Software Engineering GP02 Project
Design Specification

Authors:	Jack Book [jab153], Thomas Roethenbaugh [tpr3], Will Abbott [wia14], Michael Stamp [mjs36], Micah Barendse [mib60], Lance Sebastian [lvs1]
Config Ref:	SE.G02.DesignSpec
Date:	5th May 2023
Version:	1.2.0
Status:	Release

Department of Computer Science
Aberystwyth University
Aberystwyth
Ceredigion
SY23 3DB

CONTENTS

1. INTRODUCTION

1.1 Purpose of this Document

The purpose of this document is to give insight into how the game will be designed. It will describe the decomposition of the programs and the modules that fall within them, including any significant classes and how the classes overlap with the functional requirements. It will also describe the relationships and dependencies the modules have between each other. Additionally, it will provide a description of the interfaces used and how programmers or testers will use the facilities provided by a module. Finally, it will lay out the internal details of any modules that are not obvious to an external reader [1].

1.2 Scope

The document should be read by all members of the project. Readers should have a good understanding of the User Interface Specification document [3].

1.3 Objectives

The objective of this document is to allow the reader to understand the inner mechanisms of the classes used in the project and go into further detail of how more complicated classes work.

2. DECOMPOSITION DESCRIPTION

2.1 Programs in the System

The system as designed only contains one program and is run as a JVM process. This singular program handles both the logic and graphics of the game in several packages and classes. The backend logic relies on multiple packages, notably the game, moves, and pieces packages. These packages contain Java classes which are used as objects by the final program. The frontend relies on the GUI package, the classes utilizing JavaFX. The saving and loading functionalities are isolated in the gameSave package. Further, all JUnit classes are contained in the tests package.

2.2 Significant Classes

The program is broken down into separate packages, each package containing relevant classes and materials needed to run certain aspects of the program. Each of the sections below will describe the packages, and briefly explain the classes used.

2.2.1 game

The game package contains four classes – Game, Board, Square, and Player, these are used to build the game and holds the majority of the backend structure.

Game - collects the relevant information from other classes in the package and holds the properties of the current game in progress.

Board – stores the chessboard, which is an array of Squares, and contains methods directly related to the overall chessboard, such as creating the board at the start of the game.

Square - is responsible for looking after each cell on the chessboard and what those cells contain.

Player - collects the information regarding each player, e.g., name, color, and captured pieces.

2.2.2 pieces

The pieces package has seven classes – Piece, Bishop, King, Knight, Pawn, Queen, and Rook. These classes contain the material for each specific piece. The individual classes representing each piece work in similar ways,

apart from the valid moves method that determines what each valid move for the pieces are and allows the movement to take place. Piece is an abstract class that is inherited by the others.

2.2.3 moves

The moves package contains five classes – MakeMove, CheckChecker, Castle, EnPassant, and Promote. Castle, EnPassant, and Promote classes are considered 'special moves' that are not called as often as the other classes; as such, they are in an internal package.

MakeMove – This class holds the methods for moving a piece on the chessboard. It contains methods that are specifically used for making the moves that involve EnPassant and Castling.

CheckChecker – This class is responsible for checking whether either of the King's is currently in check or checkmate.

Castle – This class is for checking if either the Black King or White King can Castle.

EnPassant – This class holds the method to get the valid moves for enPassant.

Promote – This class promotes a Pawn to the specified piece type.

2.2.4 GUI

This package includes all the frontend classes, as well as all images required by the program, such as the menu background and all the individual pieces.

GameGUI – This is the class that runs the main graphical user interface for the game. This class will show the game in progress and display the game in the correct manner, for the game to run correctly, in accordance with the users' inputs. It includes 2 methods. The first initializes the main menu, displaying the text boxes for the users' names, the selection box for assigning the user colors, and any buttons that are needed in accordance to the User Interface Design Specification.[2]

BoardGUI -

2.2.5 util

The classes within this package provide simple utilizes to other package classes. For this program, these classes define the colors and pieces available within the chess game.

Color – This class simply defines the two available colors within chess: Black and White.

Type – This class defines the pieces available within chess. These pieces are King, Queen, Rook, Bishop, Knight and Pawn.

2.2.6 gamesave

There is one class in this package – GameSaveManager. This class contains the functions to save and load ongoing games and replays.

GameSaveManager – Stores the functions for loading and saving replays and ongoing games

FilePicker – A class for opening a directory picker and returning a file.

2.3 Requirements to Classes Mapping

Functional Requirements	Classes providing requirement
FR1	Board, Game, Player, Square
FR2	Player, Game
FR3	Board
FR4	makeMove, Bishop, King, Knight, Pawn, Piece, Queen, Rook

FR5	makeMove, Bishop, King, Knight, Pawn, Piece, Queen, Rook, Castle, enPassant, CheckChecker
FR6	CheckChecker
FR7	CheckChecker
FR8	GameGui, BoardGUI
FR9	GameGui, BoardGUI
FR10	GameGui, GameSaveManager, BoardGUI
FR11	GameGui, GameSaveManager, BoardGUI

3. DEPENDENCY DESCRIPTION

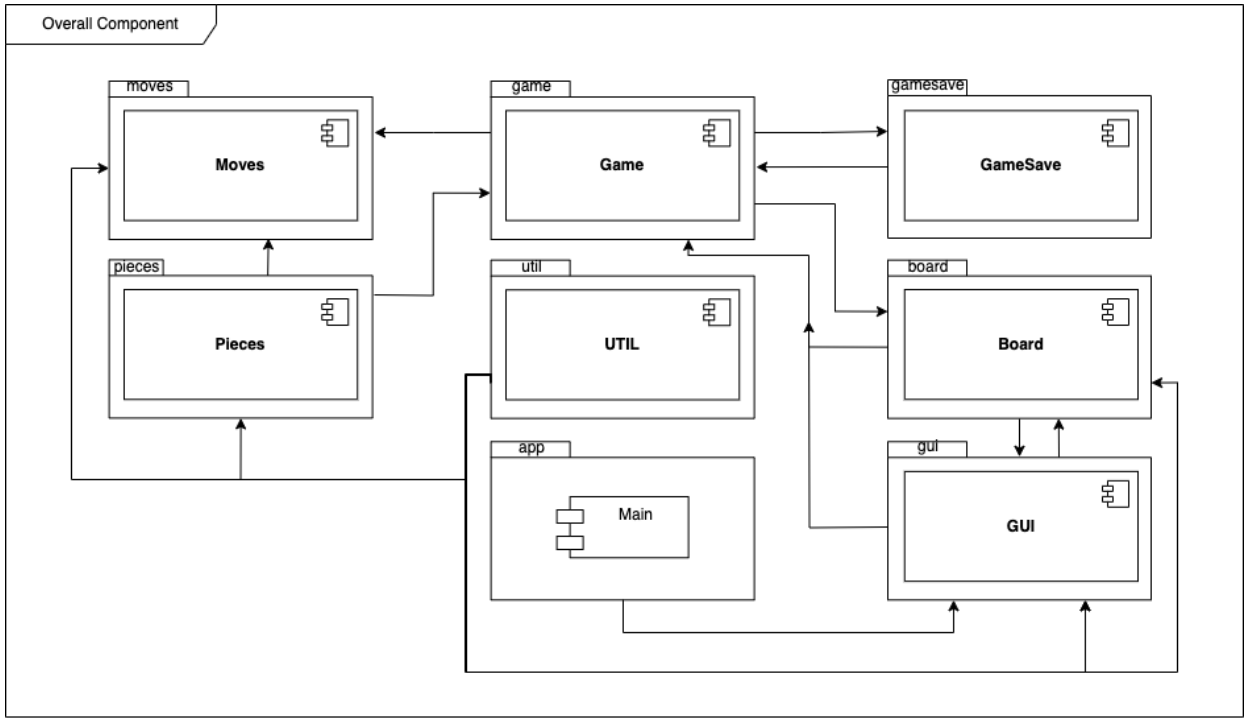

Figure 1 - Overall Component Diagram

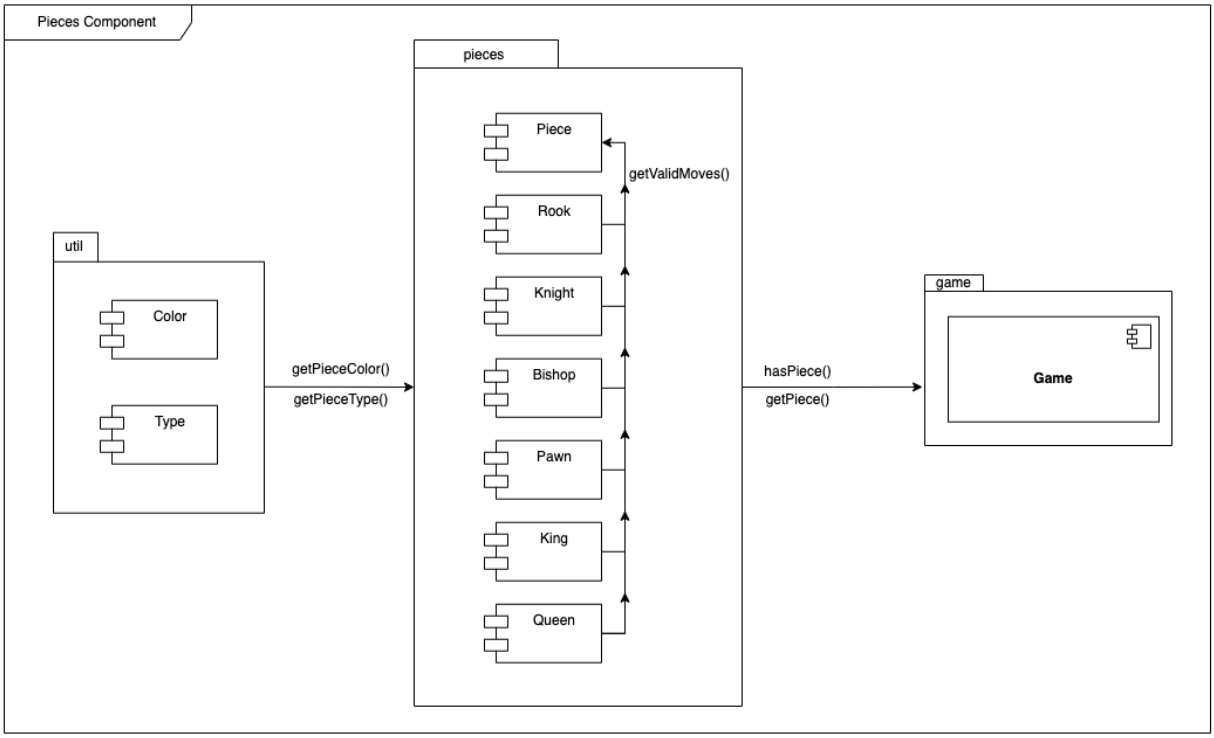

Figure 2 - Pieces Component Diagram

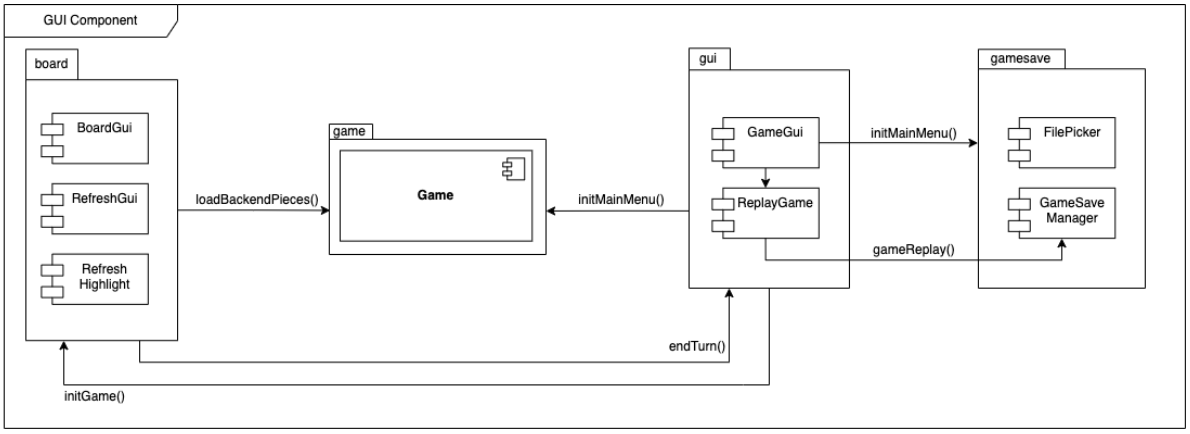

Figure 3 - GUI Component Diagram

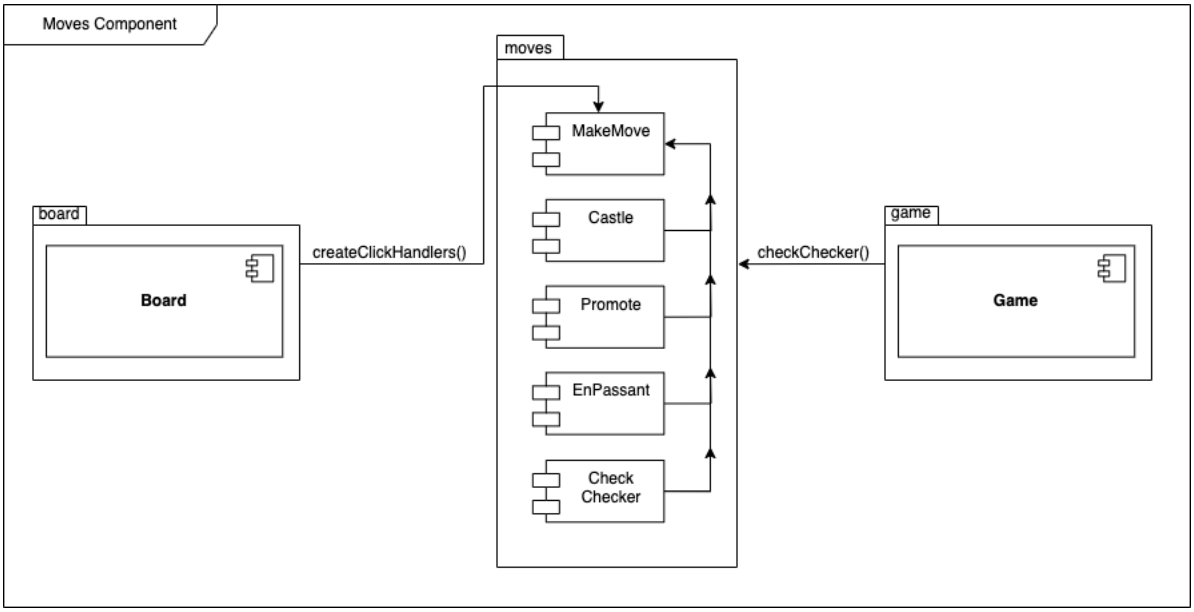

Figure 4 - Moves Component Diagram

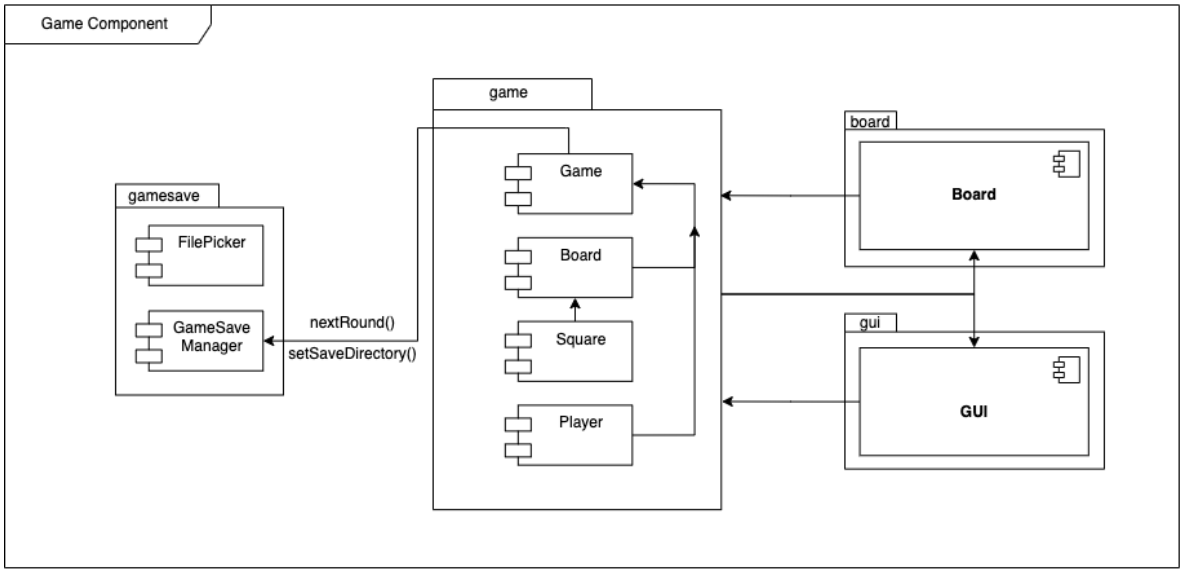

Figure 5 - Game Component Diagram

4. INTERFACE DESCRIPTION

4.1 main Package

- Type: Public
- Extends: Application
- Public Methods

Type	Name	Parameters	Description
Void	start()	primaryStage; Stage	
Void	main()	args; String[]	

4.2 game Package

4.2.1 Game

- Type: Public
- Extends: Nothing
- Public Methods

Type	Name	Parameters	Description
	Game()		Constructor. Creates a board object, sets the turn to White, and sets the roundCount to 1.
Board	getBoard()		Returns the board field.
void	setWhitePlayer()	name; String	Sets the whitePlayer field to a new player object with the passed String as the name.
void	setBlackPlayer()	name; String	Sets the blackPlayer field to a new player object with the passed String as the name.
Player	getWhitePlayer()		Returns the whitePlayer field.
Player	getBlackPlayer()		Returns the blackPlayer field.
Color	getTurn()		Returns the turn field.
void	setTurn()	turn; Color	Sets the turn field to the passed Color.
int	getRoundCount()		Returns the roundCount field.
void	nextRound()		Increments the roundCount field.
String	getSaveDirectory()		Returns the saveDirectory field.
String	setSaveDirectory()	name; String	Sets the game save directory

4.2.2 Board

- Type: Public
- Extends: Nothing
- Public Methods

Type	Name	Parameters	Description

	Board()		Constructor. Creates a new 2D Square object at 8 x 8. Sets the enPassantPiece field to {-1,-1} and then calls the boardReset method.
Square[][]	getBoardArray()		Returns the boardArray field.
int[]	getEnPassantPiece()		Returns the enPassantPiece field.
void	setEnPassantPiece()	enPassantPiece; int[]	Sets the enPassantPiece field as the passed int array.
void	boardReset()		Resets the boardArray field to the starting configuration of pieces.
void	clearBoard()		Clears the boardArray field so new pieces are on the board.
void	printBoard()		Used for debugging, prints the boardArray field to the console.
void	setSquare()	square; Square row; int col; int	
Board	clone()		

4.2.3 Square

- Type: Public
- Extends: Nothing
- Public Methods

Type	Name	Parameters	Description
	Square()		Constructor. Used for Square's that do not contain a Piece. Sets hasPiece field to false.
	Square()	piece; Piece	Constructor. Used for Square's that do contain a Piece. Sets hasPiece field to true.
Piece	getPiece()		Returns the piece field.
boolean	hasPiece()		Returns the hasPiece field.

4.2.4 Player

- Type: Public
- Extends: Nothing
- Public Methods

Type	Name	Parameters	Description
	Player()		
	Player()	color; Color,	Constructor. Sets color and name field as passed values.

		name; String	Initializes takenPieces field as an ArrayList.
Color	getColor()		Returns the color field.
String	getName()		Returns the name field.
List<Piece>	getTakenPieces()		Returns the takenPieces field
void	addTakenPieces()	piece; Piece	Adds passed Piece to takenPieces field.

4.3 pieces Package

4.3.1 Piece

- Type: Public
- Extends: Nothing
- Public Methods

Type	Name	Parameters	Description
	Piece()		
	Piece()	color; Color	Constructor. Sets passed color as color field and sets hasMoved field to false.
Color	getPieceColor()		Returns the color field.
Color	getEnemyPieceColor()		Using color field, returns opposing sides color.
Type	getPieceType()		Returns the ID field.
boolean	hasMoved()		Returns the hasMoved field.
void	setHasMoved()		Sets the hasMoved field to true.
List<int[]>	getValidMoves()	board; Board, piece; Piece, row; int, col; int	Abstract method to be redefined by inherited classes. Returns a List<int[]> of valid move coordinates.
boolean	addValidMove()	boardArray; Square[][], piece; Piece, row; int, col; int	A method to check if a specific coordinate is valid, if so, it adds it to the returnArray field.
	getPossibleMoves()		

4.3.2 Bishop

- Type: Public
- Extends: Piece
- Public Methods

Type	Name	Parameters	Description

	Bishop()	color; Color	Constructor. Uses the Piece constructor and sets ID field to Bishop type.
List<int[]>	getValidMoves()	board; Board, piece; Piece, row; int, col; int	Overrides inherited method with commands to retrieve correct coordinates for the Bishop piece.

4.3.3 King

- Type: Public
- Extends: Piece
- Public Methods

Type	Name	Parameters	Description
	King()	color; Color	Constructor. Uses the Piece constructor and sets ID field to King type.
List<int[]>	getValidMoves()	board; Board, piece; Piece, row; int, col; int	Overrides inherited method with commands to retrieve correct coordinates for the King piece.

4.3.4 Knight

- Type: Public
- Extends: Piece
- Public Methods

Type	Name	Parameters	Description
	Knight()	color; Color	Constructor. Uses the Piece constructor and sets ID field to Knight type.
List<int[]>	getValidMoves()	board; Board, piece; Piece, row; int, col; int	Overrides inherited method with commands to retrieve correct coordinates for the Knight piece.

4.3.5 Pawn

- Type: Public
- Extends: Piece
- Public Methods

Type	Name	Parameters	Description

	Pawn()	color; Color	Constructor. Uses the Piece constructor and sets ID field to Pawn type.
List<int[]>	getValidMoves()	board; Board, piece; Piece, row; int, col; int	Overrides inherited method with commands to retrieve correct coordinates for the Pawn piece.

4.3.6 Queen

- Type: Public
- Extends: Piece
- Public Methods

Type	Name	Parameters	Description
	Queen()	color; Color	Constructor. Uses the Piece constructor and sets ID field to Queen type.
List<int[]>	getValidMoves()	board; Board, piece; Piece, row; int, col; int	Overrides inherited method with commands to retrieve correct coordinates for the Queen piece.

4.3.7 Rook

- Type: Public
- Extends: Piece
- Public Methods

Type	Name	Parameters	Description
	Rook()	color; Color	Constructor. Uses the Piece constructor and sets ID field to Rook type.
List<int[]>	getValidMoves()	board; Board, piece; Piece, row; int, col; int	Overrides inherited method with commands to retrieve correct coordinates for the Rook piece.

4.4 moves Package

4.4.1 MakeMove

- Type: Public
- Extends: Nothing
- Public Methods

Type	Name	Parameters	Description

void	movePiece()	board; Board, player; Player, pieceRow; int, pieceCol; int, targetRow; int, targetCol; int	A method that moves a piece on the board. This method is only for standard move.
void	movePiece()	board; Board, player; Player, pieceRow; int, pieceCol; int, targetRow; int[], targetCol; int[]	A method designed to move a Pawn if a enPassant move is selected.
void	movePiece()	board; Board, player; Player, pieceRow; int[], pieceCol; int[], targetRow; int[], targetCol; int[]	A method designed to move a King and Rook if a Casting move is selected.
void	clearEnPassantPiece()	board; Board	A method to call the Board class method setEnPassantPiece passing in {-1,-1}, thus resetting the field.
void	addTakenPiece()	player; Player, piece; Piece	A method to add the piece to the players takenPieces list, using the Player class method addTakenPieces.

4.4.2 CheckChecker

- Type: Public
- Extends: Nothing
- Public Methods

Type	**Name**	**Parameters**	**Description**
List<int[]>	checkChecker()	board; Board, row; int, col; int attackingColor : Color	A method that returns a list of coordinates of pieces that are attacking the square. Returns empty if the square is not in check.
boolean	checkMateChecker()	board; Board, row; int, col; int	A method that returns a True or False depending on whether the King is in Check Mate.

void	removeMovesThatEndangerKing()	board; Board piece; Piece row; int col; int returnArray; List<int[]>	A function to remove any moves from a list of moves that endanger the friendly king.
void	inDangerChecker()	board; Board row; int col; int attackingColor; Color returnArray: List<int[]>	Function to remove possible king moves if they place him in check – this is used when a piece is moved to avoid recursion between checkChecker() and removeMovesThatEndangerKing()

4.4.3 Castle

- Type: Public
- Extends: Nothing
- Public Methods

Type	Name	Parameters	Description
List<int[]>	canCastle()	board; Board, piece; Piece, row; int, col; int	A method to check if a King can castle with a Rook. If so, it returns a List<int[]> containing the valid coordinates. If not, it returns the List empty.
int[]	checkKingSide()	board; Board, piece; Piece, row; int, col; int start; int, end; int, direction; int	A method that checks for a Rook on the specified side of the King. This method is exclusively called by canCastle.

4.4.4 EnPassant

- Type: Public
- Extends: Nothing
- Public Methods

Type	Name	Parameters	Description
List<List<int[]>>	getMoves()	board; Board, piece; Piece, row; int,	A method to return coordinates to be used when making an EnPassant move. It returns two Lists contained in a single list. The first List

		col; int	contains the coordinates the Pawn will move to. The second List contains the coordinates of the Pawn to be taken.

4.4.5 Promote

- Type: Public
- Extends: Nothing
- Public Methods

Type	Name	Parameters	Description
void	promotePawn()	board; Board, row; int, col; int, newPiece; Type	A method to promote a Pawn to a specified piece type.

4.5 GUI Package

4.5.1 GameGui

- Type: Public
- Extends: Nothing
- Public Methods

Type	Name	Parameters	Description
	GameGui()	stage; Stage	Constructor.
Void	initMainMenu()	primaryStage; Stage	A method to initiate the main menu as well as the Start Game Menu.
Void	players()	name1; TextField, name2; TextField, colorSelectWhite; RadioButton, colorSelectBlack; RadioButton, start; Button	Method for setting functionality for the user input in the startGameMenu.
Void	initGame()	primaryStage; Stage	Method to initiate the Game screen.
Void	requestDraw()		Method to offer the player a draw

Void	resign()		Method for player to decide to resign.
Void	winByResignation()	primaryStage; Stage	Method for player to win by resignation.
Void	backToMain()		Method to bring you back to the main menu.
Void	gameWinner()		Method to bring you back to the main menu/save game at checkmate.
BorderPane	getBorderGame()		A getter method to return the boarder game variable.
VBox	getCheckmate()		A getter method to return the checkmate variable.

4.5.2 ReplayGame

- Type: Public
- Extends: Nothing
- Public Methods

Type	**Name**	**Parameters**	**Description**
Void	gameReplay()	gameGUI; GameGUI, replay; BorderPane, primaryStage; Stage, replayGame; Game, saveDirectory, String	Method to initiate the replay game function and bring up the replay screen.

4.5.3 BoardGUI

- Type: Public
- Extends: Nothing
- Public Methods

Type	Name	Parameters	Description
	BoardGUI		Constructor.
GridPane	createBoard	newGame; Game	A method to create the board.
Void	loadBackendPieces()		A method to read the backend board, and show the pieces on the GridPane.
Void	createImage	Filename; String, Col; int, Row; int	A method to create an image and place it on the GridPane.
Void	endTurn()		A method to end the users turn.
String	whoseTurnName		A method to get the name of the current player.
GridPane	getBoard()		A method to return the GridPane of the board.
String	whoseTurnNotName		A method to get the name of the opposing player.

4.5.4 RefreshGUI

- Type: Public
- Extends: Nothing
- Public Methods

Type	Name	Parameters	Description
	RefreshGUI		Constructor.
Void	refreshBoard()		A method to refresh the board to show updated pieces.
GridPane	createCapturedBlackPiecesPane()		A method for creating a grid to store black's captured pieces.
GridPane	createCapturedWhitePiecesPane()		A method for creating a grid to store white's captured pieces.
Void	removeCapturedPieces()	capturedPieces; GridPane	A method for removing captured pieces from both GridPanes.

Void	addCapturedPieces()	capturedPieces; GridPane, color; Color	A method to add captured pieces to the captured pieces GridPane.
Void	updateCapturePiecesPane()		A method to update the CapturedPiece GridPane.
HBox	whoseTurnNameBox()		A method to create the whoseTuenNameBox.
Void	updateTurnName()		A method to update the whoseTurnNameBox.
HBox	addPromotionBox()		A method to create the promotion box.
Void	removePromotionPiece()		A method to remove the promotion pieces after a Pawn has been removed.
Void	updatePromotionBox()	targetCoordinate; int[]	A method to update the promotion box.

4.5.5 RefreshHighlight

- Type: Public
- Extends: Nothing
- Public Methods

Type	Name	Parameters	Description
	RefreshHighlight		Constructor.
Void	highlightMoves()		A method to highlight the valid moves of the selected piece.
Void	resetHighlight()		A method to reset (remove) the highlights currently on the board.
Void	isKingInCheck()		A method to show the king is in check.
Boolean	isKingCheckMate		A method to show King is in CheckMate.

4.6 gamesave Package

4.6.1 GameSaveManager

- Type: Public
- Extends: Nothing
- Public Methods

Type	Name	Parameters	Description
Game	loadReplayRound()	directory; String, round; int	Loads a specific round of a saved replay.
Game	loadOngoingGame()	directory; String	Loads an ongoing game.
void	saveOngoingGame()	game; Game	Saves an ongoing game. This should be called at the end of every turn. Requires Game.saveDirectory to be set.
void	saveReplay()	game; Game	Saves a replay.
int	getNumRoundsInReplay()	directory; String	Gets the number of rounds in a saved replay directory
boolean	saveReplay()	game: Game	Saves a replay
boolean	moveTempFiles()	directory: String	Moves save files from the temp file directory to a new directory

4.7 util Package

4.7.1 Color

- Type: Public
- Extends: Nothing
- Constants

Name
WHITE
BLACK

4.7.2 Type

- Type: Public
- Extends: Nothing
- Constants

Name
KING
QUEEN
ROOK
BISHOP
KNIGHT
PAWN

5. DETAILED DESIGN

5.1 UML Sequence Diagrams

5.1.1 Main Menu UML (UC01, 02, 04, 05)

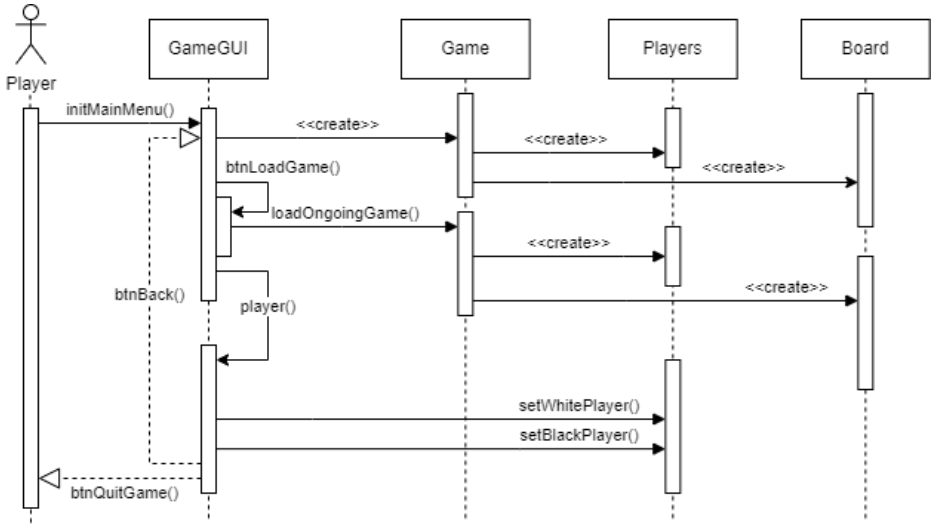

Figure 6 - Main Menu UML

5.1.2 Replay UML (UC03)

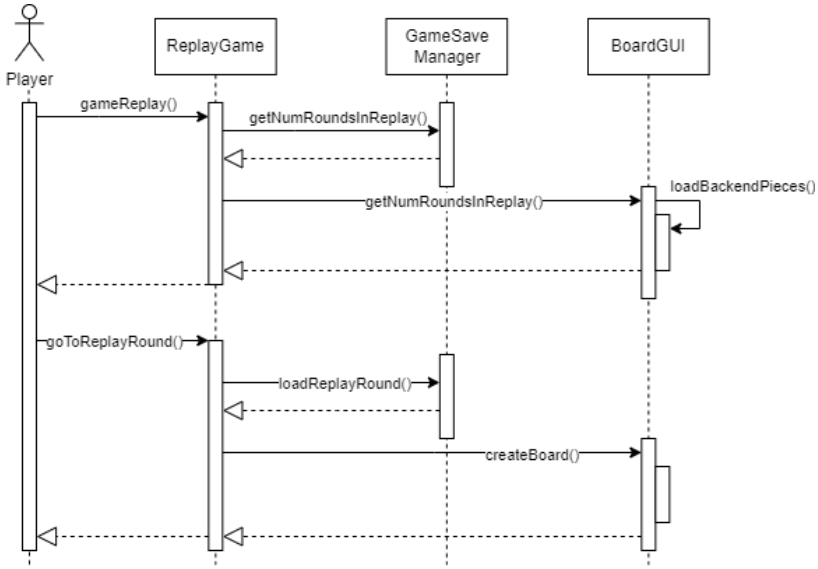

Figure 7 - Replay UML

5.1.3 Move Pieces (UC06, 07, 08, 12,13)

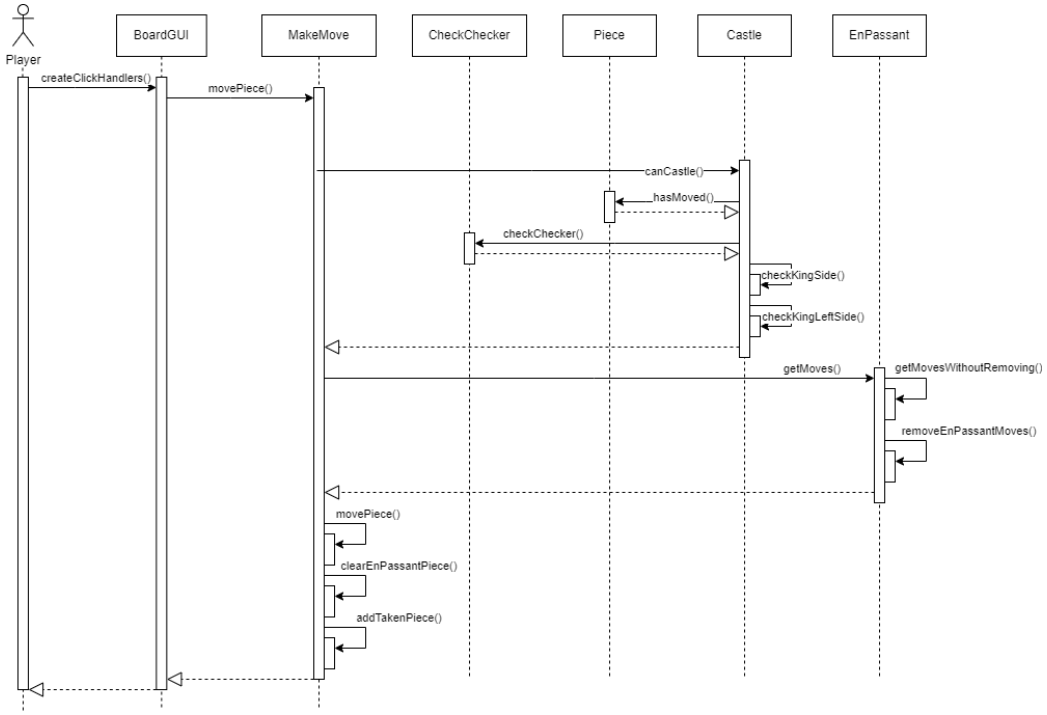

Figure 8 - Move Pieces

5.1.4 Special Moves(UC06, 09, 12, 13)

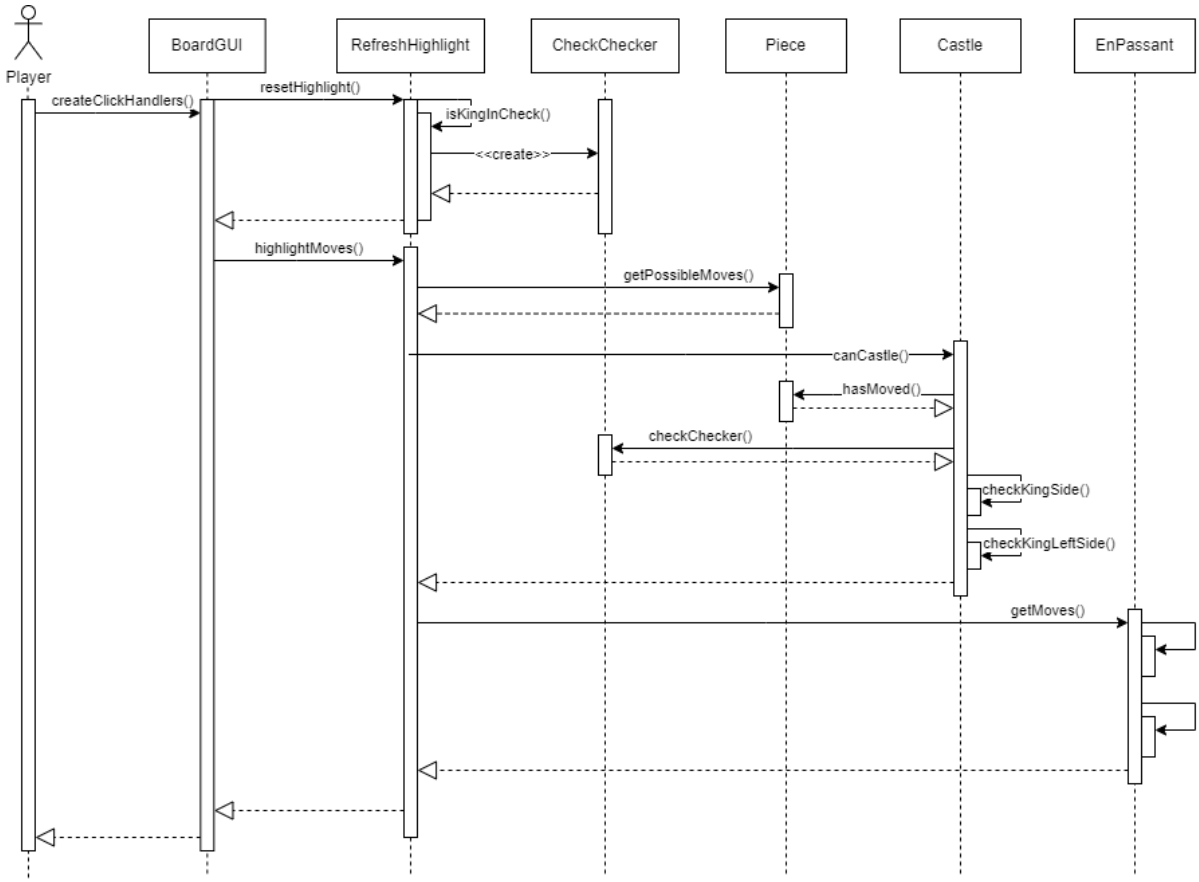

Figure 8 – Highlight and special moves

5.1.5 Check & Checkmate (UC14, 15)

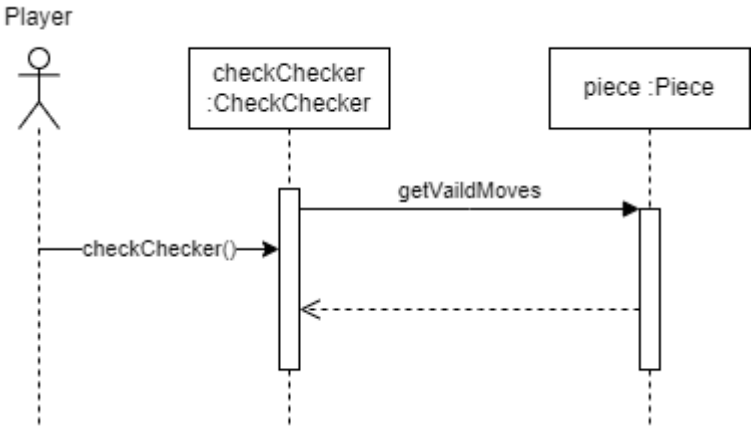

Figure 9 - Check and Checkmate

5.1.6 End game (UC18, 19)

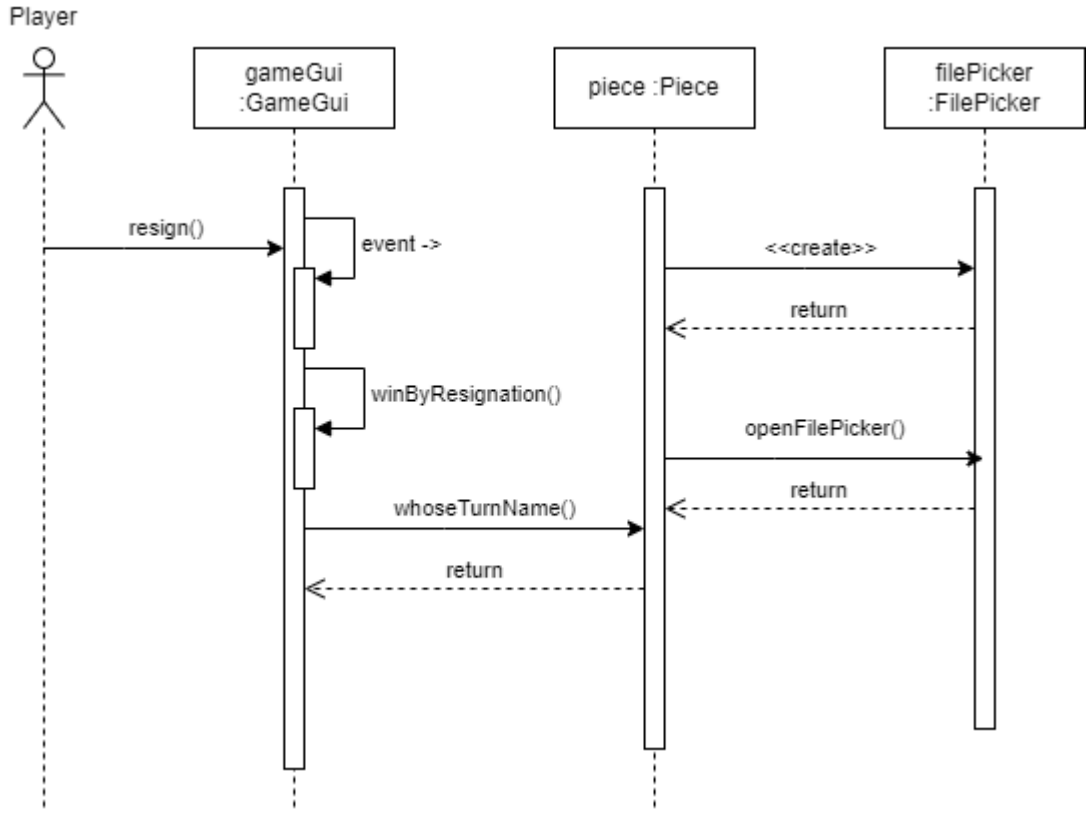

Figure 10 - Eng Game

5.2 UML Class Diagram

This diagram shows how the classes within the Chess Tutor game link between each other, to make the game run correctly.

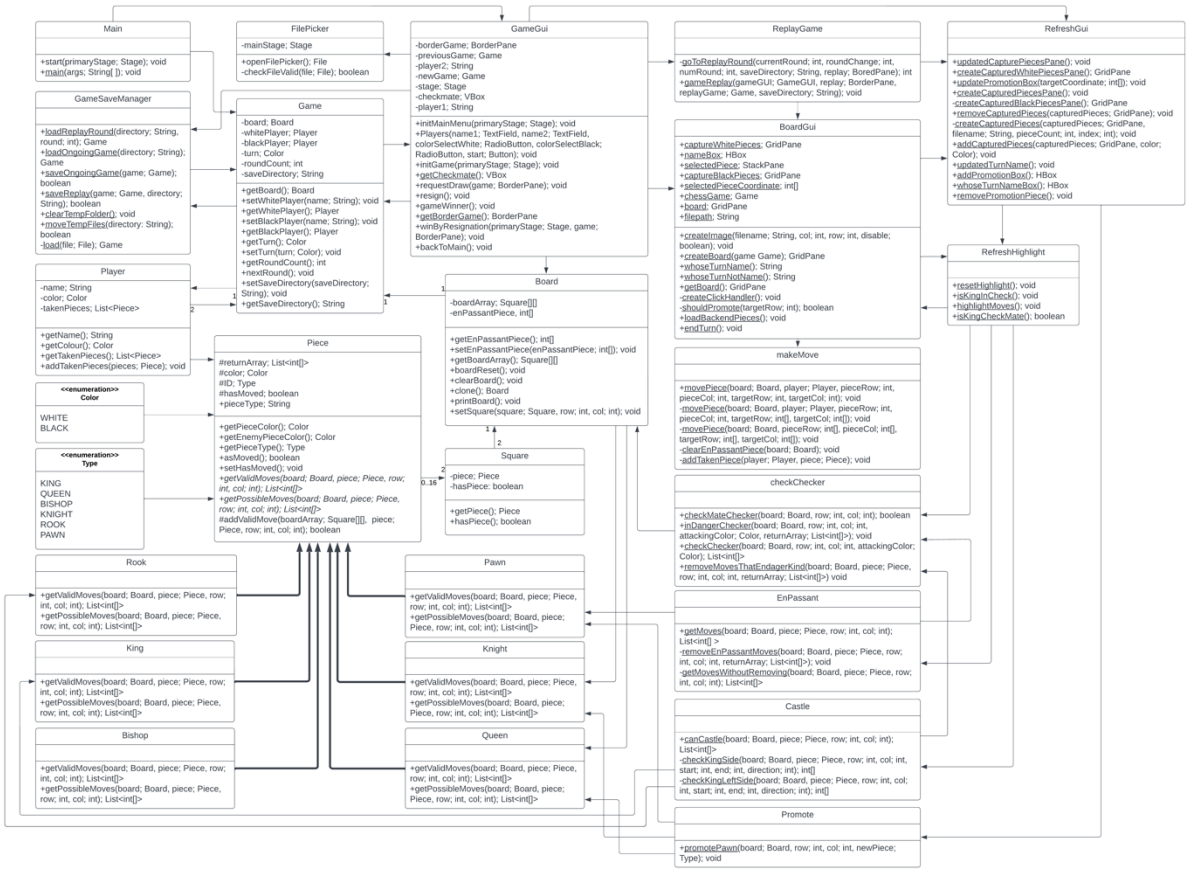

Figure 11 - UML Class Diagram

5.3 Significant Algorithms

5.3.1 Move a Piece

To move a piece on the chessboard the user would first click a piece. It would then display all the valid moves on the screen. This is done by calling the getPossibleMoves method. This returns coordinates which the GUI would then represent to the user. If the piece is a King or Pawn, it also calls the Castling method and enPassant method respectively. It would then proceed to display these on the GUI. The user can then decide which move they wish to take. Or if they do not want to take any of those moves, click on another piece and repeat the process.

Once they have decided to move a piece, the coordinates are passed through to the move method. It moves a piece by setting the target square to the value of the original square. It then resets the original square to represent an empty square. If the target square was an enemy piece, it is added to the players taken piece list. The move method has a check to ensure if the move is a Castling or enPassant move, it is handled correctly.

5.3.2 Castling

The Castling method is separated from the getPossibleMoves method. Therefore, it has to be specifically called by the program when a King piece is selected.

To Castle a King a user would select their King on the chessboard. It would then run the standard getPossibleMoves method and the canCastle method. The canCastle method checks first that the King has not moved in the game yet, if so, it checks for a Rook on either side of the King. If there is a piece blocking the King and the Rook, or there is no Rook found, the method returns an empty list. If an unblocked Rook is discovered, the method returns a List with several coordinates; the position the King will move to, the position of the Rook, the position the Rook will move to. The Castling moves are then disaplyed on the GUI with gold pieces of the King and Rook showing where they will move to if the user accepts the Castling move. If the user

accepts the Castling move, then the makeMove method is called. The makeMove method will determine if the move is a standard move or Castling move, then call the correct overloaded method. This method works in a similar way to the standard makeMove method, clearing the previously occupied squares.

5.3.3 enPassant

The enPassant method is separated from the getPossibleMoves method. Therefore, it has to be specifically called by the program when a Pawn piece is selected.

The enPassant method has two key components. First, if a Pawn has moved two spaces in its first move, it's coordinates are set to a field variable which will hold these coordinates for one turn, resetting after the opponent has moved. The second component does three things, it first checks if there is a piece parallel to it and that piece is a Pawn. It then checks if the square behind that Pawn is available to move to. It then checks if that Pawn's coordinates are the coordinates currently saved in the field variable.

If all these conditions are met, the move is displayed on the GUI. A gold Pawn is shown where the users Pawn will move to. A gold circle is displayed over the opponent's Pawn indicating it will be taken.

If the user accepts the enPassant move, the coordinates are passed to the makeMove method. The makeMove method will determine if the move was a standard move or an enPassant move and make the move correctly.

5.3.4 Check Checker

To detect whether a square is in Check a function has been made called checkChecker. This function takes a Board and two coordinates as arguments. The board is the main board of the game instance, and the coordinates are the row and column coordinates of the square. The function returns a list of coordinates of pieces that are attacking the square, or an empty list if the square is not under attack.

The function gets the piece at the row and column coordinates, and iterates through each enemy piece, gets the possible moves for that piece, and if any of the possible moves lands the enemy piece on the square the function is checking, the coordinates of the enemy piece are added to a list which is then returned by the function once all the pieces have been iterated over.

5.3.5 Checkmate Checker

To detect whether a king is in Checkmate a function has been made called checkMateChecker. This function takes a Board and two coordinates as arguments. The board is the main board, and the coordinates are the row and col coordinates of the king. The function will return true if the king is in checkmate and false otherwise.

The function first gets the attacking color and the piece on the square that the function is checking. If the piece is not a King, then the function returns false. The function then gets a list of pieces attacking the King and creates a board to simulate the moves on. If the list is empty the function returns false as it cannot be checkmate.

If there is more than one attacker the king is in double check, and must move to escape checkmate, the function iterates over the king's moves and if any of them put him out of check the function returns false as the king is not in checkmate, otherwise if none of them put him out of check it returns true.

If there is one attacker the king is in check and not double check, in which case the function iterates through the king's possible moves and simulates them on the board, if the move gets the king out of check the function returns false, otherwise the simulation board is reset for the next move.

At this point none of the King's moves can get him out of check, so the only way to escape checkmate is for a friendly piece to capture or block the attacker. The function iterates over each friendly piece and simulates a turn for each of the piece's moves. If the move can block the attack or capture the attacking piece, the function returns false as it is not checkmate.

If the function has reached the end and has not returned, the king must be in checkmate, so the function returns true.

5.3.6 Update GUI

5.3.6.1 RefreshGUI

RefreshGUI contains methods to refresh the GUI. This includes refreshBoard, which removes all the Pieces from the GUI and then readds them based on the backend. It also creatures and updates, usually by removing

and readding, the Captured Pieces which appear on either side of the board, the Name Box which displays who's turn it is, and the Promotion Box which appears and disappears based on whether a Pawn can promote.

5.3.6.2 RefreshHighlight

RefreshHighlights contains methods to refresh the highlights on the bored. It is also where the frontend calls the getPossibleMoves, enPassant, and Castling for the selected piece. The method highlightMoves calls these methods and then displays the appropriate annotations on the bored, showing the moves. This method has a twin, resetHighlights, which removes all the highlights added by the original method.

The class also contains methods to show the King in Check an in Checkmate. The Checkmate method returns a Boolean which triggers the end of the game.

5.4 Significant data structures

5.4.1 Board Array

The board array data structure is a 2D array (matrix) of Square objects. Each square acts as a container for each part of the board which can contain any type of game piece or be empty. The board array does not handle any of the logic of the chess game, only acting as a 'flat surface' from which the chess game will be played on.

5.4.2 Valid moves

Each piece on the board will contain their own arrayLists of the valid moves they can take, which can then be used by the makeMove class to help its calculations for where the piece can be moved to.

5.4.3 Saving & loading

For saving and loading games, both for continuing running games as well as replaying previous game each step in the sequence of moves will be stored in an XML file. This is handled by the JAXB parser and the code in GameSaveManager. What data is saved and loaded and what data is excluded is controlled by JAXB annotations found in the major classes.

So that users can save game replays at the end of the game, each turn is saved to a temporary file. When the game ends and the user choses to save a replay, they specify a directory to save the replay in and the contents of the temporary file are copied into the directory and the temporary file is deleted. If the user does not want to save the replay the temporary file is deleted and not copied over.

REFERENCES

[1] Software Engineering Group Projects: General Documentation Standards. C.W. Loftus. SE.QA.05. 2.3 For Release
[2] Software Engineering Design Document (correct formatting needed here)
[3] Software Engineering GP02 Project User Interface Specification

DOCUMENT HISTORY

Version	*Issue No.*	*Date*	*Changes made to document*	*Changed by*
0.1	N/A	27-02-2023	N/A - original version	JAB153
0.2	N/A	07-03-2023	Added interface descriptions	JAB153
0.3	N/A	08-03-2023	Added UML sequence diagram and started significant algorithms	TPR3
0.4	N/A	09-03-2023	Added interface descriptions	JAB153
0.5	#43	11-03-2023	Finished first UML sequence diagram as well added more descriptions to significant algorithms	TPR3
0.5.1	N/A	15-03-2023	Added more significant classes	WIA14
0.5.2	N/A	16-03-2023	Added method types	WIA14
0.5.3	N/A	19-03-2023	Updated Significant Classes section	MJS36
0.5.4	N/A	19-03-2023	Updated Game class interface description	MJS36
0.5.5	N/A	19-03-2023	Updated Board class interface description	MJS36
0.5.6	N/A	19-03-2023	Updated Square and Player class interface descriptions	MJS36
0.5.7	N/A	19-03-2023	Updated Piece package interface description	MJS36
0.5.8	N/A	19-03-2023	Updated Moves and Util package interface description	MJS36
0.5.9	N/A	19-03-2023	Changed from boardArray:Square[][] to board:Board.	MJS36
0.6	#43	19-03-2023	Added all sequence diagrams	TPR3
0.6.1	N/A	19-03-2023	Move a Piece algorithm description added	MJS36
0.6.2	N/A	20-03-2023	Added detail to significant classes. Formatting changes to interface descriptions. Deleted test package	WIA14
0.6.4	N/A	20-03-2023	Added Check and CheckMate functions to significant algorithms	MIB60
0.6.5	N/A	20-03-2023	Added game save package to interface description, added more detail to saving and loading in significant data structures, added game save package to decomposition description	MIB60
0.7	N/A	20-03-2023	Added UML class diagram, corrected file versions	WIA14
0.7.1	N/A	20-03-2023	Fixed merge – saveReplay was missing in interface description	MIB60
0.7.2	N/A	20-03-2023	Added more detail to Saving/Loading under significant data structures	MIB60
0.7.3	N/A	20-03-2023	Fixed grammar and sentence structure issues in section 1 and 2. Updated GameSaveManager interface description to reflect same style as others	MJS36
0.7.4	N/A	20-03-2023	Added more detail to Check and CheckMate in significant algorithms	MIB60
0.7.5	#42	20-03-2023	Added component diagram	JAB153
0.7.6	#45	20-03-2023	Castling Significant Algorithm updated	MJS36
0.7.7	#69	20-03-2023	Added reference to section 1. Corrected functional requirements mapped to modules	JAB153
0.7.8	#69	20-03-2023	Changed component diagram to a transparent version	MJS36
1.0	#69	21-03-2023	Submitted for review	WIA14
1.1	#69	25-04-2023	Added BoardGUI method	WIA14
1.1.1	#69	26-04-2023	Replaced all instances of x and y, with row and column	WIA14
1.1.2	#69	02-05-2023	Added new component diagrams	JAB153

Version	*Issue No.*	*Date*	*Changes made to document*	*Changed by*
1.1.3	#69	03-05-2023	Updated GUI interface descriptions	JAB153
1.1.4	#69	03-05-2023	Updated Significant Algorithms	MJS36
1.1.5	N/A	04-05-2023	Updated component diagrams	JAB153
1.1.6	N/A	04-05-2023	Added new method	JAB153
1.1.7	N/A	04-05-2023	Updated UML Class Diagram	WIA14
1.1.8	#69	05-05-2023	Updated and added UML Sequence Diagrams	LVS1
1.1.9	#69	05-05-2023	Added new sequence diagrams	TPR3
1.2.0	#69	05-05-2023	Added final sequence diagrams	LVS1