



CCP6224 OOAD

Trimester 2430

by <<MEOWZAM>>

Team Leader:

Name	Phone Number	Email
SARA EMILIA BINTI SHARIFUDIN	019-2914108	1221102332@student.mmu.edu.my

Team Members:

Name	Phone Number	Email
IZZA NELLY BINTI MOHD NASIR	017-7031856	1211111583@student.mmu.edu.my
AMALIA SORFINA BINTI MAHDZIR	013-2716912	1211111891@student.mmu.edu.my
NURUL SYAHAFIZA BINTI NAZIRON	017-9625944	1221103588@student.mmu.edu.my

Table Of Content

Table Of Content	2
1. Compile and run instructions	2
2. UML Class Diagram	6
2.1 Class Table	6
2.2 Class Diagram	9
2.3 Classes Utilizing Design Patterns	10
3. Use Case Diagram	11
3.1 Use Case Diagram	11
3.2 Use Case Description	11
4. Sequence Diagram	13
4.1 Sequence Diagram Save Current Game	13
4.2 Sequence Diagram Delete Save Game	13
4.3 Sequence Diagram Load Save Game	14
4.4 Start New Game	15
4.5 Display Board and Pieces	15
4.6 Move Pieces	16
4.7 Simulate Move	17
4.8 Switch Turn	18
5. User Documentation	20
5.1 Introduction	20
5.2 User Interface Overview	20
5.2.1 Main Screen	20
5.2.2 Start New Game Screen	21
5.2.3 Kwazam Chess Board Game Screen	21
5.2.4 Load Saved and Delete Game Screen	22
5.2.5 Winner Declaration Screen	25
5.3 Gameplay Mechanics	25
5.3.1 Drag-to-Move Functionality	26
5.3.2 Board Flipping Mechanics	26
5.5 Game Instructions	29
5.6 File Management Information	30
5.6.1 Save file format	30
5.6.2 Usage in the Kwazam Chess Game	31

1. Compile and run instructions

The prerequisite to running the application includes:

1. The device must have a Java Runtime Environment (JRE) to be able to run Java programs.
2. The zip file containing all Java files is downloaded.

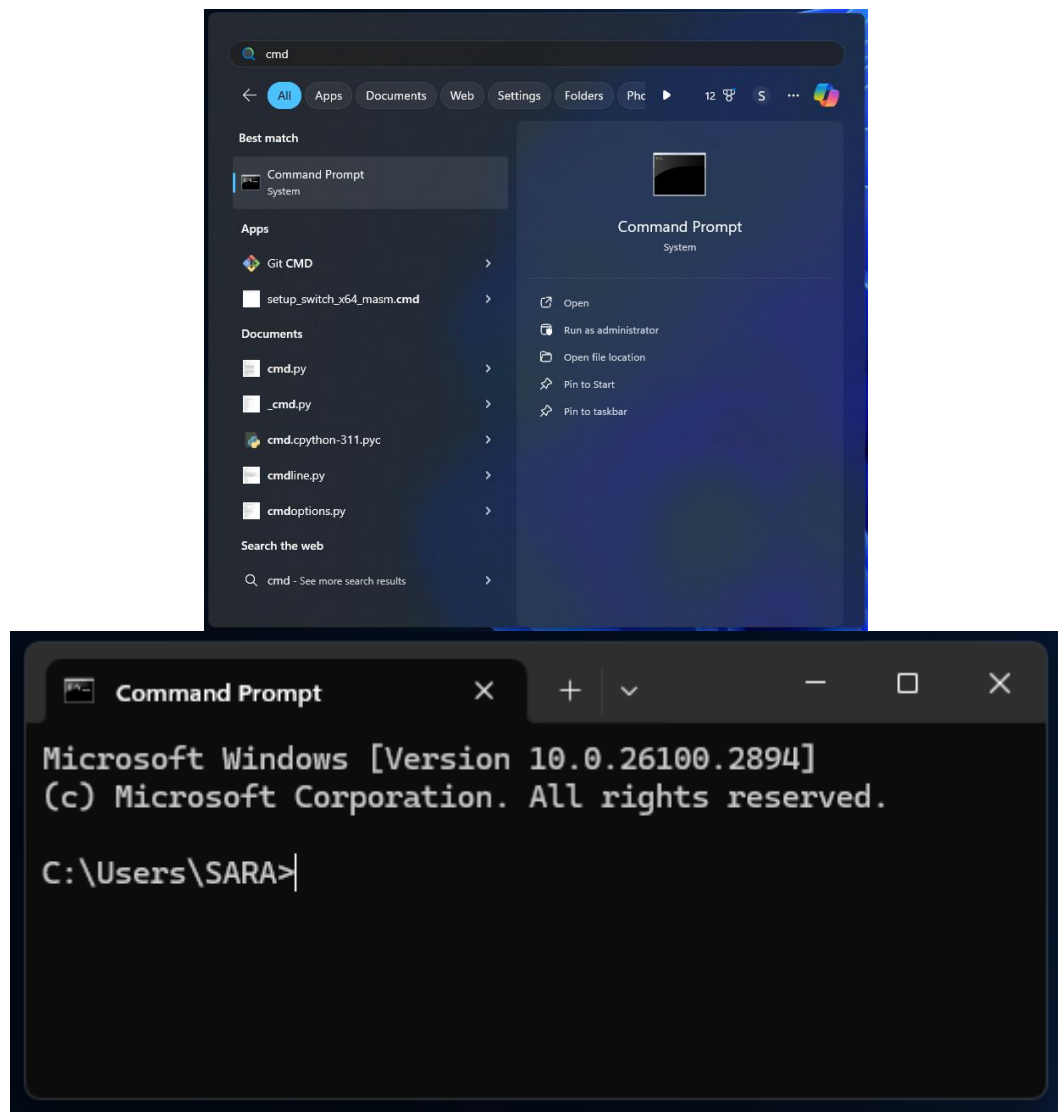
The following instructions are compatible with the Windows operating system. Different commands are needed in the terminal to run the main Java file for other operating systems.

1) **File Extraction**

Extract all the files in the zip file. When the folder called “MEOWZAM_OOAD” is selected and opened, all the source code files can be seen.

2) **Open folder through command prompt**

Search on “Start” of your computer, and type “CMD” in the search bar. Select the “Command Prompt” following the example below for Windows and it will open the command prompt.



3) **Navigate to the “MEOWZAM_OOAD” folder.**

You may use the `cd “folder name”` command to navigate to the folder. The “MEOWZAM_OOAD” folder contains a file called “Main.java”. The “Main.java” file contains the main and is runnable.

```
Command Prompt

Microsoft Windows [Version 10.0.26100.2894]
(c) Microsoft Corporation. All rights reserved.

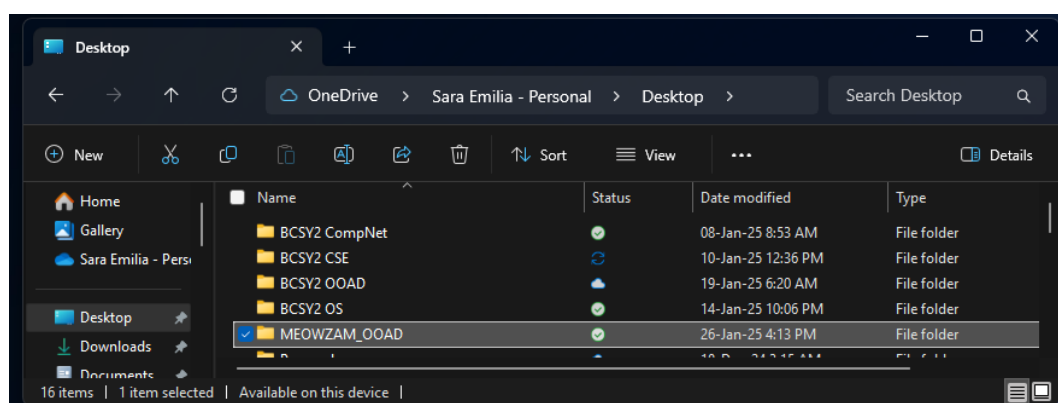
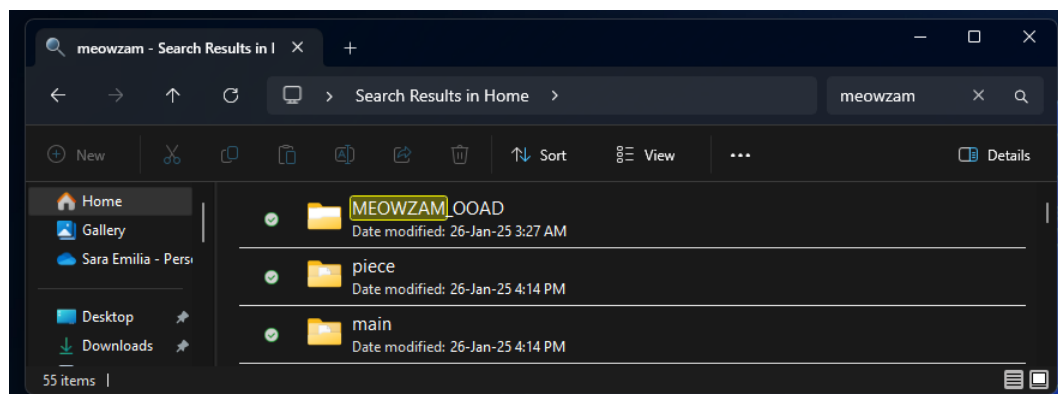
C:\Users\SARA>cd OneDrive

C:\Users\SARA\OneDrive>cd Desktop

C:\Users\SARA\OneDrive\Desktop>cd MEOWZAM_OOAD

C:\Users\SARA\OneDrive\Desktop\MEOWZAM_OOAD>
```

If you did not know the directory of the folder. Search in file explorer “MEOWZAM_OOAD”. From that you can determine the path by referencing below, as example “OneDrive\Desktop\..”



4) Compile the files and Execute the Application

Once you have navigated to the “MEOWZAM_OOAD” folder as shown above. Type and run the following command:

1. Compile the file :
javac -d out main\Main.java

2. Execute the application:
java -cp out main.Main

```
Command Prompt - java -cp X + v
Microsoft Windows [Version 10.0.26100.2894]
(c) Microsoft Corporation. All rights reserved.

C:\Users\SARA>cd OneDrive

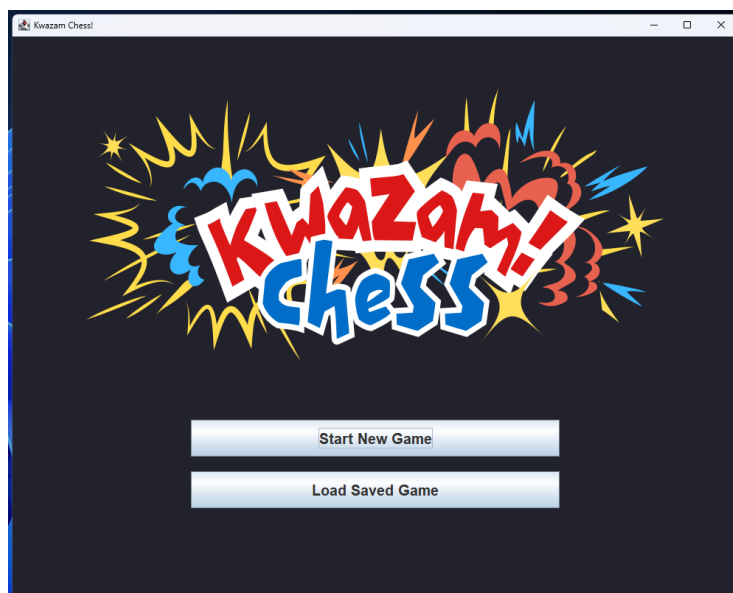
C:\Users\SARA\OneDrive>cd Desktop

C:\Users\SARA\OneDrive\Desktop>cd MEOWZAM_OOAD

C:\Users\SARA\OneDrive\Desktop\MEOWZAM_OOAD>javac -d out main\Main.java

C:\Users\SARA\OneDrive\Desktop\MEOWZAM_OOAD>java -cp out main.Main
|
```

The application will open as below, and you are ready to go.



Note: A JAR File is included as an addition if the application is run on another operating system as the instructions given above are Windows operating system specific. To run the JAR File, simply double-click it and the game program will pop up

2. UML Class Diagram

2.1 Class Table

The table below describes the classes in the class diagram.

Class	Description	Attributes/Operations			
Main Package					
Main	Purpose: The entry point of the application. It sets up the main <i>JFrame</i> , initializes the <i>Controller</i> , and manages the <i>CardLayout</i>	<table><tr><th>Main</th></tr><tr><td>+ main(String[] args): void</td></tr></table>	Main	+ main(String[] args): void	
Main					
+ main(String[] args): void					
Controller	Purpose: Acts as the main coordinator for the game. Manages UI transitions, game state saving/loading, and player turn logic. It provides options to start a new game,load a saved game, or delete game slots. Static lists <i>pieces</i> and <i>simPieces</i> are used for managing the active pieces in the game Design Pattern: <ul style="list-style-type: none">- Controller. Separating game logic and UI logic to handle transitions and user input.- Stratergy. Separating the logic for saving and updating the game state into different methods (`saveGame`, `updateGame`, `saveBoardState`)	<table><tr><th>Controller</th></tr><tr><td>- gamePanel: JPanel - startButton: JButton - loadButton: JButton - bluePlayer: String - redPlayer: String - saveNumber: int - currentColor: int + pieces: ArrayList<Piece> + simPieces: ArrayList<Piece> + BLUE: int {static final} + RED: int {static final}</td></tr><tr><td>+ Controller(cardPanel: JPanel) - showNewGamePanel(cardPanel: JPanel): void - showLoadGamePanel(cardPanel: JPanel): void + loadGame(saveNumber: int): Map<String, Object> + saveGame(saveNumber: int, currentColor: int, bluePlayer: String, redPlayer: String, gamePanel: JPanel): int + deleteGame(saveNumber: int, cardPanel: JPanel): void - updateGame(saveNumber: int, currentColor: int, bluePlayer: String, redPlayer: String, gamePanel: JPanel): void - saveBoardState(writer: Appendable, gamePanel: JPanel): void + getSaveNumber(): int + setSaveNumber(saveNumber: int): void + getBluePlayer(): String + setBluePlayer(bluePlayer: String): void + getRedPlayer(): String + setRedPlayer(redPlayer: String): void + getCurrenctColor(): int + setCurrentColor(currentColor: int): void</td></tr></table>	Controller	- gamePanel: JPanel - startButton: JButton - loadButton: JButton - bluePlayer: String - redPlayer: String - saveNumber: int - currentColor: int + pieces: ArrayList<Piece> + simPieces: ArrayList<Piece> + BLUE: int {static final} + RED: int {static final}	+ Controller(cardPanel: JPanel) - showNewGamePanel(cardPanel: JPanel): void - showLoadGamePanel(cardPanel: JPanel): void + loadGame(saveNumber: int): Map<String, Object> + saveGame(saveNumber: int, currentColor: int, bluePlayer: String, redPlayer: String, gamePanel: JPanel): int + deleteGame(saveNumber: int, cardPanel: JPanel): void - updateGame(saveNumber: int, currentColor: int, bluePlayer: String, redPlayer: String, gamePanel: JPanel): void - saveBoardState(writer: Appendable, gamePanel: JPanel): void + getSaveNumber(): int + setSaveNumber(saveNumber: int): void + getBluePlayer(): String + setBluePlayer(bluePlayer: String): void + getRedPlayer(): String + setRedPlayer(redPlayer: String): void + getCurrenctColor(): int + setCurrentColor(currentColor: int): void
Controller					
- gamePanel: JPanel - startButton: JButton - loadButton: JButton - bluePlayer: String - redPlayer: String - saveNumber: int - currentColor: int + pieces: ArrayList<Piece> + simPieces: ArrayList<Piece> + BLUE: int {static final} + RED: int {static final}					
+ Controller(cardPanel: JPanel) - showNewGamePanel(cardPanel: JPanel): void - showLoadGamePanel(cardPanel: JPanel): void + loadGame(saveNumber: int): Map<String, Object> + saveGame(saveNumber: int, currentColor: int, bluePlayer: String, redPlayer: String, gamePanel: JPanel): int + deleteGame(saveNumber: int, cardPanel: JPanel): void - updateGame(saveNumber: int, currentColor: int, bluePlayer: String, redPlayer: String, gamePanel: JPanel): void - saveBoardState(writer: Appendable, gamePanel: JPanel): void + getSaveNumber(): int + setSaveNumber(saveNumber: int): void + getBluePlayer(): String + setBluePlayer(bluePlayer: String): void + getRedPlayer(): String + setRedPlayer(redPlayer: String): void + getCurrenctColor(): int + setCurrentColor(currentColor: int): void					

<i>GamePanel</i>	Responsible for rendering the game board, managing mouse interactions, and processing game logic, such as validating moves and switching turns. It includes methods for deleting save slots (<i>deleteGameSlot</i>) and displaying a winner screen (<i>showWinnerScreen</i>).	<div>GamePanel</div> <ul style="list-style-type: none"> - FPS: int {final} - gameThread: Thread - board: Board - mouse: Mouse + pieces: ArrayList<Piece> + simPieces: ArrayList<Piece> - activeP: Piece + BLUE: int{static final} + RED: int{static final} - currentColor: int - saveNumber: int - bluePlayer: String - redPlayer: String - saveButton: JButton - returnButton: JButton - bluePlayerLabel: JLabel - redPlayerLabel: JLabel - cardPanel: JPanel - canMove: boolean - validSquare: boolean <ul style="list-style-type: none"> + GamePanel(cardPanel: JPanel, controller: Controller) + launchGame(): void - initializeGameState(controller: Controller): void + setCurrentColor(currentColor: int): void + setBluePlayer(bluePlayer: String): void + setRedPlayer(redPlayer: String): void + setSaveNumber(saveNumber: int): void + getPieces(): ArrayList<Piece> + initializeDefaultPieces(): void + loadPieces(loadedPieces: ArrayList<Piece>): void - copyPieces(source: ArrayList<Piece>, target: ArrayList<Piece>): void + run(): void - update(): void - changePlayer(): void + deleteGameSlot(saveNumber: int): void - simulate(): void - showWinnerScreen(winnerImagePath: String): void 	
<i>Board</i>	Responsible for rendering the chessboard.	<div>Board</div> <ul style="list-style-type: none"> + SQUARE_SIZE: int{static final} + HALF_SQUARE_SIZE: int{static final} <ul style="list-style-type: none"> + draw(Graphics2D g2): void 	
<i>Mouse</i>	Tracks mouse events like clicks, drags, and movements, providing input or selecting and moving pieces.	<div>Mouse</div> <ul style="list-style-type: none"> + x: int + y: int + pressed: boolean <ul style="list-style-type: none"> + mousePressed(MouseEvent e): void + mouseReleased(MouseEvent e): void + mouseDragged(MouseEvent e): void + mouseMoved(MouseEvent e): void 	
Piece Package			

<i>Piece</i>	Serves as the base class for all chess pieces. It provides common attributes (e.g. position, color) and methods for movement validation, drawing, and board interactions. It also includes <i>flipped</i> attribute to track image orientation and a <i>flipImage</i> method for flipping the piece's image vertically.	<div>Piece</div> <div> +image: BufferedImage +x: int +y: int +row: int +col: int +preRow: int +preCol: int +color: int +currentColor: int +hittingP: Piece +moved: boolean +moveCount: int +flipped: boolean </div> <div> + incrementMoveCount(): void + Piece(color: int, row: int, col: int) + getImage(imagePath: String): BufferedImage + getColor(): int + getX(col: int):int + getY(row: int): int + getCol(x: int): int + getRow(y: int): int + getIndex():int + updatePosition(): void + resetPosition(): void + canMove(targetRow: int, targetCol: int): boolean + isWithinBoard(targetRow: int, targetCol: int): boolean + isSameSquare(targetRow: int, targetCol: int): boolean + getHittingP(targetRow: int, targetCol: int): Piece + isValidSquare(targetRow: int, targetCol: int): boolean + piecesOnStraightLine(targetRow: int, targetCol: int): boolean + piecesOnDiagonalLine(targetRow: int, targetCol: int): boolean + flipImage(): void + draw(g2: Graphics2D): void + flipPieces(pieces: ArrayList<Piece>): void </div>	
<i>Biz</i>	A specific chess piece with unique movement rules	<div>Biz</div> <div> + Biz(color: int, row: int, col: int) + canMove(targetRow: int, targetCol: int): boolean </div>	
<i>Ram</i>	A chess piece that can reverse its direction when reaching the opponent's side	<div>Ram</div> <div>- isReversing: boolean</div> <div> + Ram(color: int, row: int, col: int) + canMove(targetRow: int, targetCol: int): boolean + updatePosition(): void </div>	
<i>Sau</i>	Represents the “king” of this chess game. Implements movement rules to allow small-range movements.	<div>Sau</div> <div> + Sau(color: int, row: int, col: int) + canMove(targetRow: int, targetCol: int) </div>	
<i>Tor</i>	Represents a chess piece with a rook-like movement. After moving twice, it transforms into an <i>Xor</i> piece.	<div>Tor</div> <div> +Tor(color: int, row: int, col: int) + canMove(targetRow: int, targetCol: int): boolean + updatePosition(): void - transformtoXOR(): void </div>	

2.3 Classes Utilizing Design Patterns

Design Patterns	Class	Description/Reason
Main Package		
Singleton	<i>Main</i>	For ensuring a single application instance via <i>JFrame</i> .
Controller Pattern	<i>Controller</i>	Manages the flow between UI components and the underlying game logic. It coordinates tasks such as saving, loading games, handing UI transitions, and managing player states.
Observer	<i>GamePanel</i>	Listens for mouse events and updates the game's state. For example, selecting a piece and making move.
Command Pattern	<i>GamePanel</i>	Includes <i>showWinnerScreen</i> , which encapsulates the logic for displaying the winner's UI.
Adapter	<i>Mouse</i>	The <i>Mouse</i> class extends <i>MouseAdapter</i> to simplify handling mouse events (<i>mousePressed</i> , <i>mouseReleased</i> , etc.) without implementing every method of the <i>MouseListener</i> and <i>MouseMotionListener</i> interfaces.
Piece Package		
Template Method Pattern	<i>Piece</i>	Provides a general structure for <i>canMove</i> and other behaviors, allowing subclasses to implement specific movement rules.
Factory Pattern	<i>Piece</i>	Serves as the base class for specialized pieces which represent concrete implementations.
State Pattern	<i>Ram</i>	Tracks whether the piece is in a “reversing” state and updates its movement rule accordingly.
	<i>Tor</i>	Tracks the number of moves and transforms into an <i>Xor</i> piece after two moves.
	<i>Xor</i>	Tracks the number of moves and transforms into a <i>Tor</i> piece after two moves.
Decorator	<i>Tor</i>	After two moves, <i>Tor</i> transforms into an <i>Xor</i> piece, effectively decorating the original <i>Tor</i> behavior with the movement rules of <i>Xor</i> .
	<i>Xor</i>	After two moves, <i>Xor</i> transforms into a <i>Tor</i> , effectively decorating the original <i>Xor</i> behavior with the straight-line movement rules of <i>Tor</i> .

3. Use Case Diagram

3.1 Use Case Diagram



3.2 Use Case Description

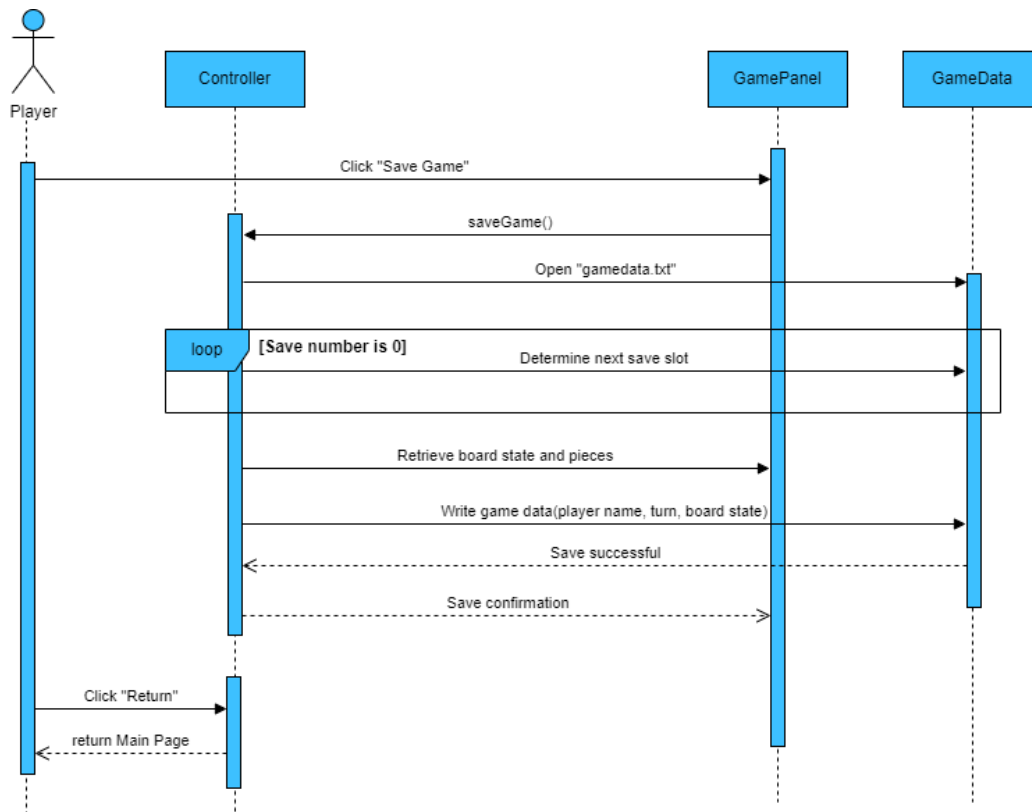
Use Case	Description
----------	-------------

Save Current Game	The player can save their current state game progress if they wish to continue to play another time.
Delete Save Game	The player can delete the save game from the list of available save slot games.
Load Save Game	The player can load their game save files to continue to play chess games that they have not finished.
Start New Game	The player can start a new game where the board and pieces are in their initial state. The player also can enter their own name to represent their own color to play in the chess. Players who play as the chosen color only can select the chosen color pieces.
Display Board and Pieces	The system will display the board and all pieces during gameplay, updating dynamically after every move, save, or load action.
Move Pieces	The player can drag the pieces and move them to a valid tile on the board. This action includes validation of the moves and updates the game state.
Simulate Move	The system validates and simulates the move by checking the tile's validity, resolving interactions with other pieces, and handling captures, including ending the game if the "Sau" is captured.
Switch Turn	The system will automatically change the turn position to the opposing player after a successful move. It will also check and upgrade eligible pieces (Xor to Tor or Tor to Xor) if they have completed 2 turns.

4. Sequence Diagram

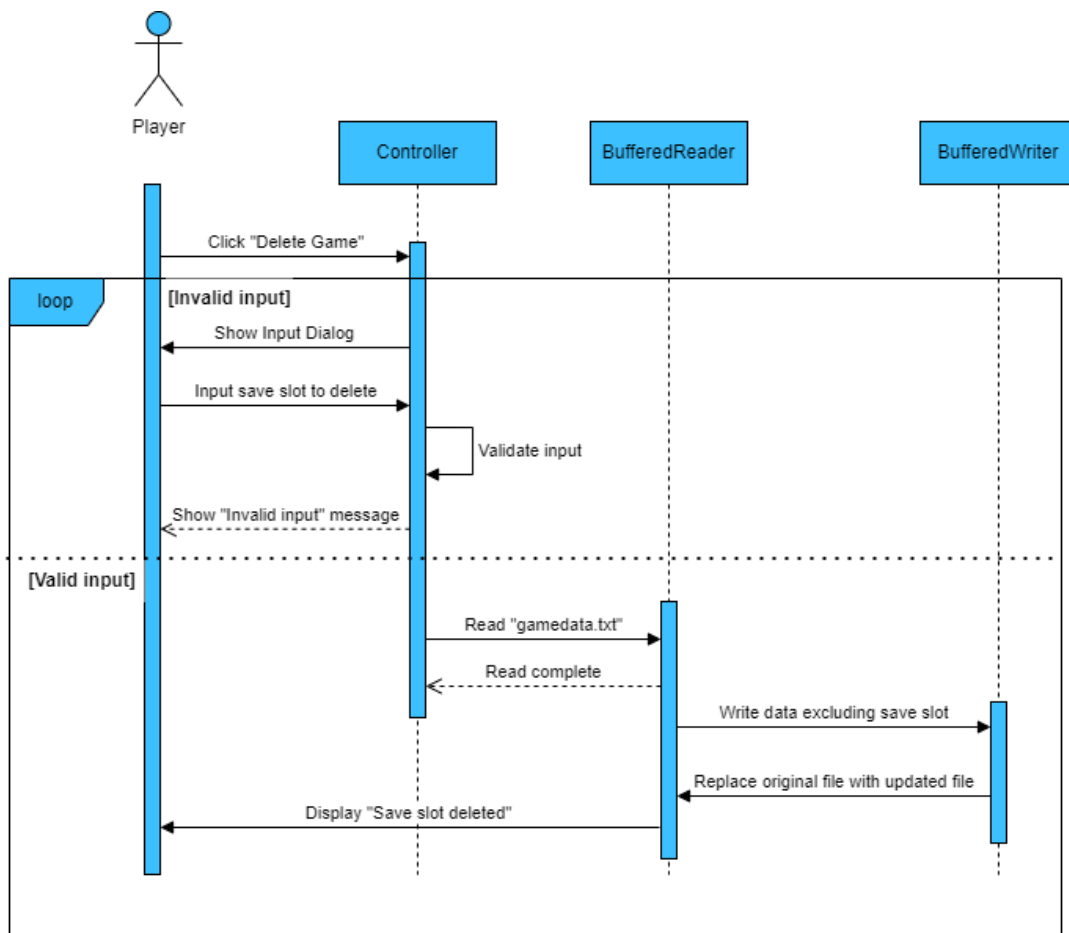
4.1 Sequence Diagram Save Current Game

The player initiates the "Save Game" action that will be handled by GamePanel and sends it to Controller via saveGame() method. The Controller opens the gamedata.txt file from GameData and checks if the save number is 0. If true, the system determines the next available save slot. Once the slot is determined, Controller receives the current board state, player names, and turns information from the GamePanel. The data is written to gamedata.txt file to ensure the game's is saved. After the save operation is completed, the player can select "Return", and the system will navigate back to the main page.



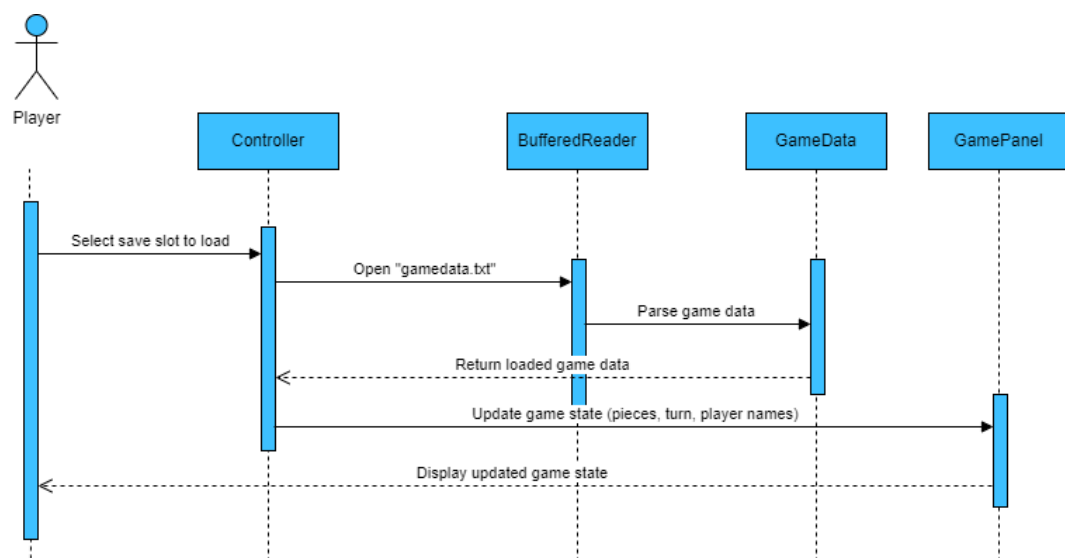
4.2 Sequence Diagram Delete Save Game

The player has done the "Delete Game" action, which prompts the Controller to display an input dialog for player to specify the save slot to delete. The Controller validates the input, and if it is invalid, an error message is shown to player, and the process terminates. If the input is valid, the Controller reads the game data from the gamedata.txt using BufferedReader, excluding the specified save slot. Once the reading is complete, the Controller writes the updated data back to the file using BufferedWriter, replacing the original file. Finally, the system confirms the deletion by displaying the "Save slot deleted" message to the player.



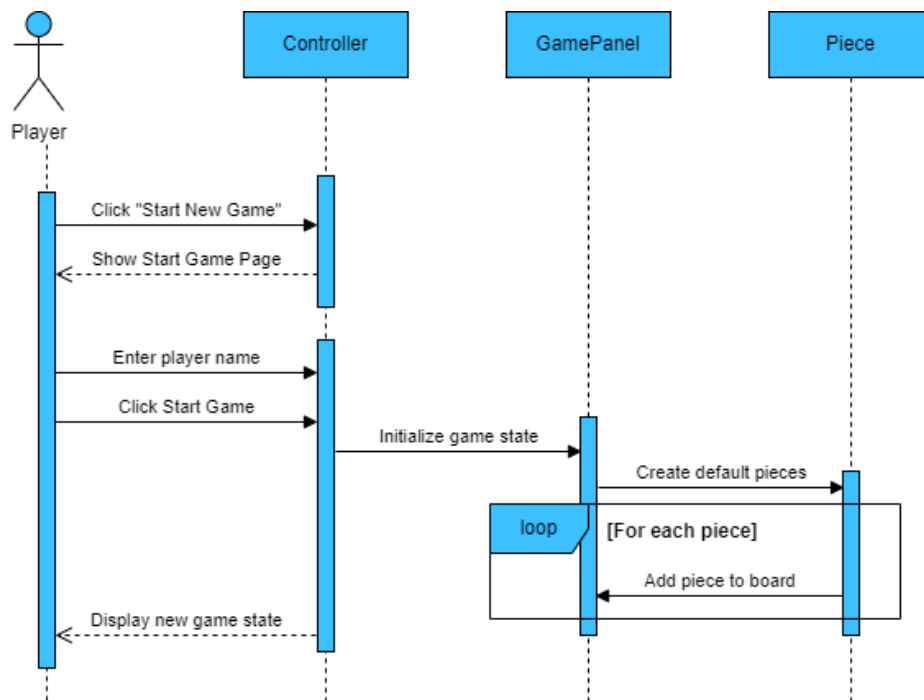
4.3 Sequence Diagram Load Save Game

The player selects the save slot to load, which will trigger the Controller to open gamedata.txt file using BufferedReader. The system then parses the game data for selected save slot to retrieve save pieces, turn, and player names from GameData. Once the data is successfully retrieved, the Controller updates the game state in the GamePanel with loaded pieces, turn and player information. Lastly, the updated game state is displayed to the player, reflecting the restored save.



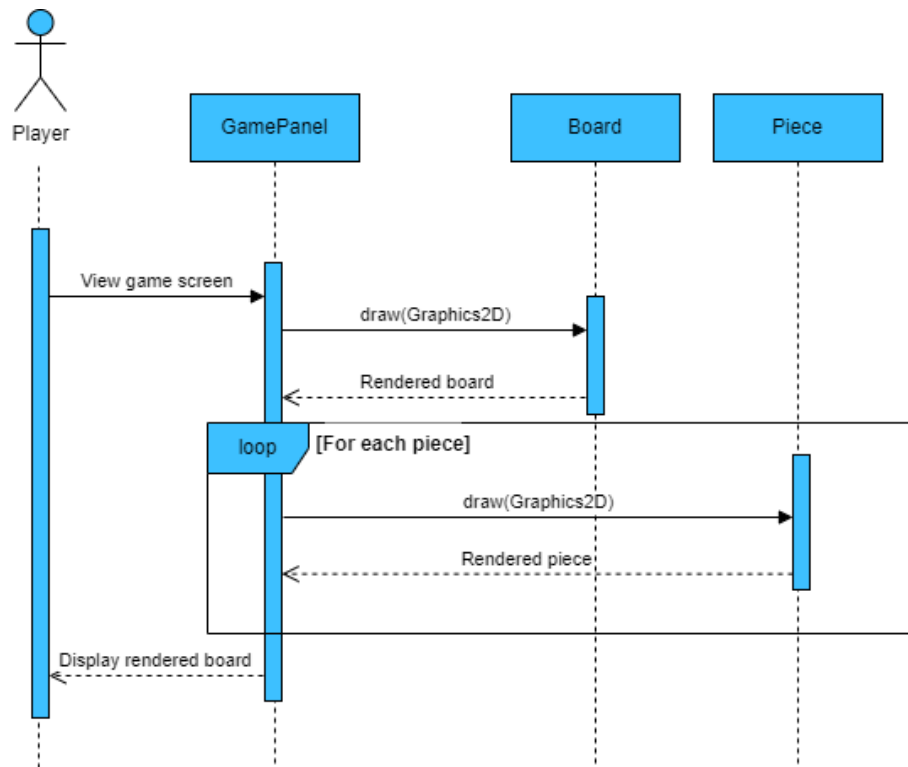
4.4 Start New Game

The player initiates the action by clicking "Start New Game", which will prompt the Controller to display the start game page. The players will enter their name and click "Start Game". The controller then initializes the game state by invoking the GamePanel. The GamePanel creates default pieces and iteratively adds each piece to the board using a loop. Once the initialization is completed, the system displays the new game state to players.



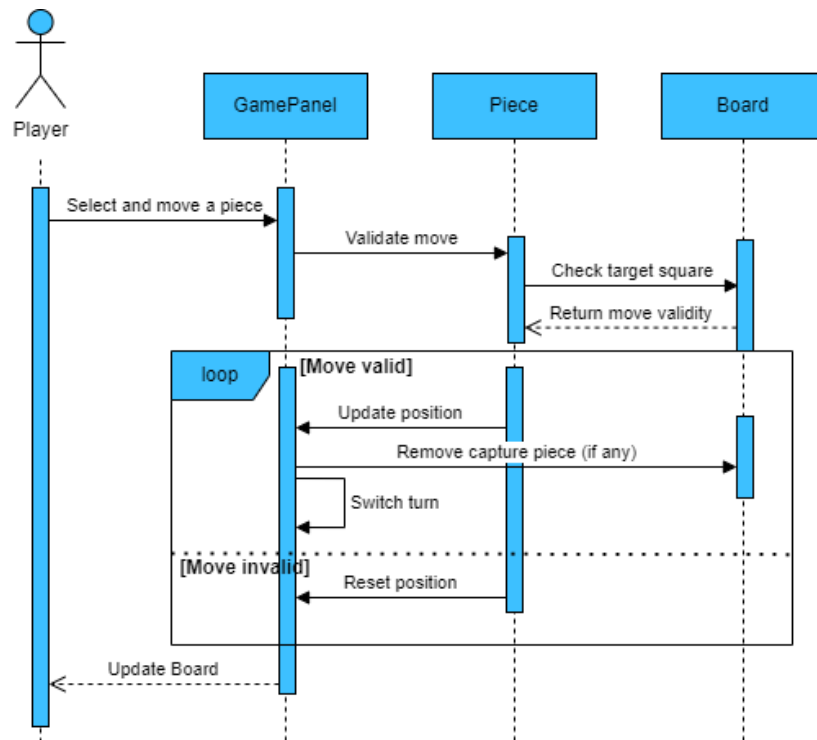
4.5 Display Board and Pieces

The player views the game screen, prompting the GamePanel to initiate the rendering process. The GamePanel first calls the Board to draw the board using the draw(Graphics2D) method, calling the draw(Graphics2D) method on each Piece to render them. Each rendered piece is returned to GamePanel. Once all pieces are drawn, the rendered board, along with pieces is displayed to players.



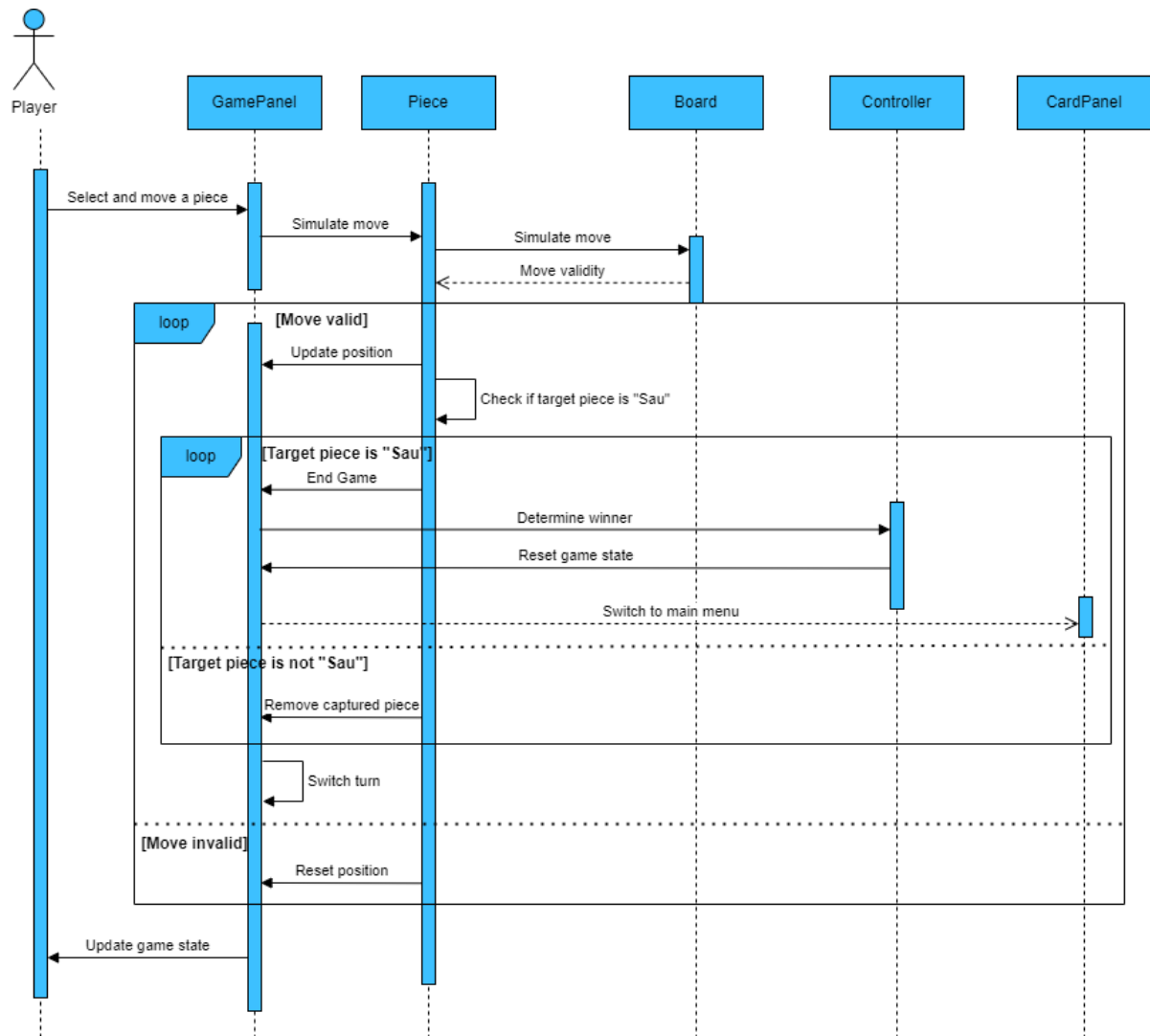
4.6 Move Pieces

Diagram illustrates the process of moving a piece where the player will select and move piece, which later will trigger the GamePanel to validate the move by interacting with the Piece. The Piece will check the target square with the Board and receive the move's validity. If the move is valid, the system will enter a loop to handle the move. The Piece updates its position, and any captured pieces are removed from the board. The system then switches the turn to another player. If the move is invalid, the Piece resets its position to previous state. The GamePanel updates the board to reflect the changes and displays the updated game state to the player.



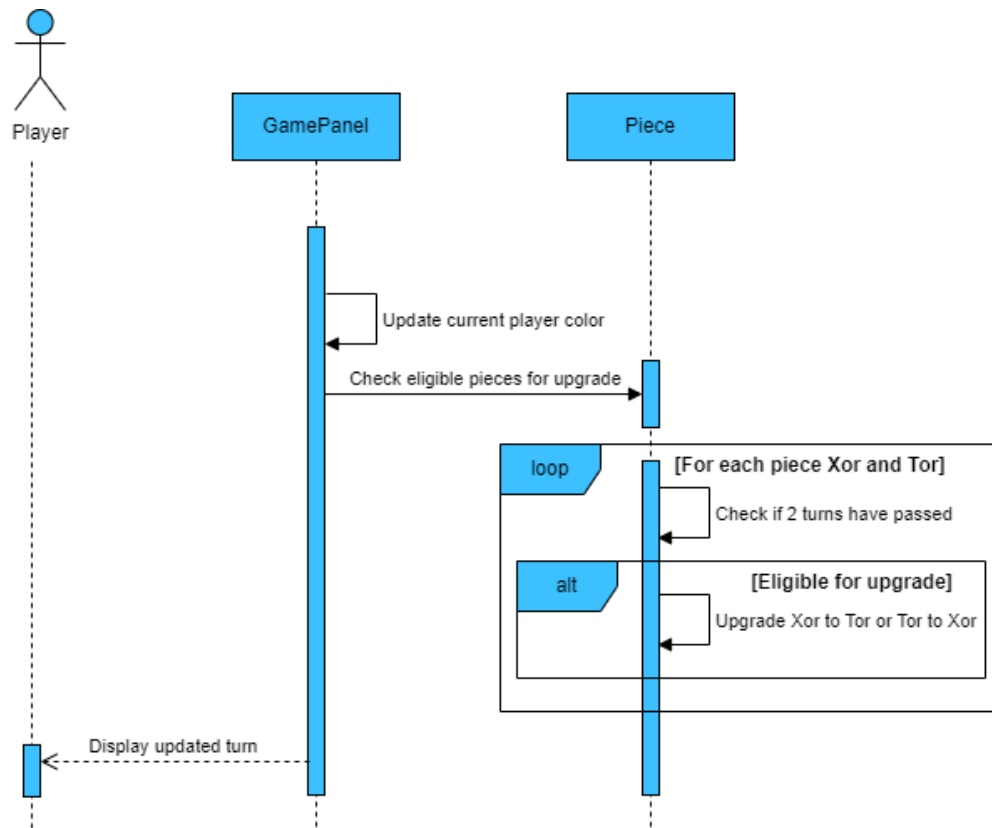
4.7 Simulate Move

The player selects and moves a piece, prompting the GamePanel to simulate the move by interacting with Piece and Board. The Board will validate the move and return the result. If the move is valid, the system will update the Piece position and check whether the target piece is a "Sau". If the target piece instructs the CardPanel to switch to the main menu. If the target piece is not "Sau", the captured piece is removed, and the turn is switching to opposite player. If the move is invalid, the Piece resets its position. Lastly, GamePanel updates the game state and displays the results to the player.



4.8 Switch Turn

The sequence diagram illustrates the process of switching turns while checking for piece upgrades. The GamePanel updates the current player's turn by switching to the opposing player. Then, it will check for Xor and Tor pieces for eligibility to upgrade. A loop shown through each piece, and for every piece, the system checks if two turns have passed. If a piece is eligible for an upgrade, an alternate flow is triggered, upgrading the piece from Xor to Tor or Tor to Xor. Once all checks and upgrades are completed, the GamePanel displays the updated turn to the player.



5. User Documentation

5.1 Introduction

Kwazam Chess is a strategy board game application built using Java with graphical user interface (GUI) by using design pattern of Model-View-Controller. This game does not implement with a computer player, it can just be a two human player game. This Kwazam Chess implements custom chess-like mechanics with features such as player turn, piece moving, flipping board, saving/loading games and user-friendly navigation.

5.2 User Interface Overview

There are 4 screen that we implement for this Kwazam Chess:

1. Main Screen
2. Start New Game Screen
3. Kwazam Chess Board Game Screen
4. Load Save and Delete Game Screen
5. Winner Declaration Screen

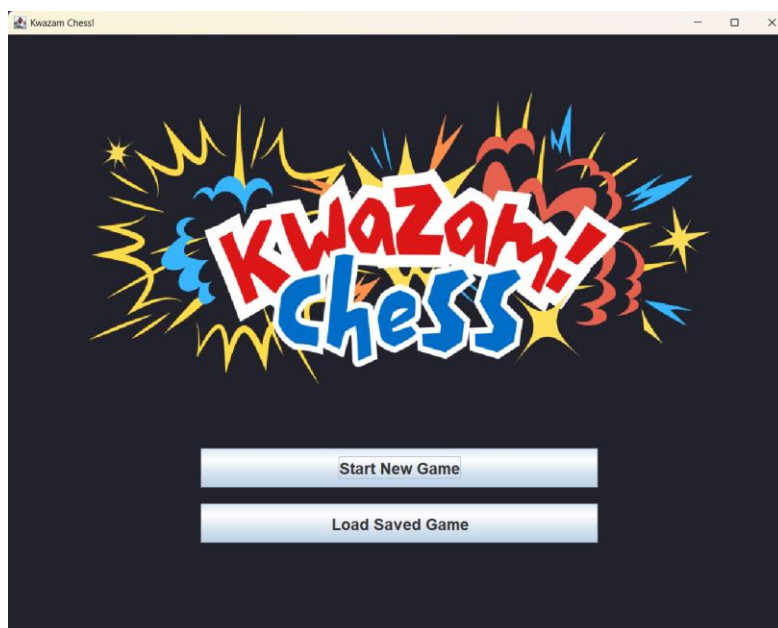
5.2.1 Main Screen

The Main Screen is a screen where the player can choose either to Start New Game or Load Saved Game.

The Main Screen contains 2 buttons:

1. Start New Game
This button will navigate the player to enter Start New Game Screen.
2. Load Saved Game
This button navigates the player to enter Load Saved Game Screen

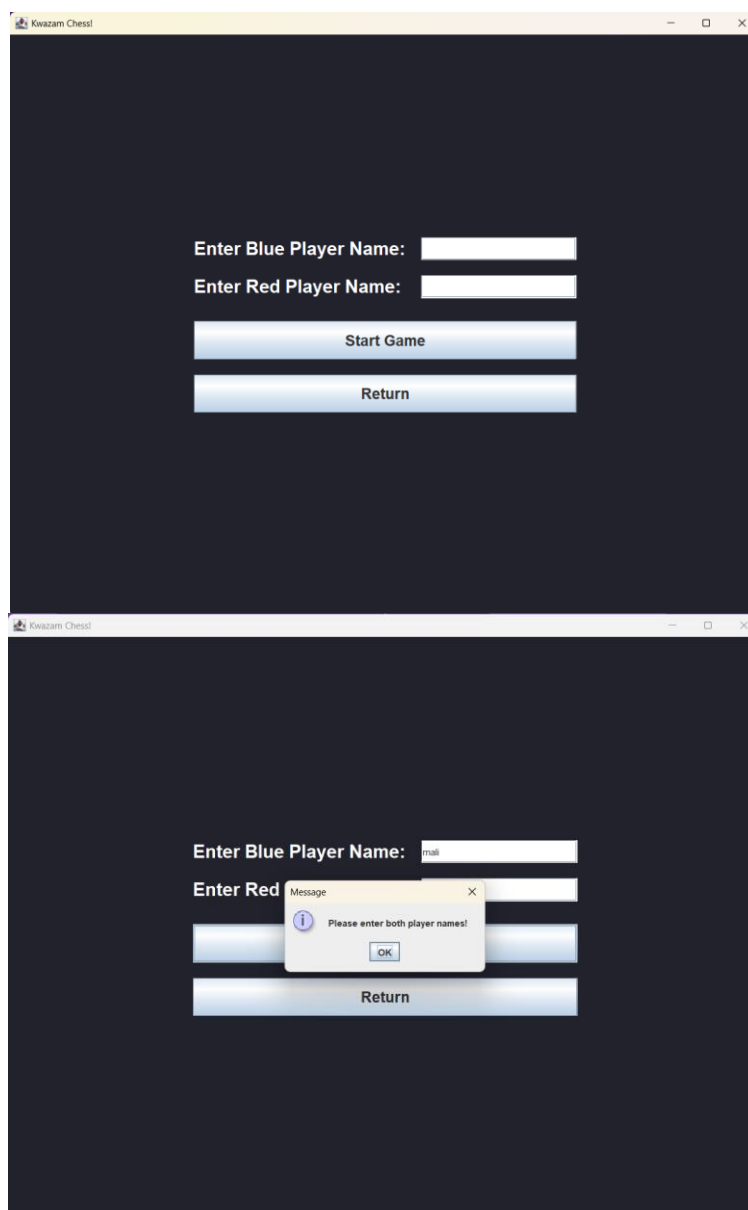
This allows friendly navigations for players to choose whether to start a new game or load saved game. Below is shown an illustration of the Main Game Screen.



5.2.2 Start New Game Screen

The Start New Game Screen for the Kwazam Chess allows players to enter their names and start a new game. Contains 2 text fields and 2 buttons.

1. Enter Blue Player Name Text Field
The name for the player controlling the blue pieces
2. Enter Red Player Name Text Field
The name for the player controlling the red pieces
3. Start Game Button
Clicking this button starts the game with the names entered in the text field. If either name or both names is left blank, the game will show the messagebox to prompt the user to fill both fields.
4. Return Button
Clicking this button returns the players to the Main Screen.

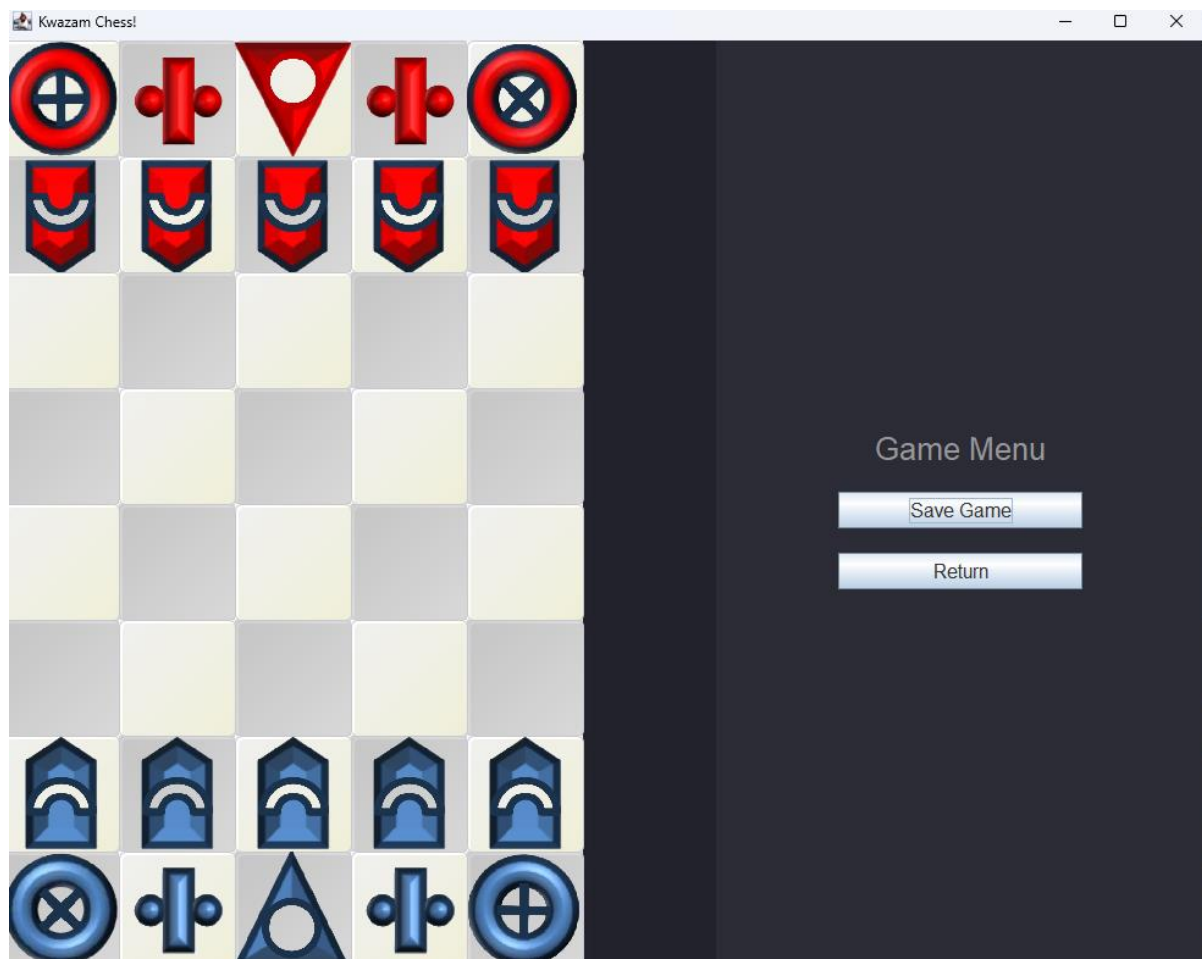


5.2.3 Kwazam Chess Board Game Screen

This is the Kwazam Chess Board Game Screen, where the main gameplay takes place. This allows 2

players to play the Kwazam Chess. It has a chessboard, game pieces, a side menu containing buttons and player information.

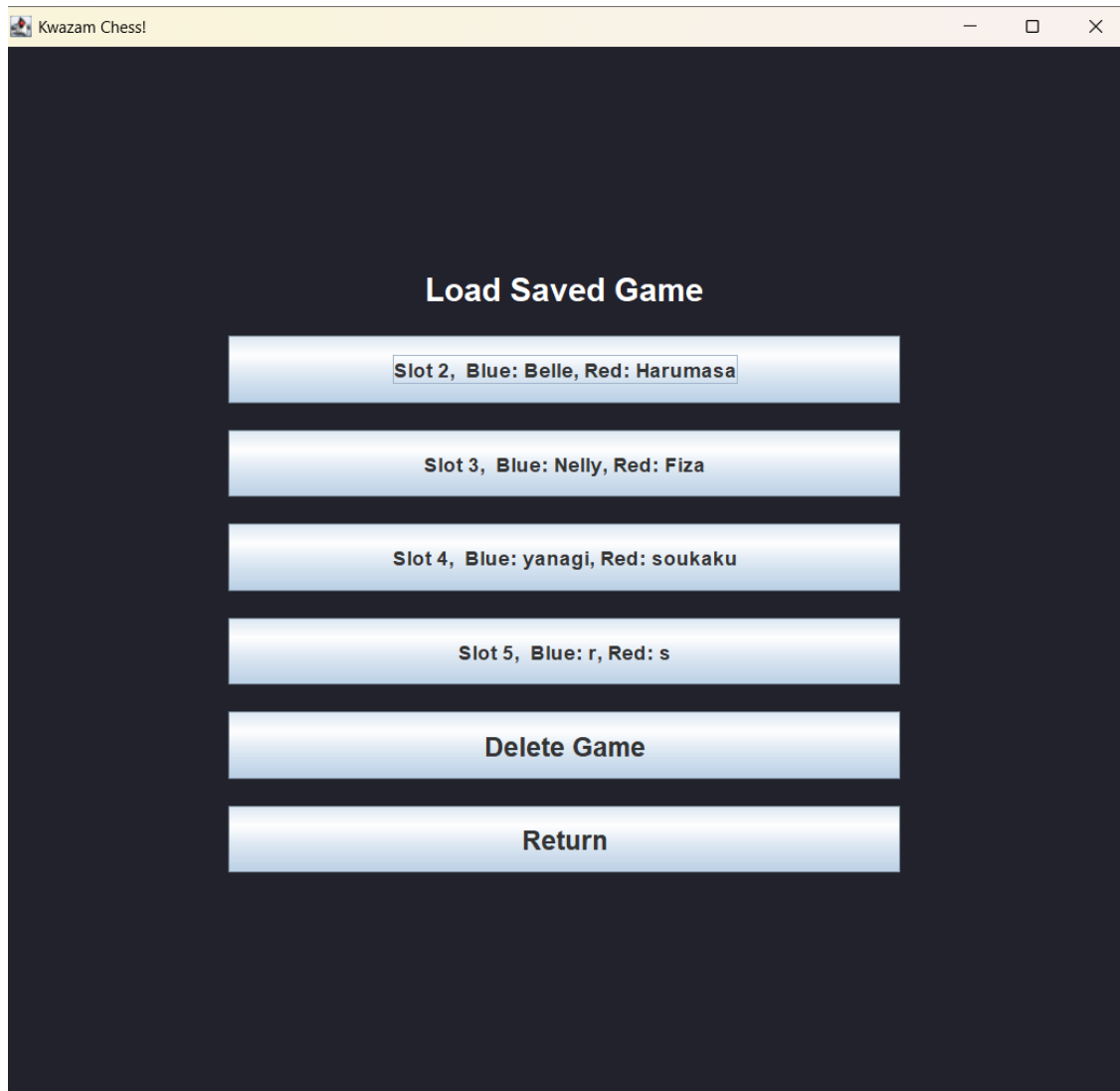
1. Chessboard
The board has an 8x5 grid layout alternating light yellow and light grey tiles. This grid serves as the playing area for the pieces.
2. Game Pieces
Red Pieces belongs to the Red Player and is initially positioned at the top of the board. Blue Pieces belongs to the Blue Player and is initially positioned at the below of the board. It contains 5 types of pieces, and each has unique shapes to represent their type.
3. Save Game Button
Saves the current state of the game to the gamedata.txt file.
4. Return Button
Takes the players back to the main menu.
5. Player Information
Display the names of the players that represent their color of pieces.

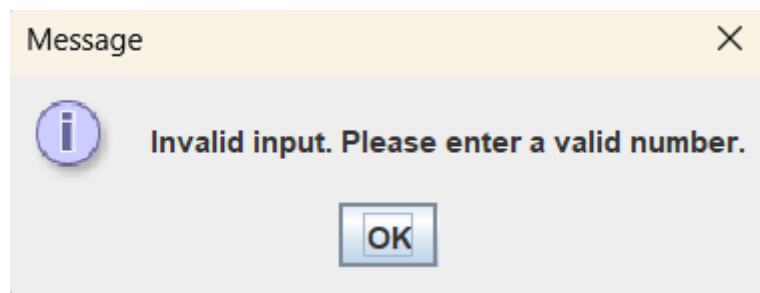
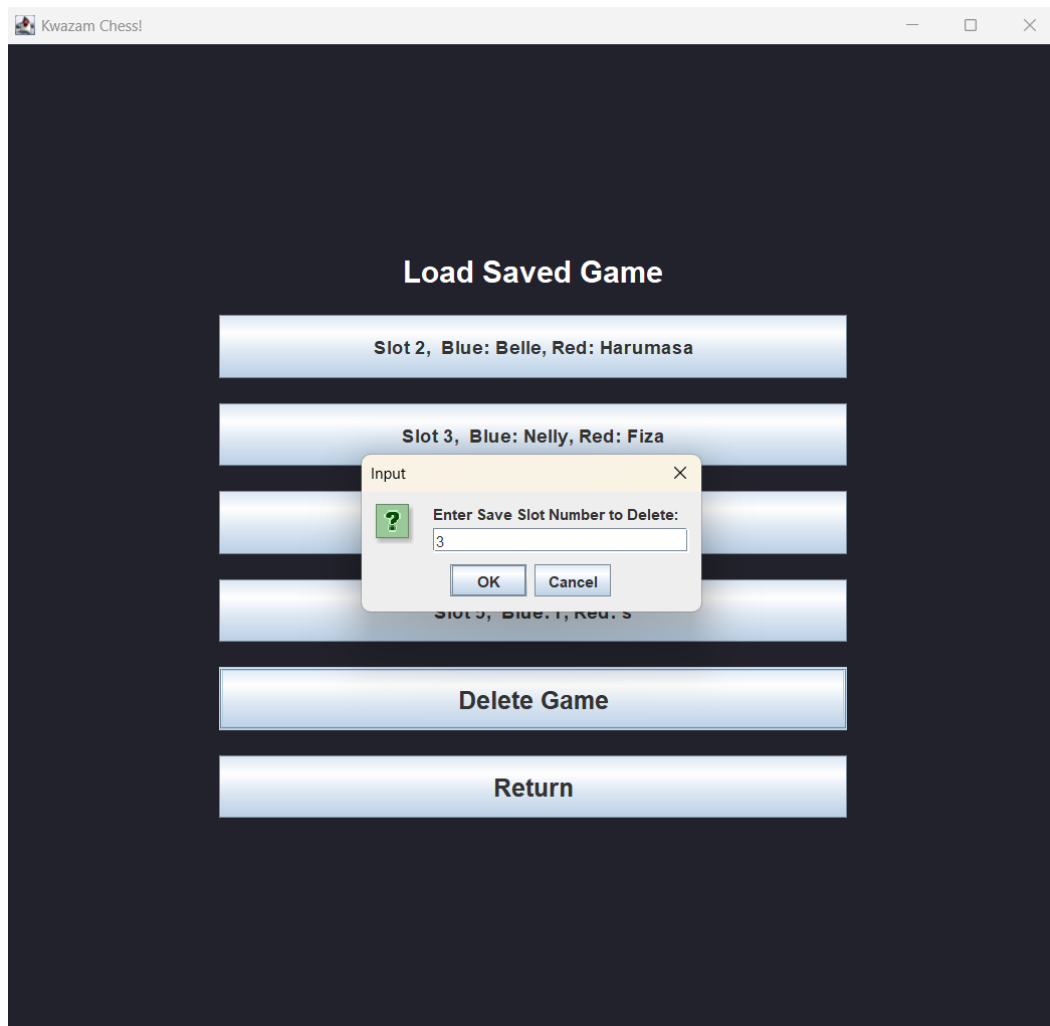


5.2.4 Load Saved and Delete Game Screen

This screen allows to load save games and delete games from the save slot. It allows players to manage and resume previously saved games. Contains buttons to allow players to resume their save slot game, delete game and return button.

1. Save Slots Button
Each button displayed for each available saved slot and labeled with slot number, blue player name and red player name. (E.g Slot 3, Blue: Nelly, Red: Fiza)
2. Delete Button
Opens the input dialog to delete a specific save slot by entering its slot number only. If the player enters the wrong number or something other than a number (e.g. alphabet, mathematical symbol), it will show message to enter a valid number.
3. Return Button
Takes the players back to the main menu.





5.2.5 Winner Declaration Screen

This screen enhances the user experience by celebrating the winner visually and offering an easy way to return to the main menu. The main purpose of this screen is to show the winner of the game using an image (e.g., "bluwins.png" or "redwins.png"). It provides a visual representation of the game's outcome. Besides, it provides a button to allow the players to return to the main menu after the winner is declared, effectively ending the game session.



5.3 Gameplay Mechanics

Kwazam features unique and interactive gameplay mechanics designed to provide an engaging

experience. Below is a detailed description of how the game operates.

5.3.1 Drag-to-Move Functionality

This shows an explanation on how the selecting a piece, valid moves highlighted and dropping the Piece.

1. Selecting a Piece

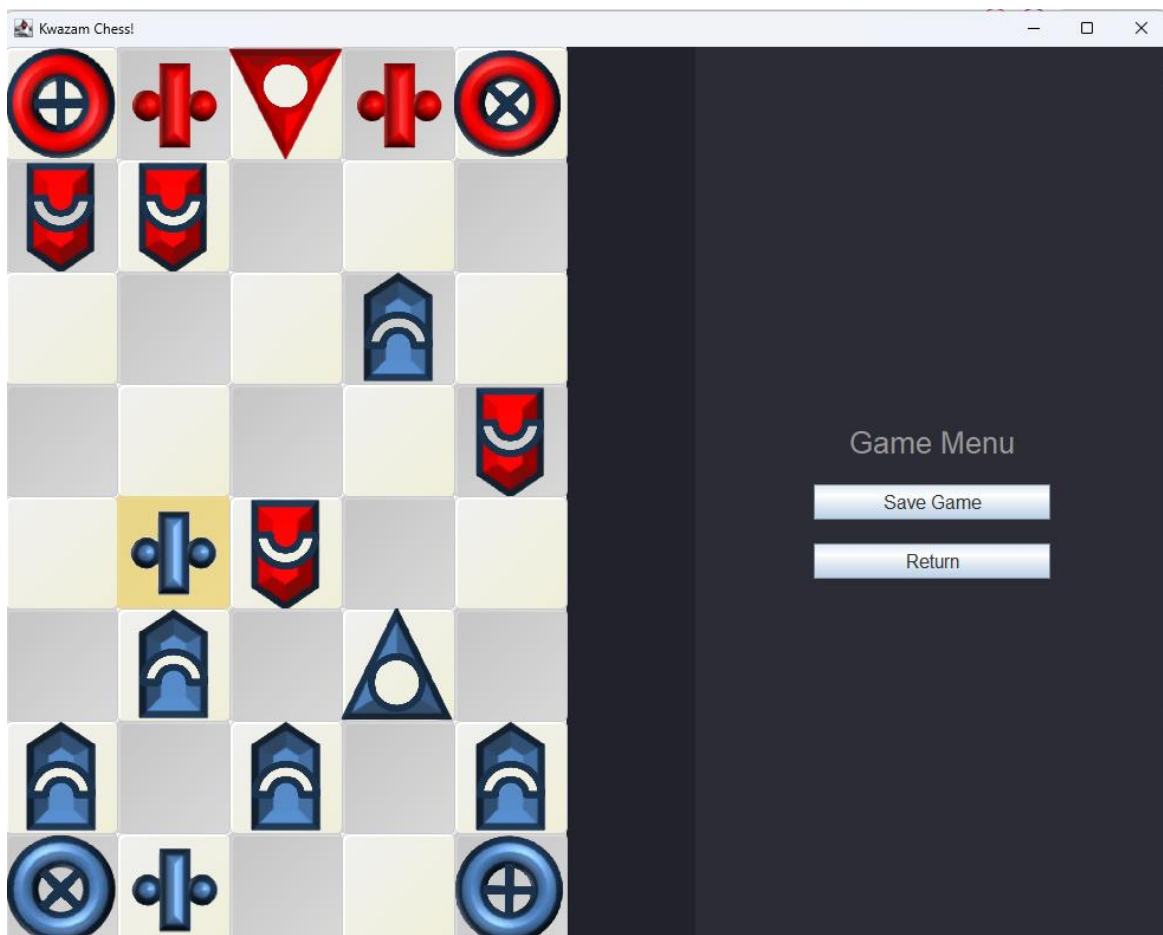
The player will press the piece to select the desired piece then the selected piece becomes the active piece. The piece will follow the player's cursor as they drag it across the board.

2. Valid Moves Highlighted

While the piece dragged on, the game highlights valid squares for movement. The valid squares depend on the piece's movement rules and current state of the board.

3. Dropping the Piece

If the player releases the piece on valid square, the piece is moved to that square and any opponent piece on the destination square is captured and removed from the board. If the player moves are invalid, the piece will snap back to its original position.



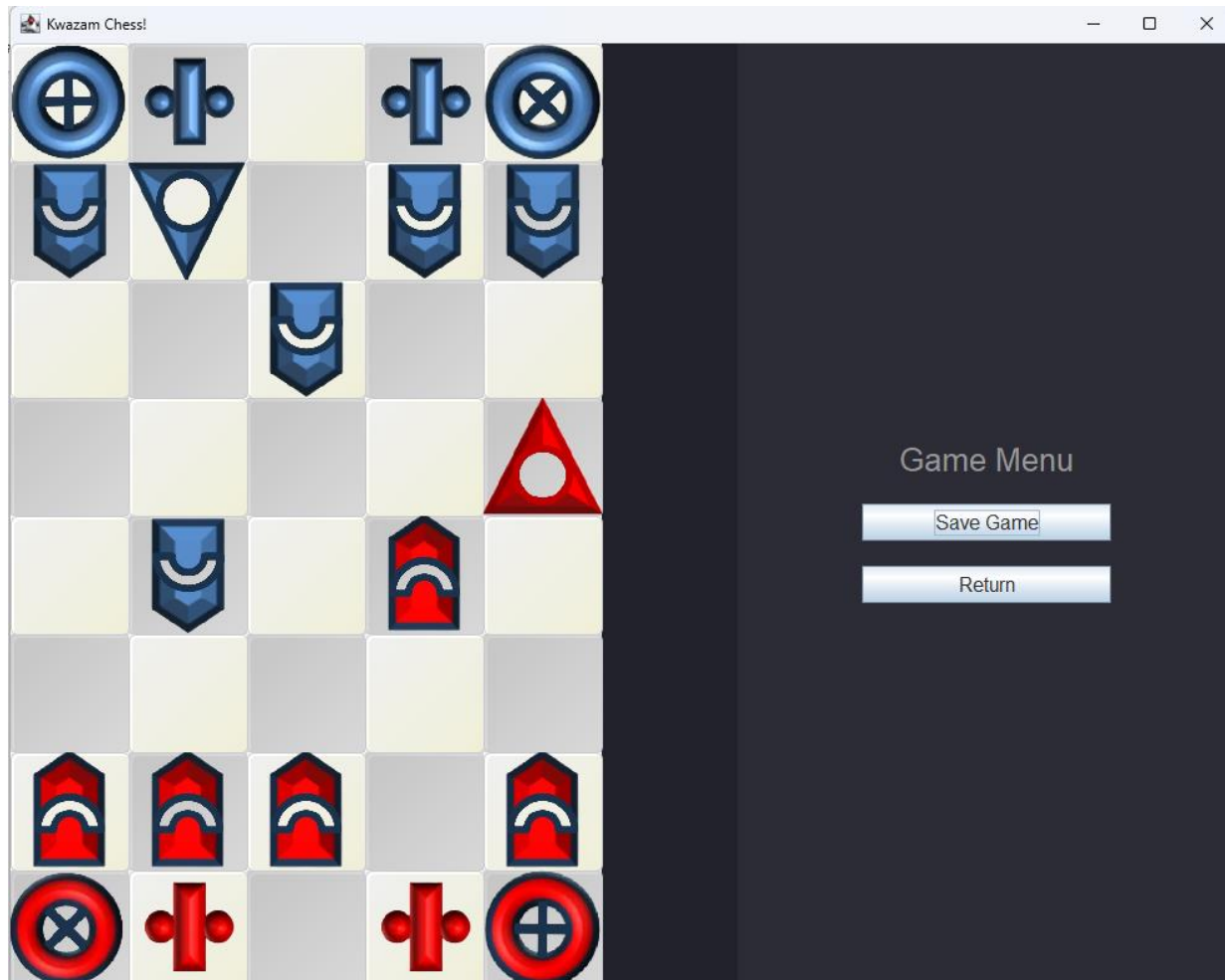
5.3.2 Board Flipping Mechanics

This shows an explanation on how the flipping board will occur in this game after one player finished their turn.

After the player make a valid move:

1. The player switches to the opponent.
2. The board will flip 180 degrees, so the next player sees their original pieces oriented at the bottom.
3. The individual image of each chess piece will flip vertically.
4. The positions and coordinates of each chess Piece are flipped. So, if previously the piece is at position (0,1), then the new position will be (7,4)






If the player were to “Save Game” after flipping, it will update the new flipped positions of chess pieces Board State in gamedata.txt.



5.4 Piece Movement Rules

There are 5 unique pieces in this Kwazam Chess for each player. Total for all these pieces that are available on the board for both layers is 10 pieces.

Piece Name	Picture	Movement	Unique Feature
------------	---------	----------	----------------

Ram		Piece can only move forward with one step.	If it reaches the end of board, it turns around and starts heading back the other way.
Biz		Piece move in a 3x2 L shape in any orientation.	The only piece that can skip over other pieces.
Tor		Piece move orthogonally only but can go any distance as long as it does not skip over other pieces.	After 2 turns, it transforms into Xor piece including the image of piece and behavior movement will also like Xor
Xor		Piece move diagonally only but can go any distance if it does not skip over other pieces	After 2 turns, it transforms into Tor piece including the image of piece and behavior movement will also like Tor
Sau		Piece move only one step in any direction.	The game ends when the Sau is captured by the other side.

5.5 Game Instructions

On this section will be explaining a simple, step-by-step guide to help players understand how to play Kwazam Chess. The game instruction is:

1. Start the Game
The Blue Player will always make their first move first. Therefore, later the Red Player will take its first move.
2. Select a Piece
A player can only select their own pieces where the Blue Player controls blue pieces and Red Player controls red pieces. By clicking on a piece to select pieces and the selected pieces will follow the mouse cursor.
3. Move the Piece
Move the selected piece into a valid square that will be highlighted on the board, it the player dragged into correct square. Later, release the mouse button to complete the move.
4. Invalid Moves
If the player attempts to drop the piece on an invalid square, it will return to its original position.
5. Capturing Opponents Pieces
If a player moves their pieces onto a square occupied by an opponent's piece, that piece is captured and removed from the board.
6. Switching turn
After a valid move being made by the player, the turn swathes to the opponent, the board will be flipping 180 degrees for their perspective.
7. Winning the Game
A player will win the game by capturing the opponent's Sau pieces. Piece will be declared the winner. This causes the game to end and the opponent of the player who resigns will be declared the winner.
8. Saving Game Progress
A player can select the "Save Game" button, and it will create a save game in the data. However, if the game has already been saved once before, it will update information of current Board State. Even if the player resigns from the game, after that they can load the saved game to play again.
9. Resign from Game
A player can select the "Return" button, and it will redirect to the Main Menu. If Exit is clicked, it will close the whole game.

5.6 File Management Information

5.6.1 Save file format

As mentioned above, the save game will save the current state of the pieces and all the information on the board, to be able to load new save game for save slot. Each save is divided into sections with a consistent format.

1. Save Slot Number

Shows the slot number of the saved game

Example:

```
Save Game : 3
```

2. Player Turn

Specify which player's turn it is to play. This will indicate by using the color of the player's possession.

Example:

```
Player Turn : BLUE
```

3. Player Names

Lists the names of the Blue and Red Players

Example:

```
Blue Player : Nelly  
Red Player : Fiza
```

4. Board State

An 8x5 grid representing the positions of all pieces on the board and each cell contains an empty square, piece code with format [Color] [Piece Type]

Example:

```
.  
RTOR RBIZ RSAU RBIZ RXOR  
RRAM RRAM ---- ---- ----  
---- ---- ---- BRAM ----  
---- ---- ---- ---- RRAM  
---- ---- RRAM ---- ----  
---- BRAM ---- BSAU ----  
BRAM ---- BRAM ---- BRAM  
BXOR BBIZ ---- BBIZ BTOR
```

Below is an example of overall gamedata.txt format with 3 saved game data is stored in it.

```

main > ≡ gamedata.txt
1  Save Game : 1
2  Player Turn : BLUE
3  Blue Player : Belle
4  Red Player : Harumasa
5  .
6  RTOR RBIZ RSAU RBIZ RXOR
7  RRAM RRAM RRAM RRAM RRAM
8  ---- ---- ---- ---- ----
9  ---- ---- ---- ---- ----
10 ---- ---- ---- ---- ----
11 ---- ---- ---- ---- ----
12 BRAM BRAM BRAM BRAM BRAM
13 BXOR BBIZ BSAU BBIZ BTOR
14
15 Save Game : 2
16 Player Turn : BLUE
17 Blue Player : Nelly
18 Red Player : Fiza
19 .
20 RTOR RBIZ RSAU RBIZ RXOR
21 RRAM RRAM ---- ---- ----
22 ---- ---- ---- BRAM ----
23 ---- ---- ---- ---- RRAM
24 ---- ---- RRAM ---- ----
25 ---- BRAM ---- BSAU ----
26 BRAM ---- BRAM ---- BRAM
27 BXOR BBIZ ---- BBIZ BTOR
28
29 Save Game : 3
30 Player Turn : BLUE
31 Blue Player : yanagi
32 Red Player : soukaku
33 .
34 RTOR RBIZ RSAU RBIZ ----
35 RRAM ---- RRAM ---- RXOR
36 ---- RRAM ---- RRAM RRAM
37 ---- ---- ---- ---- ----
38 ---- ---- ---- BRAM ----
39 ---- ---- BRAM ---- BRAM
40 BRAM BRAM ---- ---- ----
41 BXOR BBIZ BSAU BBIZ BTOR

```

5.6.2 Usage in the Kwazam Chess Game

The usage of this gamedata.txt is to save a game. When the player saves, the game will write the current state into this file as a new save slot. Secondly, is to load a game. When the game reads the selected slot and reconstructs the board, player names, and turn. Lastly, is for deleting a save slot. The specific slot can be deleted by removing the corresponding sections from this file.