Ankara Yıldırım Beyazıt University
Department of Computer Engineering

CENG 201 – Object Oriented Programming
Course Project

G08: The Chess Game

Analysis Report

Emre Karaburu,
Okan Rıdvan Gür,
Sarper Avcı,
Utku Akkuşoğlu

Instructor: Muhammed Abdullah Bülbül
Teaching Assistant: Elif Şanlıalp, Yusuf Şevki Günaydın
Date: 01/11/2024

Table of Contents

Introduction

Our project's main objective is to create an interactive and visually compelling chess game with a graphical user interface (GUI) using C++. In designing this game, we aim to incorporate all standard chess functionalities, such as accurate piece movements, board visualization, and real-time game state detection for special moves and conditions (e.g., check, checkmate, castling, and pawn promotion). By implementing a GUI, we strive to make the game not only functional but also visually appealing and user-friendly, allowing players to enjoy an immersive and seamless experience as they navigate the game's features.

Purpose of the Report:

The purpose of this report is to provide a comprehensive outline of the initial analysis, requirements, and design components essential for developing our GUI-based chess game. This report covers both functional and non-functional requirements, in addition to detailing aspects of system modeling and interface mockups. Through these sections, we aim to guide the design and implementation phases of our project, ensuring that all components work harmoniously to create a smooth and enjoyable user experience.

Why We Chose C++ as Our Development Language:

We selected C++ as the primary programming language for this project because it's particularly well-suited for developing high-performance GUI-based applications. Specifically, when combined with libraries like Qt or SFML, C++ enables us to simplify the complex processes of GUI design and game rendering. These libraries offer extensive support for graphical elements and events, which help us create a responsive, visually engaging interface. Furthermore, C++'s

efficiency in terms of memory and processing power aligns well with our need for real-time graphics and quick gameplay responses. This makes it ideal for building a chess game, where both computational efficiency and a smooth graphical presentation are crucial to the overall user experience.

2. Requirements

2.1. Functional Requirements

Our chess game relies on several core functionalities that we believe are essential to delivering a complete and enjoyable gameplay experience. We describe each of these key features in detail below:

1. **Player Controls:**
   To enhance the user experience, our game must support intuitive player controls, enabling users to move pieces using either a drag-and-drop mechanism or a click-based interface. This feature is intended to make interactions feel smooth and natural. For example, a player should be able to click on a piece to select it and then click on a target square to move it, or simply drag it to the desired position, thereby simulating the physical experience of moving pieces on a real chessboard.
2. **Piece Movement:**
   Each chess piece must be able to move according to the official rules of chess. For example, pawns should be able to move one square forward (or two squares on their initial move), rooks should move vertically or horizontally across any number of squares, bishops should move diagonally, and so forth. Additionally, the game should prevent invalid moves (e.g., a bishop moving horizontally or a pawn attempting to move backward). This functionality ensures the integrity of gameplay, allowing players to focus on strategy without worrying about rule enforcement.
3. **Board Display:**
   The visual display of an 8x8 chessboard is central to our game's interface. This feature includes a grid layout for displaying each square and the ability to update dynamically to reflect piece movements as players take turns. By displaying each piece's position on the board at all times, this layout enables players to easily track game progress and maintain situational awareness.
4. **Game State Management:**
   Our game must automatically detect and handle special game states, including check, checkmate, castling, and pawn promotion. For example, when a player's king is in check, the game should alert the player and restrict moves that would leave the king in check. If checkmate is detected, the game should automatically end. Additionally, our game will support castling (a special king-and-rook move) and pawn promotion, allowing players to replace pawns with more powerful pieces upon reaching the opposite side of the board. These state management features ensure that our game follows official chess rules and provides real-time feedback to players on the game's status.

2.2. Non-Functional Requirements

In addition to primary functionalities, our chess game needs to meet several non-functional requirements to enhance the overall user experience. These qualities will ensure that our game performs well, is easy to use, and remains stable even in unexpected scenarios.

1. **Performance:**
   To ensure a smooth and immersive experience, our game must support seamless graphics updates and quick response times. For instance, the board should refresh immediately to reflect each move, and there should be no delay when players move pieces. This level of responsiveness is particularly important in creating an enjoyable user experience, as even a slight lag can disrupt the flow of the game.
2. **Usability:**
   We aim to design a user-friendly interface that offers clear visual feedback for player actions. For example, when a player selects a piece, the interface could highlight the piece and the valid moves available to it. This feature will help players easily understand and engage with the game, regardless of their experience level. Additionally, having a simple and intuitive interface will make the game accessible to a broader audience, including users who may be new to digital chess games.
3. **Reliability:**
   Our game must be stable and capable of handling unexpected inputs, such as invalid moves or sudden interruptions, without crashing. For example, if a player tries to move a piece in an invalid way, the game should simply reject the move without affecting the gameplay flow. By implementing robust error-handling mechanisms, we aim to ensure that players can enjoy a smooth and uninterrupted experience, free from technical glitches that could detract from the game.

3. System Models

3.1. Scenarios

In this section, we conducted an analysis through scenarios and usage examples to detail the system's working logic. We aimed to highlight the important functions of the system by considering how users interact with the system and how the system reacts in various situations.

Starting a Game

1. The user selects the 'New Game' option to start the game.
2. The system creates a new chessboard in the graphical interface and places all the pieces in their starting positions.
3. The interface indicates that the game is ready to be played.

Summary: This scenario describes the process of starting the game by the user and the GUI's response in this situation.

Moving

1. The user clicks on a piece to select it.
2. The system highlights the possible moves of the selected piece.
3. The user completes his move by clicking or dragging the piece to the target square.
4. The system moves the piece to the target square and indicates the next player's turn.

Summary: This scenario describes how users select and move a piece, and how the system responds to this situation.

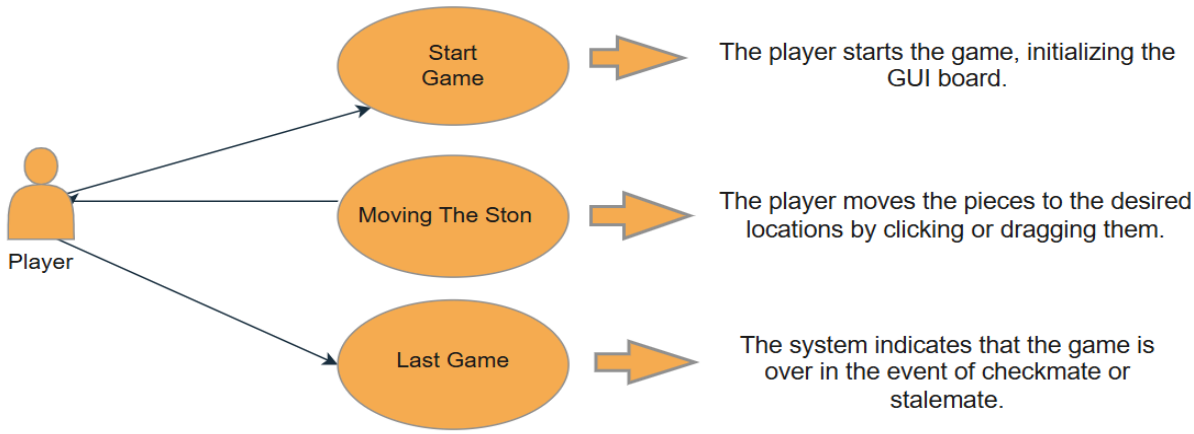
Checkmate or Stalemate

1. The user makes a move that checkmates the opponent's king, or the game ends in a stalemate.
2. The system detects a checkmate or stalemate.
3. The GUI displays a message indicating that the game has ended, and indicates the winner (in the case of checkmate) or a draw (in the case of stalemate).

Summary: This scenario describes how the game ends in a checkmate or stalemate, and the feedback the system gives to the user.

3.2. Use Cases

Using the scenarios, we identified the users who will interact with the system and the actions they will take with the system. We detailed each use case with brief explanations, visually presenting these main interactions through a use case diagram.



1. **Starting the Game**
   - The user starts the game, and the system prepares the chessboard and places the pieces in the starting position.

2. **Moving the Piece**
   - The user moves the pieces to the desired positions by clicking or dragging. The system performs these moves and moves on to the next player.

3. **End Game**
   - The game ends when checkmate or stalemate occurs. The system provides feedback to the users indicating the end of the game and the result.

3.3. Object and Class Model

Piece Class (Main Class for All Chess Pieces)

The `Piece` class is the main model for all chess pieces (like Pawns, Knights, Bishops, etc). It has attributes and methods that every chess piece uses.

Attributes:

- `position`: Tells us where the piece is on the board (like coordinates).
- `color`: Shows the color of the piece (either "white" or "black").

Methods:

- `move(target_position)`: This is a method that each type of piece (Pawn, Knight, etc.) will use to move. It is "virtual," which means each piece type will make its own rules for moving.
- `validateMove(target_position, board)`: Checks if a move is allowed by chess rules. For example, it ensures Pawns move one square ahead or two squares on their first move, or that Bishops move diagonally.
- `getPosition()` and `getColor()`: These methods give information about the piece's position and color.

The classes for each piece type (like `Pawn` or `Knight`) will use this base class (`Piece`) and create their own move rules.

Board Class

The `Board` class manages the 8x8 grid. It keeps track of where each piece is and handles moves.

Attributes:

- `grid`: An 8x8 structure that holds each `Piece` in its square, or marks the square as empty.

Methods:

- `display()`: Shows the board and the pieces on it. It is used to update the game's view for players.
- `placePiece(piece, position)`: Places a piece on a square. This is used at the start of a game to set up all pieces.
- `removePiece(position)`: Removes a piece from a square when it's captured.
- `updatePosition(piece, new_position)`: Moves a piece to a new square and updates its position.

The `Board` checks with each piece to see if moves are allowed and keeps an accurate record of the board.

Game Class

The `Game` class controls the entire game. It keeps track of whose turn it is, if the game is still going, and if a player's king is in check.

Attributes:

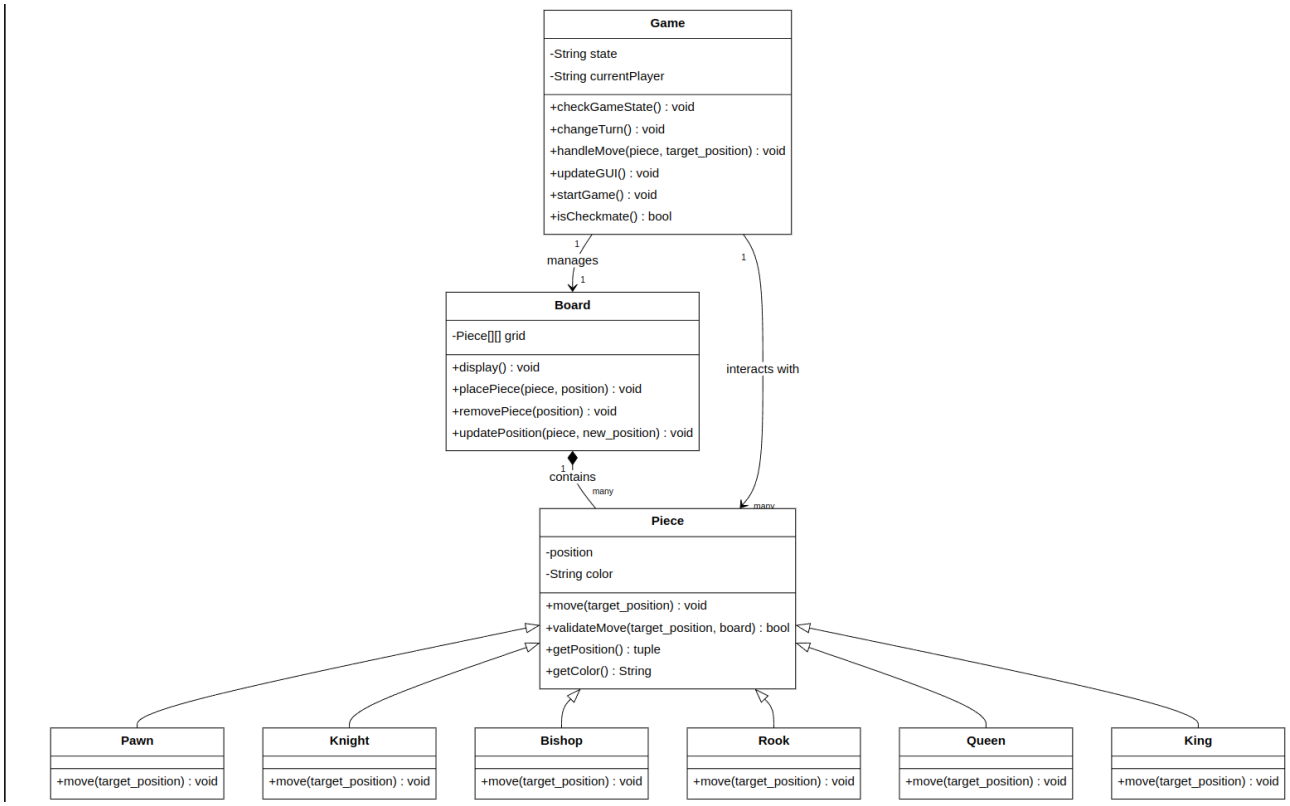
- `state`: Shows the current game situation, such as "in progress," "check," or "checkmate."
- `currentPlayer`: Tells us whose turn it is ("white" or "black").

Methods:

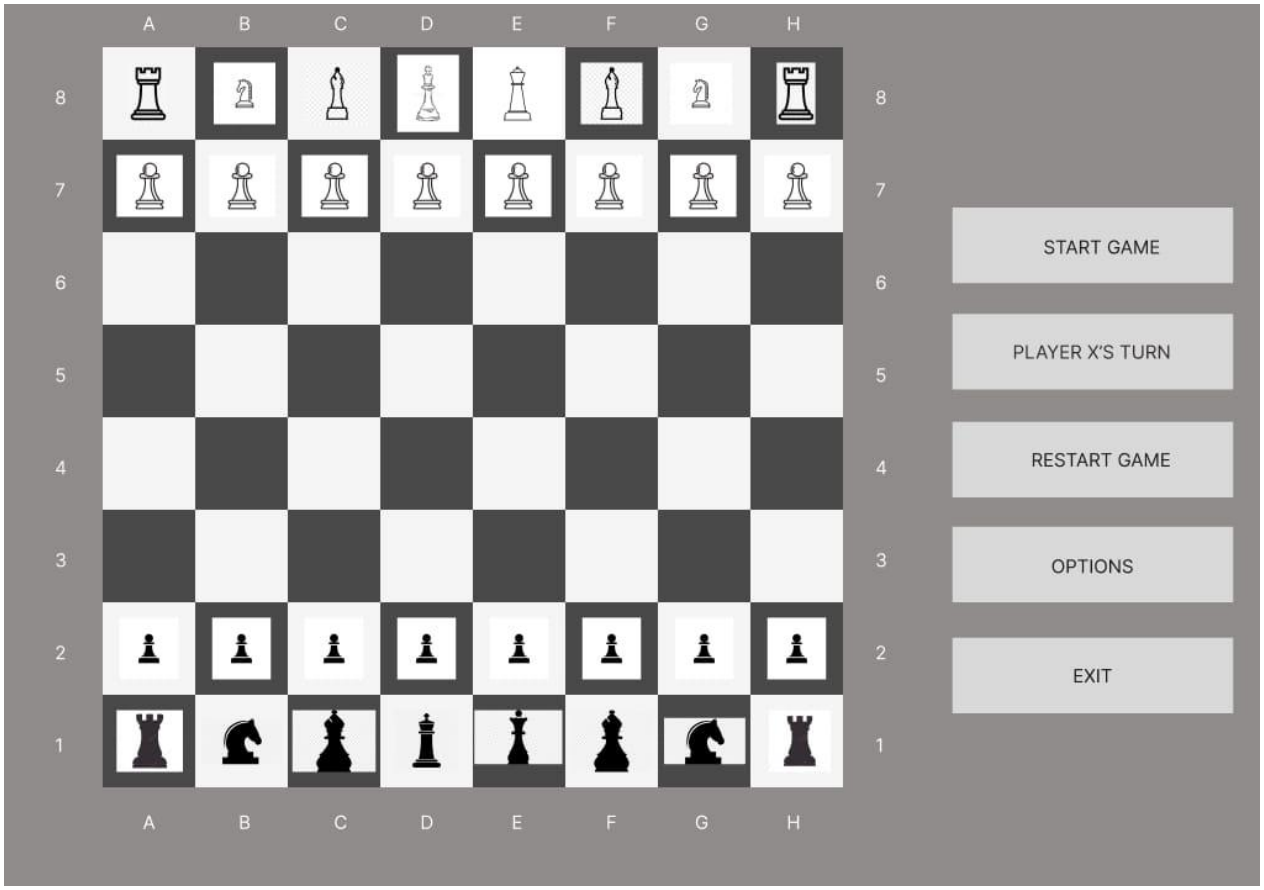
- `checkGameState()`: Checks the board and sees if a player's king is in danger (check or checkmate).
- `changeTurn()`: Switches turns between the two players.
- `handleMove(piece, target_position)`: Moves a piece to a new square if it's a legal move. It then changes the turn.
- `updateGUI()`: Refreshes the view of the board after each move.
- `startGame()`: Sets up all pieces on the board and begins the game.
- `isCheckmate()`: Looks for checkmate, which means the player's king cannot move anywhere safely.

The `Game` class brings everything together, making sure that moves are allowed, turns are managed, and the board view is updated for players.

UML Class Diagram



3.4. User Interfaces



4. Conclusion

In summary, our main objective in this project is to present a visually appealing chess game that adheres to Object Oriented Programming principles and requirements.

To complete our project smoothly, there are requirements we need to follow. Functionally, we must successfully implement features such as player control, piece movement, board display, and game state management. To ensure a user-friendly experience, we should also succeed in non-functional qualities like performance, usability, and reliability. Additionally, to offer a unique experience compared to other chess games, we plan to include primarily visual customization options.

Based on our analysis, we envision the game scenario as follows: after configuring the settings, the game will start with a "start game" option, followed by the creation of the board in a graphical interface, allowing players to move

pieces by clicking. The user's moves need to be displayed with effects, and the system should check for checkmate or stalemate.
We will have three main classes: Piece Class(main class for all chess pieces), Board Class(to track the pieces and manage their movements) and Game Class(to control the game state)

5. Contribution:

In our report, Emre Karaburu was responsible for the Introduction and Requirements sections, Okan Rıdvan Gür for the Scenarios and Use Cases section, Sarper Avcı for the Object and Class Model section, and Utku Akkuşoğlu for the User Interfaces and Conclusion section.