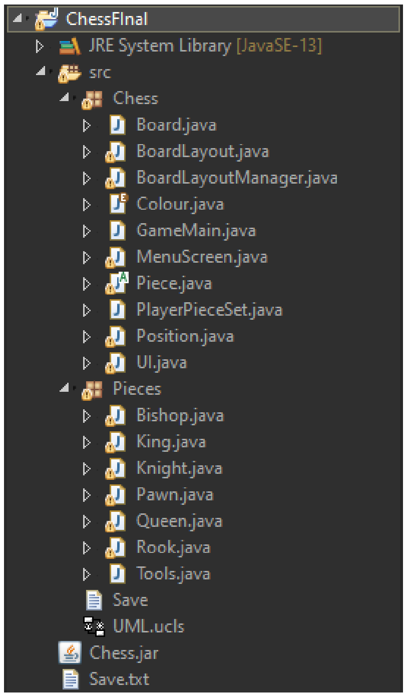
Criterion C – Development

Total word count: 1007

Contents

Source folder



```
ChessFInal
  JRE System Library [JavaSE-13]
  src
    Chess
      Board.java
      BoardLayout.java
      BoardLayoutManager.java
      Colour.java
      GameMain.java
      MenuScreen.java
      Piece.java
      PlayerPieceSet.java
      Position.java
      UI.java
    Pieces
      Bishop.java
      King.java
      Knight.java
      Pawn.java
      Queen.java
      Rook.java
      Tools.java
    Save
    UML.ucls
  Chess.jar
  Save.txt
```

<u>Chess Package</u> - Contains classes for running the program

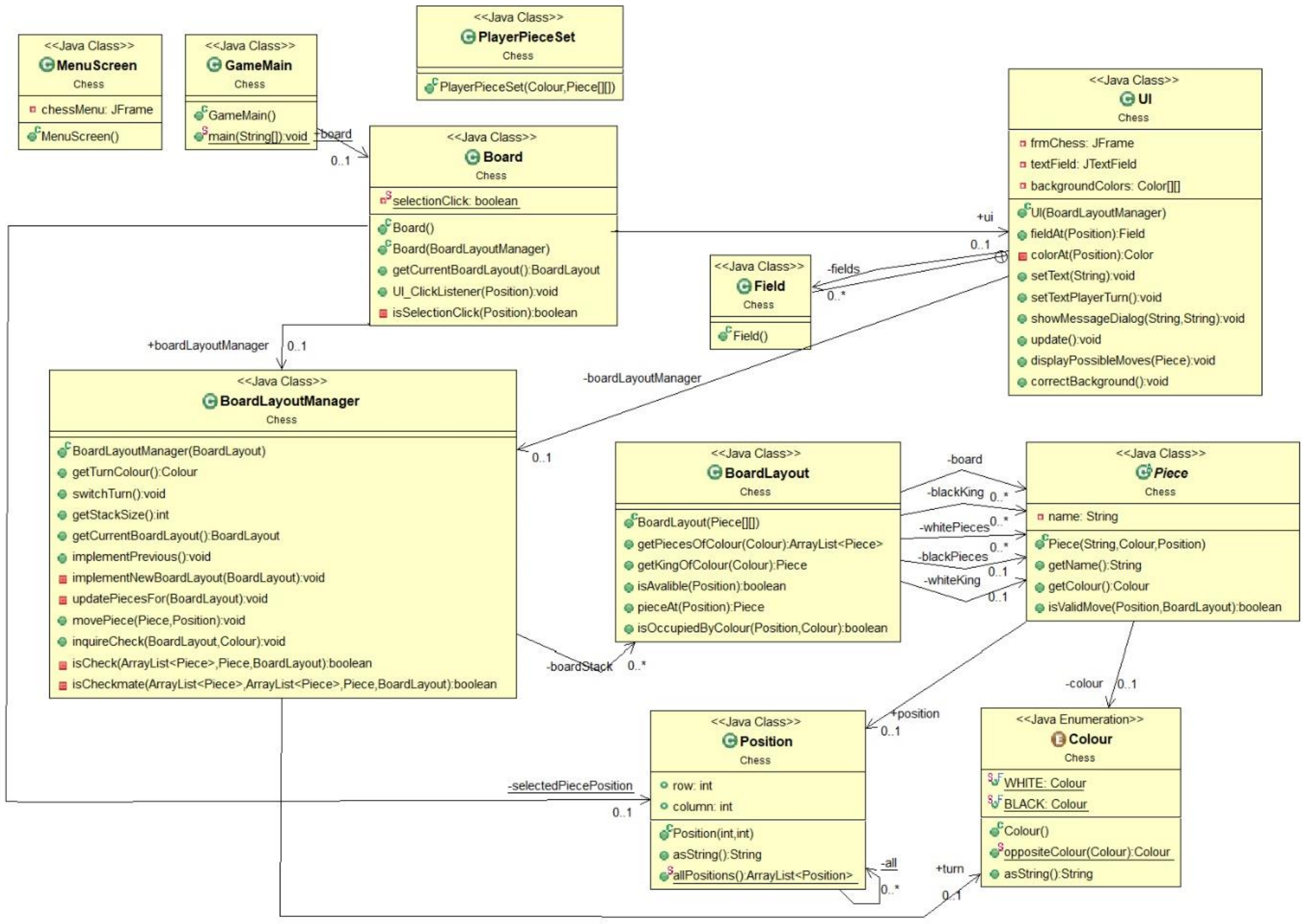
<u>Pieces Package</u> – Contains classes for each chess Piece

<u>Libraries used</u>

- Java System Lang
- Javax Swing
- Java AWT
- Java IO
- Java Util

Section word count: 30

UML class diagram



UML diagram created using Eclipse IDE

Complexities

Encapsulation

Encapsulation allows me to protect the data in a class from being altered in undesirable ways by other classes in the program. I use encapsulation in the Piece class. Some variables for each piece object, such as the name and colour, will always stay the same and never need to be changed. Therefore, I set them to be private variables and created getter methods, which only retrieve these values, negating the possibility of changing them.

```
Public abstract class Piece implements Serializable {

        private String name;
        private Colour colour;

        public Position position;

        public String getName() {
                return name;
        }

        public Colour getColour() {
                return colour;
        }
```

Inheritance

Inheritance allows me to create classes that build upon classes that already exist in the program, eliminating repetition of code. An example of its use is the Knight class, as it builds upon the original Piece class. A Knight moves in a specific way compared to other pieces, so original isValidMove() method is overridden and is programmed to assert whether a Knight Piece can move to an inputted position. Each Piece will return a different result, based on how they move in chess.

```
Public class Knight extends Piece {

        public Knight (Colour c, Position position) {

                super("Knight", c, position);
        }

        @Override
        public boolean isValidMove(Position position, BoardLayout boardLayout) {}
```

A major benefit of using inheritance is that it allows me to store and interact with different types of Pieces in the same way. When sorting Pieces on a BoardLayout (A data object that stores a particular layout of Pieces on a chess board), I can add all white Pieces into one ArrayList, no matter whether a Piece is a knight or any other piece.

```
        Private ArrayList<Piece> whitePieces;
```

Polymorphism

Polymorphism allows me program methods of the same name to do different things, based on what is inputted into the method as an argument, which makes certain methods more adaptive. I use polymorphism in the Board class, where I have two constructor methods. When this method is called without an input argument, the Board class will start the game with a new BoardLayoutManager object (An object that controls the movement of Pieces). However, when this method is called with a BoardLayoutManager as an input argument, it will start the game using it instead (Used for the BoardLayoutManager object from the save file).

```
Public class Board {

        public Board() {

                this.boardLayoutManager = new BoardLayoutManager();

                initialiseChessUI();
        }

        public Board(BoardLayoutManager BLM) {

                this.boardLayoutManager = BLM;

                initialiseChessUI();
        }
```

ArrayLists

An ArrayList is a dynamic data structure for storing variables and its size does not need to defined upon its creation, which allows for shorter and more flexible code. ArrayLists are used when creating a new BoardLayout object. The constructor reads all Pieces from a Piece array and sorts them into lists based on their colour. Since each BoardLayout will have a different number of pieces currently in play, an ArrayList is used to store them, without having to state how many there are.

```
        whitePieces = new ArrayList<Piece>();
        blackPieces = new ArrayList<Piece>();

        for (Position position : Position.allPositions()) {

                if (!isAvalible(position)) {

                        Piece piece = pieceAt(position);

                        switch(piece.getColour()) {
                        case WHITE:
                                whitePieces.add(piece);
                                if () {}

                        case BLACK:
                                blackPieces.add(piece);
                                if () {}
                        }
                }
        }
```

Stacks

A stack is an abstract data structure which incorporates a “first in, last out” system for arranging variables. This limits the of access to variables in the stack and eliminates the possibility of altering the order. A stack is the key component in my BoardLayoutManager class. New BoardLayout objects are added to the stack whenever a Piece is moved and together, they represent the history of the game. Using a stack is an appropriate solution as the order needs to stay intact and interaction is only with the BoardLayout on top, which is currently in play.

```
Public class BoardLayoutManager implements Serializable {

        private Stack<BoardLayout> boardStack;

        public BoardLayout getCurrentBoardLayout() {
                return boardStack.peek();
        }

        public void implementPrevious() {

                boardStack.pop();

                updatePiecesFor(getCurrentBoardLayout());
        }

        private void implementNewBoardLayout(BoardLayout newBoardLayout) {

                boardStack.push(newBoardLayout);

                updatePiecesFor(getCurrentBoardLayout());
        }
```

Enumerations

An enumeration is a type of data that can only be set to predefined values and allows for additional functionality with it. The Colour class in my program is an enumeration, as there are only two colour values in the game; White and Black. Although this data could be represented as a String, enumerations reduce the chance of alterations, as the values are predefined. Enumerations are also beneficial in other ways like their simple use in switch cases.

```
Public enum Colour {

        WHITE,
        BLACK;

        public Colour oppositeColour() {

                switch (this) {
                case WHITE:
                        return Colour.BLACK;
                case BLACK:
                        return Colour.WHITE;
                }
                return null;
        }
```

File input/output (IO)

Reading and writing a file allows me to process and store information for the program when it is not running, which can enable the program act differently, based on the contents of the file. File IO is used to save information about the state of the game. When the user presses the save button, the baordLayoutManager object currently in play is written to the save file. When the program is started and the user presses the load game button, the program will read the baordLayoutManager from the file and puts it into play.

```
        FileOutputStream fo = new FileOutputStream(new File(“Save.txt”));
        ObjectOutputStream oo = new ObjectOutputStream(fo);

        oo.writeObject(boardLayoutManager);

        oo.close();
        fo.close();
```

```
        FileInputStream fi = new FileInputStream(new File(“Save.txt”));
        ObjectInputStream oi = new ObjectInputStream(fi);

        BoardLayoutManager 6oardLayoutManager = (BoardLayoutManager) oi.readObject();

        oi.close();
        fi.close();
```

Error handling

Error handling allows the program to act appropriately in sections where an error could potentially occur and prevents issues such as crashes if the program encounters an error. Error handling is used when the program writes to the save file, since it is possible for the file to be missing etc. If no errors occur, the program writes to the file. However, it notifies the user if one occurs.

```
        Try {

        FileOutputStream fo = new FileOutputStream(new File(“Save.txt”));
        ObjectOutputStream oo = new ObjectOutputStream(fo);

        oo.writeObject(boardLayoutManager);

        oo.close();
        fo.close();

        showMessageDialog(“Success”,”Game has been saved.”);

        setTextPlayerTurn();

        } catch (FileNotFoundException exeption) {
                showMessageDialog(“Error”,”Save file is missing.”);
        } catch (IOException exeption) {
                showMessageDialog(“Error”,”File handling error.”);
        }
```

2D arrays

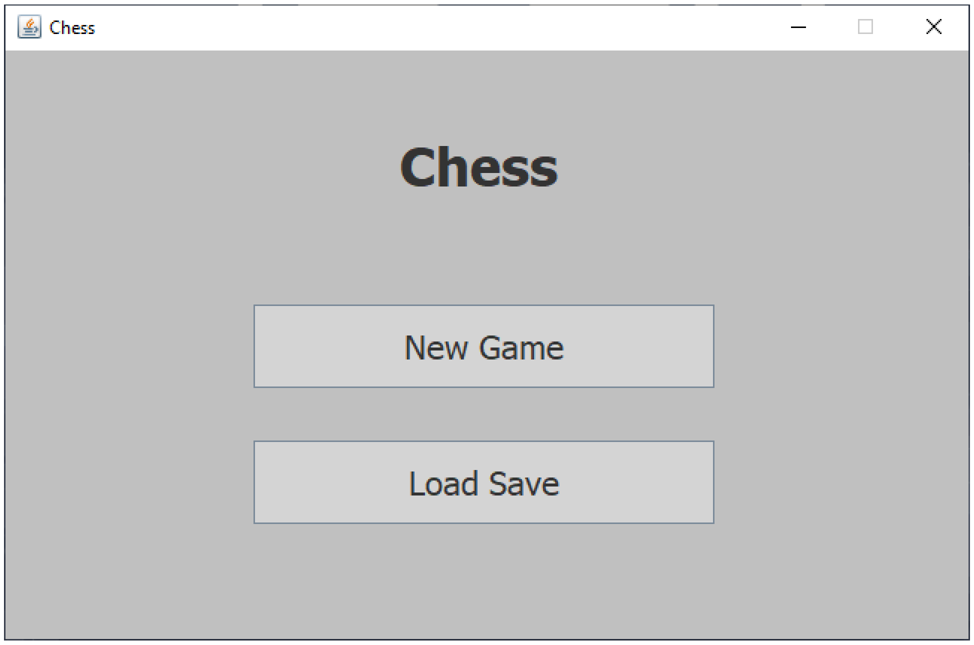
A 2D array is a static data structure similar to a standard array. However, values stored in the 2D array are accessed using two indexes. As this array includes two dimensions, it can represent as a grid and thus, also fields of a chess board. Therefore, a position (row and column) can be used as indexes for the 2D array to store Pieces and their location on the board. If a pieces is not present at a position, null is stored as the value. This is used in the PlayerPieceSet class, where a loop takes Pieces from a list and places them in a 2D array based on their position.

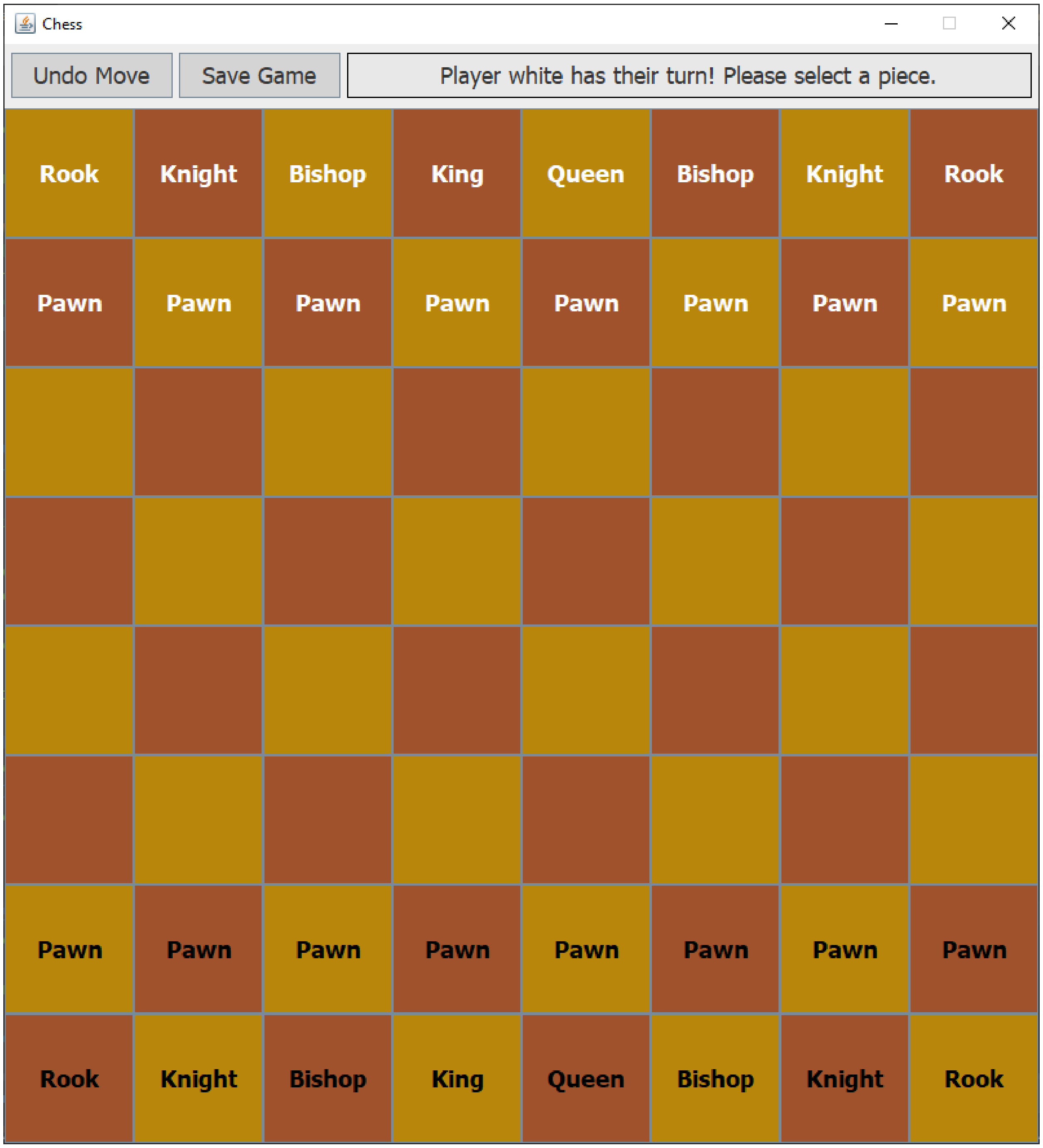
```
For (Piece piece : pieces) {

        firstBoard[piece.position.row][piece.position.column] = piece;
}
```

Graphical user interface (GUI)

A GUI acts as a tool to interact with the user and acts as a layer of abstraction which allows for easier communication between the user and the program. Communicating with the program is done through a GUI and this is necessary since my client is technologically illiterate and would struggle to interact with tools such as the console. My menu screen houses two buttons to start the game and the chess game window is a compilation of buttons that allow the user to interact with the pieces on board, as well as other buttons for undoing moves and saving the state of the game.





Player white has their turn! Please select a piece.

Rook	Knight	Bishop	King	Queen	Bishop	Knight	Rook
Pawn	Pawn	Pawn	Pawn	Pawn	Pawn	Pawn	Pawn
Pawn	Pawn	Pawn	Pawn	Pawn	Pawn	Pawn	Pawn
Rook	Knight	Bishop	King	Queen	Bishop	Knight	Rook

Section word count (not including code): 977

Sources

Publisher	Link	Benefit
CodeCademy	https://www.codecademy.com/	Helped me understand basic features and components of Java including OOP programming, Polymorphism, and Inheritance.
Geeksforgeeks	https://www.geeksforgeeks.org/design-a-chess-game/	Gave a basic idea of how to approach creating a chess game in Java. Main idea taken from this source was to use a 2D array to store pieces and their location on the board.
Youtube	https://www.youtube.com/watch?v=xaJxBsxqkyM	I learnt the importance of using an object orientated approach to making the program. I will be able include a variety of more complex features, while keeping my code simple and organized.
Youtube	https://www.youtube.com/watch?v=6VPYQ-CmDwU	Greater understanding of the use of inheritance and how to implement it in my chess program – Using an abstract Piece object as the parent class to each specific chess piece class e.g. Knight, Queen. This allows each piece to have unique functionality.
Oracle	https://docs.oracle.com/javase/tutorial/java/javaOO/enum.html	Better understanding of enumerations in Java. Showed me the additional functionality of enumerations such as their use in switch cases.
YouTube	https://www.youtube.com/watch?v=VPNs0OA4isk	Helped me install the "Window builder" plug-in the Eclipse IDE and understand how to easily build a graphical user interface.
Mkyong	https://mkyong.com/java/how-to-read-and-write-java-object-to-a-file/	Allowed me to understand how to read and write objects to a text file using the Java Serializable library
Reddit	https://www.reddit.com/r/learnpython/comments/bdfoq0/what_algorithms_could_be_used_to_identify/	The following reddit comment helped inspire my checkmate algorithm: *"If each piece checks for legality, as soon as a king is put into check have it check all available locations around it to see if any are legal. If none are, then the game is over, if there's still at least one legal move, then pass turn back to the other player to make the move."* However, the final algorithm has many additions to make it fully functional and is more complex.
StackExchange	https://codereview.stackexchange.com/questions/71790/design-a-chess-game-using-object-oriented-principles	A function in the Player.Java class on this website helped me develop my PlayerPieceSet class which a positions a new set of colored pieces in 2D array.
W3schools	https://www.w3schools.com/java/java_try_catch.asp#:~:text=Java%20try%20and%20catch,occurs%20in%20the%20try%20block.	Showed me the implementation of error handling in Java. The use of Try-catch blocks and how they function.
JavaTpoint	https://www.javatpoint.com/java-joptionpane	Assisted me in implementing dialog boxes in the program.