CCP6224 Project

Trimester 2430

by Group T16L C

Team Leader: Mohammed Amena Mohammed Abdulkarem, 01162237057, 1221305728@student.mmu.edu.my

Team members:

Mohammed Yousef Mohammed Abdulkarem, 01162238871, 1221305727@student.mmu.edu.my

Gavin Yee Hou Jien, 01110672263, 1221103458@student.mmu.edu.my

Muhammad Faiz bin Ilyasa, 0122462882, 1221309435@student.mmu.edu.my

Table of Contents

1. Introduction

Overview

Kwazam Chess is a special kind of chess that is played on a 5x8 with specialized pieces that move and change according to their own rules. This project focuses on developing a Java graphical user interface program where two players can play against each other in a fun and interactive way.

The application follows the Model-View-Controller (MVC) architecture, a pattern that supports a clear separation of concerns for a modular, maintainable, and scalable codebase. Furthermore, advanced object-oriented programming (OOP) principles and design patterns, including Singleton, Observer, and Template Method are incorporated to enhance flexibility, reusability, and extensibility.

The primary objective of this project is to develop a complete and fully functional chess game to be played by players of all skill levels following the rules of Kwazam Chess, while also following the better practices in software design and development. Key features of the application include:

- **Interactive Gameplay:** The game allows the players to make their moves by clicking on the board but the game is programmed to give visual feedback on validated and rejected moves, depending on position, and as well help in selecting a piece.
- **Save and Load Functionality:** Players can save their current game state and resume it at a later time.
- **Dynamic Board Flipping:** The board automatically flips to provide the correct perspective for each player.
- **Resizable Window:** The application supports adjustable window sizes to accommodate player preferences.

These features are designed to provide an intuitive and enjoyable user experience, enabling players of all abilities to interact with the game effortlessly.

Design Patterns

To ensure a robust and maintainable design, the following design patterns have been employed:

Model-View-Controller (MVC):

- Divides program codebase structure into three cohesive components: Model (game logic), View (user interface), and Controller (input handling).
- This segmentation ensures that the individual modules are dedicated to their designated functions and consequently makes maintenance and future development easier.

Singleton:

- Ensures that there are no two instances of the `Board` class alive in the application's runtime.

- Provides a single source of truth about the state of the board, preventing inconsistencies.

Observer:

- Allows the GUI to receive updates dynamically using the `GameView` to be notified of any changes in the `Game` state.
- Enables decoupled logic for the game logic and the user interface.

Template Method:

- Defines the structure of the `move` method in the `Piece` class, while allowing subclasses to further customize selected steps based on the piece type.
- Promotes code reuse and maintains consistent standards for all piece types.

Hence, these design models come together to create a dynamic, scalable, and maintainable code with respect to the standards of quality software engineering.

Scope

The scope of this project includes the following key areas:

Game Implementation:

- A 5x8 chessboard with unique pieces (Ram, Biz, Tor, Xor, Sau) and their respective movement rules.
- Transformation logic for Tor and Xor pieces after every two turns.
- Win/loss conditions based on the capture of the Sau piece.

User Interface:

- A graphical user interface (GUI) using Java Swing.
- Interactive board with click handling functionality.
- Visual feedback for valid moves and piece selection.

Game Management:

- Save and load game functionality using human-readable text files.
- Support for resizable windows and board flipping for player convenience.

Design Patterns:

- Model-View-Controller (MVC) pattern for separating game logic, UI, and control flow.
- Singleton pattern to maintain a single instance of the game board.
- Observer pattern to dynamically update the GUI based on game state changes.
- Template Method pattern to define the structure of the *`move`* algorithm in the *`Piece`* class.

Documentation:

- Complete documentation for all classes, methods, and design patterns used.
- UML diagrams, use case diagrams, and sequence diagrams to illustrate the system design.

Project Objectives

- To accurately implement all game rules, movement mechanics, and piece transformations according to Kwazam Chess specifications.
- To ensure seamless gameplay, allowing players to compete from start to finish with well-defined win/loss conditions.
- To develop an interactive GUI with click-to-move functionality, visual indicators for valid moves, and real-time feedback.
- To apply object-oriented programming (OOP) principles, including inheritance, encapsulation, polymorphism, and delegation, for a structured and maintainable code.
- To integrate design patterns, such as MCV, Singleton, and Observer to improve scalability and maintainability.
- To enhance user experience with resizable windows, board flipping, and save/load features for accessibility.
- To provide complete documentation, including UML, use case, and sequence diagrams, to support future enhancements.

Overview of the classes

1- Game

Purpose: Manages the core logic and state of the game.

Responsibilities:

- Tracks the current player, game state, and board state.
- Manages turn switching, win/loss conditions, and legal move validation.
- Notifies observers of state changes using the Observer pattern to update the game view.

Key Functionalities:

- Maintains a list of players and their pieces.
- Switches turns between players after each valid move.
- Detects when a player loses (if their Sau piece is captured).
- Provides methods for saving/loading game state.

2- Board (Singleton)

Purpose: Represents the 5x8 chessboard and manages piece placement.

Responsibilities:

- Stores the positions of all pieces on the board.
- Initializes the board and places pieces at the start of the game.
- Provides methods to retrieve and update piece positions.

Key Functionalities:

- Uses the Singleton pattern to ensure only one instance of the board exists.
- Updates the board whenever a piece moves or is captured.
- Resets the board to start a new game.

3- Player

Purpose: Represents a player (Red or Blue) and manages their pieces, moves, and game state.

Responsibilities:

- Manages the player's name, colour, and list of pieces.
- Handles piece movements and checks if the player has lost (e.g., if the Sau piece is captured).

Key Functionalities:

- Adds and removes pieces from the player's collection dynamically as the game progresses.
- Implements logic for making moves and capturing opponent pieces.

4- Piece (Abstract Class)

Purpose: Serves as the base class for all chess pieces, defining common properties and behaviours.

Responsibilities:

- Defines common attributes (colour, position, type).
- Provides abstract methods for movement validation, possible moves, and copying pieces.

Key Functionalities:

- Uses inheritance to create specialized piece classes (Ram, Biz, Tor, Xor, Sau).
- Implements encapsulation to protect piece state (e.g., position, captured status).
- Stores move history to track piece movement.

5- Piece Subclasses

Each subclass extends `Piece` and has unique movement rules.

- **Ram**

  Purpose: Represents the Ram piece, which moves forward and changes direction when reaching the board edge.

  Functionalities:

  - Moves one step forward in its current direction.
  - Reverses direction upon reaching the end of the board, continuing in the opposite direction.

- **Biz**

  Purpose: Represents the Biz piece, which moves in an L-shaped pattern, allowing for strategic positioning.

  Functionalities:

  - Moves in an L-shape, covering three squares in one direction followed by two squares perpendicular.
  - Can jump over other pieces, making it the only piece that can bypass obstacles.

- **Tor**

  Purpose: Represents the Tor piece, which moves orthogonally and undergoes transformation after a set number of turns.

  Functionalities:

  - Moves horizontally or vertically across any number of squares.
  - Transforms into Xor after two turns, altering its movement capabilities.

- **Xor**

  Purpose: Represents the Xor piece, which moves diagonally and transitions into the Tor piece over time.

  Functionalities:

  - Moves diagonally across the board without jumping over pieces.
  - Transforms into Tor after two turns, modifying its movement rules.

- **Sau**

  Purpose: Represents the Sau piece, the most critical piece as its capture determines the winner.

  Functionalities:

  - Moves one step in any direction (horizontally, vertically, or diagonally).
  - The game ends if the Sau is captured, making it the most important piece on the board.

6- GameView (View)

Purpose: Handles the Graphical User Interface (GUI), allowing players to interact with the game visually.

Responsibilities:

- Displays the chess board and pieces using Java Swing.
- Handles mouse clicks for selecting and moving pieces.
- Dynamically updates the board whenever a move is made (Observer Pattern).

Key Functionalities:

- Supports board flipping to ensures correct perspective for each player and piece highlighting for better usability.
- Implements a resizable and interactive chessboard with Buttons for starting, saving, and ending the game.

7- GameController (Controller)

Purpose: Acts as the intermediary between the Game (Model) and GameView (View).

Responsibilities:

- Handles user clicks and translates them into game actions.
- Ensures separation of concerns between logic and UI.

Key Functionalities:

- Ensures turn-based play by managing moves.
- Manages the start and progression of the game.
- Communicates updates to the GameView.

8- GameObserver (Interface)

Purpose: Defines the contract for observers to receive updates from the Game class.

Responsibilities:

Provides an `update(Game game)` method to notify observers of game state changes.

Key Functionalities:

Used by *GameView* to dynamically update the GUI when the game state changes.

9- PieceType (Enum)

Purpose: Defines the types of pieces in Kwazam Chess.

Responsibilities:

Enumerates the possible piece types: RAM, BIZ, TOR, XOR, SAU to ensures consistent usage of piece names.

Key Functionalities:

Provides a clear and type-safe way to represent piece types in the game.

2. Compile and Run Instructions

2.1 Prerequisites

Before compiling and running Kwazam Chess, ensure the following are set up on your PC:

- **Programming Environment:** Ensure you have the required IDE (e.g., BlueJ, VS, etc) or a code editor with Java support.
- **Java Development Kit (JDK):** Version 11 or higher installed and configured.
- **Dependencies:** Ensure all .java files are in the correct directory structure.

2.2 Setting Up the Project

- Extract the downloaded file and place it in folder of choice.
- Open BlueJ: Launch the BlueJ IDE.
- Open the Project Folder:

  Go to Project > Open Project.

  Navigate to the folder containing the Kwazam Chess .java files and select it.

  Ensure the folder contains the KwazamChess.java file (the main class).

- Load the Files:

  BlueJ will automatically load all .java files in the folder.

2.3 Compiling and Running in BlueJ

- Click “Compile” on the left-side of the screen to compile all the codes.
- Verify that the “KwazamChess” class exists and has been compiled.
- Right-click the “KwazamChess” class and select “void main(String[] args)”.
- A pop-up will appear for the method call. Click “OK” to run the application.
- The application’s main menu will show up and the Kwazam Chess game can be played.

2.4 Compiling and Running using Command Prompt (CMD/CLI)

- Navigate to the project directory by using this command:

cd <project directory>

Example:

```
C:\Users\the46>cd C:\Muhammad Faiz bin Ilyasa\ooad\bluej\asgntest

C:\Muhammad Faiz bin Ilyasa\ooad\bluej\asgntest>
```

- Compile the codes by using this command:

```
javac *.java
```

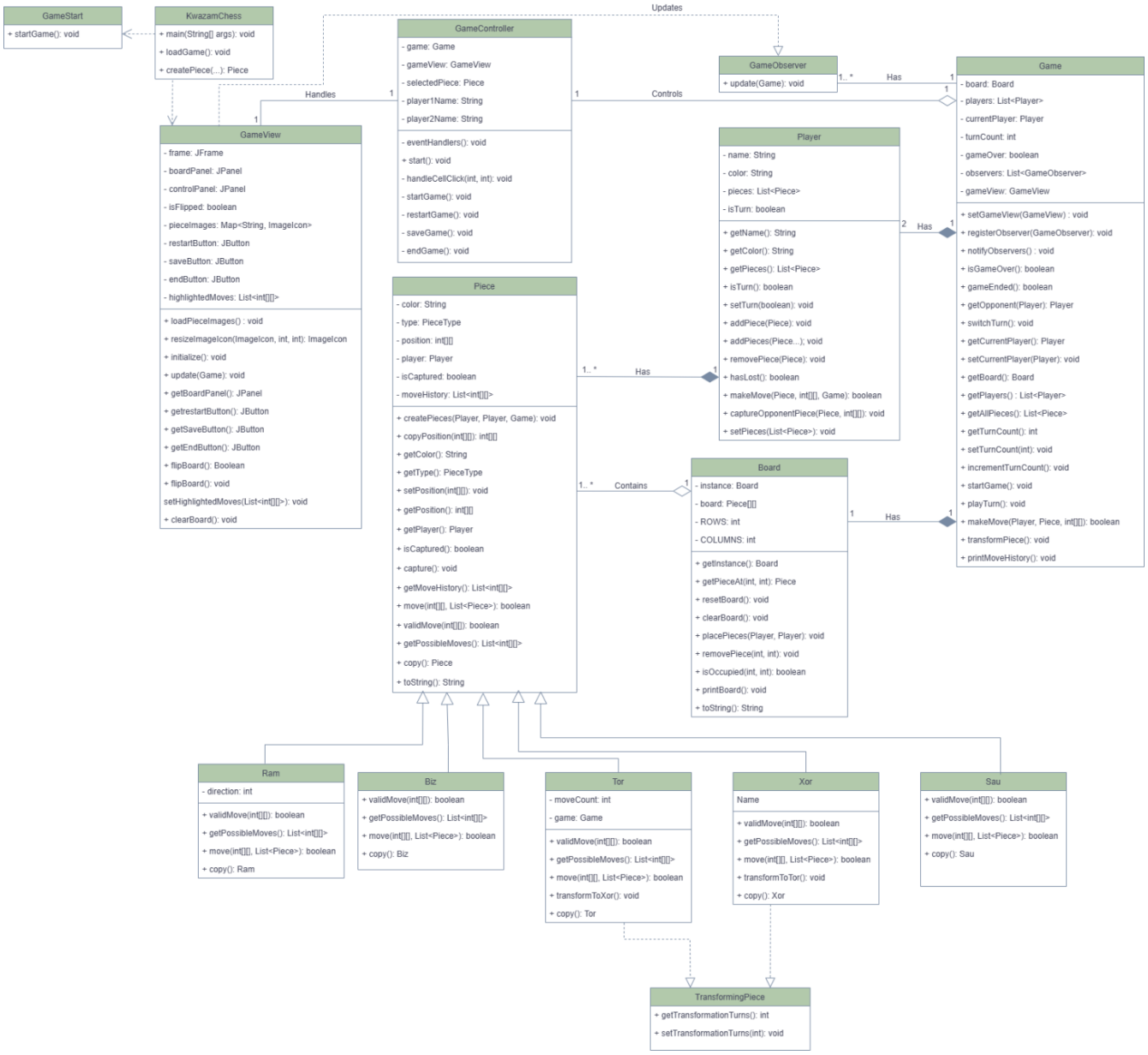
- To run the program, type:

```
java KwazamChess
```

- The program will start running and Kwazam Chess can be played.

3. UML Class Diagram

The UML Class Diagram provides a comprehensive visual representation of the system architecture, illustrating the relationships, attributes, and methods of the classes involved in the Kwazam Chess game. Serving as a blueprint for system design, it ensures clarity, consistency, and modularity in the application's implementation.



The provided class diagram illustrates the object-oriented design for a chess game. The diagram showcases a well-defined structure with clear responsibilities assigned to each class and well-defined relationships between them.

Key Classes and Relationships:

- **GameController:** This class serves as the central orchestrator, managing the overall game flow. It has a composition relationship with the Game class, indicating that the GameController owns and controls the Gameinstance.
- **Game:** This class encapsulates the core game logic, including game state management, rules enforcement, and player interactions. It has a composition relationship with both the Board and Player classes. This implies that the Game owns the Board and the instances of Player participating in the game.
- **Player:** This class represents a player in the game. It typically stores information about the player's name, color, and potentially other relevant attributes.
- Board: This class represents the chessboard, holding information about its dimensions, grid layout, and the placement of pieces. It has a composition relationship with the Piece class, signifying that the Board contains and manages the Piece objects.
- **Piece:** This is an abstract class, meaning it cannot be instantiated directly. It acts as a blueprint for concrete piece classes, defining common properties and behaviors shared by all pieces. The Piece class has an inheritancerelationship with concrete piece classes like Ram, Biz, Tor, Xor, and Sau, which inherit its properties and behaviors while implementing their unique movement rules and logic.
- **GameView:** This class is responsible for the visual representation of the game, interacting with the user interface elements. It has a composition relationship with BoardPanel and ControlPanel, indicating that these components are parts of the GameView but can potentially exist independently.
- **KwszamChess:** This appears to be the entry point of the application, responsible for initializing the game and handling user interactions.

Overall, the class diagram demonstrates a well-structured and modular design. The use of inheritance, composition, and other relationships promotes code reusability, maintainability, and flexibility in the game's implementation.

4. Use Case Diagram

The following use case diagram illustrates the interactions between the Player (actor) and the Kwazam Chess system. It highlights the core functionalities available to the player and the relationships between these functionalities.

Actor:

- Player: A user of this system who acts as the actor by initiating major activities like initiating a new game, making moves, capturing pieces, and managing the lifecycle of the games.

Use Cases:

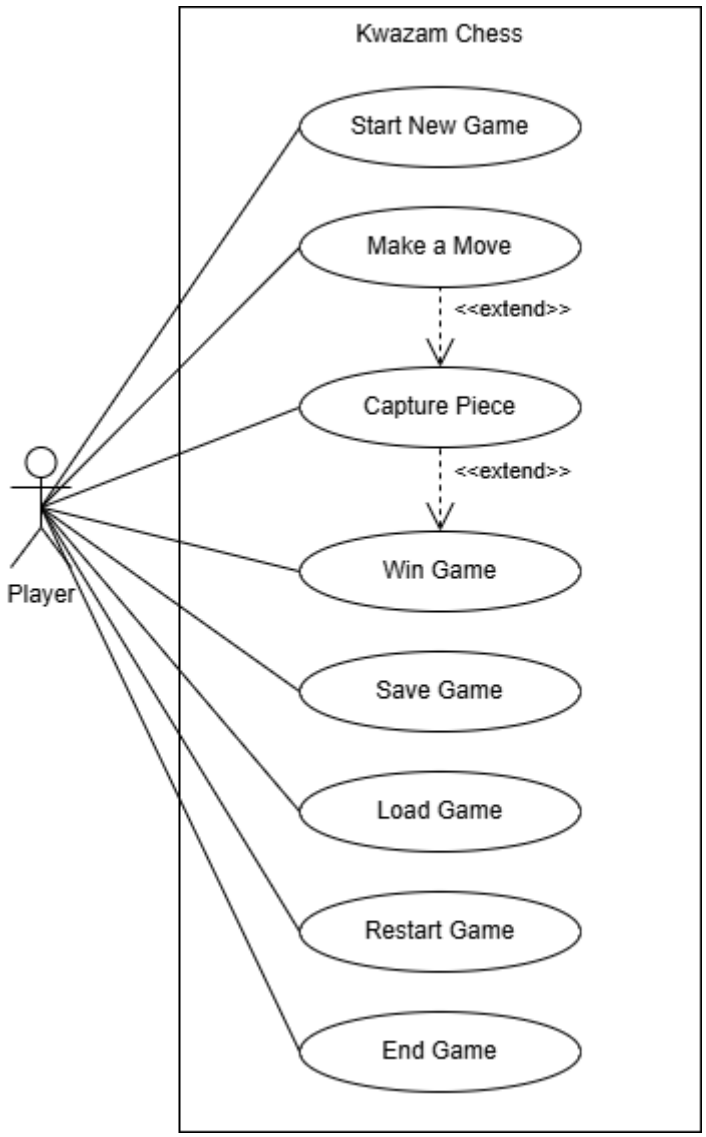
1. Start New Game:
   - Allows the player to begin a new game session.
2. Make a Move:
   - Represents the main gameplay interaction where the player moves chess pieces on the board.
3. Capture Piece:
   - An extension of the "Make a Move" use case, triggered when the player moves a piece to capture an opponent's piece.
4. Win Game:
   - An extension of the "Capture Piece" use case, triggered when the player moves a piece to capture the opponent Sau piece.
5. Save Game:
   - Enables the player to save the current state of the game for future continuation.
6. Load Game:
   - Allows the player to load a previously saved game and resume gameplay.
7. Restart Game:
   - Allows the player to restart the game with the same players, with no need to renter the player's names.
8. End Game:
   - Provides the player with the option to end the game session voluntarily.

Relationships:

- <<extend>>:
  - "Capture Piece" extends "Make a Move" since capturing is an optional event during a move.
  - "Win Game" extends "Capture Piece" as it occurs only when the Sau piece is captured.

Purpose of the Diagram:

This use case diagram explains in detail the functionality and fundamental interaction of the game exposed to the player, particularly in regard to the relationship between the standard actions of gameplay and those being auxiliary. The use of the <<extend>> relationship between “Make a Move”, “Capture Piece”, and “Win Game” captures the optional nature of capturing a piece during a move, and the possibility of that piece being Sau, adding clarity to the game mechanics.


Kwazam Chess
Start New Game
Make a Move
<<extend>>
Capture Piece
<<extend>>
Win Game
Save Game
Load Game
Restart Game
End Game
Player

5. Sequence diagrams

The sequence diagrams in this section illustrate the interactions between system components during the gameplay processes in the Kwazam Chess application. These diagrams provide a detailed visualization of how objects communicate over time, ensuring clarity in the execution flow and system behavior.

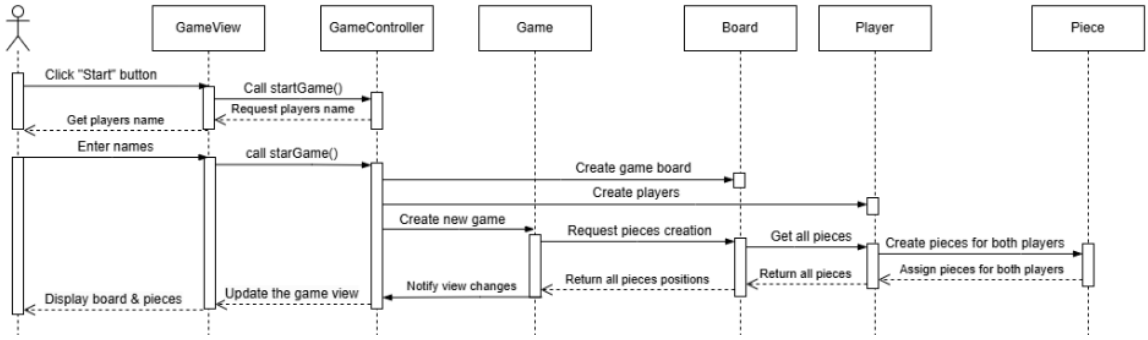
5.1 Start New Game

This sequence diagram outlines the process of initiating a new game for two players. The sequence begins when the User clicks the "Start" button, prompting GameView to initiate the game setup by invoking the startGame() method within GameController. GameController then prompts the User to provide names for the two players participating in the game.

Following this, GameController creates a new instance of the Game object, which serves as the central entity encapsulating the core game logic. The Game object then proceeds to create the game board and instantiate two Player objects, one for each player.

Next, the Game object requests the Board to generate the necessary pieces for both players. The Board fulfills this request and returns the newly created pieces positions from the player. Finally, the Game object assigns these pieces to the corresponding players and place them in the board.

Once the game setup is complete, GameController notifies GameView about the changes made. GameView then updates its display to reflect the current state of the game, including the rendered board and the positions of the pieces for both players. This sequence diagram effectively illustrates the interactions between different objects and the flow of control throughout the process of initializing a new two-player game.



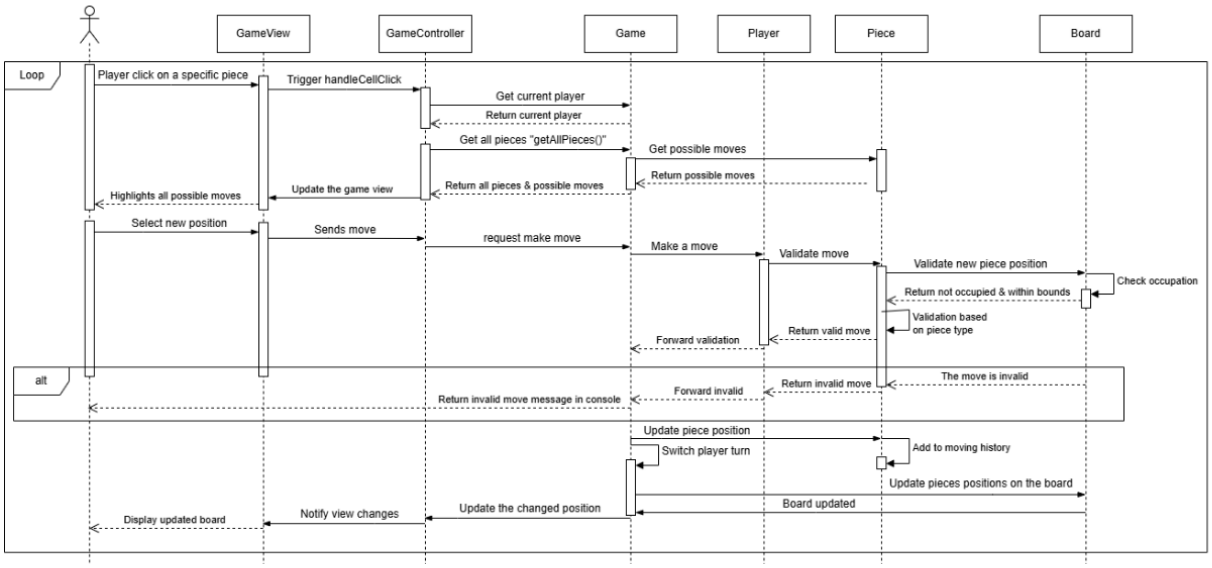
5.2 Make a Move

This sequence diagram details the process of making a move in a turn-based game. The sequence begins with the User clicking on a specific piece on the game board. GameView detects this click and triggers the handleCellClick() method within GameController. GameController then determines the current player and retrieves all the pieces from the Game object. Subsequently, the Game object calculates the possible moves for the selected piece and returns this information to GameController. GameView then updates the game view to highlight these possible moves, potentially using different colors to distinguish between legal and invalid moves.

The player selects a new position for the piece, and GameView sends this move to GameController. GameController requests the Game object to make the move. The Game

object validates the new position by checking if it is within the board's boundaries and if it is not occupied by the player's own pieces. Additionally, the Piece object validates the move based on its specific rules, such as movement limitations and capture rules. If the move is valid, the Game object updates the piece's position on the board. If the move is invalid, the Game object returns an error message to GameView. In case of a valid move, the Game object may optionally switch the turn to the next player. The Board then updates its internal representation of the game state. Finally, GameView is notified about the changed positions on the board and updates its display accordingly.

The diagram incorporates an alternative path to handle invalid moves. If the player selects an invalid move, the process repeats from the "Select new position" step until a valid move is chosen. The sequence in a loop to ensure the game runs continuously until a win condition by capture SAU piece. This sequence diagram provides a comprehensive and detailed view of the interactions between different objects and the flow of control throughout the "Make a Move" process in a turn-based game.



5.3 Capture a Piece

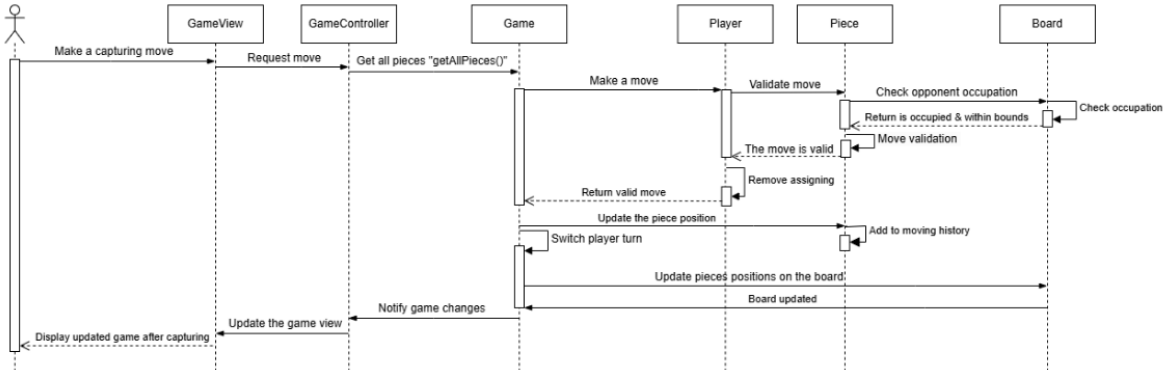
The sequence diagram illustrates the process of capturing an opponent’s piece in a turn-based game, detailing the interactions between the Player, GameView, GameController, Game, Piece, and Board components. The process begins when the player make a capturing move. This action is first received by the GameView, which forwards it to the GameController through the Request move method. The GameController then retrieves all pieces from the Game object using the getAllPieces() method to ensure it has the current state of the board for validation.

The GameController delegates the move to the selected Piece by invoking its Make a move method. At this stage, the Piece begins validating the move by checking key conditions. It ensures the target position is within the board’s bounds, unoccupied by the player’s own pieces, and, if capturing is intended, occupied by an opponent’s piece. The Board plays a crucial role here by assisting with the validation. It checks whether the target position is occupied and within bounds, returning the results to the Piece for further decision-making.

Once all validations are successfully passed, the Piece executes the move. It updates its position to the new location and records the move in its history for tracking purposes. Simultaneously, the Piece notifies the Board of the updated position, prompting the Board to adjust its internal representation of the game state to reflect the captured piece and the new placement of the moving Piece.

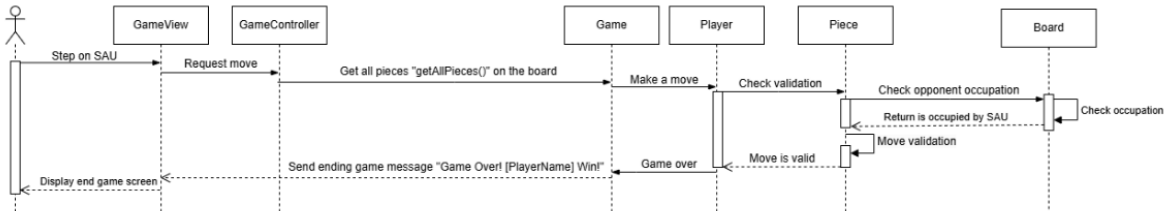
With the move completed, the Game object switches the turn to the next player, signifying the end of the current turn. Finally, the GameController notifies the GameView of the changes made during the move. The GameView updates its display to reflect the new board state, including the captured piece and the updated positions of all other pieces.

This sequence demonstrates the collaborative interactions between the system's components to ensure accurate rule validation, state management, and user interface updates. By breaking down the process into clear steps, the diagram offers a comprehensive view of the logic and coordination required for a capturing move in the game.



5.4 Win game

This sequence diagram illustrates the process of determining and declaring a "Step on SAU" win condition in a turn-based game. The sequence initiates when the User performs a move that triggers this specific win condition. GameView receives this input and subsequently triggers the Request move method within GameController. GameController then retrieves all pieces from the Game object. Subsequently, the Game object directs the selected Piece to execute the Make a move method. The Piece proceeds to validate the move by checking its adherence to game rules, including boundary checks and collision detection with other pieces. Following validation, the Piece determines if the move results in the "Step on SAU" condition. If this condition is met, the Piece signals a win condition to the Game object. The Game object then declares the game over and identifies the player who achieved the "Step on SAU" as the winner. Subsequently, the Game object transmits an "ending game message" to GameView, indicating the victorious player. GameView then displays an "end game screen" to the User, informing them of the win and the identity of the victorious player. This sequence diagram effectively captures the interactions between various objects and the steps involved in recognizing and handling this unique win condition within the game.

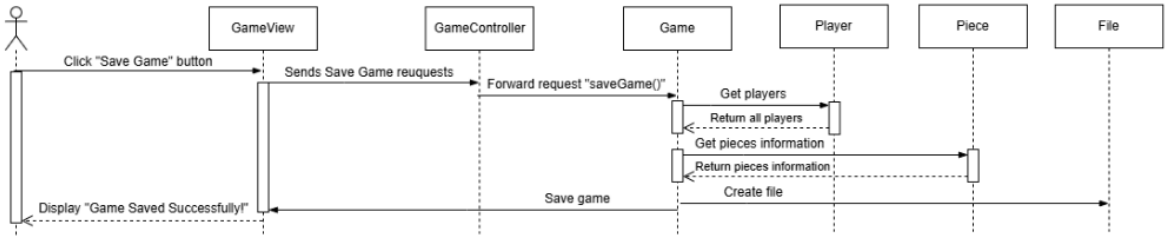


5.5 Save Game

The "Save Game" sequence diagram demonstrates the process of saving the current game state to a file. It begins when the User clicks the "Save Game" button. This action is received by the GameView, which sends a Save Game request to the GameController. The GameController then forwards the request to the Game object by invoking the saveGame() method.

The Game object gathers the necessary information for saving. It retrieves player data from the Player component and the current state of all pieces on the board from the Piece component. With this information, the Game object creates a save file (e.g., player1VSplayer2.txt). encapsulating all relevant game data to enable future restoration. Once the file is successfully created, a confirmation message is sent back to the GameView, which displays "Game Saved Successfully!" to the player, signifying the successful completion of the save operation.

This diagram highlights the collaboration between the components to ensure the accurate capture and storage of the game state, providing a seamless save experience for the User.



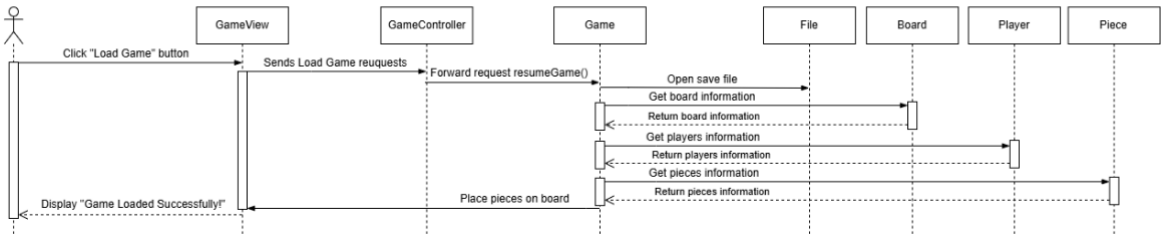
5.6 Load Game

The "Load Game" sequence diagram illustrates the process of restoring a previously saved game state. The process begins when the player clicks the "Load Game" button in the main menu. This action is received by the GameView, which sends a Load Game request to the GameController. The GameController forwards this request to the Game object by invoking the resumeGame() method.

The Game object opens the saved game file and retrieves the necessary data. First, it accesses the saved board information and restores the state of the board. Next, it retrieves player data and reinitializes the players. Finally, the Game gathers the positions and states of all pieces from the file. After reconstructing the game state, the pieces are placed back on the board.

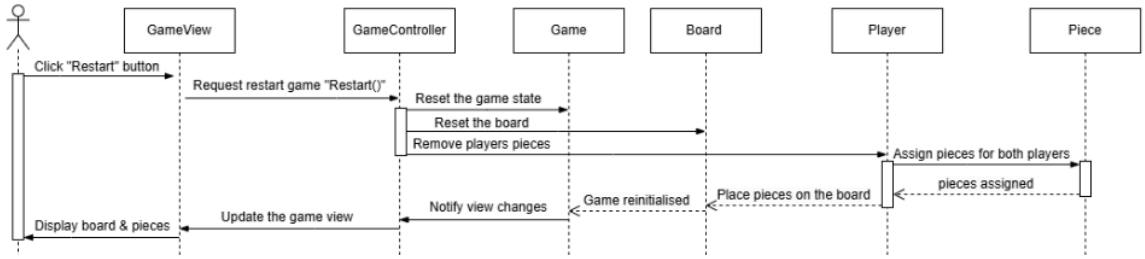
Once the loading process is complete, the Game notifies the GameController, which in turn updates the GameView. The GameView displays "Game Loaded Successfully!" to the player, confirming that the saved game has been restored accurately.

This sequence diagram underscores the system's ability to reconstruct the game state from a saved file, ensuring continuity and a smooth user experience for resuming gameplay.



5.7 Restart Game

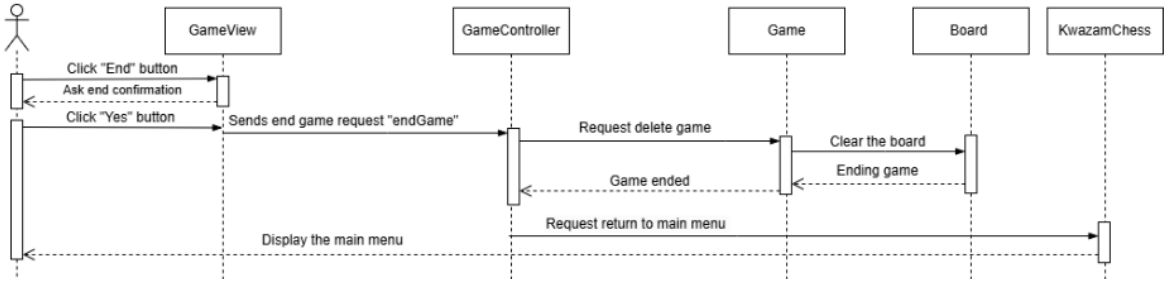
This sequence diagram illustrates the process of restarting a game within a turn-based gaming system. The sequence is initiated when the player clicks the "Restart" button within the GameView. This action triggers the restartGame() method within GameController. GameController then instructs the Game object to reset its internal game state. Subsequently, the Game object directs the Board to reset its state, effectively clearing the board of all existing pieces. The Game object then instructs the Player objects to remove their respective pieces from the board and subsequently assigns new sets of pieces to both players. Following this, the Game object directs the Board to place these newly assigned pieces on the board according to the game's initial setup. Once the board has been updated with the new pieces, the Game object notifies GameView about these changes. GameView then updates its display to accurately reflect the newly initialized game state, including the updated board and the positions of the pieces. This sequence diagram effectively depicts the interactions between various components and the steps involved in reinitializing the game state, ensuring a smooth and consistent user experience when restarting a game.



5.8 End Game

This sequence diagram illustrates the process of ending a game. The sequence initiates when the User clicks the "End" button within the GameView. GameView then displays a confirmation dialog to the User, asking them to confirm their intention to end the game. If the User confirms by clicking the "Yes" button, GameView sends an "endGame" request to GameController. Upon receiving this request, GameController instructs the Game object to delete the current game state. The Game object then clears the board and notifies GameController that the game has ended. Finally, GameController requests the

KwazamChess object to display the main menu, allowing the User to start a new game or perform other actions.



6. User Documentation

6.1 Launch the Game

When you start Kwazam Chess, the main menu will appear with the following options:

- Start Game: Begin a new game of Kwazam Chess.
- Load Game: Continue a previously saved game
- Instructions: View the rules and instructions for the game.
- Exit: Close the program.

6.2 Starting New Game

The game board consists of a 5x8 grid.

Each side has 5 unique pieces: Ram, Biz, Tor, Xor, and Sau.

There are two players playing the game, and each player takes turns to move their pieces.

The board will be flipped each time the player moves their pieces.

Use mouse to select and move pieces:

- Click on a piece to select it.
- When a player clicks on a piece, the game will highlight the positions the piece can move to.
- Click on the highlighted position to move the piece.
- If a move is invalid, an error message will be displayed.

6.3 Saving the game

You can save your current game progress at any time.

- Click the Save Game button in the menu.
- The game will create a save file and confirm once the save is successful.
- Saved games can be loaded later using the Load Game option.

6.4 Load the game

Players can save multiple game files during gameplay.

- To load a saved game:
    - Select the Load Game option from the main menu.
    - Enter the save file name
    - The game continues where it was left off.

6.5 Game ended

The game ends when one of the Sau pieces is captured.

A message will appear on the screen declaring the winner (e.g., "Player X won!").

- After the game ends, players can:
    - Start a new game by selecting the start game option from the game main menu.
    - Exit the program using the exit button.