Software Engineering Group Project
Design Specification

Author:	George Cooper [gwcl], Kieran Foy [kif11], James Owen [jco3], Tate Moore [tam41]
Config Ref:	SE_GP17_DesignSpecification
Date:	10th May 2023
Version:	1.10
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 outline the design specification for the Chess Tutor Application, to be presented along with the final draft of the design presentation in week beginning 20th March 2023.

1.2 Scope

The document has been created in line with the General Documentation Standards [1] and covers all sections described in the Design Specification Standards [2].

There are mentions of the use cases given in the User Interface Specification [3] as well as the functional requirements from the Requirements Specification [4]

The documents referenced will need to be looked through and understood before reading on.

1.3 Objectives

The objectives of this document are:

- To justify and describe the breakdown of the system into classes inside a singular program and its purpose, as well as that of each significant class'.
- Show that the functional requirements have been met by relating them to the classes.
- Using a component diagram, create an understanding of the relationships and dependencies between modules, or in other words how parts of the system fit in together.
- Give an outline specification for each class for a developer to understand them.
- Show the relationships between classes using UML class diagrams and how they work together using UML sequence diagrams in addition to how they satisfy the use cases [3].
- Explain the planning of difficult or significant algorithms using pseudocode.

2. DECOMPOSITION DESCRIPTION

2.1 Program in System

The system is made up of a singular program which runs as a JVM process. The program will allow 2 users to play a game of chess on a single computer.

When the program loads, they will be faced with a main menu where they will be able to:

- Quit the program.
- Load an unfinished game and continue it or a finished game and replay it.
- Start a new game.

Confirming and successfully loading an old game or starting a new one will take the players to a new scene where it will display the current chessboard state to the users.

The program will keep track of game statistics including:

- Whose turn it is.
- The colour and name of each player.
- The position of the pieces on the chess board.
- The history of all past moves.
- The taken pieces for each colour.

For an unfinished game, at the start of each new turn the program will indicate whose turn it is, this player can either select a piece, see all its moves, and choose one or deselect the piece.

If a user's king is in…

- Check – the program will alert the payers.
- Checkmate – the program will alert the players and display the winner. The program will give the users the option to save the game.

The program will allow the users to save and quit the game at any time. Saved games will be automatically saved in case of a crash.

For a finished game, the user will be allowed to go forwards and backwards through the game.

2.2 Significant Classes

Class Name	Description
EndScreenController	Handles all button inputs on the end screen scene where users can save the game, go back to the main menu, quit or start a new game.

Game	Game handles all systems to do with checking and updating the chessboard array.
	
	• Checks if…
	  ○ a piece moving from square to another is legal.
	  ○ it is the right player's turn.
	  ○ a king is in check or checkmate.
	  ○ an En Passant move is legal.
	• Updating the board when a piece is moved.
	
	Pawn promotion – updates pawn to a given new piece.

GameScreenController	Handles input to the game chessboard scene such as selecting and moving pieces. It also can rotate the board and showing legal moves.
NewGameScreenController	Handles input to the new game scene where the users can enter player names, go back to the main menu and confirm entered player names and move to the game chessboard scene.
Piece	Piece will act as a superclass to all specific piece classes. It will include attribute shared by all chess pieces. Its subclasses include Pawn, Knight, Bishop, Rook, Queen and King, each subclass x will act as a blueprint for all x pieces in the game, it will store x specific attributes such as legal moves.
PauseScreenController	Handles input to the pause menu where the users can continue a game, draw or resign.

PawnPromotionScreenController	Handles input to the pawn promotion scene where the users can pick to promote a pawn piece to a queen, knight, bishop or rook.
Replay	Replay will handle… • Loading and displaying previously played games. • Players stepping backwards and forwards throughout loaded finished games. • Automatic saving of the game every turn as well as manual done by the player upon finishing the game or quitting.
Setup	Initialises the starting board with all its pieces at their starting positions and generate every scene

2.3 Mapping From Requirements to Classes

Requirement [4]	Classes providing requirement
FR1	GameScreenController, NewGameScreenController, Setup
FR2	Game
FR3	Game, GameScreenController
FR4	Game, GameScreenController
FR5	Bishop, Game, GameScreenController, King, Knight, Pawn, Queen, Rook
FR6	Bishop, Game, GameScreenController, King, Knight, Pawn, Queen, Rook
FR7	Bishop, EndScreenController, Game, King, Knight, Pawn, Queen, Rook
FR8	EndScreenController, Game, PauseScreenController
FR9	PauseScreenController, Replay
FR10	Game, GameScreenController, Replay
FR11	Game, Replay

3. DEPENDENCY DESCRIPTION

3.1 Component Diagram For Model Code

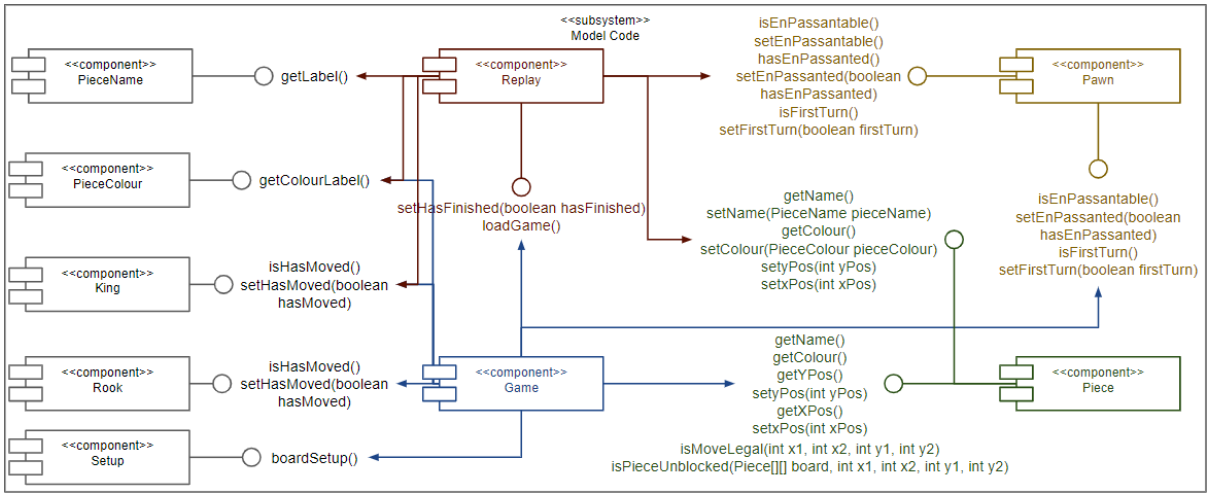


Figure 1: Component Diagram of the Chess Tutor Game System showing what methods classes call from other classes. These methods can be categorised as model code, which means they access stored data in the system.

3.2 Component Diagram For View Code

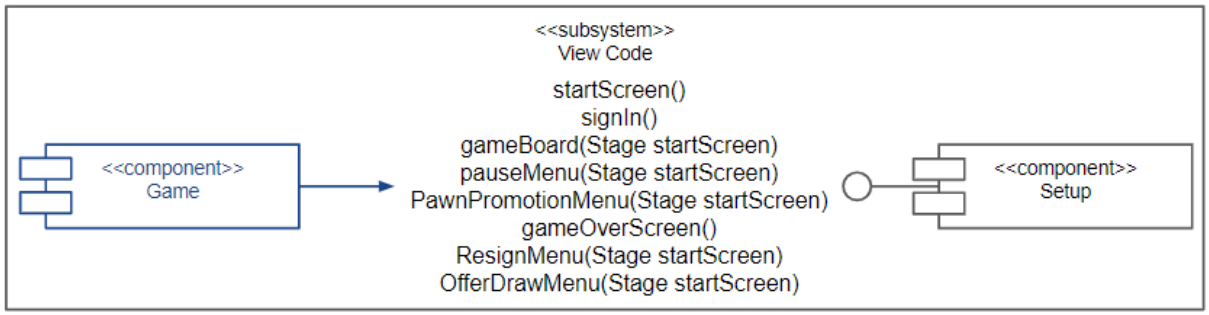


Figure 2: Component Diagram of the Chess Tutor Game System showing what methods classes call from other classes. These methods can be categorised as view code, which means they are what generates the appearance of the GUI.

3.3 Component Diagram For Controller Classes

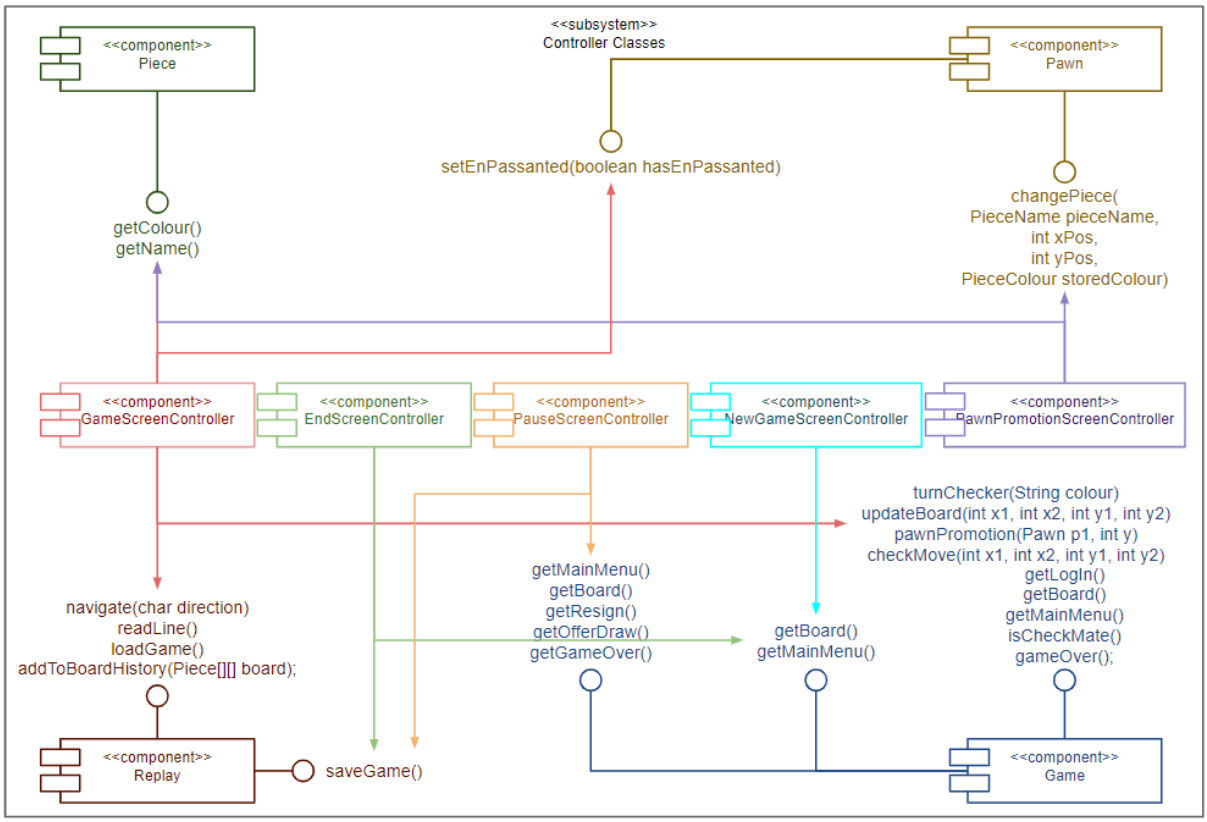


Figure 3: Component Diagram of the Chess Tutor Game System showing what methods controller classes call from other classes.

4. INTERFACE DESCRIPTION

4.1 Controller Classes

4.1.1 EndScreenController

- Type: public
- Extends: Game – holds the static replay object where a method is called from to save the game.
- Public methods:
  - **EndScreenController()** – constructor
  - **void handleSaveQuitButton(ActionEvent actionEvent)** –handles when the save and quit button on the end of game scene and will take you to the main menu scene as well as save the game for future replaying.

4.1.2 GameScreenController

- Type: public
- Extends: Game – to access the board and turn tracker static variables.
- Public methods:
  - **GameScreenController()** – constructor
  - **void addToImages()** – Adds the images of all squares on the chessboard to an ArrayList.
  - **void handleBackwardButton(ActionEvent actionEvent)** – Pressed when replaying the game, goes backwards 1 turn.
  - **void handleBoardRotation(KeyEvent actionEvent)** – rotates the board and calls rotateAllNodes(Parent p) to rotate all the pieces.

- **void handleForwardButton(ActionEvent actionEvent)** – Pressed when replaying the game, goes forwards 1 turn.
- **void handleLoadOldGameButton(ActionEvent actionEvent)** – Handles the event when the "Load Game" button is clicked. Calls a Replay object to load an old game.
- **void handlePiece(MouseEvent mouseEvent)** – handles when a square is selected.
- **void handleQuitGame(ActionEvent actionEvent)** – handles when the quit button is pressed.
- **void handleRequestMove(MouseEvent mouseEvent)** – handles when a request to move 1 piece to another.
- **void handleStartNewGame(ActionEvent actionEvent)** – handles switching scenes from main menu to game board scene.
- **void movePiece(int x2, int x2, int y1, int y2)** – updates scene for when a piece moves – puts the moving piece's image at x2,y2 and removes old piece image at x1,y1.
- **void resetImages()** – resets the images on the squares of the chessboard.
- **void rotateAllNodes(Parent p)** – rotates all the pieces.
- **void showMoves(int x1, int y1)** – shows the legal moves a piece can make.

4.1.3 NewGameScreenController

- Type: public
- Extends: nothing
- Public methods: Game – to access the static method to retrieve the board scene and the static variables holding the player names
  - **NewGameScreenController()** – constructor
  - **void handleCancelButton(ActionEvent actionEvent)** – handles when the cancel button on the new game screen is clicked on and will go back to the main menu scene.
  - **void handleSubmitButton(ActionEvent actionEvent)** – handles when the submit button on the new game screen is clicked on and will go onto the game chessboard.
  - **void playerNames(ActionEvent actionEvent)** – fetches the entered player names from the text fields.

4.1.4 PauseScreenController

- Type: public
- Extends: Game – to access the static method to retrieve the board scene
- Public methods:
  - **PauseScreenController()** – constructor
  - **void handleAccept(ActionEvent actionEvent)** – When a draw is offered, handles when the opposing player presses the decline button then will end the game.
  - **void handleContinueButton(ActionEvent actionEvent)** – handles when the continue button on the pauses screen is clicked on, program will go back to game chessboard scene.
  - **void handleDecline(ActionEvent actionEvent)** – When a draw is offered, handles when the opposing player presses the decline button then will continue the game.
  - **void handleMainMenuButton(ActionEvent actionEvent)** – handles when the main menu button is clicked on, program will move to the main menu scene.
  - **void handleOfferDrawButton(ActionEvent actionEvent)** – handles when the draw button is clicked on, program offers the other player to draw.
  - **void handleResignButton(ActionEvent actionEvent)** – handles when the resign button is clicked on, program will end the game.
  - **void handleSave(ActionEvent actionEvent)** – handles when the save button is clicked on, program will save the game.
  - **void handleSaveAndQuitButton(ActionEvent actionEvent)** – handles when the save & quit button is clicked on, program will save the game then close.

4.1.5 PawnPromotionScreenController

- Type: public
- Extends: Game – to access the static variable holding the board of pieces.
- Public methods:
  - **PawnPromotionScreenController()** – constructor

- **Node getNode(int x, int y)** – Tells another class what the value is from a private variable node
- **PieceName handleBishop(ActionEvent actionEvent)** – handles when the bishop piece is picked (for the pawn to be promoted to).
- **PieceName handleKnight(ActionEvent actionEvent)** – handles when the knight piece is picked (for the pawn to be promoted to).
- **PieceName handleQueen(ActionEvent actionEvent)** – handles when the queen piece is picked (for the pawn to be promoted to).
- **PieceName handleRook(ActionEvent actionEvent)** – handles when the rook piece is picked (for the pawn to be promoted to).

4.2 Enumerations

4.2.1 PieceColour

- Type: enum
- Extends: nothing
- Public methods:
  - **PieceColour(String label)** – constructor.
  - **String getColourLabel()** – gets the string label value of the enumeration value.

4.2.2 PieceName

- Type: enum
- Extends: nothing
- Public methods:
  - **PieceName(String label)** – constructor.
  - **String getLabel()** – gets the string label value of the enumeration value.

4.3 Model Classes

4.3.1 Game

- Type: public
- Extends: nothing
- Public methods:
  - **Game()** – constructor
  - **boolean checkMove(int x1, int x2, int y1, int y2)** – checks if a given move is legal.
  - **void checkOrCheckmate()** – detects if the King of the opposite colour of the player who just had their turn is in check or checkmate.
  - **Piece[][] copyBoard(Piece[][] board)** – Copies a given chessboard (Piece[][] board) and returns the new copy of it.
  - **void displayBoardCmd(Piece[][] board)** – Prints the board on the command line.
  - **void gameOver()** – method to handle when a game over is detected.
  - **boolean isCheck(int x1, int y1)** – checks if a king of opposing colour is within the movement range of a piece, that is at x1, y1.
  - **boolean isOpponentCheck(int x1, int x2, int y1, int y2)** – detects if the opponent's king is in check.
  - **boolean pawnPromotion(Pawn p1, int y)** – checks if a pawn that has been moved is eligible to be promoted.
  - **void startUp(Stage startScreen)** – initialiser function for when a new game is started – initialises all the scenes, the starting chessboard array and the turn tracker.
  - **void updateBoard(int x1, int x2, int y1, int y2)** – updates the board array after a move, turn tracker incremented and game saves.

4.3.2 Piece

- Type: public
- Extends: nothing

- Public methods:
  - **Piece(PieceColour colour, PieceName name, int xPos, int yPos)** – parameterised constructor.
  - **Piece()** – default constructor.
  - **abstract boolean isMoveLegal(int x1, int x2, int y1, int y2)** – the super class's abstract method – checks if a move is legal – which all child / sub classes will override.
  - **abstract boolean isPieceBlocked(int x1, int x2, int y1, int y2)** – the super class's abstract method – checks if a move is blocked by another piece in the way – which all child / sub classes will override.

4.3.3 Bishop

- Type: public
- Extends: Piece
- Public methods:
  - **Bishop(PieceColour colour, PieceName name, int xPos, int yPos)** – constructor which calls Piece's parameterised constructor (super class).
  - **Bishop()** – constructor.
  - **@Override boolean isMoveLegal(int x1, int x2, int y1, int y2)** – checks if a given move is in the movement range of the bishop piece type.
  - **@Override boolean isPieceBlocked(Piece[][] board, int x1, int x2, int y1, int y2)** – checks if a given move is blocked by a piece in its way.

4.3.4 King

- Type: public
- Extends: Piece
- Public methods:
  - **King(PieceColour colour, PieceName name, int xPos, int yPos)** – constructor which calls Piece's parameterised constructor (super class).
  - **King()** – constructor.
  - **@Override boolean isMoveLegal(int x1, int x2, int y1, int y2)** – checks if a given move is in the movement range of the king piece type.
  - **@Override boolean isPieceBlocked(Piece[][] board, int x1, int x2, int y1, int y2)** – checks if a given move is blocked by a piece in its way.

4.3.5 Knight

- Type: public
- Extends: Piece
- Public methods:
  - **Knight(PieceColour colour, PieceName name, int xPos, int yPos)** – constructor which calls Piece's parameterised constructor (super class).
  - **Knight()** – constructor.
  - **@Override boolean isMoveLegal(int x1, int x2, int y1, int y2)** – checks if a given move is in the movement range of the knight piece type.
  - **@Override boolean isPieceBlocked(Piece[][] board, int x1, int x2, int y1, int y2)** – checks if a given move is blocked by a piece in its way.

4.3.6 Pawn

- Type: public
- Extends: Piece
- Public methods:
  - **Pawn(PieceColour colour, PieceName name, int xPos, int yPos)** – constructor which calls Piece's parameterised constructor (super class).
  - **Pawn()** – constructor.
  - **void changePiece(PieceName pieceName, int xPos, int yPos, PieceColour storedColour)** – Promotes a pawn at a given position to the selected piece

- **@Override boolean isMoveLegal(int x1, int x2, int y1, int y2)** – checks if a given move is in the movement range of the pawn piece type.
- **@Override boolean isPieceBlocked(Piece[][] board, int x1, int x2, int y1, int y2)** – checks if a given move is blocked by a piece in its way.
- **void pawnPromotion(int x, int y)** – if a pawn is its opposite end of the board, the program will show the pawn promotion scene.
- **boolean takePiece(int x1, int x2, int y1, int y2)** – checks if a pawn can take a piece

4.3.7 Queen

- Type: public
- Extends: Piece
- Public methods:
  - **Queen(PieceColour colour, PieceName name, int xPos, int yPos)** – constructor which calls Piece's parameterised constructor (super class).
  - **Queen()** – constructor.
  - **@Override boolean isMoveLegal(int x1, int x2, int y1, int y2)** – checks if a given move is in the movement range of the queen piece type.
  - **@Override boolean isPieceBlocked(Piece[][] board, int x1, int x2, int y1, int y2)** – checks if a given move is blocked by a piece in its way.

4.3.8 Rook

- Type: public
- Extends: Piece
- Public methods:
  - **Rook(PieceColour colour, PieceName name, int xPos, int yPos)** – constructor which calls Piece's parameterised constructor (super class).
  - **Rook()** – constructor.
  - **@Override boolean isMoveLegal(int x1, int x2, int y1, int y2)** – checks if a given move is in the movement range of the rrook piece type.
  - **@Override boolean isPieceBlocked(Piece[][] board, int x1, int x2, int y1, int y2)** – checks if a given move is blocked by a piece in its way.

4.3.9 Replay

- Type: publ**ic**
- Extends: nothing
- Public methods:
  - **Replay(String playerOneName, String playerTwoName)** – parameterised constructor to be called when a new game is created.
  - **Replay()** – ~~constructor~~ Default constructor to be called when loading a game.
  - **void gameName()** – generates the file name for the file in which a new game will be stored.
  - **void loadGame()** – loads the game data including player names, whether the game has finished and the history of all past boards / turns.
  - **void navigate(char direction)** – Navigates forwards or backwards through the moves the players have played
  - **void printBoard(Piece[][] pieces)** – updates the board visually to a given board of pieces.
  - **boolean readLine()** – allows the user to choose a game file to be loaded in. Returns true if a file is found and it is a json file, otherwise returns false.
  - **void saveGame()** – saves the game data including player names, whether the game has finished and the history of all past boards / turns

4.4 View Classes

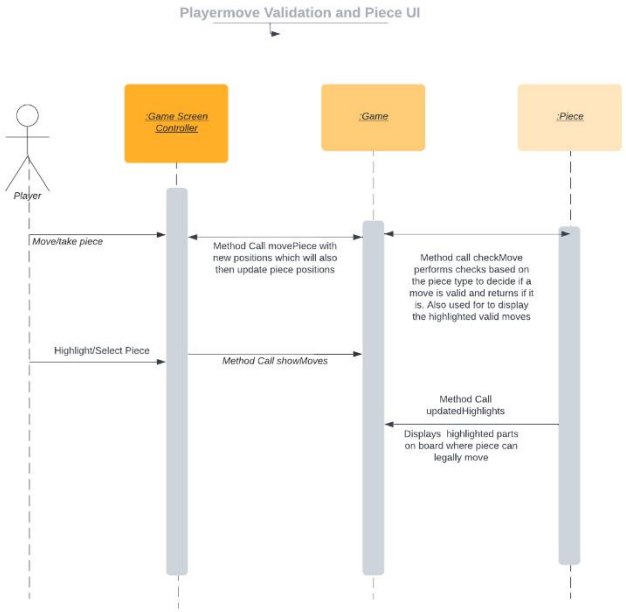
4.4.1 Setup

- Type: public
- Extends: nothing
- Public methods:
  - **Setup()** – constructor
  - **Piece[][] boardSetup()** – sets up the chessboard array that the game will be played on and adds all pieces to their starting positions. Called every time a new game is started. This is of a model method however the majority of this class can be classified as view code.
  - **Scene gameBoard(Stage startScreen)** – generates the main menu scene.
  - **Scene gameOverScreen(Stage startScreen)** – generates the game over scene.
  - **Scene OfferDrawMenu(Stage startScreen)** – generates the offer draw scene.
  - **Scene pauseMenu(Stage startScreen)** – generates the pause menu scene.
  - **Scene PawnPromotionMenu(Stage startScreen)** – generates the pawn promotion scene.
  - **void printBoard (Piece[][] chessBoard)** – prints a given chessboard to the console.
  - **Scene ResignMenu(Stage startScreen)** – generates the resign menu scene.
  - **Scene signIn()** – generates the new game scene.
  - **Scene startScreen()** – loads up the startup scene.

5. DETAILED DESIGN

5.1 Sequence diagrams

Player Move Validation Diagram:



Game State Management

Replay

5.2 Significant algorithms

5.2.1 Move Generation

As stated in FR5 pieces must display all legal moves when highlighted. In chess there are some notable exceptions to the normal piece behaviours we will have to run checks on when displaying these moves.

Check – If the king will get checked it will not display it as a valid move.

Here is a brief explanation of the check/discovered check peudocode:

First, find the location of the king that belongs to the player that just moved. Then if it is the correct king it checks if it is in check by calling a method that will return true if any pieces on the opposing team have legal moves attacking the king. If the king is in check, it checks whether is in checkmate or not. If none of the player's pieces have a legal move to take the king out of check then it is in checkmate. If the king is not in check the code does nothing.

Since every move of the game will be saved, we do not need to iterate through the board to find the king. **

```
function checkOrCheckmate():
        king = findKing(turnTracker % 2) // Finds king for player whose turn it is
        opponentAttacks = getAllAttacks(king.getColour()) if king.getPosition() in opponentAttacks:
        if canMoveOutOfCheck(king):
            checkFlag = true
        else:
            checkmateFlag = true
```

The pseudocode for discovered check runs on the same code.

En Passant – Here is a brief explanation of the En Passant pseudocode:

First check if the last move was a pawn double move. If so then a flag is set to show that en passant is possible next turn. The file and rank of the pawn in question is also saved. On the next turn check if a pawn moves adjacently behind the stored pawn. This captures the saved pawn.

```
//At the end of each turn, check if en passant is possible
 if lastMove.isPawnDoubleMove(): //Set a flag to indicate that en passant is possible
        enPassantFlag = true //Save the file and rank of the pawn that made the double move
        enPassantPawn = lastMove.getToSquare()

 //Next turn, check if an adjacent pawn moves behind the en passant pawn
if currentMove.isPawnMove() and currentMove.getToSquare().isAdjacent(enPassantPawn):
        if enPassantFlag: //En passant is possible, capture the en passant pawn
        capturedPawn = getPieceAt(enPassantPawn)
        removePieceAt(enPassantPawn)
```

Castling

First, you must check whether the king and the rook are in their initial positions. This is because for a castle to be legal both must be in their initial positions. Next, check if the squares between the king and rook are empty. If so, then check if the king is in check or passes through check.

```
if king.hasMoved() or rook.hasMoved():
        return false
if king.getPosition().getRank() != rook.getPosition().getRank(): return false if
board.isSquareOccupied(king.getPosition().getNextSquareTowards(rook.getPosition())):
        return false
for square in king.getPosition().getSquaresTowards(rook.getPosition()):
        if board.isSquareOccupied(square):
            return false
if board.isAttackedByOpponent(king.getPosition(), playerColour) or
board.isAttackedByOpponent(king.getPosition().getNextSquareTowards(rook.getPosition()), playerColour):
return false if
```

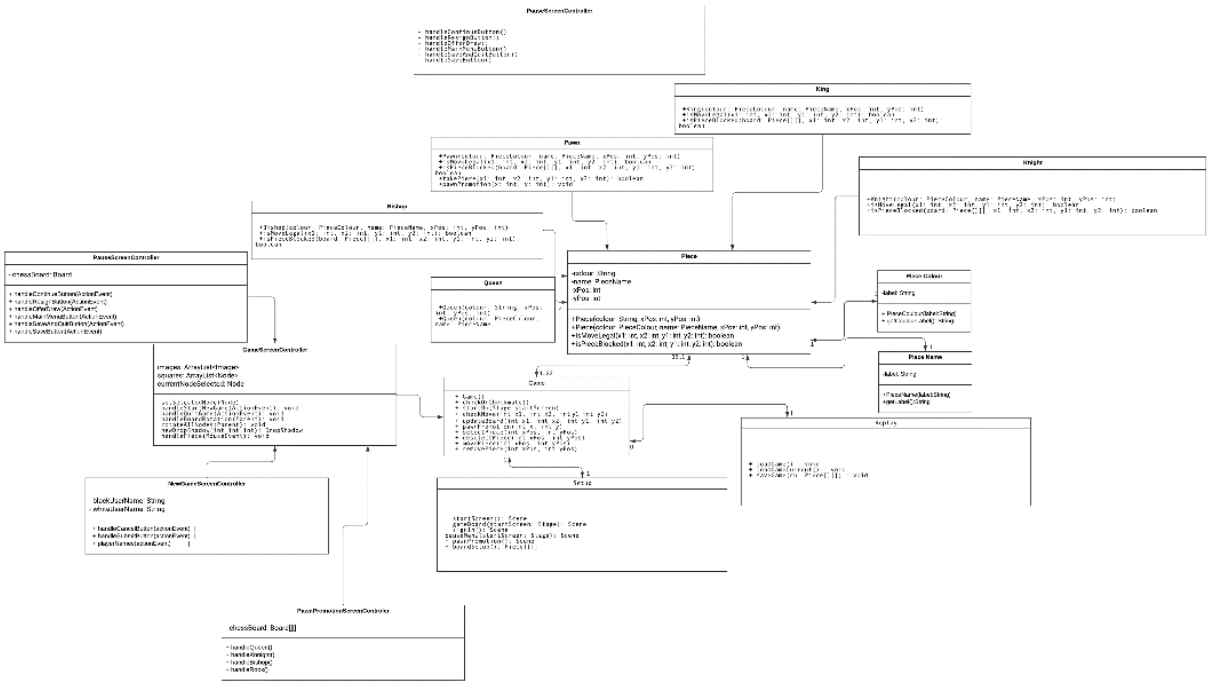
board.isAttackedByOpponent(king.getPosition().getNextSquareTowards(rook.getPosition()), playerColour) or board.isAttackedByOpponent(rook.getPosition().getPreviousSquare(), playerColour): return false

return true

//Then move the pieces to the new positions using a special method for castling

5.3 UML Class Diagrams

This diagram shows the relationships between the necessary classes to form majority of the working game and UI.



5.4 Significant data structures

The save file for the game is a JSONObject data structure which is updated each turn as the game progresses. This is so when the game is read via the continue button on the main menu, the game can resume from the latest possible moment. It is also so that the game can be replayed move by move.

We are using “Gson” library to convert the data (Piece positioning) into a JSON string and writing it to a file. Then we are reading that sting from the file and using GSON to convert it back to its original data structure.

The pieces are stored in a 2-dimensional array (Piece[][]). Each element in the array represents a position of the game board and contains information about the piece located there such as its colour and type.

**Object diagram

**State chart Diagram

REFERENCES

[1] Loftus C.W., “Software Engineering Group Projects –
General Documentation Standards”, 2.6, Computer Science Department, 10th January 2023

[2] Loftus C.W., “Software Engineering Group Projects –
Design Specification Standards”, 2.3, Computer Science Department, 6th December 2021

[3] Moore T., Enache A., “Software Engineering Group 17 User Interface”, 1.0, SE_GP17_UISpecification, 27th February 2023

[4] Loftus C.W., “Software Engineering Group Project - Chess Tutor Requirement Specification”, 1.2, Computer Science Department, 28th February 2023

Figure 1: Google. 2006. Google Slides. [Software]. [Accessed 10th May 2023].

Figure 2: Google. 2006. Google Slides. [Software]. [Accessed 10th May 2023].

Figure 3: Google. 2006. Google Slides. [Software]. [Accessed 10th May 2023].

DOCUMENT HISTORY

Version	*Issue No.*	*Date*	*Changes made to document*	*Changed by*
0.1	N/A	7/02/23	N/A - original draft version	GWC1
0.2	11	14/02/23	Filled out section 1: Introduction	KIF11
0.3	11	14/02/23	Filled out section 2: Decomposition Description	GWC1
0.4	11	04/03/23	Adjusted section 2: Decomposition Description Added almost finished diagram (Figure 1)	TAM41
0.5	11	05/03/23	Added to section 4: Interface Description	TAM41
0.6	11	07/03/23	Added in references	TAM41
0.7	11	13/03/23	Added to section 5: Detailed Design	JCO3
0.8	11	13/03/23	Adjusted sections to review meeting feedback	TAM41
0.9	11	14/03/23	Adjusted references and scope.	TAM41
0.10	11	19/03/23	Adjusted section 5 Detailed Design	JCO3
0.11	25	20/03/23	Made vast adjustments to section 4 Interface Description.	TAM41
0.12	26	20/03/23	Adjusted section 2 and redid diagram for section 3	TAM41
0.13	11	20/03/23	Adjusted section 5	JCO3
1.0	11	20/03/23	First release	GWC1
1.1	11	16/04/23	Adjusted section 5	JCO3
1.2	11	23/04/23	Adjusted various sections	JCO3
1.3	35	26/04/23	Adjusted 2.1	TAM41
1.4	35	30/04/23	Adjusted and added to section 4.	TAM41
1.5	35	30/04/23	Adjusted and added to section 2.2	TAM41
1.6	N/A	03/05/23	Adjusted section 5.2 and 5.3	JCO3
1.7	N/A	04/05/23	Completed section 5	JCO3
1.8	N/A	08/05/23	Updated sections 4 and 2	TAM41
1.9	N/A	09/05/23	Reordering in sections 4 and 2	TAM41
1.10	35	10/05/23	Redid component diagrams, their descriptions and references for them.	TAM41