Assignment Cover Letter

(Individual Work)

Student Information:	**Surname**	**Given Names**	**Student ID Number**
1.	**Tandra**	**Vincentius**	**2301894804**

Course Code : **COMP6510** **Course Name** : **Programming Languages**

Class : **L2AC** **Name of Lecturer(s)** : **Jude Joseph Lamug Martinez**

Major : **CS**

Title of Assignment (if any) :

Type of Assignment : **Final Project**

Submission Pattern

Due Date : **20/06/20** **Submission Date** : **20/06/20**

The assignment should meet the below requirements.

1. Assignment (hard copy) is required to be submitted on clean paper, and (soft copy) as per lecturer's instructions.
2. Soft copy assignment also requires the signed (hardcopy) submission of this form, which automatically validates the softcopy submission.
3. The above information is complete and legible.
4. Compiled pages are firmly stapled.
5. Assignment has been copied (soft copy and hard copy) for each student ahead of the submission.

Plagiarism/Cheating

BiNus International seriously regards all forms of plagiarism, cheating and collusion as academic offenses which may result in severe penalties, including loss/drop of marks, course/class discontinuity and other possible penalties executed by the university. Please refer to the related course syllabus for further information.

Declaration of Originality

By signing this assignment, I understand, accept and consent to BiNus International terms and policy on plagiarism. Herewith I declare that the work contained in this assignment is my own work and has not been submitted for the use of assessment in another course or class, except where this has been notified and accepted in advance.

Signature of Student: (Name of Student)

1. Vincentius Gabriel Tandra

TABLE OF CONTENTS

Name: Vincentius Gabriel Tandra

ID : 2301894804

I. Project Specification

Introduction:

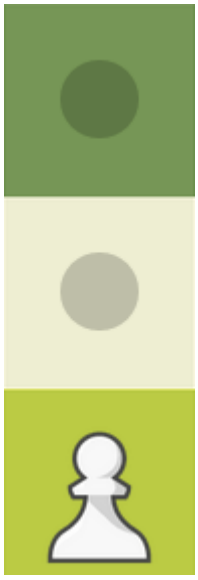
For my Java final project for this semester, I decided to create a Chess program from scratch. The reason I chose to do this is because it satisfies the requirements for the project and I had recently picked up chess so I decided that my app could be a way for myself to learn chess on my own. In the beginning, I was quite clueless as to how to begin, I understood most of how chess worked as a game, the many rules and exceptions and the way each piece worked to win a game. However, I immediately understood that translating my knowledge into Java code would be a difficult task. So I looked for resources online and I found a youtube series for exactly what I was making and decided to use it as reference when I had trouble. I was also planning to make it a proper GUI application similar to a normal chess game. Most of the code was referenced from this playlist, a lot of the code within the videos was dependent on an external library, but I decided that my chess code would be runnable without any dependencies and so I had to change some equations and replaced some functions with pre-existing internal libraries.However, many of the algorithms and variable names were replicated so it was easier for myself to understand. While I would have liked to customise the code more to my own liking, it was difficult to deviate from the rules of chess, epescially creating it from scratch. At the end of the day, I could have changed the variables and functions to suit my own reading but most of the variable names and terms used within the code were already according named specific to chess terms.

II.a. Design/Plan

Before we begin with our code, I would like to explain most of the rules within chess so the concepts I use in my code are more easily understood. Within chess two players compete each

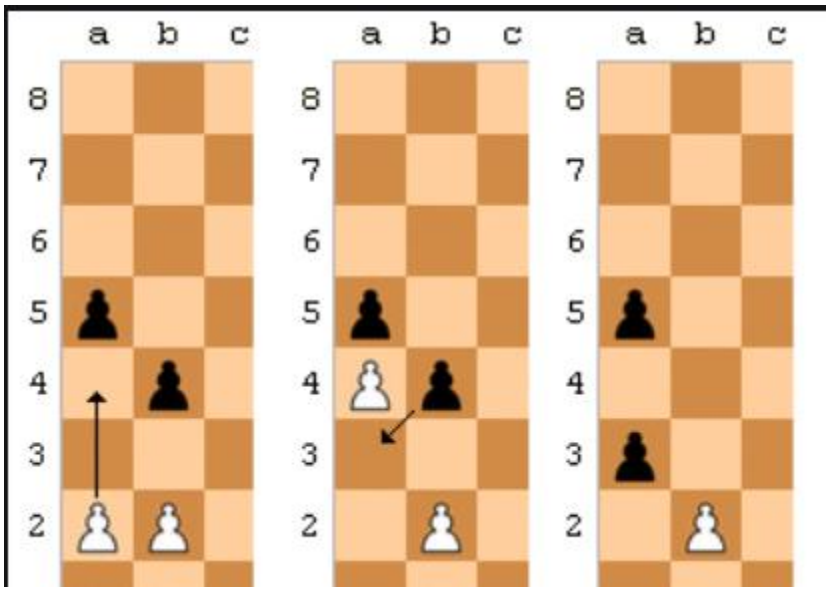
other in a conquest to conquer the opponent's king. It is a turn-based game where the two players are assigned a side, a black side and a white side. Each player takes turns moving a piece a turn and outsmarting their opponents by taking their pieces before eventually winning the game. However, chess has many rules that set it apart from its basics. A beginner of the game cannot be so easily compared to a much stronger player such as a grand master and this knowledge of the game is what makes it fun, the manipulation and knowledge of the strengths and weaknesses of each piece matters and winning a chess game is not as simple as one might expect. In chess, in order to win, one player must checkmate their opponent's king. Within chess a check is a type of move where either player's king is under attack and when a player's king has nowhere to move after a check it is checkmate and a player wins the game. However, chess has two other outcomes when the game ends. The game can end in a stalemate, where the king is not in check and it is unable to move because it would be attacked in that position and there is also a draw, where both opponents are left with only their kings. In either of these gamestates there is no actual winner and a tie is determined. Now that we have explained the general ruels of chess we can explain each piece and how they work.

Pawn:



The pawn is at face value, the simplest piece in chess. Either side of the board has 8 pawns at the beginning of the game. In its beginning position it is able to make two moves, a single move

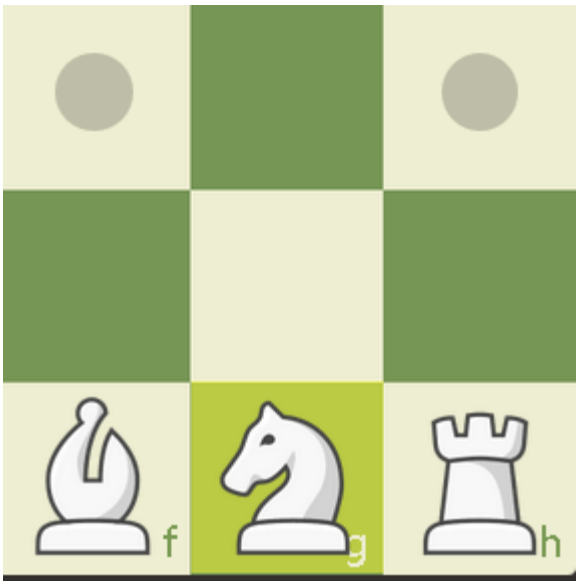
forward and it can also skip that tile in order to jump forward, a pawn is also only able to move forward. Once it does this, it is unable to jump again and will only move one tile forward. A pawn can capture any piece within one tile of it's front diagonal positions, a pawn is special because when it reaches the 8th rank from its position or the opposite end of the baord, it can promote itself into any other piece. The pawn is also able to make a special move known as En Passant, in this move if a pawn jumps from its original positon, it can still be captured like it would be normally despite it not being in the usual attacking position.



En Passant in action ^ (because it's really difficult to understand if not visualised.)

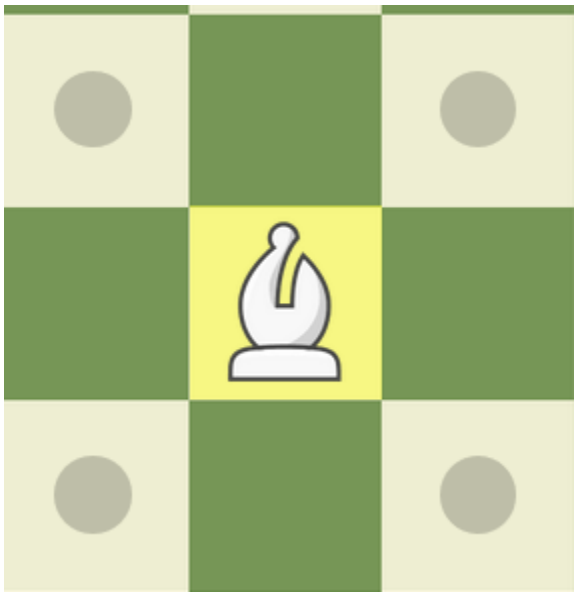
This is why the pawn is actually one of the most special pieces in chess and was the most complex to actually implement.

Knight:



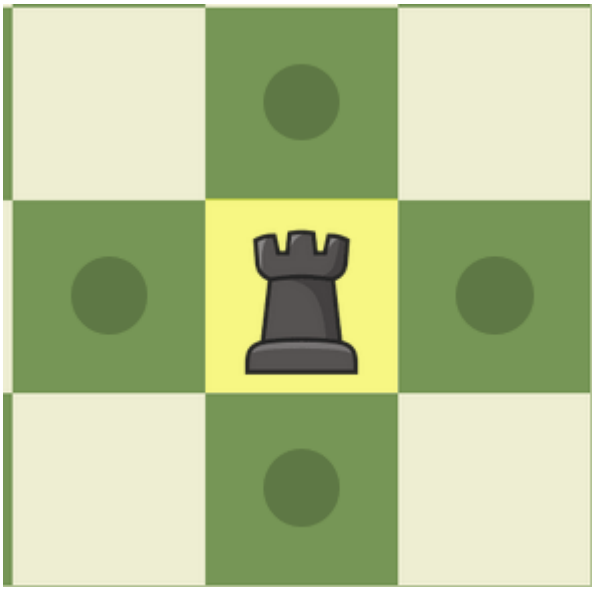
Next we have the knight, the knight has 8 possible moves in L directions from its position, unlike other pieces the knight is able to jump over other pieces in order to capture, it is also able to move forwards and backwards.

Bishop:



Next we have the bishop, the bishop is able to move in any direction of its diagonals, uniquely from other pieces, a bishop is limited to the color of its square, so on a chessboard there is a light square bishop as seen above and a dark square bishop.

Rook:

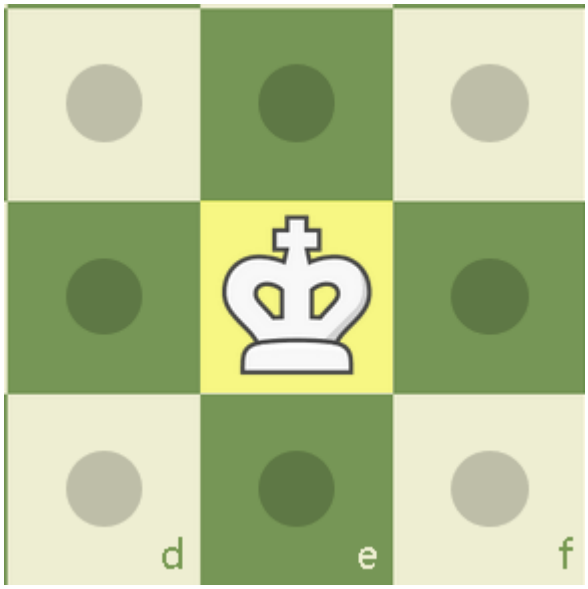


The rook can move in 4 directions in the NSWE directions, to the front and its sides. The rook is able to make a special move that we will define with the king piece knows as a castle. The rook is the second strongest piece on the board.

Queen:

The queen is the strongest piece on the board, it is a combinaion of the rook and bishop and is the main attackign force for each side in chess. You can win a game of chess quite easily by taking your opponent’s queen.

King:

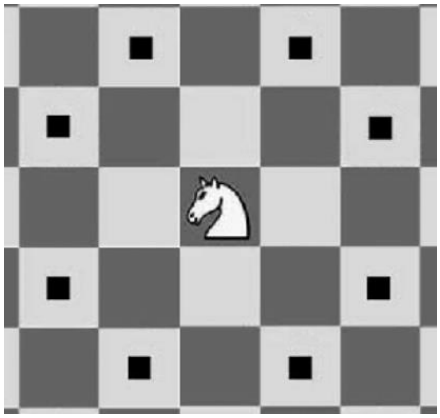


Last but certainly not least, we have chess' most important piece, the king. The king is the focal point of every game where an attack on the king wins a player the game. The king is able to move in all positions but only by one tile. Its moves can be limited by a check where it is being attacked by another piece. When a check is made, the king under attack must move out of the position or blocked by an allied piece. The king is also able to make a special move known as a castle as we mentioned before. A castle is a move wher the king trades positions with the rook and so it is protected behind it, a castle cannot be made if the king or rook has been moved prior to being able to castle, no pieces must be blocking the king rook and the king cannot be under attack while castling. There are two types of castling moves, a castle with the kign side rook and the queen side rook.The queen side rook takes longer to execute since the queen must be moved from its position.

That's all about the pieces and the general rules of chess. Lets get into the code!

So I began my chess project with two things in mind, I had to design an 8x8 chessboard, 32 pieces in total and a black side and a white side to compete against each other. We begin by creating our chessboard and initial pieces using the **Board** class and a **Piece** class. Within both classes, our base for the game is designed, we separate the black and the white pieces, determine the position of each piece on the board which we can use later and a way to calculate the amount of legal moves for a certain piece. The **Piece** class is defined as an abstract class which we will use when implementing each othe pieces later, each piece will have their own legal moves due to their differences within the game. For example, a queen will move differently from a knight would and so we must be able to determing the legal moves for each piece. By taking a one dimensional approach to the chessboard, we can assume that since there are 64 tiles on a chess board, we will take the first tile as the top left corner tile and the last tile as the bottom right corner tile, this makes detemining a piece's position on the board quite easily. However, there is a catch to this one dimensional approach when approaching the movement of certain pieces. Certain moves in very specific positions would not be possible for certain pieces.

For example, we can take the movement of a knight:



As seen here, the knight has these possible moves when at the center of the board. However, if the knight were to be in the right or left border columns of the board the number of moves they have becomes limited. Our approach sees the knight's movement caculated using offsets based on the current position of the knight. For example, these 8 moves would be the offsets -17, -15, 10,-6, 6, 10, 15, 17, we know this because from the knight's current position, if we add any of these

numbers to the knight's position, we will get the position of each of these black dots, these become the possible moves for a knight.

Here is an example where the white knight is on the left corner of the board, if we assumed it's positon on this board using our dimensional approach, it would be on the tile position numbered 41, if we were to calculate the knight's possible moves on its left side using our previous offsets, it would end up on the right side of the board, making these moves invalid. This applies to all of the other pieces, which despite having different offsets of movement would have their own exceptions when on these parts of the board, this is what I call an edge case and it is one of the issues of using a one-dimesional approach, these edge cases differ for each piece since they move in different ways but each one of them has their own edge cases. Once we can calculate the moves for each piece and define their exclusions, we are able to create our chessboard and set up the pieces and their default positions through our command line. Next we are going to define the movements of each piece, since we know how a piece needs to move we can determing the kind of moves we can make within our **Move** class. In chess a piece can do two kinds of moves, normal movement and what is known as a capture or an attacking move. Capturing a piece is the act of taking that piece using one of your own. There is a specific way of translating one of these moves into our code. We do this by redrawing the position of a piece when it is moved and checking whether that move is legal or not, after the piece is moved into the position, the condition of each piece is determined and the board is redrawn after the move is made, we do these within our

MoveTransition and **MoveStatus** classes. This is the backbone of the chess engine and all that's left is to establish the rest of the rules within the other pieces and exceptions such as En Passant and special game states such as check, checkmate and more. For the GUI section of the game I decided to use Swing. We had a few classes introducing us to GUI using Swing and that's why I am using it over JavaFX. Most of my time had been spent doing the coding for the chess but the GUI was a nice change and was something I could use as visual feedback to check if my code worked. Within the GUI, a move history tab was implemented and a tab that showed which pieces were now off the board, with the GUI, it is also possible to flip the board to change the player's perspective and do things such as highlight possible moves or checks. However, I was unable to implement the highlighting features and while I am ultimately disappointed for missing a feature, it just asserts the fact that I still have much to learn about GUI programming and that my inexperience led to this mishap. Many more GUI features could have been added such as a takeback or undo move or possibly an offer draw button, there were more plans for funner additions towards the GUI that didn't make it into the fnal program. The GUI also enables a game over screen when a player is checkmated or stalemated. Most of the special moves were aso implemented quite well, these include moves such as En Passant and the promotion of a pawn. The caveat is that certain moves such as pawn promotion was limited due to limitations with my own knowledge of my code, I was unable to promote the pawn into any piece I wanted and so I settled on the default promotion to be a queen since that is the most common choice for promoting a pawn, this means that one part of the game is somewhat incomplete and I may return to finish it in the future so it plays properly. But the largest struggle within the code was implementing castling. Despite being one of the most important moves in chess being the difference in a game ending in 10 moves to one in 20 moves. After lots of time debugging and looking for a solution, this is the only special move in the game that I was unable to implement. Initially, I was thinking of implementing an AI opponent for this project since games such as tic tac toe, checkers and other board games can easily implement a computer opponent using what is known as the minimax algorithm. However, due to time limitations, I decided to make it a local player vs player game.

II.b Code and Class Analysis

There are a grand total of 21 files within my project, each with their own different functions. We can take a dive into some of the features of each class and why they may have been designed in a

certain way to fit the final product. A more detailed analysis of specific functions and code are within the comments of my code

Some but not all of the classes will be discussed here.

The Piece Class

```java
package chessProject;

import java.util.Collection;
//the abstract class each of the piece classes inherit from
public abstract class Piece {
    //the piece has a piece type, its position on the board, the side/alliance it's on and a boolean value for its first move
    //and a hash code used when overriding the pre-defined hashcode() function
    protected final PieceType pieceType;
    protected final int piecePosition;
    protected final Alliance pieceAlliance;
    protected final boolean isFirstMove;
    private final int cachedHashCode;

    //the constructor for the piece class
    Piece(final PieceType pieceType,
          final int piecePosition,
          final Alliance pieceAlliance,
          final boolean isFirstMove) {
        this.pieceType=pieceType;
        this.pieceAlliance=pieceAlliance;
        this.piecePosition=piecePosition;
        this.isFirstMove=isFirstMove;
        this.cachedHashCode=computeHashCode();
    }
    //hash function used to calculate the hash code
    private int computeHashCode() {
        int result=pieceType.hashCode();
        result=31*result+pieceAlliance.hashCode();
        result=31*result+piecePosition;
        result=31*result+(isFirstMove? 1:0);
        return result;
    }
```

Here is an example of the abstract piece class which all of the other piece classes inherit from. Many of the classes within the project are designed in an immutable fashion since we are working with collections, many of these variables are set as constants to prevent their mutation.

The Knight Class

```java
import java.util.ArrayList;
//This is the code for the knight piece
public class Knight extends Piece{
    //These are the coordinate offsets calculated for a knight using our one dimensional approach
    private final static int[] CANDIDATE_MOVE_COORDINATES = {-17, -15, -10, -6, 6, 10, 15, 17};
    //A knight is constructed with its side and its position on the board
    //It also inherits from its piece superclass
    public Knight(final Alliance pieceAlliance,
                  final int piecePosition) {
        super(PieceType.KNIGHT, piecePosition, pieceAlliance, true);
    }

    public Knight(final Alliance pieceAlliance,
                  final int piecePosition,
                  final boolean isFirstMove) {
        super(PieceType.KNIGHT, piecePosition, pieceAlliance, isFirstMove);
    }

  //This the code to calculate the legal moves for a knight
    @Override
    public Collection<Move> calculateLegalMoves(final Board board) {
        //An arraylist of legal moves is used to store all of possible legal moves
        final List<Move> legalMoves=new ArrayList<>();
        for(final int currentCandidateOffset:CANDIDATE_MOVE_COORDINATES){
            final int candidateDestinationCoordinate=this.piecePosition+currentCandidateOffset;
          //If the knight is on a valid tile and once the illegal moves have been excluded, the moves can be calculated
            if(BoardUtils.isValidTileCoordinate(candidateDestinationCoordinate)){


                if(isFirstColumnExclusion(this.piecePosition, currentCandidateOffset) ||
                        isSecondColumnExclusion(this.piecePosition, currentCandidateOffset) ||
                        isSeventhColumnExclusion(this.piecePosition, currentCandidateOffset) ||
                        isEighthColumnExclusion(this.piecePosition, currentCandidateOffset)) {
                    continue;
                }
                final Tile candidateDestinationTile=board.getTile(candidateDestinationCoordinate);
                //Assuming that no tile occupies the tile that the knight is going to be at, a normal move is added to the
                //legal moves array list
                if(!candidateDestinationTile.isTileOccupied()){
                    legalMoves.add(new Move.MajorMove(board, this, candidateDestinationCoordinate));
                } else{
                    //Assuming that no tile occupies the tile that the knight is going to be at, a normal move is added to the
                    //legal moves array list
                    final Piece pieceAtDestination= candidateDestinationTile.getPiece();
                    final Alliance pieceAlliance= pieceAtDestination.getPieceAlliance();

                    if(this.pieceAlliance != pieceAlliance) {
                        legalMoves.add(new Move.MajorAttackMove(board, this, candidateDestinationCoordinate, pieceAtDestination));
                    }
                }
            }
        }
      //Once the moves have been collected they are returned as an immutable list of moves
        return Collections.unmodifiableList(legalMoves);
    }
  //An overridden method describing the movement of a knight
    @Override
```

```java
  //An overridden method describing the movement of a knight
    @Override
    public Knight movePiece(final Move move) {
        return new Knight(move.getMovedPiece().getPieceAlliance(), move.getDestinationCoordinate());
    }
  //An overridden method describing the toString() method for a knight
    @Override
    public String toString(){
        return PieceType.KNIGHT.toString();
    }
    //A knight has some edge cases regarding the first column,second, seventh and on the eight, the following offsets are invalid
    //because they would cause the piece to move off the board and onto a wrong position.
    private static boolean isFirstColumnExclusion(final int currentPosition, final int candidateOffset){
        return BoardUtils.FIRST_COLUMN[currentPosition] && (candidateOffset == -17 || candidateOffset == -10 ||
                candidateOffset == 6 || candidateOffset == 15);
    }

    private static boolean isSecondColumnExclusion(final int currentPosition, final int candidateOffset) {
        return BoardUtils.SECOND_COLUMN[currentPosition] && (candidateOffset == -10 || candidateOffset == 6);
    }

    private static boolean isSeventhColumnExclusion(final int currentPosition, final int candidateOffset) {
        return BoardUtils.SEVENTH_COLUMN[currentPosition] && (candidateOffset == -6 || candidateOffset == 10);
    }

    private static boolean isEighthColumnExclusion(final int currentPosition, final int candidateOffset) {
        return BoardUtils.EIGHTH_COLUMN[currentPosition] && (candidateOffset == -15 || candidateOffset == -6 ||
                candidateOffset == 10 || candidateOffset == 17);
    }

}
```

Here is the knight class in full so we can show how the code in other pieces would look like, the knight's position is defined and its movement patterns as well, it inherits a method from the piece class used to calculate the legal moves for a knight, the moves are then stored in an immutable list of legal moves to be used later. As specified earlier, we must handle the edge cases for each piece on the board so that certain invalid moves cannot be made. For the knight, these certain moves are invalid on the first column , the second column, the seventh column and the eighth column. The other pieces such as the bishop, rook and queen have similar code but with different offsets for **movement, different exclusions and different legal moves.**

The Alliance enum

```java
package chessProject;
// An enum is used here to define the black side and the white side in the game of chess.
// All functions defined within this enum are abstract so they can be overridden within the black and white enum values

public enum Alliance {
    WHITE {

        @Override
        public int getDirection() {
            return -1;
        }

        @Override
        public int getOppositeDirection() {
            return 1;
        }

        @Override
        public boolean isWhite() {
            return true;
        }

        @Override
        public boolean isBlack() {
            return false;
        }

        @Override
        public boolean isPawnPromotionSquare(int position) {
            return BoardUtils.EIGHTH_RANK[position];
        }

        @Override
        public Player choosePlayer(final WhitePlayer whitePlayer,
                                   final BlackPlayer blackPlayer) {
            return whitePlayer;
        }
        @Override
        public String toString() {
            return "White";
        }
    },
```

Activate Windows

The Alliance class is shown, this is the code for the white side, here an enum is used instead of a class, the enumerable white contains values for the white side of the board, functions for the direction are defined because the white pieces have to move towards the black pieces, the getOppositeDirection() method is used so that a piece is eligible to also move backwards, an isWhite() and isBlack() method here is used so we know if a piece is black or is white if it returns true and vice versa. We have a function that defines the pawn promotion square which is the square on which a pawn promotes and is on the opposite side of the board for a given alliance. There is also a choosePlayer() function used to choose which player is eligible to make a move and follows that turn order and a toString() method for the piece's alliance. The black enumerable overrides the same functions but is pretty much the opposite of the white player.

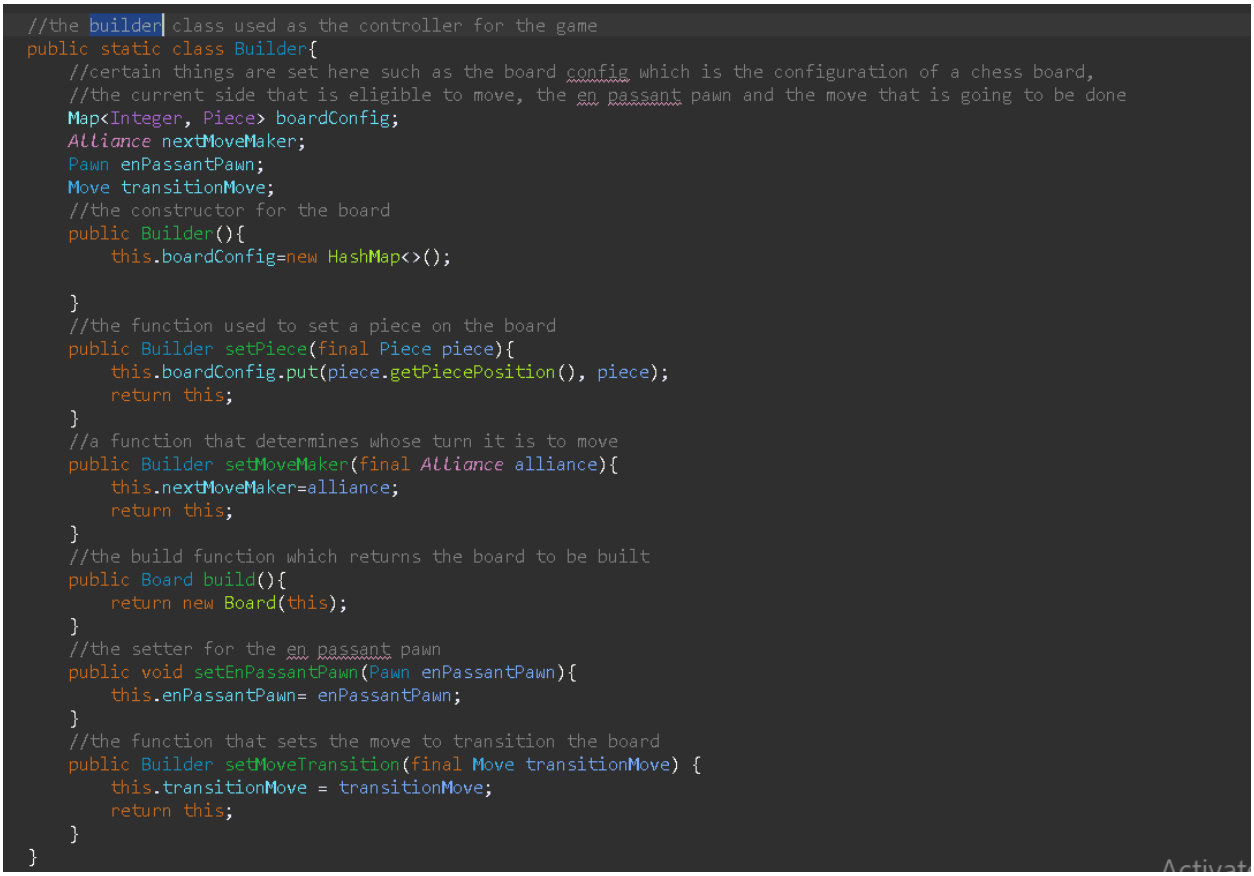
```java
//the board class used for the chessboard
public class Board {

    //here we have a list of tiles that make up the game board, the white pieces and black pieces
    //, the white and black player and the current player which is the player eligible to move
    //there is also a check for the en passant pawn since there can only be one en passant pawn
    //on the board at any given time.
    private final List<Tile> gameBoard;
    private final Collection<Piece> whitePieces;
    private final Collection<Piece> blackPieces;
    private final WhitePlayer whitePlayer;
    private final BlackPlayer blackPlayer;
    private final Player currentPlayer;
    private final Pawn enPassantPawn;

    //the constructor for a chessboard using the builder class, the board, pieces and players are given their values here
    private Board(final Builder builder){
        this.gameBoard=createGameBoard(builder);
        this.whitePieces=calculateActivePieces(this.gameBoard, Alliance.WHITE);
        this.blackPieces=calculateActivePieces(this.gameBoard, Alliance.BLACK);
        this.enPassantPawn = builder.enPassantPawn;
        final Collection<Move> whiteStandardLegalMoves=calculateLegalMoves(this.whitePieces);
        final Collection<Move> blackStandardLegalMoves=calculateLegalMoves(this.blackPieces);
        this.whitePlayer = new WhitePlayer(this, whiteStandardLegalMoves, blackStandardLegalMoves);
        this.blackPlayer = new BlackPlayer(this, whiteStandardLegalMoves, blackStandardLegalMoves);
        this.currentPlayer = builder.nextMoveMaker.choosePlayer(this.whitePlayer, this.blackPlayer);
    }

    //the to string method used to generate a string output of the chessbard's starting position, since each tile
    //has their own to string and the pieces have names, the board will be generator with the white pieces in upper case
    //and the black pieces in lower case and the empty tiles will have the - string
    @Override
    public String toString(){
        final StringBuilder builder=new StringBuilder();
        for(int i=0;i<BoardUtils.NUM_TILES;i++){
            final String tileText=this.gameBoard.get(i).toString();
            builder.append(String.format("%3s", tileText));
            if((i+1) % BoardUtils.NUM_TILES_PER_ROW==0){
                builder.append("\n");
            }
        }
        return builder.toString();
    }
```

Activate Windo
Go to Settings to ac



```
//the builder class used as the controller for the game
public static class Builder{
    //certain things are set here such as the board config which is the configuration of a chess board,
    //the current side that is eligible to move, the en passant pawn and the move that is going to be done
    Map<Integer, Piece> boardConfig;
    Alliance nextMoveMaker;
    Pawn enPassantPawn;
    Move transitionMove;
    //the constructor for the board
    public Builder(){
        this.boardConfig=new HashMap<>();

    }
    //the function used to set a piece on the board
    public Builder setPiece(final Piece piece){
        this.boardConfig.put(piece.getPiecePosition(), piece);
        return this;
    }
    //a function that determines whose turn it is to move
    public Builder setMoveMaker(final Alliance alliance){
        this.nextMoveMaker=alliance;
        return this;
    }
    //the build function which returns the board to be built
    public Board build(){
        return new Board(this);
    }
    //the setter for the en passant pawn
    public void setEnPassantPawn(Pawn enPassantPawn){
        this.enPassantPawn= enPassantPawn;
    }
    //the function that sets the move to transition the board
    public Builder setMoveTransition(final Move transitionMove) {
        this.transitionMove = transitionMove;
        return this;
    }
}
```

Here we have two screenshots of the **Board** class which we use to create the chess board for chess, since most of the code is explained within the file, a short rundown of the code and its functions can be explained here.Certain things such as the black and white players and their respective pieces are being instantiated, the legal moves of either player are taken into account and using some functions we can even choose a player to make a move before moving on, we have methods to construct the board as an 8x8 chessboard and then assign the pieces to their default positions. An important class within the Board class is the Builder class, it acts as the controller for the game, the builder class creates the configuration for the board, is used to set down pieces for the board and is also the one selecting the current move maker, it sets the en passant pawn and also the moe we need to transition the board to the new board where a move is made.

The Player Class

```java
package chessProject;

import java.util.Collection;
import java.util.Collections;
import java.util.stream.Collectors;

import chessProject.Piece.PieceType;

//The abstract class player which the black and white player classes inherit from
public abstract class Player {
    //The member variables of a player, the board, each player's respective king and the amount of legal moves they have,
    //a boolean value is also used to see whether the king piece is being attacked
    protected final Board board;
    protected final King playerKing;
    protected final Collection<Move> legalMoves;
    protected final boolean isInCheck;
    //Within the constructor, we have the board and two collections of moves, the amount of legal moves that player has
    //and their oppopnent's legal moves.
    Player(final Board board,
           final Collection<Move> playerLegals,
           final Collection<Move> opponentLegals) {
        this.board = board;
        //a function used to establish a king on either player's side
        this.playerKing = establishKing();
        //a function used to narrow down the meaning of a check which in this case we deduce whether the king is being attacked
        //and this would mean the king is in check
        this.isInCheck = !calculateAttacksOnTile(this.playerKing.getPiecePosition(), opponentLegals).isEmpty();
        this.legalMoves = Collections.unmodifiableCollection(playerLegals);
    }
    //function used to check
    public boolean isInCheck() {
        return this.isInCheck;
    }
    //function used to check for checkmate, like in real chess, checkmate occurs when the king is currently under check
    //and it has no where else to move or no legal moves to make. this results in a game over for the player.
    public boolean isInCheckMate() {
       return this.isInCheck && !hasEscapeMoves();
    }
    //function used to check for stalemate, in chess a stalemate occurs when the king is not being checked. However, it is unable
    //to move because the tiles within its legal moves are being attacked and so it is unable to move. This is also a game over
    //for either player
    public boolean isInStaleMate() {
        return !this.isInCheck && !hasEscapeMoves();
    }
```

Activate Windows
Go to Settings to activate Windows.

Here we have the player class which is an abstract class later inherited by the BlackPlayer and the WhitePlayer classes, within the class we have the board we're moving on, the king of the player and the legal moves of each player. Functions are used to establish the player king by looping through each piece until the piece of piece type king is found, there are also functions used to check whether the player's king is in check, checkmate and stalemate, there are many other functions such as checking for castling, a way to gather all legal moves and the way a player makes a move.

The Board Class

```java
package chessProject;

import chessProject.Board.Builder;
//the abstract class move, here all the types of moves are defined
public abstract class Move {
    //the board that the move is being made on is defined, the destination location of the move
    //the piece being moved and whether or not the piece being moved has moved yet
    protected final Board board;
    protected final int destinationCoordinate;
    protected final Piece movedPiece;
    protected final boolean isFirstMove;
    //constructor for the move
    private Move(final Board board,
                 final Piece pieceMoved,
                 final int destinationCoordinate) {
        this.board = board;
        this.destinationCoordinate = destinationCoordinate;
        this.movedPiece = pieceMoved;
        this.isFirstMove = pieceMoved.isFirstMove();
    }
  //constructor for an invalid move
    private Move(final Board board,
                 final int destinationCoordinate) {
        this.board = board;
        this.destinationCoordinate = destinationCoordinate;
        this.movedPiece = null;
        this.isFirstMove = false;
    }
    //An overridden function of the hash code function , we use our own implementation because a class will not be able to function properly
    //when working with hash based collections if .equals() is overridden.
    @Override
    public int hashCode() {
        int result = 1;
        result = 31 * result + this.destinationCoordinate;
        result = 31 * result + this.movedPiece.hashCode();
        result = 31 * result + this.movedPiece.getPiecePosition();
        result = result + (isFirstMove ? 1 : 0);
        return result;
    }
    //overridden equals function
    @Override
    public boolean equals(final Object other) {
        if (this == other) {
```

Activate Windows
Go to Settings to activate Windows.

```java
        if (this == other) {
            return true;
        }
        if (!(other instanceof Move)) {
            return false;
        }
        final Move otherMove = (Move) other;
        return getCurrentCoordinate() == otherMove.getCurrentCoordinate() &&
               getDestinationCoordinate() == otherMove.getDestinationCoordinate() &&
               getMovedPiece().equals(otherMove.getMovedPiece());
    }
    //getter method for the board
    public Board getBoard() {
        return this.board;
    }
  //getter method for the current location of a piece
    public int getCurrentCoordinate() {
        return this.movedPiece.getPiecePosition();
    }
  //getter method for the destination location
    public int getDestinationCoordinate() {
        return this.destinationCoordinate;
    }
  //getter method for moved piece
    public Piece getMovedPiece() {
        return this.movedPiece;
    }
  //method to check for an attack
    public boolean isAttack() {
        return false;
    }
  //method to check for a castling move
    public boolean isCastlingMove() {
        return false;
    }
  //method to check where a piece is being attacked
    public Piece getAttackedPiece() {
        return null;
    }
    //this function is used to execute a move, a builder is initialized and then the active pieces for each player are
    //added and the moved piece is set on the board, the move is made by the current player and then transitioned
    //and the move is finally made when the builder is built again.
    public Board execute() {
        final Board.Builder builder = new Builder();
```

Activate Windows
Go to Settings to activate Windows.

```
public Board execute() {
    final Board.Builder builder = new Builder();
    this.board.currentPlayer().getActivePieces().stream().filter(piece -> !this.movedPiece.equals(piece)).forEach(builder::setPiece);
    this.board.currentPlayer().getOpponent().getActivePieces().forEach(builder::setPiece);
    builder.setPiece(this.movedPiece.movePiece(this));
    builder.setMoveMaker(this.board.currentPlayer().getOpponent().getAlliance());
    builder.setMoveTransition(this);
    return builder.build();
}


//a function used to return a string after the position of the pieces have been changed in order to keep track of the move and its
//notation value
String disambiguationFile() {
    for(final Move move : this.board.currentPlayer().getLegalMoves()) {
        if(move.getDestinationCoordinate() == this.destinationCoordinate && !this.equals(move) &&
           this.movedPiece.getPieceType().equals(move.getMovedPiece().getPieceType())) {
            return BoardUtils.getPositionAtCoordinate(this.movedPiece.getPiecePosition()).substring(0, 1);
        }
    }
    return "";
}
```

This is the **Move** class where most of the different kinds of moves are being defined. All of the kinds of moves such as a normal move, an attacking move or more special moves such as pawn jump and en passant and castling and pawn promotion . Each of these different moves are going to have different implementations and have different interactions on the board. Most of these moves are then used for the calculation of legal moves for each of the pieces. More details and most of the analysis is located within the code and the attached comments.

The Table Class

```
26 public class Table {
27     //the frame for the game
28     private final JFrame gameFrame;
29     //the game history panel
30     private final GameHistoryPanel gameHistoryPanel;
31     //a panel that shows which pieces were taken on the board
32     private final TakenPiecesPanel takenPiecesPanel;
33     //the panel used to display the board
34     private final BoardPanel boardPanel;
35     //a log of moves which is used for the game history
36     private final MoveLog moveLog;
37     //the chessboard
38     private Board chessBoard;
39     //the tile that a piece is currently on
40     private Tile sourceTile;
41     //the tile that a piece is going to move to
42     private Tile destinationTile;
43     //the piece that is going to be moved
44     private Piece humanMovedPiece;
45     //the board direction
46     private BoardDirection boardDirection;
47     private boolean highLightlegalMoves;
48     //the window dimensions for the gui program
49     private final static Dimension OUTER_FRAME_DIMENSION=new Dimension(920, 800);
50     //the dimensions for the chess board panel
51     private final static Dimension BOARD_PANEL_DIMENTION=new Dimension(400, 350);
52     //the dimensions for each tile on the board
53     private final static Dimension TILE_PANEL_DIMENSION=new Dimension(50,50);
54     //the path for the piece icons
55     private static String defaultPieceImagesPath="art/pieces/";
56     private static String HighlightPath="art/misc/green_dot.png";
57     //the default colors of the light and dark squares
58     private final Color lightTileColor=Color.decode("#eeeed2");
59     private final Color darkTileColor=Color.decode("#769656");
60     public Table(){
61         //the constructor for the table class where all of the panels and menus are created
62         this.gameFrame=new JFrame("OHChess");
```

```
        this.gameFrame.setLayout(new BorderLayout());
        final JMenuBar tableMenuBar= createTableMenuBar();
        this.gameFrame.setJMenuBar(tableMenuBar);
        this.gameFrame.setSize(OUTER_FRAME_DIMENSION);
        this.chessBoard=Board.createStandardBoard();
        this.gameHistoryPanel = new GameHistoryPanel();
        this.takenPiecesPanel = new TakenPiecesPanel();
        this.boardPanel=new BoardPanel();
        this.moveLog = new MoveLog();
        this.boardDirection=BoardDirection.NORMAL;
        this.highLightlegalMoves= false;
        this.gameFrame.add(this.takenPiecesPanel, BorderLayout.WEST);
        this.gameFrame.add(this.boardPanel, BorderLayout.CENTER);
        this.gameFrame.add(this.gameHistoryPanel, BorderLayout.EAST);
        this.gameFrame.setVisible(true);
    }
    //the top bar of the gui known as the menu, it has a file menu and a preferences menu
    private JMenuBar createTableMenuBar() {
        final JMenuBar tableMenuBar=new JMenuBar();
        tableMenuBar.add(createFileMenu());
        tableMenuBar.add(createPreferencesMenu());
        return tableMenuBar;
    }
    //planned feature to add chess games by pgn file which is used to import pre-existing games of chess
    private JMenu createFileMenu() {
        final JMenu fileMenu=new JMenu("File");
        final JMenuItem openPGN= new JMenuItem("Load PGN File");
        openPGN.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent actionEvent) {
                System.out.println("Open up that PGN file!");
            }
        });
        fileMenu.add(openPGN);
        //an exit button that exits the program
        final JMenuItem exitMenuItem= new JMenuItem("Exit");
        exitMenuItem.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent actionEvent) {
                System.exit(0);
            }
        });
        fileMenu.add(exitMenuItem);
```

Activate Windows
Go to Settings to activate Windows.

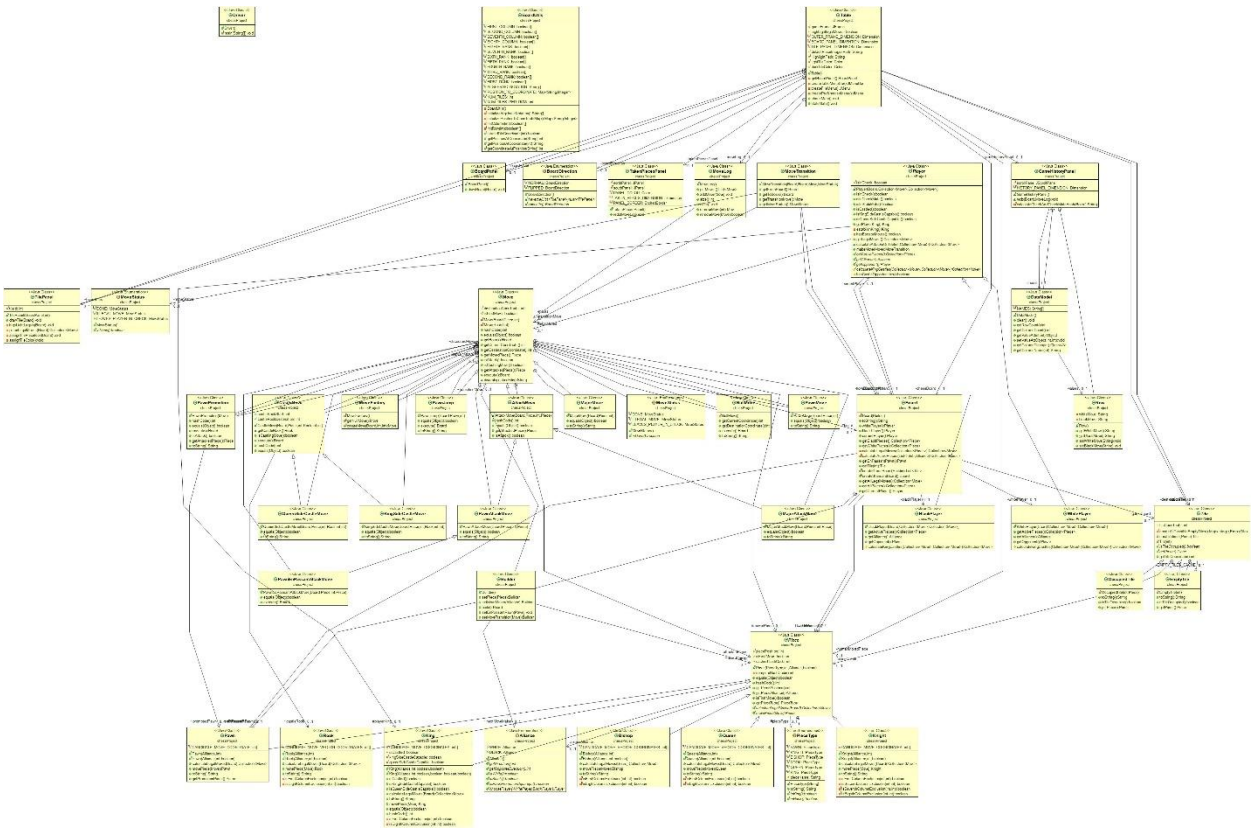
```
        fileMenu.add(exitMenuItem);
        return fileMenu;
    }
    //the preferences menu, here the board perspective can be flipped and the legal moves of a piece can be highlighted
    private JMenu createPreferencesMenu(){
        final JMenu preferencesMenu= new JMenu("Preferences");
        //flips the board's perspective
        final JMenuItem flipBoardMenuItem=new JMenuItem("Flip Board");
        flipBoardMenuItem.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(final ActionEvent e) {
                boardDirection=boardDirection.opposite();
                boardPanel.drawBoard(chessBoard);
            }
        });
        preferencesMenu.add(flipBoardMenuItem);
        preferencesMenu.addSeparator();
        //the highlight feature is implemented as a toggle option
        final JCheckBoxMenuItem legalMoveHighLighterCheckBox = new JCheckBoxMenuItem("HighLight legal moves", false);
        legalMoveHighLighterCheckBox.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                highLightlegalMoves = legalMoveHighLighterCheckBox.isSelected();
            }
        });

        preferencesMenu.add(legalMoveHighLighterCheckBox);
        return preferencesMenu;
    }
```

This is the main GUI class where the game window is made, other features were implemented in different classes such as panels that display the amounts of moves made and the pieces that had been taken by any of the players. The colors and image icons are set here and the mouse click events are also recorded here. As of now, the GUI could have had more features but it works well enough to play a decent game of chess without bugs.

II.c Class Diagram

For clarification, the class diagram shown was autogenerated using an external library, the image can be downloaded on the github link where this project is located



III.a. Lessons that Have Been Learned

There were many things that I learnt while creating this project. I think that this project was a great learning experience regarding OOP which was taught during a lot of our Programming Languages class. Many of the concepts defined there were difficult to grasp but after numerous exercises it became much easier to understand. I feel like this project was a good test of how well I understood these concepts and how they could be used. Many of the parts of the code required a lot of calculation and I feel like I learnt a lot more about the Java programming language.

One of the things that my resources for this project also taught me was the usage and importance of immutability and constants. This was an important concept I wish I had learnt about previously, I had trouble dealing with my variables in previous projects such as my Python projcet last semester. Lots of the bugs within that code could have been avoided if I had designed my

classes with this practice in mind. Within the code is also a large use of the Java Colllections framework and so many of the classes and variables were defined with this in mind.

We were taught a lot about inheritance and abstract classes within class and chess is a perfect example of how you can define each piece using abstract classes since they are all pieces but they work in different ways. It helped me to better understand the concept and why it is important to do so. I also had my fair share of learning about exceptions and how we could use them within our code. There were some applications of try and catch exception handling within the code. However, I felt like exception handling is still my weak link and I still have much more to learn about before I can use it regularly within my code.

Another concept we were taught is the use of overriding functions and within this code, many of the classes defined have simliar functions and just have different uses within each one. Prior to working on this project, my only knowledge of overriding functions was to override the built-in toString() function within Java but here I was able to override different functions within subclasses and built-in functions as well.

One thing I learnt about that was outside of our class topic is the use of the Enum in Java, I would have never thought abut using an enum for the sides within chess. I had initially thought of using a simple boolean value but after being introduced to the enum, the capabilities that you could use to implement such ideas becomes much more vast. This is another weak link for me because I feel like that it might be specific to certain uses but I hope that I could use the enum in future projects since it was so useful in this context.

Overall, I had an enjoyable time creating this project and writing the report. chess has become quite popular online and after I began playing and watching, I enjoyed my progress within my project more and more. However, I had a lot of issues regarding bugs since I had never worked with a project so far outside my comfort zone and I lost motivation at times when my code that I took hours upon hours to write resulted in errors and more errors. Overall, I like to think that stepping outside my zone was something that I really needed to do in order to improve. Even so, there was so much more that could have been added to the code to make it a more interactive engine like the ones so easily found online.

Resources :

https://www.youtube.com/watch?v=h8fSdSUKttk&list=PLOJzCFLZdG4zk5d-1_ah2B4kqZSeIlWtt&index=1

https://stackoverflow.com/

Chess piece textures courtesy of Chess.com

IV. Source Code

All of the code is located at this github link:

V. Working Program Evidence of code (Screenshots)

This is the GUI of the code when first started

Here we can see the GUI when a piece is taken and you can see the history of moves made on the right by white and by black.

This is an example of some moves being played out resulting in a checkmate.