Design Overview for Chess in C#

Name: Nhu Gia Bao Nguyen
Student ID: 104690366

Summary of Program

Create a chess game with basic chess rules such as how the pieces move, and some special rules such as castling, pawn promotion, en passant. I have also integrated 2 design patterns:
MVC Pattern:

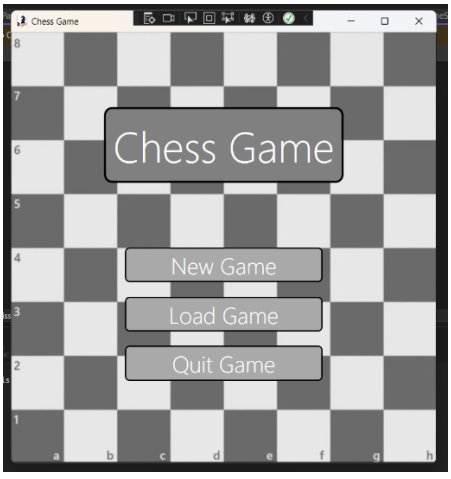
- The Model classes (GameState, Board, Piece, Move) encapsulate the data and business logic of the game.
- The View classes (MainWindow.xaml, EndMenu.xaml, PromoMenu.xaml) handle the presentation logic and user interface.
- The Controller classes (MainWindow.xaml.cs, EndMenu.xaml.cs, PromoMenu.xaml.cs) manage user inputs, interact with the model to update the game state, and refresh the view to reflect changes.

Prototype Pattern (Elements):

The cloning or copying mechanism in GameState for testing moves is an example of creating new instances based on existing ones. For example, your GameState class might use a method to clone the current state to simulate a move.

Observer Pattern: The EndMenu and PauseMenu classes use the Observer pattern to notify other parts of the application when certain events occur. They define events (OptionsChanged) and allow other classes to subscribe to these events and be notified when the events are raised.

The game is also expandable with functions such as timer clocks, multiplayer modes, custom chess pieces with different functions, more modes such as 4 players in 1 board or double sized chess board and more depends on the creator's creativity.



The game main menu

Game started

Ending screen

There are simply too many things that my program does so I will list some important and special aspects:

Moving:

The execution of the move will happen in gameState, each piece has an execute() method in their class (inherits from Piece.cs base class) to do their own move, there are 5 different moves: DoublePawn (pawn move 2 square if the first time), EnPassant, Castle, Normal (normal moves)

```
public void Moving(Move move)
{
    Board.SetPawnSkippedSquares(CurrentTurn, null);
    move.Execute(Board);
    CurrentTurn = CurrentTurn.Opponent(); //change turn
    stateString = new StateString(CurrentTurn, Board).ToString(); //update stateString
    CheckEnded();
}
```

Checking algorithm: When the player move, a copy of a board will be created, then the move will be excecuted in the board copy. If after the move, the current player's king isn't in check, then that move is legal.

```
2 references
public Board Copy()
{
    Board copy = new Board();

    foreach (Positions pos in PiecesPos())
    {
        //check if the piece is null or not
        Piece piece = this[pos];
        if (piece != null)
        {
            //if not null, get a copy of the piece
            Piece pieceCopy = piece.Copy();
            if (pieceCopy != null)
            {
                //if not null, assign the position to the copied board
                copy[pos] = pieceCopy;
            }
        }
    }

    return copy;
}
```

Code snippet

State String: A stateString is similar to a FEN string, it is devided into 4 parts, rules are:

- 1st 2nd 3r 4th, seperate by a space
- 1st part is piece postions, the rules are:
    - CAPITAL for White pieces
    - lower for black pieces
    - Each row is seperate by "/"(a full string will have all 8 rows)
    - The number represent how many empty squares are between each pieces:
    - Example: 4k2r means: from the top left, 4 empty squares, next square is black king, next is 2 empty squares, next square is black rook
- 2nd part is the current player turn: w for white and b for black
- 3rd part is catsling rights: Q for queenside and K for king side, again, CAPITAL is white rights and lowercase is black rights. Ex: Qk means Queen side for white and k means king side for black.
- Last part is en passant rights:
    - Example: g3, this is the square that a black pawn could move to to do an en passant.

```
2 references
public StateString(Player currentTurn, Board board)
{
    AddPiecesPositions(board);
    sb.Append(' ');
    AddCurrentTurn(currentTurn);
    sb.Append(' ');
    AddCastling(board);
    sb.Append(' ');
    AddEnPassant(board, currentTurn);
}
```

StateString constructor

```
public void Moving(Move move)
{
    Board.SetPawnSkippedSquares(CurrentTurn, null);
    move.Execute(Board);
    CurrentTurn = CurrentTurn.Opponent(); //change turn
    stateString = new StateString(CurrentTurn, Board).ToString(); //update stateString
    CheckEnded();
}
```

It is updated after every move in GameState

Loading and saving:

- **Loading:** We decrypt the stateString that is saved in a gamefile, the path of that file is located in MainWindow.xaml.cs:

```
SaveFilePath = System.IO.Path.Combine(Environment.GetFolderPath(Environment.SpecialFolder.Desktop), "saved_game.txt");
```

A board can be load using the btnLoadGame_Click event handler located in MainMenu:

```
// Check if the save file exists
if (File.Exists(SaveFilePath))
{
    mainMenuControl.btnLoadGame.Visibility = Visibility.Visible;
}
else
{
    mainMenuControl.btnLoadGame.Visibility= Visibility.Collapsed;
}
```

The button only appear if there is a save file

```
private void btnLoadGame_Click(object sender, RoutedEventArgs e)
{
    InitializeGameState(false);
}

2 references
private void InitializeGameState(bool isNewGame)
{
    InitializedBoard();

    if (isNewGame)
    {
        gameState = new GameState(Player.White, Board.SetUp());
        Draw(gameState.Board);
        CurrentTurnText.Text = $"Current Turn: {gameState.CurrentTurn}";
    }
    else
    {
        if (File.Exists(SaveFilePath))
        {
            Board loadedBoard = Board.LoadedBoard(SaveFilePath);
            Player currentTurn = GetCurrentTurn(loadedBoard);
            gameState = new GameState(currentTurn, loadedBoard);
            Draw(gameState.Board);
            CurrentTurnText.Text = $"Current Turn: {gameState.CurrentTurn}";
        }
```

When you click the button it will load using a loadedBoard located in Board.cs

- **Saving:**

```
//Save the current stateString to a savefile
1 reference
public void Save(string filename)
{
    StreamWriter writer = new StreamWriter(filename);

    try
    {
        writer.WriteLine(stateString);
    }
    catch (Exception e)
    {
        Debug.WriteLine("Error saving file: " + e.Message);
    }
    finally
    {
        writer.Close();
    }
}
```

Save the current stateString in GameState

Required Roles

Describe each of the classes, interfaces, and any enumerations you will create. Use a different table to describe each role you will have, using the following table templates.

Table 1: <<Board>> details – duplicate

Responsibility	Type Details	Notes
Create chessboard	Board SetUp() return Board	
Place chess pieces	AddInitialPieces() void	
Check piece's positions	IEnumerable<Positions> PiecesPos() return IEnumerable<Positions>	
Check piece's positions of each color	IEnumerable<Positions> PiecePosColor(Player p) return IEnumerable<Positions>	
Check piece moves inside board	Inside(Positions pos) void	
Check position is empty	Empty(Positions pos) void	
Check if a player is in check	InCheck(Player p) return boolen	
Make a copy of the current	Board Copy() return Board	Prototype pattern

Table 2: <<GameState>> details – duplicate

Responsibility	Type Details	Notes
Move the piece	Moving(Move move) void	
Check a piece's legal moves	IEnumerable<Move> LegalMoves(Positions pos) return IEnumerable<Move>	
Check all legal moves of a player	IEnumerable<Move> AllLegalMoves(Player p) return IEnumerable<Move>	To check stalemate of checkmate
Get result of the game	CheckEnded() void	
Change player's turn	Moving(Move move) void	

Table 3: <<PieceType>> details

Value	Notes
Pawn	
Bishop	
Knight	
Queen	
King	
Rook	

Table 4: <<Player>> details

Value	Notes
Black	
White	
None	

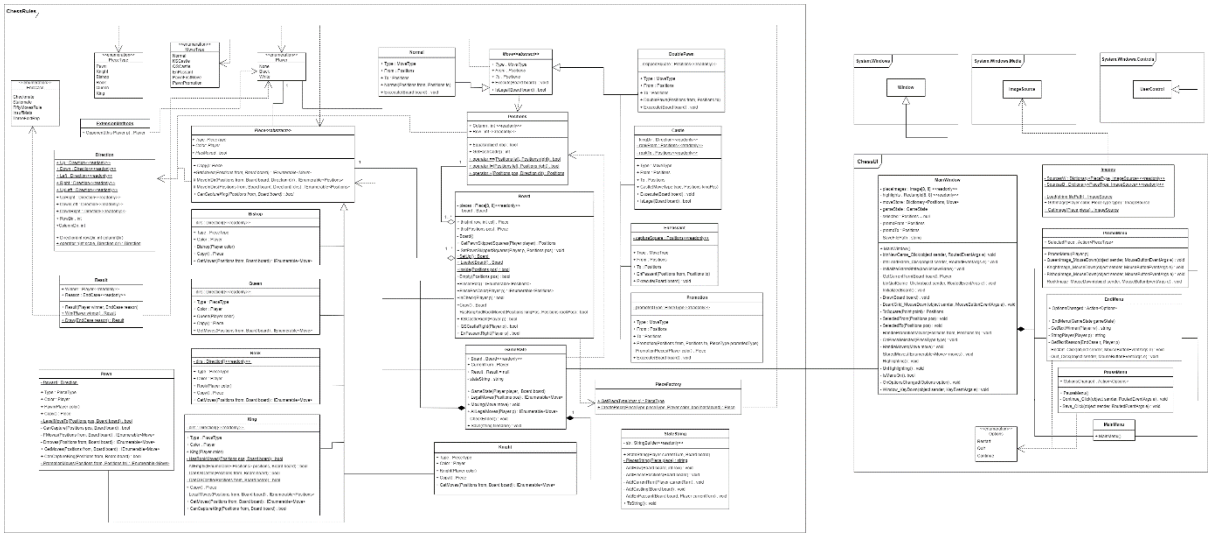
Table 5: <<MoveType>> details

Value	Notes
Normal	
KSCastle	
QSCatsle	
EnPassant	
PawnFirstMove	
PawnPromotion	

Table 6: <<EndCase>> details

Value	Notes
Checkmate	
Bishop	
Knight	
Queen	
King	
Rook	

Class Diagram



The draw.io link if the image is too hard to see: [Click me]

GitHub link for program code [(Click me)]