

Alquerque

Danny Nicolai Larsen, Mikkel Brix Nielsen & Steffen Bach

December 9, 2021

CONTENTS

Contents

| | | |
|----------|---|-----------|
| 1 | Introduction | 3 |
| 2 | Design choices | 3 |
| 2.1 | <i>Move</i> | 3 |
| 2.2 | <i>Board</i> | 3 |
| 2.2.1 | <i>Board()</i> | 4 |
| 2.2.2 | <i>black()</i> & <i>white()</i> | 4 |
| 2.2.3 | <i>isLegal()</i> | 4 |
| 2.2.4 | <i>legalMoves()</i> | 4 |
| 3 | Implementation & functionality | 5 |
| 3.1 | The methods <i>black()</i> & <i>white()</i> | 5 |
| 3.2 | The method <i>isLegal()</i> | 5 |
| 4 | Testing | 8 |
| 4.1 | Test-approach for <i>isLegal()</i> | 8 |
| 4.2 | Test of <i>legalMoves()</i> | 9 |
| 4.3 | Sketchy take-moves | 9 |
| 4.4 | Proof of <i>isLegal()</i> 's functionality | 11 |
| 4.5 | Test of <i>finishedGames()</i> | 15 |
| 4.6 | Test of <i>black()</i> & <i>white()</i> | 15 |
| 5 | Conclusion | 16 |
| 6 | Appendix | 17 |
| 6.1 | Move class | 17 |
| 6.2 | Board class | 17 |
| 6.3 | MainTest | 21 |

1 Introduction

For phase 2 of the project, we have been tasked with implementing the two classes *Board* and *Move* for the board game, Alquerque, by developing two classes in accordance with the contract for phase 2. The new *Board* and new *Move* class must be compatible with the previously developed Alquerque interfaces from phase 1. Furthermore, the *Board* and *Move* class must be developed as generally as possible to ensure its compatibility, not only with our interface, but also other interfaces developed in accordance with the contract for phase 1. The classes do not have to be executable, meaning neither of them have a main method. Testing the classes thus requires that a separate executable main method is developed to test the functionality of the two classes independently and coherently. All the provider classes, except for the ones being developed in phase 2, are precompiled, and thus we shall focus on the classes for *Board* and *Move* during this phase.

2 Design choices

2.1 *Move*

To keep the class simple and easily understandable, the class *Move* was developed with two attributes, int *from* and int *to*, which keep track of where from the move should start and where to the move should end, respectively. These attributes have been made private, as it should not be possible to change a move once it has been made. Due to the nature of the private access modifier, these attributes are not accessible through *object.attribute* syntax. For the *Board* class to have access to where from and where to, a move should be made, respective getters for both attributes were made. There are no setters since the only time it should be possible to set the value of from and to is when a new instance of *Move* is created.

2.2 *Board*

The *Board* class and its methods have been developed based on a char array. This class has six attributes, which include a char array *board*, an int *turn*, which keeps track of how many turns have been played, a boolean *isWhite*, which keeps checks whether it is white's turn, a boolean *isGameDone*, which

2 DESIGN CHOICES

ensures that int *finishedGames* can only be incremented once for each instance of *Board*, a static int *finishedGames*, which keeps track of how many games have been played, and a static final char (constant) *EMPTY*, which is the character to represents empty cells on the board.

2.2.1 *Board()*

The board's constructor contains no arguments and, when called, it creates a char array of length 26 in the game's starting position. Index 0 is empty and indices 1 through 12 are filled with black pieces, represented by a 'B', the 13th space is empty, and indices 14 through 25 are filled with white pieces, represented by a 'W'. The turn being set to 1, *isWhite* is set to be based upon what turn it is, and *isGameDone* set to false.

2.2.2 *black()* & *white()*

There are two methods, *black()* and *white()*, which individually loop through the char array that represents the board, each returning an int-array containing all the positions corresponding to their respective pieces. This is done by checking whether a cell contains either a black piece, 'B', or a white piece, 'W', and adding their position to their respective array, through their respective methods.

2.2.3 *isLegal()*

For *isLegal()*, it is designed to go through every combination of characteristics that would cause a move to be illegal according to the rules of Alquerque. In other words, every move with a set of characteristics is prohibited, is filtered away, and so only a move that passes through this filter is considered legal.

2.2.4 *legalMoves()*

For the design of *legalMoves()*, it goes through all cells where a piece is located and checks if it can make a legal move to any of all the other cells. If it encounters a legal move it is added to an array of Move objects and returned. To make this a bit more effective, it skips moves that would start from an empty cell.

3 Implementation & functionality

3.1 The methods *black()* & *white()*

The implementation of the methods *black()* and *white()* are identical except the fact that *white()* looks for white pieces and *black()* looks for black pieces. The methods work by creating a new *ArrayList<Integer>()* and using a for-loop to go through *this.board* looking for a corresponding piece and adding its index to the respective created arraylist. Afterwards, the arraylist is converted to an integer-array with the size of the created arraylist. This is done by initializing *int[] nameOfIntegerArray = new int[nameOfArrayList.size()]* before going through the created arraylist with a for-loop, making the index in the integer-array equal the index in the arraylist. The method then returns the integer-array containing all the positions of the white and black pieces, respectively.

```

1  /**
2  * Returns the positions of all white pieces on the board.
3  * @return the positions of all white pieces on the board.
4  */
5  public int[] white() {
6      ArrayList<Integer> whitePieces = new ArrayList<Integer>();
7      for (int i = 1; i <= 25; i++)
8          if (this.board[i] == 'W')
9              whitePieces.add(i);
10     int[] white = new int[whitePieces.size()];
11     for (int i = 0; i < whitePieces.size(); i++)
12         white[i] = whitePieces.get(i);
13     return white;
14 }

```

For black: ‘W’ would be replaced with ‘B’ and the variable names would be different, but the implementation is the same.

3.2 The method *isLegal()*

The boolean method, *isLegal()*, works by taking an instance of the class *Move* as an argument and checking whether it is legal according to the rules of Alquerque. A series of if- and else-if-statements filters away any move with a set of characteristics that would make it illegal, and so, whenever a move passes through, *isLegal()* returns true.

This “filter” is implemented by checking the following criteria:

3 IMPLEMENTATION & FUNCTIONALITY

The position returned by *to()* must always be empty.

The position returned by *from()* must be a piece corresponding to whoever's turn it is. So whenever *isWhite* is true, the *from()* position must contain a 'W' and whenever *isWhite* is false, the *from()* position must contain 'B'. Speaking of which; *isWhite* is an instance variable with a boolean value that is used in the code as a replacement to continuously write "turn % 2 == 1" / "...0" to check whether it is white or black to move, and for the code to be more easily serviceable as well as improved code-readability.

The absolute difference in columns between *from()* and *to()* must never be greater than 2. For this, we made the auxiliary method, *fileDiff()*. The method, *fileDiff()*, returns an int, calculated by subtracting 1 from the positional value returned by *to()* and *from()*, respectively, modulo 5, and adding 1 back to both, before subtracting one from the other, and returning the absolute value thereof.

The difference between the *from()* position and the *to()* position must be within -4 and -6 for white and within 4 and 6 for black. For this, we made the auxiliary method, *pieceDiff()*, which works by subtracting the value of *to()* by the value of *from()*. By defining the legal difference to be -4, -5, and -6 for white, we ensure that only moves in the correct directions are allowed. The same thing applies to black with 4, 5, and 6 as legal positional differences between *to()* and *from()*.

Normal moves from even-numbered positions must always be to an odd-numbered position, according to the rules of the game.

Take-moves from odd-numbered positions must always have an absolute positional difference of either 2, 8, 10 or 12.

And lastly, take-moves from even-numbered positions must have an absolute positional difference of either 2 or 10.

If an instance of *Move* breaks none of the aforementioned criteria, thereby passing through the filter, *isLegal()* recognizes it as a valid move, in accordance with the current state of the board, on which it was called, and returns true.

The auxiliary method, *isTakeMove()*, checks whether a move is considered a take-move by returning true if the *pieceDiff()* is greater than 6 or less than 4, albeit only if the piece taken is an opponent's piece, which is checked with the average positional value of *to()* and *from()*.

The method *legalMoves()* The method *legalMoves()* is implemented using

3 IMPLEMENTATION & FUNCTIONALITY

an arraylist, which is then converted to an integer-array. Firstly a new arraylist of type move, called *legalList* is created. Then a for-loop is used to go through all cells of *this.board* and if that cell is not empty, then calculate all *legalMoves* from that cell to any other cells using another for-loop and creating new instances of *Move*, where the outer for-loops iterator variable is used as the origin of the move and the inner for-loops iterator variable is used as the destination for an instance of *Move*. *isLegal()* is then called with that instance of *Move* as its argument, and if *isLegal()* returns true, that instance of move is added to the arraylist, *legalList*. Afterwards, the arraylist, *legalList*, is converted to an integer-array, *legalMove*, with the size of *legalList*. This is done by initializing *int[] legalMove = new int[legalList.size()]* before going through *legalList* with a for-loop, making the index in the *legalMove* equal the index in *legalList*. The method then returns the integer-array, *legalMoves*, containing all the *legalMoves*.

The method *move()* The method *move()* takes an instance of the *Move* class as a parameter and updates the *board* array accordingly with the *from()* and *to()* getters for that move, it then checks if the move is a take-move, using the auxiliary *isTakeMove()* method, and if this is true, it calculates the average positional value between *to()* and *from()* and removes the piece at that cell. After exiting or skipping the if-statement it increments the *turn* counter, and updates the *isWhite* variable. Finally it uses the method *isGameOver()* to check if the game has ended, and if it has, it increments the *finishedGames* variable, and then sets the *isGameDone* variable to true.

The methods *isGameOver()* & *finishedGames()* The methods *isGameOver()* and *finishedGames()* are implemented in such a way that *isGameOver()* returns whether there are no pieces remaining in *white()* or whether there are no pieces left in *black()* or whether there are no moves left in *legalMove()* to indicate when the game represented by this board is over.

The static method *finishedGames()* was implemented with the same functionality as a getter for the static class variable *finishedGames*, meaning that when *finishedGames()* is called on the class, *Board*, it returns the value of the attribute *finishedGames*.

The methods *copy()*, *equals()* & *hashCode()* Copy The method *copy()* is implemented such that it creates a new instance of *Board*, from the constructor, *Board()*, called *newBoard*. The respective char at each position on the board is copied via a for-loop to *newBoard*, so that the current game gamestate of *newBoard* matches the original board. Afterwards, the instance variables *turn*, *isWhite*, and *isGameDone* for *newBoard* are changed to equal the value of the original boards *turn*, *isWhite*, and *isGameDone*, respectively. Lastly,

4 TESTING

newBoard is returned as a copy of the instance of *Board*, which *copy()* was called upon.

Equals() The *equals()* method was implemented due to *copy()* being overwritten. Overwriting *copy()* could, potentially, make the original *equals()* method return false on two copies of the same board. Therefore, a new *equals()* was implemented, which created a filter of tests that ensures that the compared object does not equal null, which would return false, and that checks whether the instance of *board* and the other object have the same memory address, which would return true. The last part of the filter checks whether the other object is an instance of *Board*, which, if it is not, returns false, otherwise, *other* is typecast to be a *Board* called *otherBoard*. Thereafter, the contents of each cell on *otherBoard* is compared to the same cell on the instance of *board*, that *equals()* was called upon, through a while-loop, which increments a counter once every time both cells contain the same character. The loop is stopped at any point where two compared cells do have identical contents, or when the counter equals the length of the char array board. Finally, *equals()* returns whether: The counter is equal to the length of the char array AND both boards are on the same *turn* AND *isGameDone* has the same boolean value for both.

HashCode() The *hashCode()* method was implemented due to *equals()* being overwritten. This method is implemented by returning an integer based on the values of the attributes on the instance of the *Board* that it is called upon. The attributes used to create the hashcode are *board* and *turn*. Even though *board* is a char array, it needs to be converted to an integer first. This is done by calling *hashCode()* for char arrays on *board*. In short; the method returns the sum of *board.hashCode()* and *turn* multiplied by 31.

4 Testing

4.1 Test-approach for *isLegal()*

To thoroughly test whether *isLegal()* worked as desired, it was convenient to use a testclient and create specific scenarios on a board. The method, *isLegal()*, works like a filter for moves that are not allowed to be made. By testing several different sets of characteristics for a move to see whether it was allowed or not, it could effectively be ensured that no illegal move would ever pass through this so-called filter. Theoretically, this means that all 625 combinations of moves have been checked, but in a generalized sense

4 TESTING

that took a fraction of the time that otherwise would have been required to ensure proper functionality.

4.2 Test of *legalMoves()*

To test the method *legalMoves*, we modified our *MainTest* class to print out the contents of the array while playing. This seemed as an effective way of testing two things at the same time. One: That it stores valid moves correctly in the array; and two: That *isLegal* works properly, which calculates all the valid moves available without returning any invalid moves.

Below is a sample output of the console during these tests.

```

  A   B   C   D   E
1 [B]-[ ]-[B]-[B]-[B] 1
  | \ | / | \ | / |
2 [B]-[B]-[B]-[B]-[B] 2
  | / | \ | / | \ |
3 [ ]-[B]-[W]-[W]-[W] 3
  | \ | / | \ | / |
4 [W]-[W]-[W]-[W]-[W] 4
  | / | \ | / | \ |
5 [W]-[ ]-[B]-[W]-[ ] 5
  A   B   C   D   E

It's black to move these are the legal moves:
From: 6, to: 11
From: 7, to: 11
From: 12, to: 22
From: 23, to: 11
From: 23, to: 25
```

4.3 Sketchy take-moves

The move from 19 to 13 would remove the piece on 16, regardless of color.

The same logic applied to the move 17 to 13, which would instead remove the piece on 15. Two things went wrong in these instances: Not only did white remove a white piece, it removed a piece whilst not being a take-move.

So to kill two birds with one stone, we made the auxiliary boolean method, *isTakeMove()*, to check whether the absolute positional difference between the value returned by *from()* and the value returned by *to()* was greater than 6 or less than 4, and to check whether the piece taken was an opponent piece.

4 TESTING

| Boards starting positon. | White moved from 19 to 13. |
|--|--|
| <pre> A B C D E 1 [B]-[B]-[B]-[B]-[B] 1 \ / \ / 2 [B]-[B]-[B]-[B]-[B] 2 / \ / \ 3 [B]-[B]-[]-[W]-[W] 3 \ / \ / 4 [W]-[W]-[W]-[W]-[W] 4 / \ / \ 5 [W]-[W]-[W]-[W]-[W] 5 A B C D E </pre> | <pre> A B C D E 1 [B]-[B]-[B]-[B]-[B] 1 \ / \ / 2 [B]-[B]-[B]-[B]-[B] 2 / \ / \ 3 [B]-[B]-[W]-[W]-[W] 3 \ / \ / 4 []-[W]-[W]-[]-[W] 4 / \ / \ 5 [W]-[W]-[W]-[W]-[W] 5 A B C D E </pre> |

We later discovered certain cases where, as an example, a piece on 15 could take 16 and move to 17, which implied that more had to be done to check for the legality of a take-move.

| Boards starting positon. | White moved from 15 to 17. |
|--|--|
| <pre> A B C D E 1 []-[]-[]-[]-[] 1 \ / \ / 2 []-[]-[]-[]-[] 2 / \ / \ 3 []-[]-[]-[]-[W] 3 \ / \ / 4 [B]-[]-[]-[]-[] 4 / \ / \ 5 []-[]-[]-[]-[] 5 A B C D E </pre> | <pre> A B C D E 1 []-[]-[]-[]-[] 1 \ / \ / 2 []-[]-[]-[]-[] 2 / \ / \ 3 []-[]-[]-[]-[] 3 \ / \ / 4 []-[W]-[]-[]-[] 4 / \ / \ 5 []-[]-[]-[]-[] 5 A B C D E </pre> |

As the absolute positional difference of the move was 2, which is less than 4, and the piece on 16 was an opponent piece, it fully qualified for a legal take-move.

To fix this as generally as possible, we made the auxiliary method, *fileDiff()*, to check the difference between the from-file and the to-file. For a take-move to be legal, this would always have to be either 2 or 0.

This method conveniently could have been used for non-take-moves as well, but that seemed redundant, as non-take-moves were only allowed to be within

4 TESTING

a specific range, which is a piece difference of min. 4, max. 6 for black and min. -6, max. -4 for white.

4.4 Proof of *isLegal()*'s functionality

Below is a selection of examples of legal and illegal moves used to test the general functionality, which, as a result, should give sufficient confidence in the method, *isLegal()*.

Moves in the right direction are valid moves

```
Where do you want to move from?  
17  
Where do you want to move to?  
13  
B-B-B-B-B  
I\I/I\I/I  
B-B-B-B-B  
I/I\I/I\I  
B-B-W-W-W  
I\I/I\I/I  
W-*-W-W-W  
I/I\I/I\I  
W-W-W-W-W
```

4 TESTING

Moving pieces on the wrong turn are not valid moves

```
*--B-B-B
|\|/\|/|
*--W-*--
|/\|/\|
*--W-*--
|\|/\|/|
*--*--*--
|/\|/\|
*-W-*--W
It is Black's turn.
Where do you want to move from?
3
Where do you want to move to?
13
That is not a valid move!
Where do you want to move from?
```

Take-moves in all directions are valid moves.

```
W--B-B-B
|\|/\|/|
*--W-*--
|/\|/\|
*--*--*--
|\|/\|/|
*--B-B-*
|/\|/\|
B-W-W-W-W
It is White's turn.
Where do you want to move from?
23
Where do you want to move to?
13
```

```
W--B-B-B
|\|/\|/|
*--W-*--
|/\|/\|
*--W-*--
|\|/\|/|
*--*--B-*
|/\|/\|
B-W-*--W-W
It is Black's turn.
Where do you want to move from?
21
Where do you want to move to?
23
```

```
W--B-B-B
|\|/\|/|
*--W-*--
|/\|/\|
*--W-*--
|\|/\|/|
*--*--B-*
|/\|/\|
*--B-W-W
It is White's turn.
Where do you want to move from?
24
Where do you want to move to?
22
```

4 TESTING

Take-moves across the board are not allowed. Normal moves in a backwards direction are not valid moves.

```
W-*-B-B-B
|\|/\|/\|
*-*-*-W-B
|/\|/\|/\|
*-*-B-*-*
|\|/\|/\|
B-*-W-*-*
|/\|/\|/\|
B-W-W-W-W
It is White's turn.
Where do you want to move from?
9
Where do you want to move to?
11
That is not a valid move!
Where do you want to move from?
9
Where do you want to move to?
14
That is not a valid move!
```

Moves to a non-empty position are not allowed.

```
*-*-B-B-B
|\|/\|/\|
*-*-W-*-*
|/\|/\|/\|
*-*-W-*-*
|\|/\|/\|
*-*-*-*-*
|/\|/\|/\|
*-W-*-*-W
It is Black's turn.
Where do you want to move from?
3
Where do you want to move to?
13
That is not a valid move!
Where do you want to move from?
```

4 TESTING

Moves from an empty position are not allowed.

```
*--B-B-*
|\|/\|/\|
*--W-*--*
|/\|/\|/\|
*--W-*--B
|\|/\|/\|
*--*--*--W
|/\|/\|/\|
*-W-*--*
It is White's turn.
Where do you want to move from?
12
Where do you want to move to?
7
That is not a valid move!
```

4 TESTING

4.5 Test of *finishedGames()*

To test *finishedGames*, an impromptu client was made to initiate a game with ‘CPU vs. CPU’. After a game is finished, the user can choose to play again, which will initiate a new game. If the user chooses not to play again, the program will end. The *finishedGames* attribute is incremented in *move()* whenever *isGameOver()* returns true.

To check functionality, *finishedGames()* was printed as “Games Played”, as seen on the image below. Note that the total number of wins, losses, and draws, which indicate the actual number of games played, does not match “Games Played” played when the next game is initiated. This is due to the fact that every copy of *board* made by MiniMax, whenever an end-state is reached, also counts towards the total number of *finishedGames*.

```
  A  B  C  D  E
1 [ ]-[ ]-[ ]-[W]-[ ] 1
  | \ | / | \ | / |
2 [ ]-[ ]-[W]-[ ]-[ ] 2
  | / | \ | / | \ |
3 [W]-[B]-[ ]-[ ]-[ ] 3
  | \ | / | \ | / |
4 [W]-[B]-[ ]-[ ]-[W] 4
  | / | \ | / | \ |
5 [W]-[W]-[B]-[B]-[W] 5
  A  B  C  D  E

Draw.
Stats:
White's wins: 0
Black's wins: 0
Draws: 1

Do you want to continue with another game? (y/n): y
Games Played: 376
```

4.6 Test of *black()* & *white()*

By printing the respective array of positions for black and white pieces, we ensure that *black()* and *white()* work as intended, which is that they return an array with the correct positions of their respective pieces on the board. Two of the tested board positions can be seen in the images below.

5 CONCLUSION

| Starting positon of the board: | Other position of board: |
|---|--|
| <pre> A B C D E 1 [B]-[B]-[B]-[B]-[B] 1 \ / \ / 2 [B]-[B]-[B]-[B]-[B] 2 / \ / \ 3 [B]-[B]-[]-[W]-[W] 3 \ / \ / 4 [W]-[W]-[W]-[W]-[W] 4 / \ / \ 5 [W]-[W]-[W]-[W]-[W] 5 A B C D E Position of white's pieces: 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, Position of black's pieces: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12,</pre> | <pre> A B C D E 1 []-[]-[]-[W]-[] 1 \ / \ / 2 []-[]-[W]-[]-[] 2 / \ / \ 3 [W]-[B]-[]-[]-[] 3 \ / \ / 4 [W]-[B]-[]-[]-[W] 4 / \ / \ 5 [W]-[W]-[B]-[B]-[W] 5 A B C D E Position of white's pieces: 4, 8, 11, 16, 20, 21, 22, 25, Position of black's pieces: 12, 17, 23, 24,</pre> |

As seen on picture 1, the starting position for white and black pieces are printed correctly on and below the board. As seen on picture 2, which is an arbitrary end-of-game position, the positions on the board match the ones printed from the respective arrays.

5 Conclusion

Through the development and testing of the classes *Board* and *Move*, both separately and coherently, a few issues occurred. These issues, as described in the test phase of the report, were eliminated. As a result of resolving these issues, each class then worked properly, as intended, and in accordance with the contract for phase 2. Furthermore, as seen through the test-section, the tests have been conducted with the help of a separate test class, developed for that purpose. Along with this, the classes have been tested with our own version of the Alquerque interface, as well as the precompiled version from phase 2. With a significant level of confidence, it can be concluded that the developed versions of classes *Board* and *Move* will work with all other interfaces developed in accordance with the contract from phase 1 of this project, as well as other interfaces developed in a similar fashion adhering to the documentation for *Board* and *Move*.

6 Appendix

6.1 Move class

```

1 public class Move {
2     private int from;
3     private int to;
4     /**
5      * Creates a new move with given origin and destination.
6      * @param from the place to move the piece from.
7      * @param to the place to move the piece to.
8      */
9     public Move(int from, int to) {
10         this.from = from;
11         this.to = to;
12     }
13
14     /**
15      * Returns the origin of this move.
16      * @return the origin of this move.
17      */
18     public int from() {
19         return from;
20     }
21
22     /**
23      * Returns the destination of this move.
24      * @return the destination of this move
25      */
26     public int to() {
27         return to;
28     }
29 }

```

6.2 Board class

```

1 import java.util.ArrayList;
2
3 public class Board {
4     private char[] board;
5     private int turn;
6     private boolean isWhite;
7     private boolean isGameDone;
8     private static int finishedGames = 0;
9     private static final char EMPTY = ' ';
10
11     /**
12      * Creates a new Alquerque board in the starting state:
13      * each player has twelve pieces in their original position, and it is↵
14      * white's turn.
15      */
16     public Board() {

```

6 APPENDIX

```

16     turn = 1;
17     board = new char[26];
18     for (int i = 1; i < 26; i++) {
19         if (i < 13)
20             board[i] = 'B';
21         else if (i == 13)
22             board[i] = EMPTY;
23         else
24             board[i] = 'W';
25     }
26     isWhite = (turn % 2 == 1);
27     isGameDone = false;
28 }
29
30 /**
31  * Returns the positions of all black pieces on the board.
32  *
33  * @return the positions of all black pieces on the board.
34  */
35 public int[] black() {
36     ArrayList<Integer> blackPieces = new ArrayList<Integer>();
37     for (int i = 1; i <= 25; i++)
38         if (this.board[i] == 'B')
39             blackPieces.add(i);
40     int[] black = new int[blackPieces.size()];
41     for (int i = 0; i < blackPieces.size(); i++)
42         black[i] = blackPieces.get(i);
43     return black;
44 }
45
46 /**
47  * Returns the positions of all white pieces on the board.
48  * @return the positions of all white pieces on the board.
49  */
50 public int[] white() {
51     ArrayList<Integer> whitePieces = new ArrayList<Integer>();
52     for (int i = 1; i <= 25; i++)
53         if (this.board[i] == 'W')
54             whitePieces.add(i);
55     int[] white = new int[whitePieces.size()];
56     for (int i = 0; i < whitePieces.size(); i++)
57         white[i] = whitePieces.get(i);
58     return white;
59 }
60
61 /**
62  * Moves a piece and updates the board correspondingly.
63  * Precondition: move must be a legal between 1 and 25
64  * @param move the move to simulate.
65  */
66 public void move(Move move) {
67     board[move.to()] = board[move.from()];
68     board[move.from()] = EMPTY;
69     if (isTakeMove(move)) //if the move is a take, the taken piece ↵
70         is removed
71         board[(move.to() + move.from()) / 2] = EMPTY; //calculates ↵
72         average position value and removes piece
73     // Updates who's turn it is
74     this.turn++;
75     isWhite = (turn % 2 == 1);
76     // updates finishedGames after eachmove
77     if (isGameOver() && !isGameDone) {

```

6 APPENDIX

```

76         finishedGames++;
77         isGameDone = true;
78     }
79 }
80
81 /**
82  * Checks whether a move is legal.
83  * Precondition: move must be an int from 1 through 25
84  * @param move move input to evaluate.
85  */
86 public boolean isLegal(Move move) {
87     if (board[move.to()] != EMPTY) // Checks whether the player tries to
88         // move from an empty cell
89         return false;
90     else if ((isWhite && board[move.from()] != 'W') || (!isWhite && board[move.from()] != 'B'))
91         // Checks if the player tries to move the opponents piece
92         return false;
93     else if (fileDiff(move) > 2)
94         // Checks if the player tries to move to a column that is too far away, which prevents moves rolling over from one row to the next
95         return false;
96     else if (!isTakeMove(move)){ // Logic for regular moves
97         if ((isWhite && (pieceDiff(move) < -6 || pieceDiff(move) > -4)) ||
98             (!isWhite && (pieceDiff(move) < 4 || pieceDiff(move) > 6)))
99             // Checks if direction is correct and if it is within the range of allowed cells to move to
100             return false;
101         else if (move.from() % 2 == 0 && move.to() % 2 == 0)
102             // Check for moves on even cells (To confirm that it follows the lines on the board)
103             return false;
104     } else if (isTakeMove(move)) { // Logic for moves that take another piece
105         if (Math.abs(pieceDiff(move)) != 2 && Math.abs(pieceDiff(move)) != 8 &&
106             Math.abs(pieceDiff(move)) != 10 && Math.abs(pieceDiff(move)) != 12)
107             // Checks if the move is to the specified allowed cells for a take move
108             return false;
109         else if (move.from() % 2 == 0 && Math.abs(pieceDiff(move)) != 10 && Math.abs(pieceDiff(move)) != 2)
110             // Checks for moves on even cells (to confirm it follows the lines on the board)
111             return false;
112     }
113     return true;
114 }
115
116 /**
117  * Returns an array of all legal moves for this board
118  * @return an array of all legal moves for this board
119  */
120 public Move[] legalMoves() {
121     ArrayList<Move> legalList = new ArrayList<Move>();
122     for (int i = 1; i < board.length; i++)
123         if (board[i] != EMPTY)
124             for (int j = 1; j < board.length; j++)

```

6 APPENDIX

```
124         if (isLegal(new Move(i,j)))
125             legalList.add(new Move(i,j));
126     Move[] legalMoves = new Move[legalList.size()];
127     for (int i = 0; i < legalList.size(); i++)
128         legalMoves[i] = legalList.get(i);
129     return legalMoves;
130 }
131
132 /**
133  * Returns if the game is over
134  * @return if the game is over
135  */
136 public boolean isGameOver() {
137     return (white().length == 0 || black().length == 0 || legalMoves()↵
138         .length == 0);
139 }
140
141 /**
142  * Returns how many objects of type Board that represents games, that ↵
143  * are finished games.
144  * @return how many objects of type Board that represents games, that ↵
145  * are finished games.
146  */
147 public static int finishedGames() {
148     return finishedGames;
149 }
150
151 /**
152  * Returns a copy of this board
153  * @return a copy of this board
154  */
155 public Board copy() {
156     Board newBoard = new Board();
157     for (int i = 0; i < this.board.length; i++)
158         newBoard.board[i] = this.board[i];
159     newBoard.turn = this.turn;
160     newBoard.isWhite = this.isWhite;
161     newBoard.isGameDone = this.isGameDone;
162     return newBoard;
163 }
164
165 /**
166  * Checks whether this Board is equal to other Object
167  * @param other Object to check against this board
168  * @return whether this Board is equal to other Object
169  */
170 public boolean equals(Object other){
171     if (other == null) return false;
172     else if (this == other) return true;
173     else if (!(other instanceof Board)) return false;
174     Board otherBoard = (Board) other;
175     int i = 0;
176     while(i < this.board.length && this.board[i] == otherBoard.board[i]↵
177         ])
178         i++;
179     return (i == this.board.length && this.turn == otherBoard.turn && ↵
180         this.isGameDone == otherBoard.isGameDone);
181 }
182
183 /**
184  * Returns a hashCode comprised of this boards attributes
185  * @return a hashCode comprised of this boards attributes
186  */
```

6 APPENDIX

```
181     */
182     public int hashCode() {
183         return (this.board.hashCode() + this.turn*31);
184     }
185
186     /*
187     * Auxillary methods to check how far there are between the columns ↵
188     in the move
189     */
189     private int fileDiff(Move move){
190         return Math.abs(((move.from() - 1) % 5 + 1) - ((move.to() - 1) % 5 ↵
191             + 1));
192     }
193
194     /*
195     * Auxillary method to check how far there is between two pieces
196     */
196     private int pieceDiff(Move move) {
197         return (move.to() - move.from());
198     }
199
200     /*
201     * checks whether the move is a take move
202     */
203     private boolean isTakeMove(Move move) {
204         return ((Math.abs(pieceDiff(move)) > 6 || Math.abs(pieceDiff(move) ↵
205             ) < 4) &&
206             ((isWhite && board[(move.to() + move.from()) / 2] == 'B') ↵
207                 || //checks if opponent piece is taken
208                 (!isWhite && board[(move.to() + move.from()) / 2] ↵
209                     == 'W'))); //checks if opponent piece is taken
210     }
211 }
```

6.3 MainTest

This was just made for testing purposes and is not expected to run in its current state, since all is uncommented for display purposes, but the individual test segments does work.

```
1  import java.util.Locale;
2  import java.util.Scanner;
3  public class MainTest {
4      public static Board myBoard = new Board();
5      public static Board yourBoard = new Board();
6      public static Scanner reader = new Scanner(System.in);
7      public static final char EMPTY = ' ';
8
9      public static void main(String[] args) {
10
11         // test whether an instance of Move returns the correct to and ↵
12         from values
13         Move m1 = new Move(19,13);
14         System.out.println("Move 1 expected: 19, 13 - Got: " + m1.from() ↵
15             + ", " + m1.to());
```

6 APPENDIX

```

14 Move m2 = new Move(2,25);
15 System.out.println("Move 2 expected: 2, 25 - Got: " + m2.from() +↵
    ", " + m2.to());
16 Move m3 = new Move(23,46);
17 System.out.println("Move 3 expected: 23, 46 - Got: " + m3.from() ↵
    + ", " + m3.to());
18 Move m4 = new Move(-12,40);
19 System.out.println("Move 4 expected: -12, 40 - Got: " + m4.from()↵
    + ", " + m4.to());
20 Move m5 = new Move(-91,-108);
21 System.out.println("Move 4 expected: -91, -108 - Got: " + m5.from↵
    () + ", " + m5.to());
22
23 // test to see if legal moves prints the correct moves
24
25 printBoard(myBoard);
26 int from = 0;
27 int to = 0;
28 boolean isWhite = true;
29 do {
30     do {
31         System.out.println("It's " + ((isWhite) ? "white to move, ↵
            these are the legal moves:" : "black to move these are↵
            the legal moves:"));
32         for (int i = 0; i < myBoard.legalMoves().length; i++)
33             System.out.println("From: " + myBoard.legalMoves()[i].↵
                from() + ", " + "to: " + myBoard.legalMoves()[i].↵
                to());
34         System.out.println("which peice do you want to move: ");
35         from = reader.nextInt();
36         System.out.println("where do you want to move that piece")↵
            ;
37         to = reader.nextInt();
38     } while (!myBoard.isLegal(new Move(from, to)));
39     myBoard.move(new Move(from, to));
40     isWhite = !isWhite;
41     printBoard(myBoard);
42 } while (!myBoard.isGameOver());
43
44
45 // test of white() and black()
46 System.out.println("Starting positon of the board:");
47 printBoard(myBoard);
48 System.out.println("Position of white's pieces:");
49 for (int i = 0; i < myBoard.white().length; i++)
50     System.out.print(myBoard.white()[i] + ", ");
51 System.out.println("\nPosition of black's pieces:");
52 for (int i = 0; i < myBoard.black().length; i++)
53     System.out.print(myBoard.black()[i] + ", ");
54
55 boolean isWhite = true;
56 do {
57     myBoard.move(Minimax.nextMove(myBoard,5,isWhite));
58     isWhite = !isWhite;
59 } while (!myBoard.isGameOver());
60
61 System.out.println("\n");
62 System.out.println("Other position of board: ");
63 printBoard(myBoard);
64 System.out.println("Position of white's pieces:");
65 for (int i = 0; i < myBoard.white().length; i++)
66     System.out.print(myBoard.white()[i] + ", ");

```

6 APPENDIX

```
67 System.out.println("\nPosition of black's pieces:");
68 for (int i = 0; i < myBoard.black().length; i++)
69     System.out.print(myBoard.black()[i] + ", ");
70
71
72
73 // test to see if the finishedGames method works and if it is ←
    incremented by MiniMax
74 int white = 0;
75 int black = 0;
76 int draw = 0;
77 boolean end = false;
78 boolean isWhite = true;
79 do {
80     System.out.println("Games Played: " + Board.finishedGames());
81     myBoard = new Board();
82     printBoard(myBoard);
83     do {
84         Move nextMove = Minimax.nextMove(myBoard, 5, isWhite);
85         myBoard.move(nextMove);
86         printBoard(myBoard);
87         isWhite = !isWhite;
88     } while (!myBoard.isGameOver());
89     if (myBoard.white().length == 0) {
90         System.out.println("White won!");
91         black++;
92     } else if (myBoard.black().length == 0) {
93         System.out.println("White won!");
94         white++;
95     } else {
96         System.out.println("Draw.");
97         draw++;
98     }
99     System.out.println("Stats:");
100    System.out.println("White's wins: " + white);
101    System.out.println("Black's wins: " + black);
102    System.out.println("Draws: " + draw);
103
104    System.out.println();
105    System.out.print("Do you want to continue with another game? (↔
        y/n): ");
106    end = ((reader.nextLine().toLowerCase().charAt(0) == 'n') ? ←
        true : false);
107 } while (!end);
108
109
110 // testing hashCode and copy
111 System.out.println(myBoard.hashCode());
112 Move[] m = myBoard.legalMoves();
113 for (int i = 0; i < m.length; i++) {
114     System.out.println("From: " + m[i].from() + ", to: " + m[i].to←
        ());
115 }
116 Board nBoard;
117 nBoard = myBoard.copy();
118 System.out.println(nBoard.hashCode());
119
120 }
121
122 // these methods were previously developed for our own Alquerque ←
    client.
123 /**
```

6 APPENDIX

```

124      * Creates a representation of the game board with the pieces ←
125      * correctly placed
126      * in the form of a two dimensional array.
127      * Precondition: Relies on method black() and white() to return valid ←
128      * positions numbered from 1-25
129      * @return a two dimensional array 5 x 5 with the game pieces placed ←
130      * correctly
131      */
132      private static char[][] boardWithPieces(Board myBoard) {
133          char[][] boardArr = new char[6][5]; //A-E & (no 0) 1-5
134          for (int j = 1; j < boardArr.length; j++)
135              for (int i = 0; i < boardArr[j].length; i++)
136                  boardArr[j][i] = EMPTY; // Fills board with empty spaces
137          for (int i = 0; i < myBoard.black().length; i++)
138              boardArr[((myBoard.black()[i] - 1) / 5) + 1][((myBoard.black()[i] - 1) % 5)] = 'B'; // Places black pieces
139          for (int i = 0; i < myBoard.white().length; i++)
140              boardArr[((myBoard.white()[i] - 1) / 5) + 1][((myBoard.white()[i] - 1) % 5)] = 'W'; // Places white pieces
141          return boardArr;
142      }
143
144      /**
145       * prints a representation of the board to the terminal
146       */
147      private static void printBoard(Board myBoard) {
148          System.out.println(); // new line
149          int i = 0, j = 1;
150          System.out.println("  A   B   C   D   E"); //upper-coordinate ←
151          //line (A-E)
152          char[][] boardWithPieces = boardWithPieces(myBoard);
153          while (j < 6) {
154              System.out.print(j + " "); //left-hand coordinate (1-5)
155              while (i < 5) {
156                  System.out.print "[" + boardWithPieces[j][i] + " ]";
157                  if (i < 4)
158                      System.out.print "-";
159                  i++;
160              }
161              System.out.print(" " + (j)); //right-hand coordinate (1-5)
162              System.out.println("");
163              i = 0;
164              if (j % 2 == 1 && j < 5)
165                  System.out.println("  | \\ | / | \\ | / |");
166              else if (j % 2 == 0)
167                  System.out.println("  | / | \\ | / | \\ |");
168              j++;
169          }
170          System.out.println("  A   B   C   D   E"); //bottom-coordinate ←
171          //line (A-E)
172          System.out.println(""); // new line
173      }
174  }

```