getWhite()))  return true; //If it's not empty and this function above finds an attacker on the explored position, it returns true and breaks every loop.  else  break; //If it's not empty and it finds a piece that isn't a threat or is an ally, it breaks the current loop and returns false, since none of the loops found a threat.  }  }  return false;  }    private boolean checkHorizVertAttacker(int rowX, int colY, boolean isWhiteKing) {  return (this.getPosition(rowX, colY) instanceof Queen || this.getPosition(rowX, colY) instanceof Rook) && //Checks that there's either a Queen or a Rook  (!((ChessPiece) this.getPosition(rowX, colY)).getWhite() == isWhiteKing); //And that the king is not the same color as the piece.  }    //Checks if the king is threatened from any diagonal by either a Queen or a Bishop.  private boolean checkDiagonalThreat(int kingRow, int kingCol) {  int rowOffSet = ChessStatic.NEGATIVE, colOffSet = ChessStatic.NEGATIVE;  int multiplier = 1;  int rowX, colY;  rowX = kingRow + rowOffSet \* multiplier;  colY = kingCol + colOffSet \* multiplier;  while(rowX >= ChessStatic.MIN\_DIM && colY >= ChessStatic.MIN\_DIM) { //NorthWest Direction  if(this.getPosition(rowX, colY) != null) { //If the position is empty, the loop keeps running.  if(checkDiagonalAttacker(rowX, colY, ((ChessPiece) this.getPosition(kingRow, kingCol)).getWhite()))  return true; //If it's not empty and this function above finds an attacker on the explored position, it returns true and breaks every loop.  else  break; //If it's not empty and it finds a piece that isn't a threat or is an ally, it breaks the current loop and starts the next one.  }  multiplier++;  rowX = kingRow + rowOffSet \* multiplier;  colY = kingCol + colOffSet \* multiplier;  }    multiplier = 1; rowOffSet = ChessStatic.NEGATIVE; colOffSet = ChessStatic.POSITIVE;  rowX = kingRow + rowOffSet \* multiplier;  colY = kingCol + colOffSet \* multiplier;  while(rowX >= ChessStatic.MIN\_DIM && colY <= ChessStatic.MAX\_DIM) { //NorthEast Direction  if(this.getPosition(rowX, colY) != null) { //If the position is empty, the loop keeps running.  if(checkDiagonalAttacker(rowX, colY, ((ChessPiece) this.getPosition(kingRow, kingCol)).getWhite()))  return true; //If it's not empty and this function above finds an attacker on the explored position, it returns true and breaks every loop.  else  break; //If it's not empty and it finds a piece that isn't a threat or is an ally, it breaks the current loop and starts the next one.  }  multiplier++;  rowX = kingRow + rowOffSet \* multiplier;  colY = kingCol + colOffSet \* multiplier;  }    multiplier = 1; rowOffSet = ChessStatic.POSITIVE; colOffSet = ChessStatic.POSITIVE;  rowX = kingRow + rowOffSet \* multiplier;  colY = kingCol + colOffSet \* multiplier;  while(rowX <= ChessStatic.MAX\_DIM && colY <= ChessStatic.MAX\_DIM) { //SouthEast Direction  if(this.getPosition(rowX, colY) != null) { //If the position is empty, the loop keeps running.  if(checkDiagonalAttacker(rowX, colY, ((ChessPiece) this.getPosition(kingRow, kingCol)).getWhite()))  return true; //If it's not empty and this function above finds an attacker on the explored position, it returns true and breaks every loop.  else  break; //If it's not empty and it finds a piece that isn't a threat or is an ally, it breaks the current loop and starts the next one.  }  multiplier++;  rowX = kingRow + rowOffSet \* multiplier;  colY = kingCol + colOffSet \* multiplier;  }    multiplier = 1; rowOffSet = ChessStatic.POSITIVE; colOffSet = ChessStatic.NEGATIVE;  rowX = kingRow + rowOffSet \* multiplier;  colY = kingCol + colOffSet \* multiplier;  while(rowX <= ChessStatic.MAX\_DIM && colY >= ChessStatic.MIN\_DIM) { //SouthWest Direction  rowX = kingRow + rowOffSet \* multiplier;  colY = kingCol + colOffSet \* multiplier;  if(this.getPosition(rowX, colY) != null) { //If the position is empty, the loop keeps running.  if(checkDiagonalAttacker(rowX, colY, ((ChessPiece) this.getPosition(kingRow, kingCol)).getWhite()))  return true; //If it's not empty and this function above finds an attacker on the explored position, it returns true and breaks every loop.  else  break; //If it's not empty and it finds a piece that isn't a threat or is an ally, it breaks the current loop and returns false, since none of the loops found a threat.  }  multiplier++;  rowX = kingRow + rowOffSet \* multiplier;  colY = kingCol + colOffSet \* multiplier;  }  return false;  }    private boolean checkDiagonalAttacker(int rowX, int colY, boolean isWhiteKing) {  return (this.getPosition(rowX, colY) instanceof Queen || this.getPosition(rowX, colY) instanceof Bishop) && //Checks that there's either a Queen or a Bishop  (!((ChessPiece) this.getPosition(rowX, colY)).getWhite() == isWhiteKing); //And that the king is not the same color as the piece.  }  private boolean checkKnightThreat(int kingRow, int kingCol) {  boolean isWhiteKing = ((ChessPiece) this.getPosition(kingRow, kingCol)).getWhite();    if(kingRow - 2 >= ChessStatic.MIN\_DIM) { //Check if going 2 rows up is within the board range.  if(kingCol - 1 >= ChessStatic.MIN\_DIM) { //Check if going 1 column left is within the board range  if(this.getPosition(kingRow - 2, kingCol - 1) != null &&  this.getPosition(kingRow - 2, kingCol - 1) instanceof Knight &&  ((ChessPiece) this.getPosition(kingRow - 2, kingCol - 1)).getWhite() != isWhiteKing) {  return true; //If the explored position is not empty, and has a enemy Knight  }  }    if(kingCol + 1 <= ChessStatic.MAX\_DIM) { //Check if going 1 column right is within the board range  if(this.getPosition(kingRow - 2, kingCol + 1) != null &&  this.getPosition(kingRow - 2, kingCol + 1) instanceof Knight &&  ((ChessPiece) this.getPosition(kingRow - 2, kingCol + 1)).getWhite() != isWhiteKing) {  return true; //If the explored position is not empty, and has a enemy Knight  }  }  }    if(kingRow + 2 <= ChessStatic.MAX\_DIM) { //Check if going 2 rows down is within the board range.  if(kingCol - 1 >= ChessStatic.MIN\_DIM) { //Check if going 1 column left is within the board range  if(this.getPosition(kingRow + 2, kingCol - 1) != null &&  this.getPosition(kingRow + 2, kingCol - 1) instanceof Knight &&  ((ChessPiece) this.getPosition(kingRow + 2, kingCol - 1)).getWhite() != isWhiteKing) {  return true; //If the explored position is not empty, and has a enemy Knight  }  }    if(kingCol + 1 <= ChessStatic.MAX\_DIM) { //Check if going 1 column right is within the board range  if(this.getPosition(kingRow + 2, kingCol + 1) != null &&  this.getPosition(kingRow + 2, kingCol + 1) instanceof Knight &&  ((ChessPiece) this.getPosition(kingRow + 2, kingCol + 1)).getWhite() != isWhiteKing) {  return true; //If the explored position is not empty, and has a enemy Knight  }  }  }    if(kingCol - 2 >= ChessStatic.MIN\_DIM) { //Check if going 2 columns left is within the board range.  if(kingRow - 1 >= ChessStatic.MIN\_DIM) { //Check if going 1 row up is within the board range  if(this.getPosition(kingRow - 1, kingCol - 2) != null &&  this.getPosition(kingRow - 1, kingCol - 2) instanceof Knight &&  ((ChessPiece) this.getPosition(kingRow - 1, kingCol - 2)).getWhite() != isWhiteKing) {  return true; //If the explored position is not empty, and has a enemy Knight  }  }    if(kingRow + 1 <= ChessStatic.MAX\_DIM) { //Check if going 1 row down is within the board range  if(this.getPosition(kingRow + 1, kingCol - 2) != null &&  this.getPosition(kingRow + 1, kingCol - 2) instanceof Knight &&  ((ChessPiece) this.getPosition(kingRow + 1, kingCol - 2)).getWhite() != isWhiteKing) {  return true; //If the explored position is not empty, and has a enemy Knight  }  }  }    if(kingCol + 2 <= ChessStatic.MAX\_DIM) { //Check if going 2 columns right is within the board range.  if(kingRow - 1 >= ChessStatic.MIN\_DIM) { //Check if going 1 row up is within the board range  if(this.getPosition(kingRow - 1, kingCol + 2) != null &&  this.getPosition(kingRow - 1, kingCol + 2) instanceof Knight &&  ((ChessPiece) this.getPosition(kingRow - 1, kingCol + 2)).getWhite() != isWhiteKing) {  return true; //If the explored position is not empty, and has a enemy Knight  }  }    if(kingRow + 1 <= ChessStatic.MAX\_DIM) { //Check if going 1 row down is within the board range  if(this.getPosition(kingRow + 1, kingCol + 2) != null &&  this.getPosition(kingRow + 1, kingCol + 2) instanceof Knight &&  ((ChessPiece) this.getPosition(kingRow + 1, kingCol + 2)).getWhite() != isWhiteKing) {  return true; //If the explored position is not empty, and has a enemy Knight  }  }  }    return false; //If none of the above conditions were met at some point  }  //Checks if the king is threatened by a Pawn.  private boolean checkPawnThreat(int kingRow, int kingCol) {  if(((ChessPiece) this.getPosition(kingRow, kingCol)).getWhite() && kingRow > ChessStatic.MIN\_DIM + 1) { //If the king is white and not on the top 2 rows (since pawns can't be on the top row anyway).  if(kingCol + 1 <= ChessStatic.MAX\_DIM && this.getPosition(kingRow - 1, kingCol + 1) != null) { //Checking the right diagonal for a pawn.  if(!((ChessPiece) this.getPosition(kingRow - 1, kingCol + 1)).getWhite()) { //If there is a black piece on that position  if(this.getPosition(kingRow - 1, kingCol + 1) instanceof Pawn) { //If that piece is a pawn  return true; //All the conditions were met.  }  }  }    if(kingCol - 1 <= ChessStatic.MAX\_DIM && this.getPosition(kingRow - 1, kingCol - 1) != null) { //Checking the left diagonal.  if(!((ChessPiece) this.getPosition(kingRow - 1, kingCol - 1)).getWhite()) { //If there is a black piece on that position  if(this.getPosition(kingRow - 1, kingCol - 1) instanceof Pawn) { //If that piece is a pawn  return true; //All the conditions were met.  }  }  }  } else if(!((ChessPiece) this.getPosition(kingRow, kingCol)).getWhite() && kingRow < ChessStatic.MAX\_DIM - 1) { //If the king is black and not on the bottom 2 rows (since pawns can't be on the bottom row anyway).  if(kingCol + 1 <= ChessStatic.MAX\_DIM && this.getPosition(kingRow + 1, kingCol + 1) != null) { //Checking the right diagonal for a pawn.  if(((ChessPiece) this.getPosition(kingRow + 1, kingCol + 1)).getWhite()) { //If there is a white piece on that position  if(this.getPosition(kingRow + 1, kingCol + 1) instanceof Pawn) { //If that piece is a pawn  return true; //All the conditions were met.  }  }  }    if(kingCol - 1 <= ChessStatic.MAX\_DIM && this.getPosition(kingRow + 1, kingCol - 1) != null) { //Checking the left diagonal.  if(((ChessPiece) this.getPosition(kingRow + 1, kingCol - 1)).getWhite()) { //If there is a white piece on that position  if(this.getPosition(kingRow + 1, kingCol - 1) instanceof Pawn) { //If that piece is a pawn  return true; //All the conditions were met.  }  }  }  }    return false;  }    public void addCurrentPosition() {  if(this.lastPositions.get(this.toString()) == null) {  this.lastPositions.put(this.toString(), 1);  } else {  Integer repeats = this.lastPositions.get(this.toString());  if(repeats + 1 == 3) { //If this is reached, it means that it will be incremented to 3 on this iteration.  this.repetitionsRule = true;  } else {  this.lastPositions.put(this.toString(), repeats + 1);  }  }  }    public void resetBoardHistory() {  this.repetitionsRule = false;  this.lastPositions.clear();  }    public void increaseMovesWithoutAction() {  this.movesWithoutAction++;  }    public void resetMovesRules() {  this.movesWithoutAction = 0;  resetBoardHistory();  }    /\*\*  \* @return @true if there have been more than 50 moves without a capture or pawn movement of 3 repeated boards. @false otherwise  \*/  public boolean checkMovesRulesLimit() {  return this.movesWithoutAction >= 50 || this.repetitionsRule;  }    @Override  public boolean isFull() {  return false;  }  @Override  public boolean isEmpty() {  return false;  }  } |

# Appendix III. ChessMove code

<https://github.com/danielgarm/Chess-Delta/blob/master/src/chess/game/mvc/model/chessFiles/ChessMove.java>

|  |
| --- |
| public class ChessMove extends GameMove {  private int row;  private int col;  private int rowDes;  private int colDes;    private boolean actionMove; //If it is a pawn move or a capture move, for the 50 moves rule.  private boolean testMove; //If it is a test move and will be reversed/ignored.    private ChessPiece chessPiece;  private ChessBoard chessBoard;    public ChessMove() {}    public ChessMove(int row, int col,int rowDes, int colDes, Piece p) {  super(p);  this.row = row;  this.col = col;  this.rowDes = rowDes;  this.colDes = colDes;  this.chessPiece = (ChessPiece) p;  this.actionMove = false;  this.testMove = false;  }    private static final long serialVersionUID = 273689252496184587L;    //Checks if the move can be made and, if so, moves the piece to the specified position.  public void execute(Board board, List<Piece> playersPieces, List<Piece> chessPieces) {  //this.getPiece() now returns the player that made the move!!!    this.chessBoard = (ChessBoard) board;  this.actionMove = false;    ChessBoard originalBoard = (ChessBoard) this.chessBoard.copy();    if(this.chessBoard.getPosition(this.row, this.col) == null)  throw new GameError("You need to select one of your pieces to perform the move!");    if(!checkTurn(this.chessBoard)) {//Checks that the player is trying to move his own piece.  String color = "";  if(((ChessPiece) this.getPiece()).getWhite())  color = "WHITE";  else  color = "BLACK";  throw new GameError("You can only move your own pieces. It's the turn for " + color + ". Try again."); //Works properly  }    if(this.row == this.rowDes && this.col == this.colDes)  throw new GameError("You cannot move a piece to the same position it was before. Try again.");    if(this.chessBoard.getPosition(this.row, this.col) instanceof Pawn) {  executePawnMove(this.chessBoard);  this.actionMove = true; //This is used for the 50 moves rule  } else if (this.chessBoard.getPosition(this.row, this.col) instanceof Rook) {  executeRookMove(this.chessBoard);  } else if (this.chessBoard.getPosition(this.row, this.col) instanceof Knight) {  executeKnightMove(this.chessBoard);  } else if (this.chessBoard.getPosition(this.row, this.col) instanceof Bishop) {  executeBishopMove(this.chessBoard);  } else if (this.chessBoard.getPosition(this.row, this.col) instanceof Queen) {  executeQueenMove(this.chessBoard);  } else if (this.chessBoard.getPosition(this.row, this.col) instanceof King) {  executeKingMove(this.chessBoard);  } else {  throw new GameError("Piece type not recognised! This should be unreachable.");  }    if(this.chessBoard.isKingInCheck(((ChessPiece) this.getPiece()).getWhite())) { //If the King is in Check  revertBoard(this.chessBoard, originalBoard);  if(originalBoard.isKingInCheck(((ChessPiece) this.getPiece()).getWhite())) { //If the King was in Check before the move.  throw new GameError("You cannot perform that move, your King is in Check!!!");  } else { //If the King would be in check after the move.  throw new GameError("You cannot perform that move, your King would be in Check after that!!!");  }  }    //If the code reaches this point, the move has been successfully executed.    disableEnPassant(!((ChessPiece) this.getPiece()).getWhite()); //Disable En passant for every pawn of the colour that is going  //to move next turn, since it has been more than a move ago that he moved his pieces.    if(actionMove && !testMove) {  this.chessBoard.resetMovesRules();  } else if(!testMove) {  this.chessBoard.increaseMovesWithoutAction();  this.chessBoard.addCurrentPosition();  }  }    /\*\*  \* Used by ChessRules to get the list of valid moves.  \* @param testBoard  \* @return  \*/  public boolean isMoveLegal(ChessBoard testBoard) {  try {  if (this.colDes > ChessStatic.MAX\_DIM || this.rowDes > ChessStatic.MAX\_DIM ||  this.colDes < ChessStatic.MIN\_DIM || this.rowDes < ChessStatic.MIN\_DIM) //First checks the range of the move.  return false;    this.testMove = true;  this.executeTestMove(testBoard);  this.testMove = false;  } catch(Exception e) {  return false;  }  return true;  }    /\*\*  \* Used by the function above to test if a move is valid or not by capturing exceptions!  \* @param testBoard  \*/  public void executeTestMove(ChessBoard testBoard) {  this.chessBoard = (ChessBoard) testBoard.copy();    if(this.chessBoard.getPosition(this.row, this.col) == null)  throw new GameError("You need to select one of your pieces to perform the move!");    if(!checkTurn(this.chessBoard)) {//Checks that the player is trying to move his own piece.  String color = "";  if(((ChessPiece) this.getPiece()).getWhite())  color = "WHITE";  else  color = "BLACK";  throw new GameError("You can only move your own pieces. It's the turn for " + color + ". Try again.");  }    if(this.row == this.rowDes && this.col == this.colDes)  throw new GameError("You cannot move a piece to the same position it was before. Try again.");    if(this.chessBoard.getPosition(this.row, this.col) instanceof Pawn) {  executePawnMove(this.chessBoard);  } else if (this.chessBoard.getPosition(this.row, this.col) instanceof Rook) {  executeRookMove(this.chessBoard);  } else if (this.chessBoard.getPosition(this.row, this.col) instanceof Knight) {  executeKnightMove(this.chessBoard);  } else if (this.chessBoard.getPosition(this.row, this.col) instanceof Bishop) {  executeBishopMove(this.chessBoard);  } else if (this.chessBoard.getPosition(this.row, this.col) instanceof Queen) {  executeQueenMove(this.chessBoard);  } else if (this.chessBoard.getPosition(this.row, this.col) instanceof King) {  executeKingMove(this.chessBoard);  } else {  throw new GameError("Piece type not recognised! This should be unreachable.");  }    if(((ChessBoard) this.chessBoard).isKingInCheck(((ChessPiece) this.getPiece()).getWhite())) { //If the King is in Check  throw new GameError("You cannot perform that move, your King would be in Check after that!!!");  }    if(actionMove && !testMove) {  this.chessBoard.resetMovesRules();  } else if(!testMove) {  this.chessBoard.increaseMovesWithoutAction();  this.chessBoard.addCurrentPosition();  }  }    /\*\*  \* Simply executes a move that has been previously checked and is legal.  \* @param chessBoard  \*/  private void executeCheckedMove(ChessBoard chessBoard) {  chessBoard.setPosition(this.rowDes, this.colDes, chessBoard.getPosition(this.row, this.col));  deleteMovedPiece(this.row, this.col, chessBoard);  }    /\*\*  \* Simply executes a move that has been previously checked and is legal.  \* It is public so that it can be called from the AI classes to perform moves from the list of possible (valid) moves.  \* @param chessBoard  \*/  public void executeAICheckedMove(ChessBoard chessBoard) {  //TODO This condition below needs testing.  if(chessBoard.getPosition(this.row, this.col) instanceof Pawn || chessBoard.getPosition(this.rowDes, this.colDes) != null) {  chessBoard.setPosition(this.rowDes, this.colDes, chessBoard.getPosition(this.row, this.col));  deleteMovedPiece(this.row, this.col, chessBoard);  chessBoard.resetMovesRules();  } else {  chessBoard.setPosition(this.rowDes, this.colDes, chessBoard.getPosition(this.row, this.col));  deleteMovedPiece(this.row, this.col, chessBoard);  chessBoard.increaseMovesWithoutAction();  chessBoard.addCurrentPosition();  }  }    private void executeCastlingTestMove(ChessBoard testBoard, int row, int col, int rowDes, int colDes) {  testBoard.setPosition(rowDes, colDes, testBoard.getPosition(row, col));  deleteMovedPiece(row, col, testBoard);  }      private void executeCaptureMove(ChessBoard chessBoard2) {  if((((ChessPiece) chessBoard2.getPosition(this.rowDes, this.colDes)).getWhite() && (((ChessPiece) this.getPiece()).getWhite())) ||  ((!((ChessPiece) chessBoard2.getPosition(this.rowDes, this.colDes)).getWhite() && (!((ChessPiece) this.getPiece()).getWhite())))) {  //If the player tried to capture an ally piece.  throw new GameError("Destination position already occupied by an ally piece!");  } else { //If he's capturing an enemy piece.  if(chessBoard2.getPosition(this.rowDes, this.colDes) instanceof King) { //You can't capture the king!  throw new GameError("Checkmate? This point should be unreachable.");  } else {  executeCheckedMove(chessBoard2);  }  }  }    //Disable En Passant for every Pawn from the colour passed by parameter.  private void disableEnPassant(boolean isWhite) {  for(int rowX = ChessStatic.MIN\_DIM + 3; rowX <= ChessStatic.MAX\_DIM - 3; rowX++) { //Pawns that have the "En passant" attribute set, can only be in the 2 central rows  for(int colY = ChessStatic.MIN\_DIM; colY <= ChessStatic.MAX\_DIM; colY++) {  if(this.chessBoard.getPosition(rowX, colY) != null &&  this.chessBoard.getPosition(rowX, colY) instanceof Pawn &&  ((ChessPiece) this.chessBoard.getPosition(rowX, colY)).getWhite() == isWhite) {  if(!this.testMove) ((Pawn) this.chessBoard.getPosition(rowX, colY)).setPassant(false);  }  }  }  }    //Shows whether the current player is moving his own pieces or trying to move the opponent's ones.  private boolean checkTurn(Board chessBoard2) {  //this.getPiece() returns the piece to which the move belongs!  return ((((ChessPiece) this.getPiece()).getWhite() && ((ChessPiece) chessBoard2.getPosition(this.row, this.col)).getWhite()) ||  (!((ChessPiece) this.getPiece()).getWhite() && !((ChessPiece) chessBoard2.getPosition(this.row, this.col)).getWhite()));  }    //Due to the peculiar pattern of pawn movement, it is required to have two  //different functions, since white pawns move upwards, and black ones move downwards.  private void executePawnMove(ChessBoard chessBoard2) {  if(((ChessPiece) this.getPiece()).getWhite())  executeWhitePawnMove(chessBoard2);  else  executeBlackPawnMove(chessBoard2);  }    private void executeWhitePawnMove(ChessBoard chessBoard2) {  //Check if this pawn is trying to capture a piece. (Diagonal move)  if(this.rowDes == this.row - 1 && (this.colDes == this.col + 1 || this.colDes == this.col - 1)) {  if(chessBoard2.getPosition(this.rowDes, this.colDes) != null) { //The destination position isn't empty  executeCaptureMove(chessBoard2); //This already checks the kind of piece that is in the destination position and if it can be captured.  if(!this.testMove) ((Pawn) chessBoard2.getPosition(this.rowDes, this.colDes)).setFirstMove(false);  } else if((chessBoard2.getPosition(this.row, this.col + 1) instanceof Pawn) && (this.colDes == this.col + 1) || //There is a pawn on the right  (chessBoard2.getPosition(this.row, this.col - 1) instanceof Pawn) && (this.colDes == this.col - 1)) { //or the left for En Passant capture  //No need to check that destination is null, that's checked above.  if(this.colDes == this.col + 1) { //Capture to the right  if(!((ChessPiece) chessBoard2.getPosition(this.row, this.col + 1)).getWhite() && //Check that the pawn is black  ((Pawn) chessBoard2.getPosition(this.row, this.col + 1)).getPassant()) { //and it can be captured En Passant for this turn.  executeCheckedMove(chessBoard2);  if(!this.testMove) chessBoard2.setPosition(this.row, this.col + 1, null); //Deletes the captured piece  } else { //Trying to capture either your own pawn or a pawn that can't be captured En Passant  throw new GameError("Invalid Pawn move, try again.");  }  } else if (this.colDes == this.col - 1) { //Capture to the left.  if(!((ChessPiece) chessBoard2.getPosition(this.row, this.col - 1)).getWhite() && //Check that the pawn is black  ((Pawn) chessBoard2.getPosition(this.row, this.col - 1)).getPassant()) { //and it can be captured En Passant for this turn.  executeCheckedMove(chessBoard2);  if(!this.testMove) chessBoard2.setPosition(this.row, this.col - 1, null); //Deletes the captured piece  } else { //Trying to capture own piece En Passant  throw new GameError("Invalid Pawn move, try again.");  }  } else { //Unrecognised En Passant move (not left nor right?)  throw new GameError("Invalid Pawn move, try again.");  }  } else { //Pawn invalid diagonal move.  throw new GameError("Invalid Pawn move, try again.");  }  } else if(this.col == this.colDes && this.row - 1 == this.rowDes) { //Check if it's making a simple move  if(chessBoard2.getPosition(this.rowDes, this.colDes) == null) { //Working.  executeCheckedMove(chessBoard2);  if(!this.testMove) ((Pawn) chessBoard2.getPosition(this.rowDes, this.colDes)).setFirstMove(false);  } else {  throw new GameError("Invalid Pawn move, try again.");  }  } else if(this.col == this.colDes && this.row - 2 == this.rowDes) { //Check if it's making an opening move (Double).  if(((Pawn) chessBoard2.getPosition(this.row, this.col)).getFirstMove()) { //Checks if it's the pawn's first move.  if(chessBoard2.getPosition(this.rowDes, this.colDes) != null) {  throw new GameError("Invalid move, the destination position is occupied, try again.");  } else if (!checkPiecesInbetween(this.row, this.col, this.rowDes, this.colDes)) {  throw new GameError("Invalid move, you can't skip through other pieces, try again.");  } else {  executeCheckedMove(chessBoard2);  if(!this.testMove) ((Pawn) chessBoard2.getPosition(this.rowDes, this.colDes)).setPassant(true); //This pawn can be captured En Passant in the next move.  if(!this.testMove) ((Pawn) chessBoard2.getPosition(this.rowDes, this.colDes)).setFirstMove(false);  }  } else {  throw new GameError("This Pawn can't perform an opening move.");  }  } else { //The move is not valid.  throw new GameError("Invalid Pawn move, try again.");  }    //You can only get here if everything went right during the execution of the move.  if(!this.testMove && !chessBoard2.isKingInCheck(((ChessPiece) this.getPiece()).getWhite()) && checkPromotion(chessBoard2)) {  if(!Controller.isWhiteAI) {  ChessPawnPromotionDialog dialog = new ChessPawnPromotionDialog("Select the piece you would like:", true);    dialog.addWindowListener(new WindowAdapter() {  public void windowClosed(WindowEvent e) {  ChessPiece newPiece;  switch(dialog.getChosenPiece()) {  case ChessPieceID.WHITE\_ROOK\_PAWN:  newPiece = new Rook(true, false);  break;  case ChessPieceID.WHITE\_KNIGHT:  newPiece = new Knight(true);  break;  case ChessPieceID.WHITE\_BISHOP:  newPiece = new Bishop(true);  break;  case ChessPieceID.WHITE\_QUEEN:  newPiece = new Queen(true);  break;  default:  throw new GameError("Invalid piece for promotion chosen.");  }  executePromotion(chessBoard2, newPiece);  }  });  } else {  executePromotion(chessBoard2, new Queen(true)); //Always promotes to Queen because of piece value maximisation heuristics  }  } else if(this.testMove) {  executePromotion(chessBoard2, new Queen(true)); //Always promotes to Queen because of piece value maximisation heuristics  }  }    private void executeBlackPawnMove(ChessBoard chessBoard2) {  //Check if this pawn is trying to capture a piece. (Diagonal move)  if(this.rowDes == this.row + 1 && (this.colDes == this.col + 1 || this.colDes == this.col - 1)) {  if(chessBoard2.getPosition(this.rowDes, this.colDes) != null) { //The destination position isn't empty  executeCaptureMove(chessBoard2); //This already checks the kind of piece that is in the destination position and if it can be captured.  if(!this.testMove) ((Pawn) chessBoard2.getPosition(this.rowDes, this.colDes)).setFirstMove(false);  } else if((chessBoard2.getPosition(this.row, this.col + 1) instanceof Pawn) && (this.colDes == this.col + 1) || //There is a pawn on the right  (chessBoard2.getPosition(this.row, this.col - 1) instanceof Pawn) && (this.colDes == this.col - 1)) { //or the left for En Passant capture  //No need to check that destination is null, that's checked above.  if(this.colDes == this.col + 1) { //Capture to the right  if(((ChessPiece) chessBoard2.getPosition(this.row, this.col + 1)).getWhite() && //Check that the pawn is white  ((Pawn) chessBoard2.getPosition(this.row, this.col + 1)).getPassant()) { //and it can be captured En Passant for this turn.  executeCheckedMove(chessBoard2);  chessBoard2.setPosition(this.row, this.col + 1, null); //Deletes the captured piece  } else { //Trying to capture either your own pawn or a pawn that can't be captured En Passant  throw new GameError("Invalid Pawn move, try again.");  }  } else if (this.colDes == this.col - 1) { //Capture to the left.  if(((ChessPiece) chessBoard2.getPosition(this.row, this.col - 1)).getWhite() && //Check that the pawn is white  ((Pawn) chessBoard2.getPosition(this.row, this.col - 1)).getPassant()) { //and it can be captured En Passant for this turn.  executeCheckedMove(chessBoard2);  chessBoard2.setPosition(this.row, this.col - 1, null); //Deletes the captured piece  } else { //Trying to capture own piece En Passant  throw new GameError("Invalid Pawn move, try again.");  }  } else { //Unrecognised En Passant move (not left nor right?)  throw new GameError("Invalid Pawn move, try again.");  }  } else { //Pawn invalid diagonal move.  throw new GameError("Invalid Pawn move, try again.");  }  } else if(this.col == this.colDes && this.row + 1 == this.rowDes) { //Check if it's making a simple move  if(chessBoard2.getPosition(this.rowDes, this.colDes) == null) { //Working.  executeCheckedMove(chessBoard2);  if(!this.testMove) ((Pawn) chessBoard2.getPosition(this.rowDes, this.colDes)).setFirstMove(false);  } else {  throw new GameError("Invalid Pawn move, try again.");  }  } else if(this.col == this.colDes && this.row + 2 == this.rowDes) { //Check if it's making an opening move (Double).  if(((Pawn) chessBoard2.getPosition(this.row, this.col)).getFirstMove()) { //Checks if it's the pawn's first move.  if(chessBoard2.getPosition(this.rowDes, this.colDes) != null) {  throw new GameError("Invalid move, the destination position is occupied, try again.");  } else if (!checkPiecesInbetween(this.row, this.col, this.rowDes, this.colDes)) {  throw new GameError("Invalid move, you can't skip through other pieces, try again.");  } else {  executeCheckedMove(chessBoard2);  if(!this.testMove) ((Pawn) chessBoard2.getPosition(this.rowDes, this.colDes)).setPassant(true); //This pawn can be captured En Passant in the next move.  if(!this.testMove) ((Pawn) chessBoard2.getPosition(this.rowDes, this.colDes)).setFirstMove(false);  }  } else {  throw new GameError("This Pawn can't perform an opening move.");  }  } else { //The move is not valid.  throw new GameError("Invalid Pawn move, try again.");  }    //You can only get here if everything went right during the execution of the move.  if(!this.testMove && !chessBoard2.isKingInCheck(((ChessPiece) this.getPiece()).getWhite()) && checkPromotion(chessBoard2)) {  if(!Controller.isBlackAI) {  ChessPawnPromotionDialog dialog = new ChessPawnPromotionDialog("Select the piece you would like:", false);    dialog.addWindowListener(new WindowAdapter() {  public void windowClosed(WindowEvent e) {  ChessPiece newPiece;  switch(dialog.getChosenPiece()) {  case ChessPieceID.BLACK\_ROOK\_PAWN:  newPiece = new Rook(false, false);  break;  case ChessPieceID.BLACK\_KNIGHT:  newPiece = new Knight(false);  break;  case ChessPieceID.BLACK\_BISHOP:  newPiece = new Bishop(false);  break;  case ChessPieceID.BLACK\_QUEEN:  newPiece = new Queen(false);  break;  default:  throw new GameError("Invalid piece for promotion chosen.");  }  executePromotion(chessBoard2, newPiece);  }  });  } else {  executePromotion(chessBoard2, new Queen(false)); //Always promotes to Queen because of piece value maximisation heuristics  }  } else if(this.testMove) {  executePromotion(chessBoard2, new Queen(false)); //Always promotes to Queen because of piece value maximisation heuristics  }  }    private boolean checkPromotion(ChessBoard chessBoard2) { //This function should only be called from within the Pawn movement function for it to work properly.  return (((ChessPiece) chessBoard2.getPosition(this.rowDes, this.colDes)).getWhite() && this.rowDes == ChessStatic.MIN\_DIM) || //If it's white and has reached the top  (!((ChessPiece) chessBoard2.getPosition(this.rowDes, this.colDes)).getWhite() && this.rowDes == ChessStatic.MAX\_DIM); //If it's black and has reached the bottom  }    private void executePromotion(ChessBoard chessBoard2, ChessPiece p) { //The piece must be properly created in another function. Usually executeWhite/BlackPawnMove().  chessBoard2.setPosition(this.rowDes, this.colDes, p);  }  private void executeRookMove(ChessBoard chessBoard2) {  if(checkHorizVertMove()) { //Check that it's moving either horizontally or vertically.  if(chessBoard2.getPosition(this.rowDes, this.colDes) != null) { //Trying to capture a piece.  if((((ChessPiece) chessBoard2.getPosition(this.rowDes, this.colDes)).getWhite() && ((ChessPiece) this.getPiece()).getWhite()) ||  (!((ChessPiece) chessBoard2.getPosition(this.rowDes, this.colDes)).getWhite() && !((ChessPiece) this.getPiece()).getWhite())) {  //Check that you're not trying to capture your own pieces.  throw new GameError("Invalid move, the destination is occupied by an ally piece, try again.");  } else if (!checkPiecesInbetween(this.row, this.col, this.rowDes, this.colDes)) { //Check there are no pieces inbetween.  throw new GameError("Invalid move, you can't skip through other pieces, try again.");  } else { //Everything correct, we can execute the move.  executeCaptureMove(chessBoard2);  this.actionMove = true;  if(!this.testMove) ((Rook) chessBoard2.getPosition(this.rowDes, this.colDes)).setCastle(false); //This rook can't Castle since it moved.  }  } else { //The destination position is empty.  if (!checkPiecesInbetween(this.row, this.col, this.rowDes, this.colDes)) { //Check there are no pieces inbetween.  throw new GameError("Invalid move, you can't skip through other pieces, try again.");  } else { //We can proceed to the move.  executeCheckedMove(chessBoard2);  if(!this.testMove) ((Rook) chessBoard2.getPosition(this.rowDes, this.colDes)).setCastle(false); //This rook can't Castle since it moved.  }  }  } else { //It is not a vertical or horizontal move.  throw new GameError("Invalid move, try again.");  }  }    private void executeKnightMove(ChessBoard chessBoard2) {  if(((this.colDes == this.col + 1 || this.colDes == this.col - 1) &&  (this.rowDes == this.row + 2 || this.rowDes == this.row - 2)) ||  ((this.rowDes == this.row + 1 || this.rowDes == this.row - 1) &&  (this.colDes == this.col + 2 || this.colDes == this.col - 2))) { //Check if the movement is legal.  if(chessBoard2.getPosition(this.rowDes, this.colDes) == null) { //If the destination position is empty.  executeCheckedMove(chessBoard2);  } else { //If the piece is trying to capture a piece.  executeCaptureMove(chessBoard2);  this.actionMove = true;  }  } else { //The movement is illegal.  throw new GameError("Invalid movement, try again.");  }  }    private void executeBishopMove(ChessBoard board) {  if(checkDiagonalMove()) { //Checks that it's moving diagonally.  if(board.getPosition(this.rowDes, this.colDes) != null) { //Trying to capture a piece.  if((((ChessPiece) board.getPosition(this.rowDes, this.colDes)).getWhite() && ((ChessPiece) this.getPiece()).getWhite()) ||  (!((ChessPiece) board.getPosition(this.rowDes, this.colDes)).getWhite() && !((ChessPiece) this.getPiece()).getWhite())) {  //Check that you're not trying to capture your own pieces.  throw new GameError("Invalid move, the destination position is occupied by an ally piece, try again.");  } else if (!checkPiecesInbetween(this.row, this.col, this.rowDes, this.colDes)) { //Check there are no pieces inbetween.  throw new GameError("Invalid move, you can't skip through other pieces, try again.");  } else { //Everything correct, we can execute the move.  executeCaptureMove(board);  this.actionMove = true;  }  } else { //The destination position is empty.  if (!checkPiecesInbetween(this.row, this.col, this.rowDes, this.colDes)) { //Check there are no pieces inbetween.  throw new GameError("Invalid move, you can't skip through other pieces, try again.");  } else { //We can proceed to the move.  executeCheckedMove(board);  }  }  } else { //It is not a vertical or horizontal move.  throw new GameError("Invalid move, try again.");  }  }  private void executeQueenMove(ChessBoard board) {  if(checkDiagonalMove() || checkHorizVertMove()) { //Checks that it's moving diagonally or horizontally but not in any other way.  if(board.getPosition(this.rowDes, this.colDes) != null) { //Trying to capture a piece.  if((((ChessPiece) board.getPosition(this.rowDes, this.colDes)).getWhite() && ((ChessPiece) this.getPiece()).getWhite()) ||  (!((ChessPiece) board.getPosition(this.rowDes, this.colDes)).getWhite() && !((ChessPiece) this.getPiece()).getWhite())) {  //Check that you're not trying to capture your own pieces.  throw new GameError("Invalid move, the destination position is occupied by an ally piece, try again.");  } else if (!checkPiecesInbetween(this.row, this.col, this.rowDes, this.colDes)) { //Check there are no pieces inbetween.  throw new GameError("Invalid move, you can't skip through other pieces, try again.");  } else { //Everything correct, we can execute the move.  executeCaptureMove(board);  this.actionMove = true;  }  } else { //The destination position is empty.  if (!checkPiecesInbetween(this.row, this.col, this.rowDes, this.colDes)) { //Check there are no pieces inbetween.  throw new GameError("Invalid move, you can't skip through other pieces, try again.");  } else { //We can proceed to the move.  executeCheckedMove(board);  }  }  } else { //It is not a vertical or horizontal move.  throw new GameError("Invalid move, try again.");  }  }    private void executeKingMove(ChessBoard board) {  if(Math.abs(this.row - this.rowDes) <= 1 && Math.abs(this.col - this.colDes) <= 1) {  //Check that it's moving only 1 tile away.  if(board.getPosition(this.rowDes, this.colDes) != null) { //Trying to capture a piece.  if((((ChessPiece) board.getPosition(this.rowDes, this.colDes)).getWhite() && ((ChessPiece) this.getPiece()).getWhite()) ||  (!((ChessPiece) board.getPosition(this.rowDes, this.colDes)).getWhite() && !((ChessPiece) this.getPiece()).getWhite())) {  //Check that you're not trying to capture your own pieces.  throw new GameError("Invalid move, the destination position is occupied by an ally piece, try again.");  } else { //Everything correct, we can execute the move.  executeCaptureMove(board);  this.actionMove = true;  if(!this.testMove) ((King) board.getPosition(this.rowDes, this.colDes)).setCastle(false); //This King can't Castle since it just moved.  }  } else { //The destination position is empty.  executeCheckedMove(board);  if(!this.testMove) ((King) board.getPosition(this.rowDes, this.colDes)).setCastle(false); //This King can't Castle since it just moved.  }  } else if(((ChessPiece) board.getPosition(this.row, this.col)).getWhite() && ((this.row == 7 && this.col == 4) &&  (this.rowDes == 7 && (this.colDes == 2 || this.colDes == 6)))) { //Trying to Castle white king.  if(((King) board.getPosition(this.row, this.col)).getCastle()) { //Check if the King can castle.  if(this.colDes == 6) { //Short castling  if(board.getPosition(7, 7) != null && board.getPosition(7, 7) instanceof Rook) { //Check that there's a rook  if(((Rook) board.getPosition(7, 7)).getCastle()) { //Check if the Rook can castle  if(checkPiecesInbetween(this.row, this.col, 7, 7)) { //Check that there are no pieces inbetween  if(!board.isKingInCheck(ChessStatic.WHITE)) { //Check that the WHITE king is not in check  ChessBoard testBoard = (ChessBoard) board.copy(); //Duplicate the board  boolean abortMovement = false;  executeCastlingTestMove(testBoard, this.row, this.col, 7, 5); //Move to the first position it would go through  if(testBoard.isKingInCheck(((ChessPiece) this.getPiece()).getWhite()))//Check if it is in check at this point.  abortMovement = true;  executeCastlingTestMove(testBoard, 7, 5, 7, 6); //Move to the next position  if(testBoard.isKingInCheck(((ChessPiece) this.getPiece()).getWhite())) //Check if it is in check at this point.  abortMovement = true;  if(!abortMovement) { //If the king doesn't go through any danger positions  executeCheckedMove(board); //Performs Castling...  board.setPosition(7, 5, board.getPosition(7, 7));  deleteMovedPiece(7, 7, board);  } else {  throw new GameError("Cannot perform Castling, the King would be in Check during the move!");  }  } else {  throw new GameError("Cannot perform Castling, the King is in Check!");  }  } else {  throw new GameError("Cannot perform Castling, there are pieces inbetween.");  }  } else {  throw new GameError("Rook cannot perform Castling.");  }  } else {  throw new GameError("Rook is missing for Castling.");  }  } else { //Long castling  if(board.getPosition(7, 0) != null && board.getPosition(7, 0) instanceof Rook) { //Check that there's a rook  if(((Rook) board.getPosition(7, 0)).getCastle()) { //Check if the Rook can castle  if(checkPiecesInbetween(this.row, this.col, 7, 0)) { //Check that there are no pieces inbetween  if(!board.isKingInCheck(ChessStatic.WHITE)) { //Check that the WHITE king is not in check  ChessBoard testBoard = (ChessBoard) board.copy(); //Duplicate the board  boolean abortMovement = false;  executeCastlingTestMove(testBoard, this.row, this.col, 7, 3); //Move to the first position it would go through  if(testBoard.isKingInCheck(((ChessPiece) this.getPiece()).getWhite()))//Check if it is in check at this point.  abortMovement = true;  executeCastlingTestMove(testBoard, 7, 3, 7, 2); //Move to the next position  if(testBoard.isKingInCheck(((ChessPiece) this.getPiece()).getWhite())) //Check if it is in check at this point.  abortMovement = true;  if(!abortMovement) { //If the king doesn't go through any danger positions  executeCheckedMove(board); //Performs Castling...  board.setPosition(7, 3, board.getPosition(7, 0));  deleteMovedPiece(7, 0, board);  } else {  throw new GameError("Cannot perform Castling, the King would be in Check during the move!");  }  } else {  throw new GameError("Cannot perform Castling, the King is in Check!");  }  } else {  throw new GameError("Cannot perform Castling, there are pieces inbetween.");  }  } else {  throw new GameError("Rook cannot perform Castling.");  }  } else {  throw new GameError("Rook is missing for Castling.");  }  }  } else {  throw new GameError("This King cannot Castle anymore.");  }  } else if(!((ChessPiece) board.getPosition(this.row, this.col)).getWhite() && ((this.row == 0 && this.col == 4) &&  (this.rowDes == 0 && (this.colDes == 2 || this.colDes == 6)))) { //Trying to Castle black king.  if(((King) board.getPosition(this.row, this.col)).getCastle()) { //Check if the King can castle.  if(this.colDes == 6) { //Short castling  if(board.getPosition(0, 7) != null && board.getPosition(0, 7) instanceof Rook) { //Check that there's a rook  if(((Rook) board.getPosition(0, 7)).getCastle()) { //Check if the Rook can castle  if(checkPiecesInbetween(this.row, this.col, 0, 7)) { //Check that there are no pieces inbetween  if(!board.isKingInCheck(ChessStatic.BLACK)) { //Check that the BLACK king is not in check  ChessBoard testBoard = (ChessBoard) board.copy(); //Duplicate the board  boolean abortMovement = false;  executeCastlingTestMove(testBoard, this.row, this.col, 0, 5); //Move to the first position it would go through  if(testBoard.isKingInCheck(((ChessPiece) this.getPiece()).getWhite()))//Check if it is in check at this point.  abortMovement = true;  executeCastlingTestMove(testBoard, 0, 5, 0, 6); //Move to the next position  if(testBoard.isKingInCheck(((ChessPiece) this.getPiece()).getWhite())) //Check if it is in check at this point.  abortMovement = true;  if(!abortMovement) { //If the king doesn't go through any danger positions  executeCheckedMove(board); //Performs Castling...  board.setPosition(0, 5, board.getPosition(0, 7));  deleteMovedPiece(0, 7, board);  } else {  throw new GameError("Cannot perform Castling, the King would be in Check during the move!");  }  } else {  throw new GameError("Cannot perform Castling, the King is in Check!");  }  } else {  throw new GameError("Cannot perform Castling, there are pieces inbetween.");  }  } else {  throw new GameError("Rook cannot perform Castling.");  }  } else {  throw new GameError("Rook is missing for Castling.");  }  } else { //Long castling  if(board.getPosition(0, 0) != null && board.getPosition(0, 0) instanceof Rook) { //Check that there's a rook  if(((Rook) board.getPosition(0, 0)).getCastle()) { //Check if the Rook can castle  if(checkPiecesInbetween(this.row, this.col, 0, 0)) { //Check that there are no pieces inbetween  if(!board.isKingInCheck(ChessStatic.BLACK)) { //Check that the BLACK king is not in check  ChessBoard testBoard = (ChessBoard) board.copy(); //Duplicate the board  boolean abortMovement = false;  executeCastlingTestMove(testBoard, this.row, this.col, 0, 3); //Move to the first position it would go through  if(testBoard.isKingInCheck(((ChessPiece) this.getPiece()).getWhite()))//Check if it is in check at this point.  abortMovement = true;  executeCastlingTestMove(testBoard, 0, 3, 0, 2); //Move to the next position  if(testBoard.isKingInCheck(((ChessPiece) this.getPiece()).getWhite())) //Check if it is in check at this point.  abortMovement = true;  if(!abortMovement) { //If the king doesn't go through any danger positions  executeCheckedMove(board); //Performs Castling...  board.setPosition(0, 3, board.getPosition(0, 0));  deleteMovedPiece(0, 0, board);  } else {  throw new GameError("Cannot perform Castling, the King would be in Check during the move!");  }  } else {  throw new GameError("Cannot perform Castling, the King is in Check!");  }  } else {  throw new GameError("Cannot perform Castling, there are pieces inbetween.");  }  } else {  throw new GameError("Rook cannot perform Castling.");  }  } else {  throw new GameError("Rook is missing for Castling.");  }  }  } else {  throw new GameError("This King cannot Castle anymore.");  }  } else { //Illegal move.  throw new GameError("Invalid move, try again.");  }  }    //Checks if the movement is actually diagonal and not horizontal, vertical or anything else.  private boolean checkDiagonalMove() {  return (Math.abs(this.col - this.colDes) - Math.abs(this.row - this.rowDes)) == 0;  }    //Checks if the movement is actually horizontal or vertical and not diagonal or anything else.  private boolean checkHorizVertMove() {  return (this.col == this.colDes || this.row == this.rowDes);  }    /\*  \* Returns the direction of the move, being the number returned the one matching the direction  \* at the diagram underneath.  \*  \* 0 1 2  \* 7 \* 3  \* 6 5 4  \*/  private int checkDirection(int rowIni, int colIni, int rowEnd, int colEnd) {  if(rowIni == rowEnd) { //It's either E or W  if(colIni > colEnd) {  return ChessStatic.W;  } else { //colIni < colEnd (colIni always != colEnd at this point!) since both parameters can't be equal!  return ChessStatic.E;  }  } else if (rowIni > rowEnd) { //It's either NW, N or NE  if(colIni == colEnd) {  return ChessStatic.N;  } else if(colIni > colEnd) {  return ChessStatic.NW;  } else { //colIni < colEnd  return ChessStatic.NE;  }  } else { //rowIni < rowEnd //It's either SW, S or SE  if(colIni == colEnd) {  return ChessStatic.S;  } else if(colIni > colEnd) {  return ChessStatic.SW;  } else { //colIni < colEnd  return ChessStatic.SE;  }  }  }    //Check if there are any pieces between the positions selected (for bishops, rooks and queens)  //True means that the move can be performed, false means that there are pieces inbetween.  //The directions are shown on at the checkDirection() function above.  //  //Returns true if the movement can be performed, returns false if it can't be performed!!!  private boolean checkPiecesInbetween(int rowIni, int colIni, int rowEnd, int colEnd) {  int rowOffset, colOffset;  switch(checkDirection(rowIni, colIni, rowEnd, colEnd)) {  case ChessStatic.NW:  rowOffset = ChessStatic.NEGATIVE; colOffset = ChessStatic.NEGATIVE; break;  case ChessStatic.N:  rowOffset = ChessStatic.NEGATIVE; colOffset = ChessStatic.NEUTRAL; break;  case ChessStatic.NE:  rowOffset = ChessStatic.NEGATIVE; colOffset = ChessStatic.POSITIVE; break;  case ChessStatic.E:  rowOffset = ChessStatic.NEUTRAL; colOffset = ChessStatic.POSITIVE; break;  case ChessStatic.SE:  rowOffset = ChessStatic.POSITIVE; colOffset = ChessStatic.POSITIVE; break;  case ChessStatic.S:  rowOffset = ChessStatic.POSITIVE; colOffset = ChessStatic.NEUTRAL; break;  case ChessStatic.SW:  rowOffset = ChessStatic.POSITIVE; colOffset = ChessStatic.NEGATIVE; break;  case ChessStatic.W:  rowOffset = ChessStatic.NEUTRAL; colOffset = ChessStatic.NEGATIVE; break;  default:  throw new GameError("Internal error. Invalid move direction, this should be unreachable.");  }    int itCount = 0; //In case any error happens, this avoids an infinite loop.  int rowIt = rowIni, colIt = colIni;    while(itCount < ChessStatic.ITER\_LIMIT && (rowIt != rowEnd || colIt != colEnd)) {  rowIt += rowOffset;  colIt += colOffset;    if(rowIt == rowEnd && colIt == colEnd) break; //Destination position should not be checked so you can capture pieces.    if(chessBoard.getPosition(rowIt, colIt) != null) //If it found a piece on the way.  return false;    itCount++;  }    if(itCount >= ChessStatic.ITER\_LIMIT)  throw new GameError("Internal error. Move direction bugged. This should never happen.");    return true;  } //If returned false, throw an exception from the suitable function.    //Deletes a piece (used after copying it to a new cell)  private void deleteMovedPiece(int row, int col, Board board) {  board.setPosition(row, col, null);  }    public void revertBoard(Board chessBoard2, Board originalBoard) {  for(int rowX = ChessStatic.MIN\_DIM; rowX <= ChessStatic.MAX\_DIM; rowX++) {  for(int colY = ChessStatic.MIN\_DIM; colY <= ChessStatic.MAX\_DIM; colY++) {  if(originalBoard.getPosition(rowX, colY) == null) { //If the cell was empty...  if(chessBoard2.getPosition(rowX, colY) != null) { //and the cell is occupied now...  chessBoard2.setPosition(rowX, colY, null); //it deletes the contents.  } //else: It is still empty, nothing to do.  } else { //If the cell wasn't empty...  chessBoard2.setPosition(rowX, colY, originalBoard.getPosition(rowX, colY)); //we just recover the old piece, whichever it was.  }  }  }  }  /\*\* Creates a new instance of a ChessMove with parameters row, col, p.  \* \*/  protected GameMove createMove(int row, int col, int rowDes, int colDes, Piece p) {  return new ChessMove(row, col, rowDes, colDes, p);  }  /\*\* Sends a help message to help the user.  \* \*/  public String help() {  return "'Xrow Xcolumn Yrow Ycolumn', to move a piece from X to Y.";  }    /\*\* Notifies the user if they are using the wrong instructions.  \* \*/  public String toString() {  if (getPiece() == null) {  return help();  } else {  return getPiece() + " player moves (" + row + "," + col + ") to (" + rowDes + "," + colDes + ")";  }  }    @Override  public Piece getPiece() {  return this.chessPiece;  }  @Override  public GameMove fromString(Piece p, String str) {  //This is not needed in this game and should not be called.  return null;  }  public int getRow() {  return this.row;  }    public int getCol() {  return this.col;  }    public int getRowDes() {  return this.rowDes;  }    public int getColDes() {  return this.colDes;  }  } |

# Appendix IV. ChessRules code

<https://github.com/danielgarm/Chess-Delta/blob/master/src/chess/game/mvc/model/chessFiles/ChessRules.java>

|  |
| --- |
| public class ChessRules implements GameRules {    // This object is returned by gameOver to indicate that the game is not  // over. Just to avoid creating it multiple times, etc.  protected final Pair<State, Piece> gameInPlayResult = new Pair<State, Piece>(State.InPlay, null);    public ChessRules() {}  @Override  public String gameDesc() {  return "Chess";  }    @Override  public Board createBoard(List<Piece> pieces, List<Piece> pieceTypes) {  Board board = new ChessBoard();    //Sets the positions for every piece on their starting positions.  board.setPosition(0, 0, pieceTypes.get(ChessPieceID.BLACK\_ROOK\_A));  board.setPosition(0, 1, pieceTypes.get(ChessPieceID.BLACK\_KNIGHT));  board.setPosition(0, 2, pieceTypes.get(ChessPieceID.BLACK\_BISHOP));  board.setPosition(0, 3, pieceTypes.get(ChessPieceID.BLACK\_QUEEN));  board.setPosition(0, 4, pieceTypes.get(ChessPieceID.BLACK\_KING));  board.setPosition(0, 5, pieceTypes.get(ChessPieceID.BLACK\_BISHOP));  board.setPosition(0, 6, pieceTypes.get(ChessPieceID.BLACK\_KNIGHT));  board.setPosition(0, 7, pieceTypes.get(ChessPieceID.BLACK\_ROOK\_H));    board.setPosition(7, 0, pieceTypes.get(ChessPieceID.WHITE\_ROOK\_A));  board.setPosition(7, 1, pieceTypes.get(ChessPieceID.WHITE\_KNIGHT));  board.setPosition(7, 2, pieceTypes.get(ChessPieceID.WHITE\_BISHOP));  board.setPosition(7, 3, pieceTypes.get(ChessPieceID.WHITE\_QUEEN));  board.setPosition(7, 4, pieceTypes.get(ChessPieceID.WHITE\_KING));  board.setPosition(7, 5, pieceTypes.get(ChessPieceID.WHITE\_BISHOP));  board.setPosition(7, 6, pieceTypes.get(ChessPieceID.WHITE\_KNIGHT));  board.setPosition(7, 7, pieceTypes.get(ChessPieceID.WHITE\_ROOK\_H));    board.setPosition(1, 0, pieceTypes.get(ChessPieceID.BLACK\_PAWN\_A));  board.setPosition(6, 0, pieceTypes.get(ChessPieceID.WHITE\_PAWN\_A));  board.setPosition(1, 1, pieceTypes.get(ChessPieceID.BLACK\_PAWN\_B));  board.setPosition(6, 1, pieceTypes.get(ChessPieceID.WHITE\_PAWN\_B));  board.setPosition(1, 2, pieceTypes.get(ChessPieceID.BLACK\_PAWN\_C));  board.setPosition(6, 2, pieceTypes.get(ChessPieceID.WHITE\_PAWN\_C));  board.setPosition(1, 3, pieceTypes.get(ChessPieceID.BLACK\_PAWN\_D));  board.setPosition(6, 3, pieceTypes.get(ChessPieceID.WHITE\_PAWN\_D));  board.setPosition(1, 4, pieceTypes.get(ChessPieceID.BLACK\_PAWN\_E));  board.setPosition(6, 4, pieceTypes.get(ChessPieceID.WHITE\_PAWN\_E));  board.setPosition(1, 5, pieceTypes.get(ChessPieceID.BLACK\_PAWN\_F));  board.setPosition(6, 5, pieceTypes.get(ChessPieceID.WHITE\_PAWN\_F));  board.setPosition(1, 6, pieceTypes.get(ChessPieceID.BLACK\_PAWN\_G));  board.setPosition(6, 6, pieceTypes.get(ChessPieceID.WHITE\_PAWN\_G));  board.setPosition(1, 7, pieceTypes.get(ChessPieceID.BLACK\_PAWN\_H));  board.setPosition(6, 7, pieceTypes.get(ChessPieceID.WHITE\_PAWN\_H));    setStartingAttributes(board);  ((ChessBoard) board).addCurrentPosition();    return board;  }    //In the case of a game restart, the attributes do not return to their original, therefore this function is needed.  private void setStartingAttributes(Board b) {  if(b instanceof ChessBoard) {  ChessBoard board = (ChessBoard) b;  for(int rowX = ChessStatic.MIN\_DIM; rowX <= ChessStatic.MAX\_DIM; rowX++) {  for(int colY = ChessStatic.MIN\_DIM; colY <= ChessStatic.MAX\_DIM; colY++) {  if(board.getPosition(rowX, colY) != null) { //If there's a piece  if(board.getPosition(rowX, colY) instanceof Rook) {  ((Rook) board.getPosition(rowX, colY)).setCastle(true);  } else if(board.getPosition(rowX, colY) instanceof Pawn) {  ((Pawn) board.getPosition(rowX, colY)).setPassant(false);  ((Pawn) board.getPosition(rowX, colY)).setFirstMove(true);  } else if(board.getPosition(rowX, colY) instanceof King) {  ((King) board.getPosition(rowX, colY)).setCastle(true);  }  }  }  }  }  }    @Override  public Pair<State, Piece> updateState(Board board, List<Piece> pieces, Piece lastPlayer) {  Pair<State, Piece> gameState = this.gameInPlayResult; //gameInPlayResult = IF GAME IS NOT FINISHED, STILL IN PLAY    if(nextPlayer(board, pieces, lastPlayer) == null) { //If next player can't move, someone has won or there's a Stalemate!  //There must be a Checkmate or Stalemate, therefore we check the end of the game.  gameState = checkWinnerEndGame(board, pieces, lastPlayer);  } else if(((ChessBoard) board).checkMovesRulesLimit()) { //Check if no pawn has been moved or no piece has been captured in the last 50 moves  gameState = new Pair<State, Piece>(State.Draw, null);  }    return gameState;  }    /\*\*  \* This function must only be called if the game has ended for a player not being able to move. Not in any other case!!  \* This could only be called after either Checkmate or Stalemate!!  \*/  private Pair<State, Piece> checkWinnerEndGame(Board board, List<Piece> pieces, Piece lastPlayer) {  ChessPiece lastChessPlayer = (ChessPiece) lastPlayer;  //Creates an empty chess move to check if the enemy king is in check  if(((ChessBoard) board).isKingInCheck(!lastChessPlayer.getWhite())) { //Checks if the King from the player that hasn't made the last move is in Check. (It would be Checkmate)  //If there is a Checkmate  if(lastChessPlayer.getWhite()) {  //If the last one to move was White  return new Pair<State, Piece>(State.Won, pieces.get(ChessStatic.WHITE\_ID)); //White wins!  } else {  //If the last one to move was Black  return new Pair<State, Piece>(State.Won, pieces.get(ChessStatic.BLACK\_ID)); //Black wins!  }  } else {  //If there is a Stalemate  return new Pair<State, Piece>(State.Draw, null); //There is a Draw  }  }    @Override  public List<GameMove> validMoves(Board board, List<Piece> playersPieces, Piece turn) {  Board originalBoard = board;  boolean isWhiteTurn = ((ChessPiece) turn).getWhite();  Board testBoard = originalBoard.copy();  List<GameMove> legalMoves = new ArrayList<GameMove>(); //Every single possible move on the board is saved here  for (int rowX = ChessStatic.MIN\_DIM; rowX <= ChessStatic.MAX\_DIM; rowX++) { //Exploring rows and cols.  for (int colY = ChessStatic.MIN\_DIM; colY <= ChessStatic.MAX\_DIM; colY++) {  if(originalBoard.getPosition(rowX, colY) != null && //If the position is not empty...  ((ChessPiece) originalBoard.getPosition(rowX, colY)).getWhite() == isWhiteTurn){ //...and it is from the current turn.  if(originalBoard.getPosition(rowX, colY) instanceof Pawn) {  this.addPawnMoves(testBoard, originalBoard, legalMoves, rowX, colY, turn);  } else if(originalBoard.getPosition(rowX, colY) instanceof Rook) {  this.addHorizVertMoves(testBoard, originalBoard, legalMoves, rowX, colY, turn);  } else if(originalBoard.getPosition(rowX, colY) instanceof Knight) {  this.addKnightMoves(testBoard, originalBoard, legalMoves, rowX, colY, turn);  } else if(originalBoard.getPosition(rowX, colY) instanceof Bishop) {  this.addDiagonalMoves(testBoard, originalBoard, legalMoves, rowX, colY, turn);  } else if(originalBoard.getPosition(rowX, colY) instanceof Queen) {  this.addHorizVertMoves(testBoard, originalBoard, legalMoves, rowX, colY, turn);  this.addDiagonalMoves(testBoard, originalBoard, legalMoves, rowX, colY, turn);  } else if(originalBoard.getPosition(rowX, colY) instanceof King) {  this.addKingMoves(testBoard, originalBoard, legalMoves, rowX, colY, turn);  } else {  throw new GameError("Internal error, piece type not found!");  }  }  }  }  return legalMoves;  }    private void checkAndAdd(ChessMove testMove, Board testBoard, Board originalBoard, List<GameMove> legalMoves) {  if(testMove.isMoveLegal((ChessBoard) testBoard)) {  testMove.revertBoard(testBoard, originalBoard);  legalMoves.add(testMove);  }  }    private void addHorizVertMoves(Board testBoard, Board originalBoard, List<GameMove> legalMoves, int rowX, int colY, Piece turn) {  int rowDes = rowX - 1, colDes = colY;  while(rowDes >= ChessStatic.MIN\_DIM) { //North direction  this.checkAndAdd(new ChessMove(rowX, colY, rowDes, colDes, turn), testBoard, originalBoard, legalMoves);  if(originalBoard.getPosition(rowDes, colDes) != null) break;  rowDes--;  }  rowDes = rowX + 1; colDes = colY;  while(rowDes <= ChessStatic.MAX\_DIM) { //South direction  this.checkAndAdd(new ChessMove(rowX, colY, rowDes, colDes, turn), testBoard, originalBoard, legalMoves);  if(originalBoard.getPosition(rowDes, colDes) != null) break;  rowDes++;  }  rowDes = rowX; colDes = colY + 1;  while(colDes <= ChessStatic.MAX\_DIM) { //East direction  this.checkAndAdd(new ChessMove(rowX, colY, rowDes, colDes, turn), testBoard, originalBoard, legalMoves);  if(originalBoard.getPosition(rowDes, colDes) != null) break;  colDes++;  }  rowDes = rowX; colDes = colY - 1;  while(colDes >= ChessStatic.MIN\_DIM) { //West direction  this.checkAndAdd(new ChessMove(rowX, colY, rowDes, colDes, turn), testBoard, originalBoard, legalMoves);  if(originalBoard.getPosition(rowDes, colDes) != null) break;  colDes--;  }  }    private void addDiagonalMoves(Board testBoard, Board originalBoard, List<GameMove> legalMoves, int rowX, int colY, Piece turn) {  int rowDes = rowX - 1, colDes = colY - 1;  while(rowDes >= ChessStatic.MIN\_DIM && colDes >= ChessStatic.MIN\_DIM) { //NorthWest direction  this.checkAndAdd(new ChessMove(rowX, colY, rowDes, colDes, turn), testBoard, originalBoard, legalMoves);  if(originalBoard.getPosition(rowDes, colDes) != null) break;  rowDes--;  colDes--;  }    rowDes = rowX - 1; colDes = colY + 1;  while(rowDes >= ChessStatic.MIN\_DIM && colDes <= ChessStatic.MAX\_DIM) { //NorthEast direction  this.checkAndAdd(new ChessMove(rowX, colY, rowDes, colDes, turn), testBoard, originalBoard, legalMoves);  if(originalBoard.getPosition(rowDes, colDes) != null) break;  rowDes--;  colDes++;  }    rowDes = rowX + 1; colDes = colY + 1;  while(colDes <= ChessStatic.MAX\_DIM && rowDes <= ChessStatic.MAX\_DIM) { //SouthEast direction  this.checkAndAdd(new ChessMove(rowX, colY, rowDes, colDes, turn), testBoard, originalBoard, legalMoves);  if(originalBoard.getPosition(rowDes, colDes) != null) break;  colDes++;  rowDes++;  }    rowDes = rowX + 1; colDes = colY - 1;  while(colDes >= ChessStatic.MIN\_DIM && rowDes <= ChessStatic.MAX\_DIM) { //SouthWest direction  this.checkAndAdd(new ChessMove(rowX, colY, rowDes, colDes, turn), testBoard, originalBoard, legalMoves);  if(originalBoard.getPosition(rowDes, colDes) != null) break;  colDes--;  rowDes++;  }  }    private void addKnightMoves(Board testBoard, Board originalBoard, List<GameMove> legalMoves, int rowX, int colY, Piece turn) {  this.checkAndAdd(new ChessMove(rowX, colY, rowX - 2, colY - 1, turn), testBoard, originalBoard, legalMoves); //2 up 1 left  this.checkAndAdd(new ChessMove(rowX, colY, rowX - 2, colY + 1, turn), testBoard, originalBoard, legalMoves); //2 up 1 right  this.checkAndAdd(new ChessMove(rowX, colY, rowX + 2, colY - 1, turn), testBoard, originalBoard, legalMoves); //2 down 1 left  this.checkAndAdd(new ChessMove(rowX, colY, rowX + 2, colY + 1, turn), testBoard, originalBoard, legalMoves); //2 down 1 right  this.checkAndAdd(new ChessMove(rowX, colY, rowX - 1, colY - 2, turn), testBoard, originalBoard, legalMoves); //2 left 1 up  this.checkAndAdd(new ChessMove(rowX, colY, rowX + 1, colY - 2, turn), testBoard, originalBoard, legalMoves); //2 left 1 down  this.checkAndAdd(new ChessMove(rowX, colY, rowX - 1, colY + 2, turn), testBoard, originalBoard, legalMoves); //2 right 1 up  this.checkAndAdd(new ChessMove(rowX, colY, rowX + 1, colY + 2, turn), testBoard, originalBoard, legalMoves); //2 right 1 down  }    private void addPawnMoves(Board testBoard, Board originalBoard, List<GameMove> legalMoves, int rowX, int colY, Piece turn) {  if(((ChessPiece) originalBoard.getPosition(rowX, colY)).getWhite()) { //If it's a white Pawn it will explore some moves...  this.checkAndAdd(new ChessMove(rowX, colY, rowX - 1, colY, turn), testBoard, originalBoard, legalMoves); //Simple move  this.checkAndAdd(new ChessMove(rowX, colY, rowX - 2, colY, turn), testBoard, originalBoard, legalMoves); //Double move  this.checkAndAdd(new ChessMove(rowX, colY, rowX - 1, colY - 1, turn), testBoard, originalBoard, legalMoves); //Diagonal move left  this.checkAndAdd(new ChessMove(rowX, colY, rowX - 1, colY + 1, turn), testBoard, originalBoard, legalMoves); //Diagonal move right  } else { //...if it's a black one it will explore others.  this.checkAndAdd(new ChessMove(rowX, colY, rowX + 1, colY, turn), testBoard, originalBoard, legalMoves); //Simple move  this.checkAndAdd(new ChessMove(rowX, colY, rowX + 2, colY, turn), testBoard, originalBoard, legalMoves); //Double move  this.checkAndAdd(new ChessMove(rowX, colY, rowX + 1, colY - 1, turn), testBoard, originalBoard, legalMoves); //Diagonal move left  this.checkAndAdd(new ChessMove(rowX, colY, rowX + 1, colY + 1, turn), testBoard, originalBoard, legalMoves); //Diagonal move right  }  }    private void addKingMoves(Board testBoard, Board originalBoard, List<GameMove> legalMoves, int rowX, int colY, Piece turn) {  this.checkAndAdd(new ChessMove(rowX, colY, rowX - 1, colY - 1, turn), testBoard, originalBoard, legalMoves); //NW move  this.checkAndAdd(new ChessMove(rowX, colY, rowX - 1, colY, turn), testBoard, originalBoard, legalMoves); //N move  this.checkAndAdd(new ChessMove(rowX, colY, rowX - 1, colY + 1, turn), testBoard, originalBoard, legalMoves); //NE move  this.checkAndAdd(new ChessMove(rowX, colY, rowX, colY + 1, turn), testBoard, originalBoard, legalMoves); //E move  this.checkAndAdd(new ChessMove(rowX, colY, rowX, colY - 1, turn), testBoard, originalBoard, legalMoves); //W move  this.checkAndAdd(new ChessMove(rowX, colY, rowX + 1, colY - 1, turn), testBoard, originalBoard, legalMoves); //SW move  this.checkAndAdd(new ChessMove(rowX, colY, rowX + 1, colY, turn), testBoard, originalBoard, legalMoves); //S move  this.checkAndAdd(new ChessMove(rowX, colY, rowX + 1, colY + 1, turn), testBoard, originalBoard, legalMoves); //SE move    this.checkAndAdd(new ChessMove(rowX, colY, rowX, colY + 2, turn), testBoard, originalBoard, legalMoves); //Short Castling  this.checkAndAdd(new ChessMove(rowX, colY, rowX, colY - 2, turn), testBoard, originalBoard, legalMoves); //Long Castling  }  @Override  public Piece nextPlayer(Board board, List<Piece> playersPieces, Piece lastPlayer){  Piece nextPlayer;    if(lastPlayer == playersPieces.get(ChessStatic.WHITE\_ID))  nextPlayer = playersPieces.get(ChessStatic.BLACK\_ID);  else if (lastPlayer == playersPieces.get(ChessStatic.BLACK\_ID))  nextPlayer = playersPieces.get(ChessStatic.WHITE\_ID);  else  return null; //throw new GameError("Something went wrong while changing turns. Unrecognised player. This should be unreachable.");    if(validMoves(board, playersPieces, nextPlayer).isEmpty()) //Check if he can move any of his pieces  return null; //Returns null if next player can't move!  return nextPlayer;  }  @Override  public Piece initialPlayer(Board board, List<Piece> pieces) {  return pieces.get(ChessStatic.WHITE\_ID); //White always starts  }  @Override  public double evaluate(Board board, List<Piece> pieces, Piece turn, Piece p) {  //This function should never be called  /\*ChessBoardEvaluator evaluator = new ChessBoardEvaluator((ChessBoard) board, ((ChessPiece) turn).getWhite());  return evaluator.getRating();\*/  return 0;  }  @Override  public int minPlayers() {  return 2;  }  @Override  public int maxPlayers() {  return 2;  }  } |

# Appendix V. Chess pieces code

The following sections include the code for *ChessPiece, Bishop, King, Knight, Pawn, Queen* and *Rook.* The code for *ChessPiece* can also be found at:

<https://github.com/danielgarm/Chess-Delta/blob/master/src/chess/game/mvc/model/chessPieces/ChessPiece.java>

and the code for each one of the implemented pieces can be seen at:

<https://github.com/danielgarm/Chess-Delta/tree/master/src/chess/game/mvc/model/chessPieces/chessPiecesImp>

## Appendix V-a. ChessPiece code

|  |
| --- |
| public class ChessPiece extends Piece {    private static final long serialVersionUID = 1L;    private boolean isWhite;    public ChessPiece() {  super();  }    public ChessPiece(boolean isWhite) {  super();  this.isWhite = isWhite;  }    public ChessPiece(String id, boolean isWhite) {  super(id);  this.isWhite = isWhite;  }    public void setWhite(boolean isWhite) {  this.isWhite = isWhite;  }    public boolean getWhite() {  return this.isWhite;  }  public ChessPiece copyPiece() { //Not going to be used  return new ChessPiece(this.getId(), this.isWhite);  }  } |

## Appendix V-b. Bishop code

|  |
| --- |
| public class Bishop extends ChessPiece {  private static final long serialVersionUID = 1L;  public Bishop() {  super();  }    public Bishop(boolean isWhite) {  super(isWhite);  }  public Bishop(String id, boolean isWhite) {  super(id, isWhite);  }    @Override  public Bishop copyPiece() {  return new Bishop(this.getId(), this.getWhite());  }  } |

## Appendix V-c. King code

|  |
| --- |
| public class King extends ChessPiece {  private static final long serialVersionUID = 1L;  private boolean canCastle;    public King() {  super();  this.canCastle = true;  }    public King(boolean isWhite) {  super(isWhite);  this.canCastle = true;  }    public King(String id, boolean isWhite, boolean canCastle) {  super(id, isWhite);  this.canCastle = canCastle;  }  public King(String id, boolean isWhite) {  super(id, isWhite);  this.canCastle = true;  }    public boolean getCastle() {  return canCastle;  }    public void setCastle(boolean canCastle) {  this.canCastle = canCastle;  }    @Override  public King copyPiece() {  return new King(this.getId(), this.getWhite(), this.canCastle);  }  } |

## Appendix V-d. Knight code

|  |
| --- |
| public class Knight extends ChessPiece {  private static final long serialVersionUID = 1L;  public Knight() {  super();  }    public Knight(boolean isWhite) {  super(isWhite);  }  public Knight(String id, boolean isWhite) {  super(id, isWhite);  }    @Override  public Knight copyPiece() {  return new Knight(this.getId(), this.getWhite());  }  } |

## Appendix V-e. Pawn code

|  |
| --- |
| public class Pawn extends ChessPiece {  private static final long serialVersionUID = 1L;  private boolean canPassant; //If this pawn can be captured "En Passant"  private boolean firstMove; //If this pawn can perform an opening move    public Pawn() {  super();  this.canPassant = false;  this.firstMove = true;  }    public Pawn(boolean isWhite) {  super(isWhite);  this.canPassant = false;  this.firstMove = true;  }    public Pawn(String id, boolean isWhite, boolean canPassant, boolean firstMove) {  super(id, isWhite);  this.canPassant = canPassant;  this.firstMove = firstMove;  }  public Pawn(String id, boolean isWhite) {  super(id, isWhite);  this.canPassant = false;  this.firstMove = true;  }    public boolean getPassant() {  return canPassant;  }    public void setPassant(boolean canPassant) {  this.canPassant = canPassant;  }    public boolean getFirstMove() {  return firstMove;  }    public void setFirstMove(boolean firstMove) {  this.firstMove = firstMove;  }    @Override  public Pawn copyPiece() {  return new Pawn(this.getId(), this.getWhite(), this.canPassant, this.firstMove);  }  } |

## Appendix V-f. Queen code

|  |
| --- |
| public class Queen extends ChessPiece {  private static final long serialVersionUID = 1L;  public Queen() {  super();  }    public Queen(boolean isWhite) {  super(isWhite);  }  public Queen(String id, boolean isWhite) {  super(id, isWhite);  }    @Override  public Queen copyPiece() {  return new Queen(this.getId(), this.getWhite());  }  } |

## Appendix V-g. Rook code

|  |
| --- |
| public class Rook extends ChessPiece {  private static final long serialVersionUID = 1L;    private boolean canCastle;    public Rook() {  super();  this.canCastle = true;  }    public Rook(boolean isWhite) {  super(isWhite);  this.canCastle = true;  }    public Rook(boolean isWhite, boolean canCastle) {  super(isWhite);  this.canCastle = canCastle;  }    public Rook(String id, boolean isWhite, boolean canCastle) {  super(id, isWhite);  this.canCastle = canCastle;  }  public Rook(String id, boolean isWhite) {  super(id, isWhite);  this.canCastle = true;  }    public boolean getCastle() {  return canCastle;  }    public void setCastle(boolean canCastle) {  this.canCastle = canCastle;  }    @Override  public Rook copyPiece() {  return new Rook(this.getId(), this.getWhite(), this.canCastle);  }  } |

# Appendix VI. Chess Views code

The code for both documents can be found at:

<https://github.com/danielgarm/Chess-Delta/tree/master/src/chess/game/mvc/view/chessViews>

## Appendix VI-a. ChessBoardComponent code

|  |
| --- |
| @SuppressWarnings("serial")  public abstract class ChessBoardComponent extends JComponent { //Draws the board and pieces  private int \_CELL\_HEIGHT = 50;  private int \_CELL\_WIDTH = 50;  public static final String piecesPath = "./img/";    private BufferedImage whitePawn;  private BufferedImage whiteRook;  private BufferedImage whiteKnight;  private BufferedImage whiteBishop;  private BufferedImage whiteQueen;  private BufferedImage whiteKing;  private BufferedImage blackPawn;  private BufferedImage blackRook;  private BufferedImage blackKnight;  private BufferedImage blackBishop;  private BufferedImage blackQueen;  private BufferedImage blackKing;    private int rows;  private int cols;  private Board board;  /\*\*  \* Default constructor  \*/  public ChessBoardComponent() {  loadImages();  createGUI();  }    /\*\*  \* Constructor with parameters  \* @param rows  \* @param cols  \*/  public ChessBoardComponent(int rows, int cols) {  this.rows = rows;  this.cols = cols;  loadImages();  createGUI();  }    private void loadImages() {  try {  this.whitePawn = ImageIO.read(new File(piecesPath + "WhitePawn.png"));  this.whiteRook = ImageIO.read(new File(piecesPath + "WhiteRook.png"));  this.whiteKnight = ImageIO.read(new File(piecesPath + "WhiteKnight.png"));  this.whiteBishop = ImageIO.read(new File(piecesPath + "WhiteBishop.png"));  this.whiteQueen = ImageIO.read(new File(piecesPath + "WhiteQueen.png"));  this.whiteKing = ImageIO.read(new File(piecesPath + "WhiteKing.png"));    this.blackPawn = ImageIO.read(new File(piecesPath + "BlackPawn.png"));  this.blackRook = ImageIO.read(new File(piecesPath + "BlackRook.png"));  this.blackKnight = ImageIO.read(new File(piecesPath + "BlackKnight.png"));  this.blackBishop = ImageIO.read(new File(piecesPath + "BlackBishop.png"));  this.blackQueen = ImageIO.read(new File(piecesPath + "BlackQueen.png"));  this.blackKing = ImageIO.read(new File(piecesPath + "BlackKing.png"));  } catch (IOException e) {  e.printStackTrace();  }  }  /\*\*  \* Function that creates the graphic interface  \*/  private void createGUI() {  addMouseListener(new MouseListener() {  @Override  public void mouseReleased(MouseEvent e) {  //System.out.println("Mouse Released: " + "(" + e.getX() + "," + e.getY() + ")");  }  @Override  public void mousePressed(MouseEvent e) {  //System.out.println("Mouse Pressed: " + "(" + e.getX() + "," + e.getY() + ")");  }  @Override  public void mouseExited(MouseEvent e) {  //System.out.println("Mouse Exited Component: " + "(" + e.getX() + "," + e.getY() + ")");  }  @Override  public void mouseEntered(MouseEvent e) {  //System.out.println("Mouse Entered Component: " + "(" + e.getX() + "," + e.getY() + ")");  }    /\*\*  \* Gets the cell where the mouse was clicked  \*/  @Override  public void mouseClicked(MouseEvent e) {    //System.out.println("Mouse Button " + e.getButton() + " Clicked at " + "(" + e.getX() + "," + e.getY() + ")");  int col = ((e.getX() - 19) / \_CELL\_WIDTH);  int row = ((e.getY() - 16) / \_CELL\_HEIGHT);  int mouseButton = 0;  if (SwingUtilities.isLeftMouseButton(e)) {  mouseButton = 1;  } else if (SwingUtilities.isMiddleMouseButton(e)) {  mouseButton = 2;  } else if (SwingUtilities.isRightMouseButton(e)) {  mouseButton = 3;  } else {  return;  }    if(e.getX() >= 19 && e.getY() >= 16) {  //The click is only detected if the player clicked inside the chess board and not on the edges of the board.  ChessBoardComponent.this.mouseClicked(row, col, mouseButton);  }  }  });    this.setSize(new Dimension(rows \* \_CELL\_HEIGHT, cols \* \_CELL\_WIDTH));    repaint();  }    /\*\*  \* Paints the background  \*/  public void paintComponent(Graphics g) {  super.paintComponent(g);  if(board == null) {  g.drawString("Game has not started yet...", 300, 300);  return;  }  g.setColor(Color.lightGray);  g.fillRect(0, 0, this.getWidth(), this.getHeight());    //Make cells slightly smaller to be able to fit column and row numbers and letters.  \_CELL\_WIDTH = this.getWidth() / board.getCols() - 2;  \_CELL\_HEIGHT = this.getHeight() / board.getRows() - 2;    for (int i = 0; i < board.getCols(); i++)  for (int j = 0; j < board.getRows(); j++)  drawCell(i, j, g);    g.setColor(Color.black);    //Letters are 2 px wide.  //y = 12 is for the letter not to be too close to the top of the screen.  //16 is for the offset on drawCell(...), where the numbers will be.  g.drawString("a", 16 + 0 \* \_CELL\_WIDTH + (\_CELL\_WIDTH / 2), 12);  g.drawString("b", 16 + 1 \* \_CELL\_WIDTH + (\_CELL\_WIDTH / 2), 12);  g.drawString("c", 16 + 2 \* \_CELL\_WIDTH + (\_CELL\_WIDTH / 2), 12);  g.drawString("d", 16 + 3 \* \_CELL\_WIDTH + (\_CELL\_WIDTH / 2), 12);  g.drawString("e", 16 + 4 \* \_CELL\_WIDTH + (\_CELL\_WIDTH / 2), 12);  g.drawString("f", 16 + 5 \* \_CELL\_WIDTH + (\_CELL\_WIDTH / 2), 12);  g.drawString("g", 16 + 6 \* \_CELL\_WIDTH + (\_CELL\_WIDTH / 2), 12);  g.drawString("h", 16 + 7 \* \_CELL\_WIDTH + (\_CELL\_WIDTH / 2), 12);    for (int row = 1; row <= 8; row++) {  //Draws numbers on borders.  g.drawString(String.valueOf(9 - row), 6, 21 + (row - 1) \* \_CELL\_HEIGHT + (\_CELL\_HEIGHT / 2));  }  }  /\*\*  \* Draws one cell and the pieces into them  \* @param row  \* @param col  \* @param g  \*/  private void drawCell(int row, int col, Graphics g) {  int x = col \* \_CELL\_WIDTH + 19;  int y = row \* \_CELL\_HEIGHT + 16;  //IMPORTANT: 19 and 16 express the offset from the left side and top side respectively  //x and y are coordinates in the space, not rows/cols!!    //Selects the colour for each tile depending on position  if(ChessStatic.rowClicked == row && ChessStatic.colClicked == col && !ChessStatic.firstClick) {  g.setColor(new Color(45, 152, 38)); //Color for the selected tile after first click.  } else if(row % 2 == 0 && col % 2 == 1 || row % 2 == 1 && col % 2 == 0) {  g.setColor(new Color(204, 102, 0)); //Dark tiles  } else {  g.setColor(new Color(255, 217, 179)); //Light tiles  }  g.fillRect(x, y, \_CELL\_WIDTH, \_CELL\_HEIGHT);  //Checks if there's a piece on the selected position. If there is, draws it depending on the type and colour.  this.drawPiece(row, col, x, y, g);  }    private void drawPiece(int row, int col, int x, int y, Graphics g) {  if(board.getPosition(row, col) != null) {  if(this.getPiece(row, col, board) instanceof Pawn) {  if (((ChessPiece) this.getPiece(row, col, board)).getWhite()) {  g.drawImage(this.whitePawn, x, y, \_CELL\_WIDTH, \_CELL\_HEIGHT, this);  } else {  g.drawImage(this.blackPawn, x, y, \_CELL\_WIDTH, \_CELL\_HEIGHT, this);  }  } else if (this.getPiece(row, col, board) instanceof Rook) {  if (((ChessPiece) this.getPiece(row, col, board)).getWhite()) {  g.drawImage(this.whiteRook, x, y, \_CELL\_WIDTH, \_CELL\_HEIGHT, this);  } else {  g.drawImage(this.blackRook, x, y, \_CELL\_WIDTH, \_CELL\_HEIGHT, this);  }  } else if (this.getPiece(row, col, board) instanceof Knight) {  if (((ChessPiece) this.getPiece(row, col, board)).getWhite()) {  g.drawImage(this.whiteKnight, x, y, \_CELL\_WIDTH, \_CELL\_HEIGHT, this);  } else {  g.drawImage(this.blackKnight, x, y, \_CELL\_WIDTH, \_CELL\_HEIGHT, this);  }  } else if (this.getPiece(row, col, board) instanceof Bishop) {  if (((ChessPiece) this.getPiece(row, col, board)).getWhite()) {  g.drawImage(this.whiteBishop, x, y, \_CELL\_WIDTH, \_CELL\_HEIGHT, this);  } else {  g.drawImage(this.blackBishop, x, y, \_CELL\_WIDTH, \_CELL\_HEIGHT, this);  }  } else if (this.getPiece(row, col, board) instanceof Queen) {  if (((ChessPiece) this.getPiece(row, col, board)).getWhite()) {  g.drawImage(this.whiteQueen, x, y, \_CELL\_WIDTH, \_CELL\_HEIGHT, this);  } else {  g.drawImage(this.blackQueen, x, y, \_CELL\_WIDTH, \_CELL\_HEIGHT, this);  }  } else if (this.getPiece(row, col, board) instanceof King) {  if (((ChessPiece) this.getPiece(row, col, board)).getWhite()) {  g.drawImage(this.whiteKing, x, y, \_CELL\_WIDTH, \_CELL\_HEIGHT, this);  } else {  g.drawImage(this.blackKing, x, y, \_CELL\_WIDTH, \_CELL\_HEIGHT, this);  }  } else {  throw new GameError("Piece type not recognised! This point should never be reached!");  }  }  }    /\*\*  \* Returns the piece on a position  \* @param row  \* @param col  \* @param board  \* @return  \*/  private Piece getPiece(int row, int col, Board board) {  return board.getPosition(row, col);  }  /\*\*  \* Redraws the board  \* @param b  \*/  public void redraw(Board b) {  this.board = b;  repaint();  }  protected abstract void mouseClicked(int row, int col, int mouseButton);  protected abstract boolean isPlayerPiece(Piece p);  protected abstract Color getPieceColor(Piece p);  } |

## Appendix VI-b. ChessWindowSwingView code

|  |
| --- |
| public abstract class ChessWindowSwingView extends JFrame implements GameObserver { //Hace la ventana y el pane    protected Controller ctrl; //Controller  protected Observable<GameObserver> game; //Game    private Piece localPiece; //Local piece  private Piece turn; //Piece who can play  private List<Piece> pieces; //List of pieces    private Board board; //Board    private Player randPlayer;  private List<Player> aiPlayers;    private Map<Piece, Color> pieceColors;  private Map<Piece, PlayerMode> playerModes;    private JPanel backgroundPane;  private JPanel boardPanel;  private JPanel sidebarPanel;  private JTextArea statusArea;  private JComboBox<Piece> playerModePieces;  private JComboBox<PlayerMode> modes;  private PlayerInfoTableModel tableModel;    private JButton randomMove;  private JButton intelligentMove;  private JButton setMode;  private JButton restart;  private JButton quit;  private String gameDesc;    /\*\*  \* Available playermodes  \* @author Dani-MacBookPro  \*  \*/  enum PlayerMode {  MANUAL("Manual", "Manual"),  RANDOM("Dummy AI", "Dummy AI"),  MINMAX2("MinMax Depth 2", "MinMax Depth 2"),  MINMAX3("MinMax Depth 3", "MinMax Depth 3"),  MINMAX4("MinMax Depth 4", "MinMax Depth 4"),  MINMAX5("MinMax Depth 5", "MinMax Depth 5"),  AB2("AlphaBeta Depth 2", "AlphaBeta Depth 2"),  AB3("AlphaBeta Depth 3", "AlphaBeta Depth 3"),  AB4("AlphaBeta Depth 4", "AlphaBeta Depth 4"),  AB5("AlphaBeta Depth 5", "AlphaBeta Depth 5"),  AB6("AlphaBeta Depth 6", "AlphaBeta Depth 6"),  AB7("AlphaBeta Depth 7", "AlphaBeta Depth 7");  private String id;  private String desc;  PlayerMode(String id, String desc) {  this.id = id;  this.desc = desc;  }  public String getId() {  return id;  }  public String getDesc() {  return desc;  }  @Override  public String toString() {  return id;  }  }    /\*\*  \* Constructor for the Swing View  \* @param g  \* @param c  \* @param localPiece  \* @param randPlayer  \* @param aiPlayer  \*/  public ChessWindowSwingView(Observable<GameObserver> g, Controller c, Piece localPiece, Player randPlayer, List<Player> aiPlayers) {  this.game = g;  this.ctrl = c;  this.localPiece = localPiece;  this.randPlayer = randPlayer;  this.aiPlayers = aiPlayers;  this.pieceColors = new HashMap<Piece, Color>();  this.playerModes = new HashMap<Piece, PlayerMode>();    createGUI();  game.addObserver(this);  }    /\*\*  \* Creates the whole GUI  \*/  private void createGUI() {  this.backgroundPane = new JPanel(new BorderLayout()); //Crea el marco grande del fondo  this.setContentPane(backgroundPane);    boardPanel = new JPanel(new BorderLayout()); //Crea el trozo que se usará para el tablero  backgroundPane.add(boardPanel, BorderLayout.CENTER); //Lo coloca en la posición CENTER    startBoardGUI(); //Esto inicializa la tabla    sidebarPanel = new JPanel(); //Crea la barra lateral  sidebarPanel.setLayout(new BoxLayout(sidebarPanel, BoxLayout.Y\_AXIS)); //Lo establece como vertical  backgroundPane.add(sidebarPanel, BorderLayout.LINE\_END); //Lo añade al lado derecho de la ventana    startSidebarGUI(); //Esto inicializa el lateral    this.addWindowListener(new WindowListener() {  public void windowClosing(WindowEvent e) {  quit();  }  public void windowOpened(WindowEvent e) {  }  @Override  public void windowIconified(WindowEvent e) {  }  @Override  public void windowDeiconified(WindowEvent e) {  }  @Override  public void windowDeactivated(WindowEvent e) {  }  @Override  public void windowClosed(WindowEvent e) {  }  @Override  public void windowActivated(WindowEvent e) {  }    });    setExtendedState(MAXIMIZED\_BOTH);  setDefaultCloseOperation(JFrame.DO\_NOTHING\_ON\_CLOSE);  pack();  setVisible(true);  }    /\*\*  \* Quits the game (with confirmation)  \*/  protected void quit() { //Method to execute when the user attempts to quit the game  if (JOptionPane.showOptionDialog(new JFrame(), "Are you sure you want to exit the game?", "Exit confirmation", JOptionPane.YES\_NO\_OPTION, JOptionPane.QUESTION\_MESSAGE, null, null, null) == 0)  {  try  {  ctrl.stop();  }  catch (GameError \_e) {}  setVisible(false);  dispose();  System.exit(0);  }  }  /\*\*  \* Getter  \* @return  \*/  final protected Piece getTurn() {  return turn;  }    /\*\*  \* Getter  \* @return  \*/  final protected Board getBoard() {  return board;  }    /\*\*  \* Getter  \* @return  \*/  final public List<Piece> getPieces() {  return pieces;  }    /\*\*  \* Getter  \* @param p  \* @return  \*/  final protected Color getPieceColorFromSwing(Piece p) {  return pieceColors.get(p);  }    /\*\*  \* Setter  \* @param p  \* @param c  \* @return  \*/  final protected Color setPieceColor(Piece p, Color c) {  return pieceColors.put(p,c);  }    /\*\*  \* Setter for the board area  \* @param c  \*/  final protected void setBoardArea(JComponent c) {  boardPanel.add(c, BorderLayout.CENTER);  }    /\*\*  \* Adds a message to the sidebar  \* @param msg  \*/  final protected void addMsg(String msg)  {  statusArea.append(msg + "\n");  }    /\*\*  \* Makes a manual move  \* @param manualPlayer  \*/  final protected void decideMakeManualMove(Player manualPlayer) {  try {  ctrl.makeMove(manualPlayer);  } catch (GameError e) {  System.out.println(e.getMessage());  }  }    /\*\*  \* Make an auto move  \*/  final protected void decideMakeAutomaticMove() {  switch(playerModes.get(turn)) {  case MINMAX2: intelligentMove(0); break;  case MINMAX3: intelligentMove(1); break;  case MINMAX4: intelligentMove(2); break;  case MINMAX5: intelligentMove(3); break;  case AB2: intelligentMove(4); break;  case AB3: intelligentMove(5); break;  case AB4: intelligentMove(6); break;  case AB5: intelligentMove(7); break;  case AB6: intelligentMove(8); break;  case AB7: intelligentMove(9); break;  case RANDOM: randomMove(); break;  default: throw new GameError("Unknown AI Player");  }  }    protected abstract void startBoardGUI();    /\*\*  \* Starts the sidebar GUI  \*/  protected void startSidebarGUI() {  addStatusMessagesTextArea();  addPlayerInfoTable();  addPlayersModes();  addButtons();  }    /\*\*  \* Adds both button panels  \*/  private void addButtons() {  addAutoPlayersButtons();  addBottomButtons();  }    /\*\*  \* Adds quit and restart  \*/  private void addBottomButtons() {  JPanel bottomPanel = new JPanel(new FlowLayout(FlowLayout.CENTER)); //Creates the bottom panel for the buttons  quit = new JButton("Quit"); //Creates the quit button  quit.addActionListener(new ActionListener() { //Adds the action listener to the button  @Override  public void actionPerformed(ActionEvent e) {  quit(); //Quit when it is clicked  }  });  bottomPanel.add(quit); //Adds quit button to the bottom panel    restart = new JButton("Restart"); //Same as quit button  restart.addActionListener(new ActionListener() {  @Override  public void actionPerformed(ActionEvent e) {  try{  ctrl.restart(); //Calls the controller to restart the game //TODO Fix game restart for attributes or remove  }  catch(GameError \_e){}  }  });  bottomPanel.add(restart); //Adds restart button to the bottom panel    addToSidePanel(bottomPanel); //Adds bottom panel to the side panel  }  /\*\*  \* Adds the random and intelligent button  \*/  private void addAutoPlayersButtons() {  JPanel autoPlayerPanel = new JPanel(new GridLayout(0,2,5,5)); //Creates a panel with a grid of 2 columns and gaps of 5 units  autoPlayerPanel.setBorder(BorderFactory.createTitledBorder(/\*"Automatic Moves"\*/"Game Options")); //Creates and sets the label of the border  autoPlayerPanel.setSize(new Dimension(10,50)); //Sets the size for the panel  randomMove = new JButton("Random Move"); //Creates the random move button  randomMove.addActionListener(new ActionListener() { //Same as quit and restart buttons  @Override  public void actionPerformed(ActionEvent e) {  randomMove(); //Makes a random move  }  });  autoPlayerPanel.add(randomMove); //Adds the button to the Auto players panel    intelligentMove = new JButton("AlphaBeta Depth 4 Move");  intelligentMove.addActionListener(new ActionListener() {  @Override  public void actionPerformed(ActionEvent e) {  intelligentMove(6); //Makes an AlphaBeta Depth 4 Move  }  });  autoPlayerPanel.add(intelligentMove);    addToSidePanel(autoPlayerPanel);  }  /\*\*  \* Makes an intelligent move  \*/  protected void intelligentMove(int aiIndex) {  ctrl.makeMove(aiPlayers.get(aiIndex)); //TODO Test this  tableModel.refresh();  }  /\*\*  \* Makes a random move  \*/  protected void randomMove() {  ctrl.makeMove(randPlayer);  tableModel.refresh();  }  /\*\*  \* Adds the player modes panel  \*/  private void addPlayersModes() {  JPanel panel = new JPanel(new FlowLayout(FlowLayout.LEFT)); //Creates JPanel  panel.setBorder(BorderFactory.createTitledBorder("Player Modes")); //Creates titled border  playerModePieces = new JComboBox<Piece>(new DefaultComboBoxModel<Piece>()); //Creates the left comboBox  modes = new JComboBox<PlayerMode>(new DefaultComboBoxModel<PlayerMode>()); //Creates the right comboBox  modes.addItem(PlayerMode.MANUAL); //Add the options in the ComboBox from the enum  if(randPlayer != null) {  modes.addItem(PlayerMode.RANDOM);  }  if(!aiPlayers.isEmpty()) {  modes.addItem(PlayerMode.MINMAX2);  modes.addItem(PlayerMode.MINMAX3);  modes.addItem(PlayerMode.MINMAX4);  modes.addItem(PlayerMode.MINMAX5);  modes.addItem(PlayerMode.AB2);  modes.addItem(PlayerMode.AB3);  modes.addItem(PlayerMode.AB4);  modes.addItem(PlayerMode.AB5);  modes.addItem(PlayerMode.AB6);  modes.addItem(PlayerMode.AB7);  }  setMode = new JButton("Set Mode"); //Adds button and action listener  setMode.addActionListener(new ActionListener() {  @Override  public void actionPerformed(ActionEvent e) {  Piece selectedPiece = (Piece) playerModePieces.getSelectedItem();  PlayerMode selectedMode = (PlayerMode) modes.getSelectedItem();  PlayerMode currentMode = playerModes.get(selectedPiece);  playerModes.put(selectedPiece, selectedMode);  if(selectedMode != PlayerMode.MANUAL) {  if(((ChessPiece) selectedPiece).getWhite()) {  Controller.isWhiteAI = true;  } else {  Controller.isBlackAI = true;  }  } else {  if(((ChessPiece) selectedPiece).getWhite()) {  Controller.isWhiteAI = false;  } else {  Controller.isBlackAI = false;  }  }    //TODO Repaint gamemode board?    if(currentMode == PlayerMode.MANUAL && selectedMode != PlayerMode.MANUAL && ((ChessPiece) turn).getWhite() == ((ChessPiece) selectedPiece).getWhite())  decideMakeAutomaticMove();  }  });  panel.add(playerModePieces);  panel.add(modes);  panel.add(setMode);    addToSidePanel(panel);  }  /\*\*  \* Adds the player information table to the side  \*/  private void addPlayerInfoTable() {  JPanel panel = new JPanel(new BorderLayout());  panel.setBorder(BorderFactory.createTitledBorder("Player Info")); //Used to be Player information    tableModel = new PlayerInfoTableModel();  JTable table = new JTable(tableModel) {  @Override  public Component prepareRenderer(TableCellRenderer renderer, int row, int col){  Component comp = super.prepareRenderer(renderer, row, col);  comp.setBackground(pieceColors.get(pieces.get(row)));  if(pieceColors.get(pieces.get(row)) == Color.BLACK || pieceColors.get(pieces.get(row)) == Color.BLUE) {  comp.setForeground(Color.WHITE);  } else {  comp.setForeground(Color.BLACK);  }  return comp;  }  };    JScrollPane tableScroll = new JScrollPane(table);    panel.add(tableScroll, BorderLayout.CENTER);    addToSidePanel(panel);  }  /\*\*  \* Adds the status messages area  \*/  private void addStatusMessagesTextArea() {  JPanel panel = new JPanel(new BorderLayout());  panel.setPreferredSize(new Dimension(100, 150));  panel.setBorder(BorderFactory.createTitledBorder("Status Messages"));  statusArea = new JTextArea(5, 10);  statusArea.setEditable(false); //Can't be edited    JScrollPane statusAreaScroll = new JScrollPane(statusArea); //Creates a scrollPane as well  panel.add(statusAreaScroll, BorderLayout.CENTER);    addToSidePanel(panel);  }  /\*\*  \* Adds a panel to the side  \* @param panelToAdd  \*/  private void addToSidePanel(JPanel panelToAdd) {  sidebarPanel.add(panelToAdd);  }  protected abstract void activateBoard();  protected abstract void deActivateBoard();  protected abstract void redrawBoard();    //GameObserver Methods  public void onGameStart(final Board board, final String gameDesc, final List<Piece> pieces, final Piece turn) {  SwingUtilities.invokeLater(new Runnable(){  public void run(){  handleGameStart(board, gameDesc, pieces, turn);  }  });  }    /\*\*  \* What it should do at start  \* @param board  \* @param gameDesc  \* @param pieces  \* @param turn  \*/  private void handleGameStart(Board board, String gameDesc, List<Piece> pieces, Piece turn) {  this.setTitle(gameDesc);  addMsg("Game started");  this.turn = pieces.get(ChessStatic.WHITE\_ID); //White always starts  this.board = board;  this.pieces = pieces;  this.gameDesc = gameDesc;  initPlayerModes();  redrawBoard();    handleOnChangeTurn(board, pieces.get(ChessStatic.WHITE\_ID));  }  /\*\*  \* Initiates the player modes  \*/  private void initPlayerModes() {  if(playerModePieces != null) {  if(localPiece == null) {  for(Piece p: pieces) {  if(playerModes.get(p) == null) {  playerModes.put(p, PlayerMode.MANUAL);  Controller.isWhiteAI = false;  Controller.isBlackAI = false;  playerModePieces.addItem(p);  }  }  } else {  if(playerModes.get(localPiece) == null) {  playerModes.put(localPiece, PlayerMode.MANUAL);  playerModePieces.addItem(localPiece);  }  }  }  }    /\*\*  \* Enables the view  \*/  private void enableView() {  enableButtons();  activateBoard();  }    /\*\*  \* Disables the view  \*/  @SuppressWarnings("unused")  private void disableView() {  disableButtons();  deActivateBoard();  }  /\*\*  \* Disables the buttons  \*/  private void disableButtons() {  quit.setEnabled(false);  restart.setEnabled(false);  randomMove.setEnabled(false);  intelligentMove.setEnabled(false);  setMode.setEnabled(false);  }  /\*\*  \* Enables the buttons  \*/  private void enableButtons() {  quit.setEnabled(true);  restart.setEnabled(true);  randomMove.setEnabled(true);  intelligentMove.setEnabled(true);  setMode.setEnabled(true);  }  @Override  public void onGameOver(final Board board, final State state, final Piece winner) {  SwingUtilities.invokeLater(new Runnable(){  public void run(){  handleOnGameOver(board, state, winner);  }  });  }  /\*\*  \* What the program should do at the end of the game  \* @param board  \* @param state  \* @param winner  \*/  protected void handleOnGameOver(Board board, State state, Piece winner) {  this.board = board;  redrawBoard();  if(state == State.Won) {  addMsg(winner.toString() + " won the game!");  JOptionPane.showMessageDialog(new JFrame(), winner.toString() + " player wins the match!!!", "Match result", JOptionPane.INFORMATION\_MESSAGE);  } else if (state == State.Draw) {  addMsg("The match finished with a draw.");  JOptionPane.showMessageDialog(new JFrame(), "The match finished with a draw.", "Match result", JOptionPane.INFORMATION\_MESSAGE);  }  }  @Override  public void onMoveStart(final Board board, final Piece turn) {  SwingUtilities.invokeLater(new Runnable(){  public void run(){  handleOnMoveStart(board, turn);  }  });  }  /\*\*  \* What to do at the start of a move  \* @param board  \* @param turn  \*/  protected void handleOnMoveStart(Board board, Piece turn) {  //Empty method  }  @Override  public void onMoveEnd(final Board board, final Piece turn, final boolean success) {  SwingUtilities.invokeLater(new Runnable(){  public void run(){  handleOnMoveEnd(board, turn, success);  }  });    }  /\*\*  \* What to do at the end of a move  \* @param board  \* @param turn  \* @param success  \*/  protected void handleOnMoveEnd(Board board, Piece turn, boolean success) {  this.board = board;  this.turn = turn;  tableModel.refresh();  }  @Override  public void onChangeTurn(final Board board, final Piece turn) {  SwingUtilities.invokeLater(new Runnable(){  public void run(){  handleOnChangeTurn(board, turn);  }  });  }  /\*\*  \* What to do when the turn changes  \* @param board  \* @param turn  \*/  protected void handleOnChangeTurn(Board board, Piece turn) {  this.board = board;  this.turn = turn;  redrawBoard();  addMsg("Turn for player '" + turn + "'");  /\*if(Main.isMultiviews() || Main.isClientMode()) //TODO Only for online mode or multiviews  {  this.setTitle(gameDesc + ". View from: " + localPiece + " (Turn for: " + turn.getId() + ")");  if(turn.equals(localPiece) && playerModes.get(turn) == PlayerMode.MANUAL) {  enableView();  } else {  disableView();  }  } else {\*/  this.setTitle(gameDesc + " (Turn for: " + turn + ")");  enableView();  //}  if(playerModes.get(turn) != PlayerMode.MANUAL) {  addMsg("AI Player is thinking...");    switch(playerModes.get(turn)) {  case MINMAX2: intelligentMove(0); break;  case MINMAX3: intelligentMove(1); break;  case MINMAX4: intelligentMove(2); break;  case MINMAX5: intelligentMove(3); break;  case AB2: intelligentMove(4); break;  case AB3: intelligentMove(5); break;  case AB4: intelligentMove(6); break;  case AB5: intelligentMove(7); break;  case AB6: intelligentMove(8); break;  case AB7: intelligentMove(9); break;  case RANDOM: randomMove(); break;  default: throw new GameError("Unknown AI Player");  }  }  }  @Override  public void onError(final String msg) {  SwingUtilities.invokeLater(new Runnable(){  public void run(){  handleOnError(msg);  }  });  }  /\*\*  \* What to do on error  \* @param msg  \*/  protected void handleOnError(String msg) {  if((turn.equals(localPiece) && playerModes.get(turn) == PlayerMode.MANUAL) || !Main.isMultiviews())  {  addMsg(msg);  JOptionPane.showMessageDialog(new JFrame(), msg, "Error", JOptionPane.ERROR\_MESSAGE);  }  }    /\*\*  \* The class for the table model  \* (It needs to be here to get the attributes more easily)  \*/  class PlayerInfoTableModel extends AbstractTableModel {  private String[] colNames;    PlayerInfoTableModel(){  this.colNames = new String[] {"Player", "Mode", "Last Move Rating"};  }    @Override  public String getColumnName(int col){  return colNames[col];  }    @Override  public int getColumnCount(){  return colNames.length;  }    @Override  public int getRowCount(){  if(pieces == null)  {  return 0;  }  else  {  return pieces.size();  }  }    @Override  public Object getValueAt(int row, int col){  if(pieces == null){  return null;  }  Piece p = pieces.get(row);    switch(col)  {  case 0: return p;  case 1: return playerModes.get(p);  default: return board.getPieceCount(p);  }  }    public void refresh() {  fireTableDataChanged();  }  }  } |

1. Peter Shotwell (2008), “The Game of Go: Speculations on its Origins and Symbolism in Ancient China”, American Go Association. [↑](#footnote-ref-1)
2. British Go Association (December 2017), “Comparison of Some Go Rules”, <http://www.britgo.org/rules/compare> [↑](#footnote-ref-2)
3. Sid Sackson (1982), “Focus”, *A Gamut of Games*. Pantheon Books. Pp. 125-35. ISBN 0-394-71115-7 [↑](#footnote-ref-3)
4. Niklesh Kumar Jain (2014), “Hindi and the Origins of Chess”, <https://en.chessbase.com/post/hindi-and-the-origins-of-chess> [↑](#footnote-ref-4)
5. Hooper & Whyld (1992), “The Oxford Companion to Chess”, pp. 173-175. [↑](#footnote-ref-5)
6. Larry Kaufman (Sept 2009): “Calling stalemate a draw is completely illogical, since it represents the ultimate zugzwang, where any move would get your king taken.” [↑](#footnote-ref-6)
7. Fraenkel & Lichtenstein (1981), “Computing a perfect strategy for nxn chess requires time exponential in n”, Journal of Combinatorial Theory, Series A (31): pp. 199-214. [↑](#footnote-ref-7)
8. Claude Shannon (1950), “Programming a Computer for Playing Chess”, Philosophical Magazine. 41 (314) [↑](#footnote-ref-8)
9. Michiel Hazewinkel, ed. (2001) [1994], "Minimax principle", Encyclopedia of Mathematics, Springer Science+Business Media B.V. / Kluwer Academic Publishers, ISBN 978-1-55608-010-4 [↑](#footnote-ref-9)
10. Cai, Y. and Daskalakis, C., 2011, January. On minmax theorems for multiplayer games. In Proceedings of the twenty-second annual ACM-SIAM symposium on Discrete Algorithms (pp. 217-234). Society for Industrial and Applied Mathematics. [↑](#footnote-ref-10)
11. Marsland, T. A. (1991). “Computer chess and search”. Department of Computing Science, University of Alberta. [↑](#footnote-ref-11)
12. Edwards, D.J.; Hart, T.P. (4 December 1961). "The Alpha–beta Heuristic (AIM-030)". Massachusetts Institute of Technology. [↑](#footnote-ref-12)
13. “An Adaptive Sampling Algorithm for Solving Markov Decision Processes” Hyeong Soo Chang, Michael C. Fu, Jiaqiao Hu, and Steven I. Marcus Operations Research 200553:1, 126-139 [↑](#footnote-ref-13)
14. Silver D., Huang A., Maddison C.J., Guez A., Sifre L., Van Den Driessche G., Schrittwieser J., (...), Hassabis D. (2016) “Mastering the game of Go with deep neural networks and tree search” Nature, 529 (7587), pp. 484-489. [↑](#footnote-ref-14)
15. Silver, David; Hubert, Thomas; Schrittwieser, Julian (December 2017). “Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm” pp. 4-5. [↑](#footnote-ref-15)
16. Sarcar, V. (2016). Java Design Patterns. Apress. [↑](#footnote-ref-16)
17. Patwardhan, A., 2016. Analysis of Software Delivery Process Shortcomings and Architectural Pitfalls. arXiv preprint arXiv:1607.03748. [↑](#footnote-ref-17)
18. Norvig, Peter (1998). Design Patterns in Dynamic Languages. [↑](#footnote-ref-18)
19. Alur, Deepak; Crupi, John; Malks, Dan (May 2003). Core J2EE Patterns: Best Practices and Design Strategies (2nd Edition). Prentice Hall. ISBN 0-13-142246-4. [↑](#footnote-ref-19)
20. E. Guerra Sánchez (2008/09). Técnicas de Programación (UC3M). Patrones de diseño. (Lecture slides) [↑](#footnote-ref-20)
21. Systems and software engineering - Vocabulary, ISO/IEC/IEEE Std 24765:2010(E), 2010. [↑](#footnote-ref-21)
22. “IEEE Standard for Developing Software Life Cycle Processes” in IEEE Std 1074-1991, vol., no., pp.1-, 1992 doi: 10.1109/IEEESTD.1992.101080 [↑](#footnote-ref-22)
23. Standards.ieee.org. (2018). IEEE Standards for Software Lifecycle Processes. [online] All standards available at: https://standards.ieee.org/ [↑](#footnote-ref-23)
24. "Supplement 4-A, A Procedure for Requirements Analysis". Systems Engineering Fundamentals. United States Government US Army. 2001. ISBN 978-1484120835. [↑](#footnote-ref-24)
25. Dalbey, John. "Nonfunctional Requirements". Csc.calpoly.edu. [↑](#footnote-ref-25)
26. 828-2012 IEEE Standard for Configuration Management in Systems and Software Engineering. 2012. doi:10.1109/IEEESTD.2012.6170935. ISBN 978-0-7381-7232-3. [↑](#footnote-ref-26)
27. “Theory and Practice of Artificial Intelligence” Daniel Polani. School of Computer Science. University of Hertfordshire. March 9, 2017. [↑](#footnote-ref-27)
28. https://commons.apache.org/proper/commons-cli/ [↑](#footnote-ref-28)
29. Eric van Reem. "The birth of Fischer Random Chess". The Chess Variant Pages. [↑](#footnote-ref-29)
30. D. B. Pritchard (2000). Popular Chess Variants, p. 8 [↑](#footnote-ref-30)
31. The hot game of Nim – Nim theory and connections. Available at <https://web.archive.org/web/20080820154432/http://www.cut-the-knot.org/ctk/May2001.html> [↑](#footnote-ref-31)