Criterion C: Development

Techniques Used:

- Recursion and search trees for AI decision making (minimax algorithm)
- Polymorphism
- Encapsulation
- Nested loops
- GUI
- Filtering from possible moves to legal moves. (Rule Making)

Explanation of Techniques:

Recursion and Search Trees

Using a recursive method to create a search tree for the AI's decision making was a critical part to the chess engine. The chess AI used the minimax algorithm, which aims to maximize the minimal gain of the AI's moves, while returning the best playable move at a given position. The minimax algorithm also relies on a heuristic function (see appendix for reference to source code), which considers the state of the game (opening, midgame, endgame), as well as the safe mobility and positioning of all pieces, to return an accurate board rating. The minimax algorithm is also enhanced by alpha-beta pruning; a pruning method that discontinues a branch if a worse move would be played by the AI, thus reducing heap usage and increasing search depth.

```
public Piece[][] miniMax(ChessDisplay game){
    int lowestScore = Integer.MAX_VALUE;
    Piece[][] bestMove = null;

    ArrayList<Piece[][]> moves = game.getAllBoards( color: 1, game.chessboard);
    int score = 0;
    for (Piece[][] move : moves) {
        score = min(move, game,  depth: 3, game.BLACK,  alpha: -10000,  beta: 10000);
        if (score < lowestScore) {
            lowestScore = score;
            bestMove = move;
        }
    }

    return bestMove;
}
```

```java
public int min(Piece[][] board, ChessDisplay game, int depth, byte color, int alpha, int beta){
    if(depth == 0){
        return game.EvaluateBoard(board);
    }
    int highestScore = Integer.MAX_VALUE;

    byte next_color;
    if (color == game.WHITE) {
        next_color = game.BLACK;
    } else {
        next_color = game.WHITE;
    }

    ArrayList<Piece[][]> moves = game.getAllBoards(color, board); //creates the children nodes
    int score = 0;
    for (Piece[][] move : moves) {
        score = max(move, game,  depth: depth -1, next_color, alpha, beta);
        if(score <= alpha){
            return alpha;
        }
        if(score < beta){
            beta = score;
        }
        highestScore = beta;
    }

    return highestScore;
}
public int max(Piece[][] board, ChessDisplay game, int depth, byte color, int alpha, int beta){
    if (depth == 0) {
        return game.EvaluateBoard(board);
    }
    int lowestScore = Integer.MIN_VALUE;

    byte next_color;
    if (color == game.WHITE) {
        next_color = game.BLACK;
    } else {
        next_color = game.WHITE;
    }

    ArrayList<Piece[][]> moves = game.getAllBoards(color, board);
    int score = 0;
    for (Piece[][] move : moves) {
        score = min(move, game,  depth: depth-1, next_color, alpha, beta);
        if(score >= beta){
            return beta;
        }
        if(score > alpha){
            alpha = score;
        }
        lowestScore = alpha;
    }

    return lowestScore;
}
```

Ingenuity and Appropriateness:

Using the two helper methods, min and max, allowed for an easy way to pass the data from each node, as it travels up the minimax tree. The minimax algorithm is very appropriate and clever as it allows the AI to look a few moves in advance while not making a move that will lead to a serious blunder. The alpha-beta pruning of this algorithm is especially clever as it never changes the output of the tree, but only the total number of nodes, thus saving a great amount of memory and run time. The depth of the search tree is also easily controlled, allowing for the sophistication of the AI to be controlled.

References:

Minimax. (n.d.). Retrieved from Wikipedia website:

https://en.wikipedia.org/wiki/Minimax

Minimax Algorithm in Game Theory | Set 1 (Introduction). (n.d.). Retrieved from

Geegksforgeeks website: https://www.geeksforgeeks.org/minimax-algorithm-in-game-

theory-set-1-introduction/

Minimax Algorithm with Alpha-beta pruning. (n.d.). Retrieved from hackerearth website:

https://www.hackerearth.com/blog/artificial-intelligence/minimax-algorithm-alpha-

beta-pruning/

Polymorphism:

Creating abstract classes to be used in different implementations is an essential part to creating a strong inheritance structure. This program used an abstract class, an interface with basic implementation, for a general Piece object. The variables of the Piece object would then be inherited by its children, the pawns, knights, etc. The children of the Piece object, the pawns, knights, bishops, etc., would inherit the variables of the parent, and add their own specific implementation.

Each piece will have certain characteristics that are universal, and therefore were implemented within the parent. This includes drawing, moving, and converting grid values into pixel values.

Figure 1: the draw method.

```java
/**
 * Draws a string character of the piece at its given position.
 * @param g: java graphics object; allows objects to be drawn to the GUI
 */
public void draw(Graphics g){
    Graphics2D g2d = (Graphics2D)g;
    Font font = new Font( name: "Arial", Font.PLAIN,  size: 66);
    g2d.setFont(font);

    //Drawing the piece
    g2d.drawString(symbol, position[X], position[Y]);
}
```

Figure 2: the set position method. (only x is shown)

```java
/** This method converts a grid position into a pixel value that will correspond to
 * the appropriate grid on the chessboard.
 * @param gridPosX: the x position of a piece on the chessboard (0-7)
 * @return
 */
public short setPositionX(byte gridPosX){
    //Adding the board buffer is important to ensuring that the clicks align with the chessboard on screen.
    position[X] = (short)(gridPosX*60 + boardBufferX);
    return position[X];
}
```

Figure 3: the method that moves the piece

```java
/**
 * Moves a piece to a position defined in the parameter, from its original position.
 * @param pos: the position the piece is to be moved to.
 */
public void move(Position pos){
    gridPosX = pos.returnX();
    gridPosY = pos.returnY();

    //takes a grid position and converts it a pixel value using the method above.
    short x = setPositionX(gridPosX);
    short y = setPositionY(gridPosY);

    //sets the position to the grid value as calculated above.
    position[X] = x;
    position[Y] = y;
}
```

Ingenuity

This approach to handling the pieces is clever as these methods will be reused by all of the specific piece objects, and therefore would otherwise have to be written in multiple places. This firstly makes the code more understandable and is better for editing purposes, as the code would only have to be edited in one place as opposed to six places.

Appropriateness:

This approach properly aligns with the requests of the user, as it allows each piece to have the correct amount of unique implementation, as well as general implementation from the parent class. As requested by the user, each type of piece must move as it would per the rules of chess, and this approach allows all pieces to share the same general functions. With this approach, each piece can move piece in a normal manner and will make integration with the GUI much easier.

References:

Lewis, J. (2015). *Java Software Solutions* (8th ed.). Pearson.

Polymorphism in Java. (n.d.). Retrieved from beginner's book website:

https://beginnersbook.com/2013/03/polymorphism-in-java/

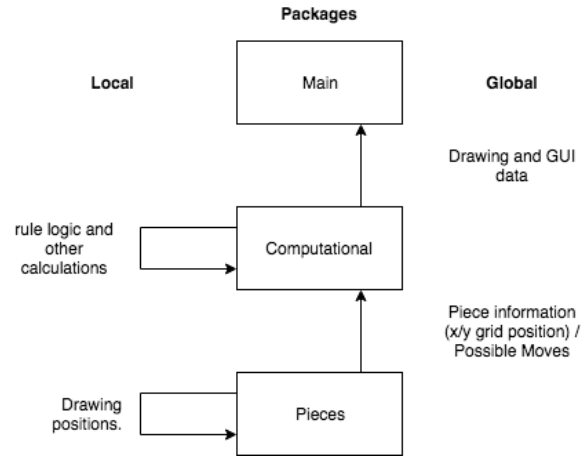
Java™ Platform, Standard Edition 7 API Specification. (n.d.). Retrieved from oracle website:

https://docs.oracle.com/javase/7/docs/api/

Encapsulation:

Encapsulation involves dividing the program into separate modules, as well as creating a distinction between local and global variables, ensuring that the flow of data is organized. Dividing up the program into separate modules allows the programmer to implement each module in different phases (or version) and still have a functioning program. This was mainly done by placing all the piece related objects, menu related objects, and logic/computing related objects into separate packages.

Fig 4: flow chart for the flow of data between packages.



Ingenuity: Separating all objects into the three packages firstly allowed for a structured hierarchy of data flow as well as created general organization with the code. The different modules can also be edited separately, thus making the process of updating versions, much easier.

Appropriateness: Each package focuses on one aspect of the program that will be used to fulfill

the success criteria. This is appropriate because a clear distinction of the purpose of each object is clearly identified.

The data and methods for each class were either set as local or global, either allowing the data to be wrapped within the class, or to allow other classes for specific methods.

References:

Lewis, J. (2015). *Java Software Solutions* (8th ed.). Pearson.

Java™ Platform, Standard Edition 7 API Specification. (n.d.). Retrieved from oracle website:

https://docs.oracle.com/javase/7/docs/api/

Graphical User Interface:

As user interactions are a key component to any software game, this program used a GUI for interacting with the program and for playing the game, per the client's requests. The java graphics library was used to create the graphical user interface for the program. The GUI contained panels with buttons as well as a playable interface for the game itself.

Fig 6: screenshot of the starting menu.

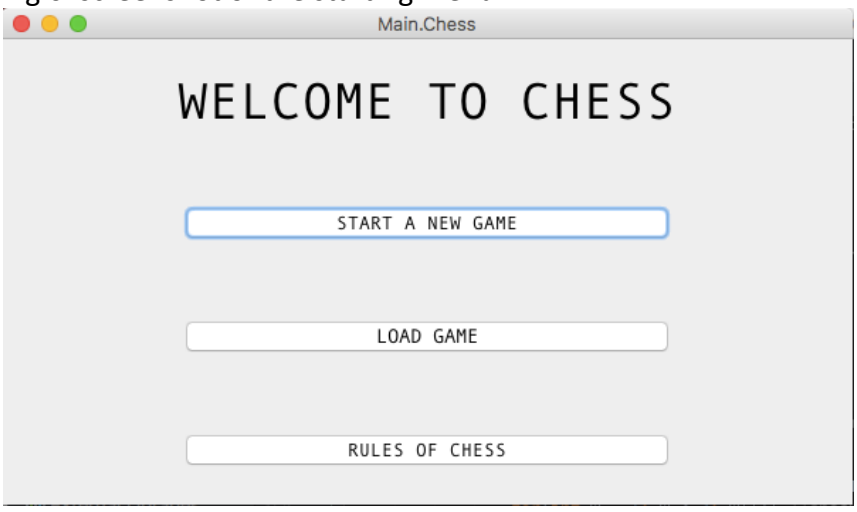


Fig 7: The playing interface

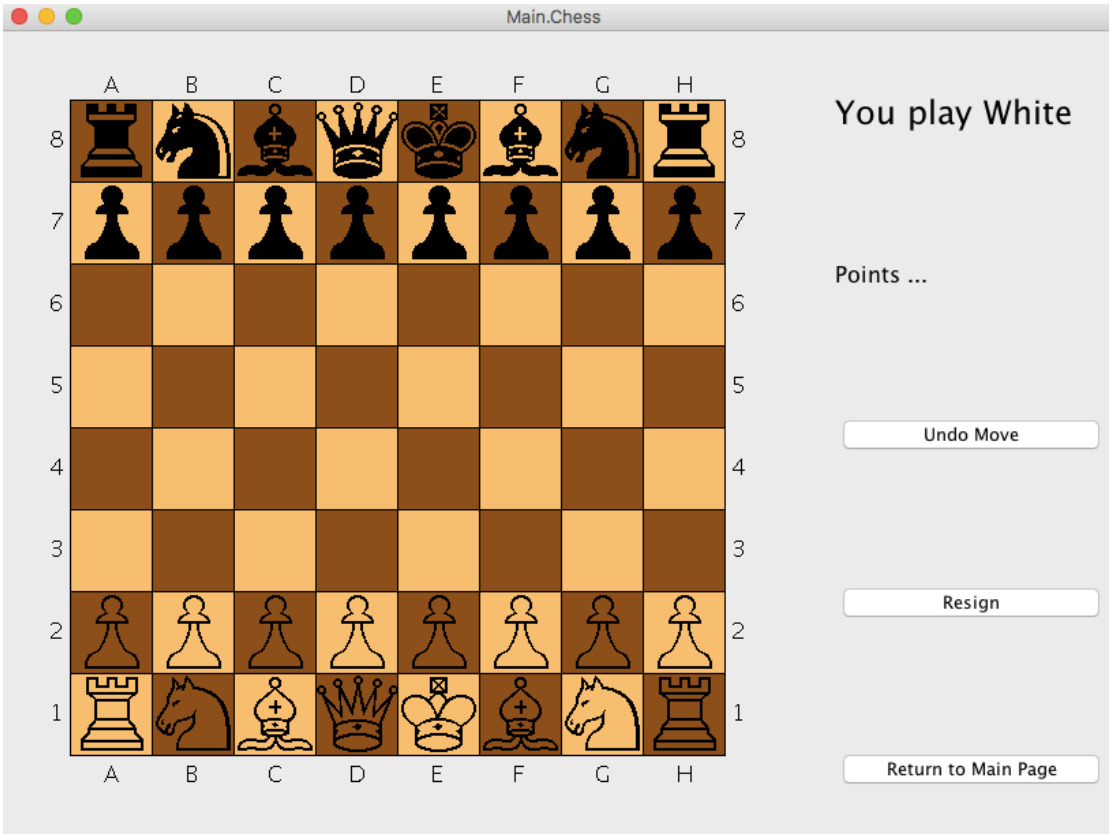


Fig 8: The playing interface mid-game

Ingenuity:

Firstly, the java graphics library allows panels to be made, which greatly facilitates and organizes the structure of the interface. This approach is preferable, as the game panel can be distinctly made and operated from the menu panel on the side. This approach also allows for easy switching between the title and game menus, which aligns with the interface requested by the user. The GUI also creates a very friendly interface for the user, as the possible moves for each piece can be visually shown. For instance, this method is superior to printing possible moves to a console, as a visual representation allows the program to have better accessibility.

Appropriateness:

Graphically showing the board, pieces and possible moves, as well as creating distinct menus was a key part to the success criteria as outlined by the client, thus making this approach very appropriate.

References:

Lewis, J. (2015). *Java Software Solutions* (8th ed.). Pearson.

Java™ Platform, Standard Edition 7 API Specification. (n.d.). Retrieved from oracle website:

https://docs.oracle.com/javase/7/docs/api/

Nested loops: (Filtering from possible moves to legal moves. (Rule Making)

Searching and analyzing the board is a key component to creating proper game logic for the pieces to follow in chess. Nested loops played an integral part in the logic (rulemaking) of the game, as it allowed in many cases each part of the board to be analyzed in multiple steps.

Here is checking for castling possibilities would look like:
Fig 9: castling possibilities

```
for (int i = 0; i < possibleMovesArr.size(); i++) {
//Checking for castling possibilities
if (p instanceof King){
    if(((King) p).getFirstMove() == true){
        int dirX1 = +1, dirX2 = -1;
        boolean pathIsClear1 = false, pathIsClear2 = false;
        for(int w = 1; w < 3; w++){ //check first direction (to the right and look for pieces blocking the path)
            if(chessboard[p.getGridX()+dirX1*w][p.getGridY()] != null){
                pathIsClear1 = false;
                break;
            } else {
                pathIsClear1 = true;
            }
        }
        for(int z = 1; z < 4; z++){ //check second direction (to the left and look for pieces blocking the path)
            if(chessboard[p.getGridX()+dirX2*z][p.getGridY()] != null){
                pathIsClear2 = false;
                break;
            } else {
                pathIsClear2 = true;
            }
        }
        if(pathIsClear1){
            //if the path is clear, then we can add the extra move to the list of possible moves for the king.
            possibleMovesArr.add(new Position((byte)(p.getGridX() +2), p.getGridY()));
        } if (pathIsClear2){
            possibleMovesArr.add(new Position((byte)(p.getGridX()-2), p.getGridY()));
        }
    }
}
```

Ingenuity:

The approach of going through each possible move a piece and then filtering out the moves that are illegal is clever, as it allows the board to evaluate the moves of only one piece at a time, without having to know of the moves of the other pieces on the board. In the instance shown above, the nested loop allows the board to check all the positions next to the king, effectively checking whether the path is clear or not.

Appropriateness:

This approach contributes to the logic of the game, and allows the board control the movement of all the pieces based on a set of rules. This method also allows each possible position to be drawn graphically, which was a requirement set by the user.

References:

Lewis, J. (2015). *Java Software Solutions* (8th ed.). Pearson.

Java™ Platform, Standard Edition 7 API Specification. (n.d.). Retrieved from oracle website:

https://docs.oracle.com/javase/7/docs/api/