Chess Game - Documentation

A GUI-Based Chess Game with Human vs Computer and Human vs Human Modes

Author and Developer:
Bilal Yaseen

Date:
December 14, 2024

Abstract

This document provides comprehensive details on the development of a GUI-based Chess game, designed as part of a Data Structures and Algorithms course. The project implements fundamental data structures, including arrays, linked lists, and hash maps, to manage chessboard states and player interactions. The game supports both Human vs Human and Human vs Computer gameplay modes, with varying difficulty levels for the AI opponent.

Table of Contents

List of Figures

List of Tables

1 Project Overview

1.1 Description

This project is a Chess Game developed using the WPF framework in C#. It leverages the .NET Framework version v4.8 and provides an engaging platform for both multiplayer and single-player modes. The single-player mode offers three difficulty levels: easy, medium, and hard. Additionally, the game adheres strictly to FIDE rules, includes time controls (1m, 3m, 5m, and 10m), and supports advanced features like Undo functionality, offering a draw, resign, and move validation. In multiplayer mode, it offers the users to play either regular Chess or Chess960 (Fischer Random Chess).

1.2 Motivation

The motivation for creating this project stemmed from two key factors:

- A desire to learn and understand the practical use of data structures in building real world projects.
- A personal admiration for the game of Chess and an eagerness to develop my own version of it with all the main features and functionalities.

1.3 Objectives

The objectives of the project include the following.

- Demonstrate the use of data structures such as arrays, lists, linked lists, stacks, and graphs in a real-world project.
- Design and implement a chess game that adheres to the FIDE rules. Also integrate the correct moves notation.
- To integrate both multiplayer and Player vs. Computer modes with varying difficulty levels.
- To enhance understanding of WPF and the.NET framework.

1.4 Target Audience

This project aims to:

- Beginners and intermediate programmers who wish to explore the practical application of data structures.
- Developers interested in creating games using WPF and the.NET Framework.
- People who enjoy both Chess and Coding and looking for some fun side activity to broaden their experiences.

1.5 Features

Table 1: Game Features

Feature	Description
Multiplayer Mode	Allows two human players to play against each other on a same device.
Chess960	Fischer Random or Chess960 also available for multiplayer gameplay.
Notation	Proper Display of Moves Notation (according to FIDE rules) for ease of users.
Player vs. Computer Mode	Enables a single player to compete against the AI with three difficulty levels: easy, medium, and hard.
Time Controls	Provides four time control options: 1 minute, 3 minutes, 5 minutes, and 10 minutes.
FIDE-Compliant Rules	Adheres to official Chess rules, including castling, en passant, and pawn promotion.
Undo Functionality	Allows players to undo their last move.
Offer Draw	Players can offer a draw during the game.
Resign	Players can also resign during the game.

1.6 Operational Details

Table 2: Technology Stack

Framework	.NET Framework v4.8.
UI Framework	WPF (Windows Presentation Foundation)
Language	C#
IDE	Visual Studio 2022 Community Edition
Coding Style	Pascal Case for Class members and methods. Camel Case for local variables.

2 Use Cases

2.1 Landing Page

Table 3: Landing Page

Name	Landing page
Actor	Player
Description	It displays the options that the player can explore in this game.

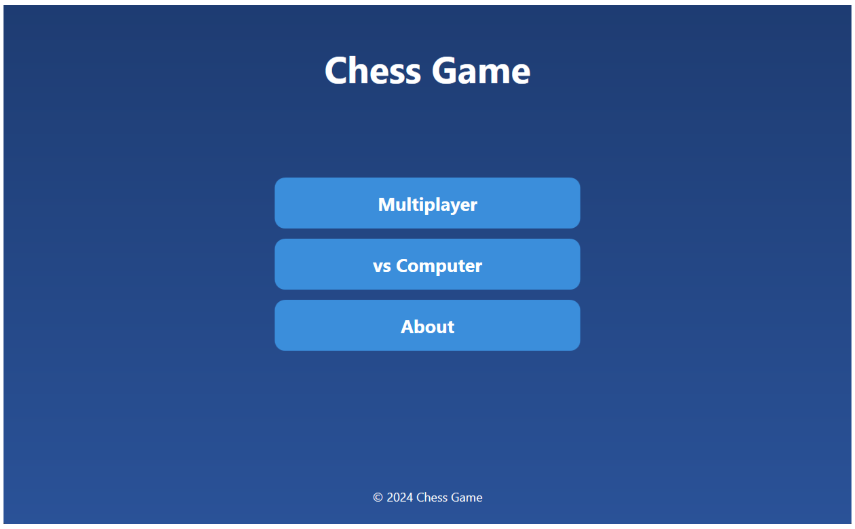
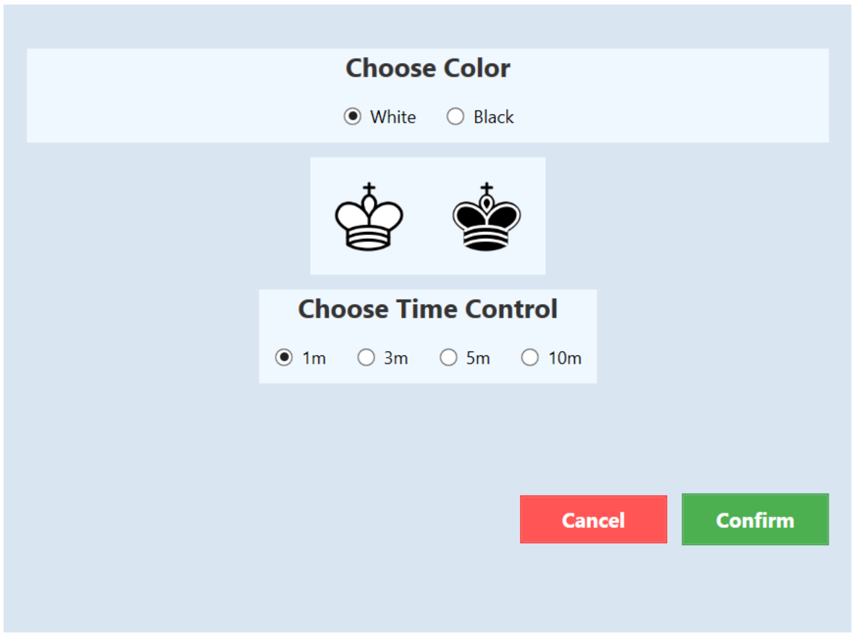


Figure 1: Landing Page

2.2 Multi-Player Select Options

Table 4: Multi-Player Select Options

Name	Multi-player pop-up
Actor	Player
Description	It displays the time controls and the color that the player can select.


Choose Color

(● White) ○ Black

Choose Time Control

(● 1m) ○ 3m ○ 5m ○ 10m

Figure 2: Multi-Player Select Options

2.3 Multi-Player Page

Table 5: Multi-player Page

Name	Game page
Actor	Player
Description	It is the main page where the game is played. It displays the board, moves, dead pieces and the additional buttons for better experience

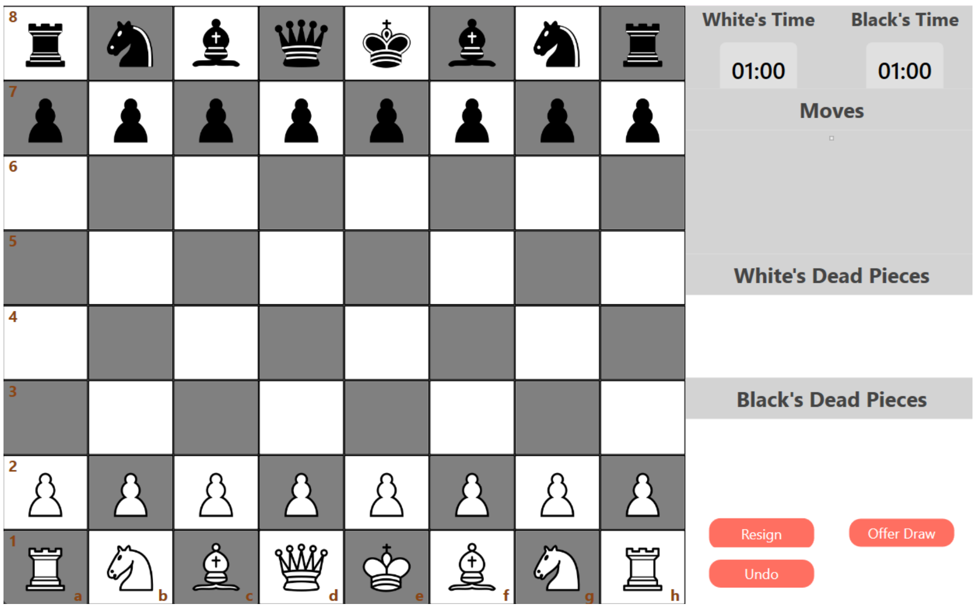


Figure 3: Multi-Player Page

2.4 Promotion Select Options

Table 6: Promotion Select Options

Name	Promotion selection page
Actor	Player
Description	It displays the options (Queen, Knight, Bishop and Rook) that your pawn can promote to on reaching enemies last rank.

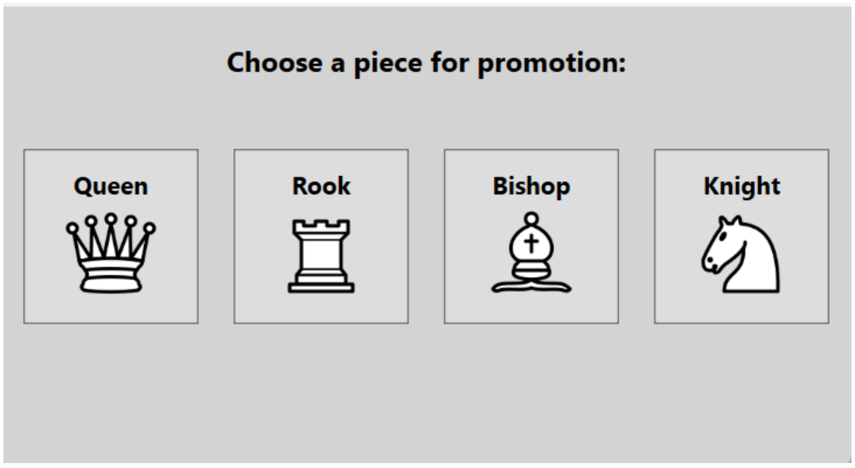


Figure 4: Promotion Select Options

2.5 Multi-Player Game Progress

Table 7: Multi-Player Game Progress

Name	Game page
Actor	Player
Description	It displays current progress of the game. On top right side are the moves notations, and below are both player's dead pieces

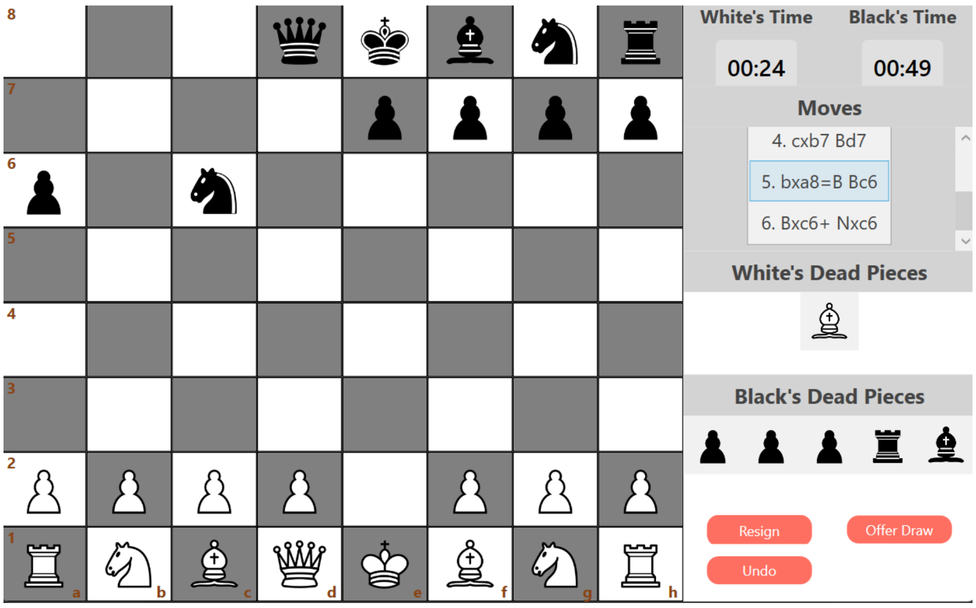


Figure 5: Multi-Player Progress

2.6 Chess960 Page

Table 8: Chess960 Initial Setup

Name	Chess960
Actor	Player
Description	It displays random initial setup of the Chess960 game.

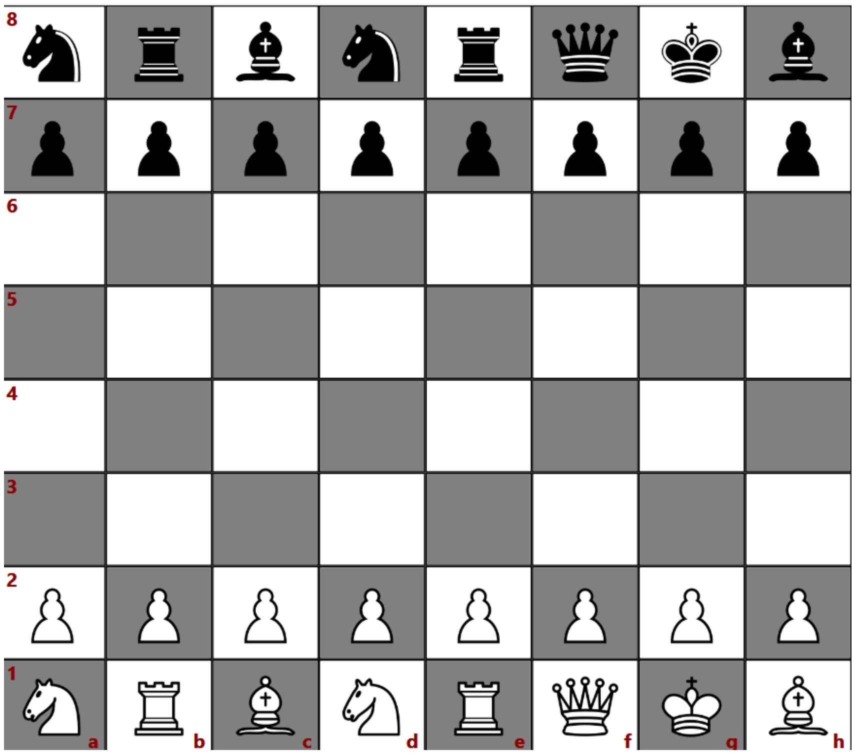
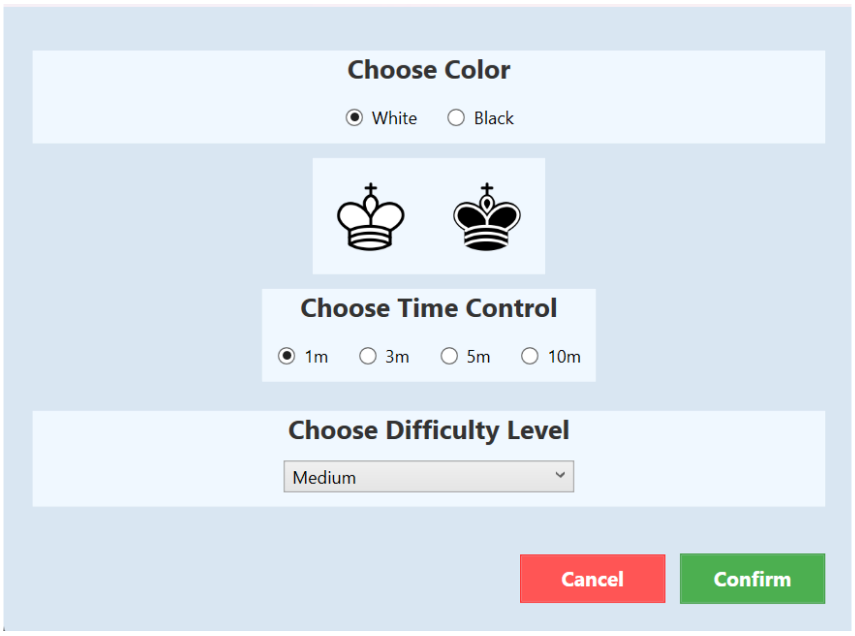


Figure 6: Chess960 setup

2.7 Vs Computer Select Options

Table 9: Vs Computer Select Options

Name	Vs Computer Select Options
Actor	Player
Description	It displays the time controls, difficulty levels and the color that the player can select.



Choose Color: [✓ White] Black

Choose Time Control: [✓ 1m] 3m 5m 10m

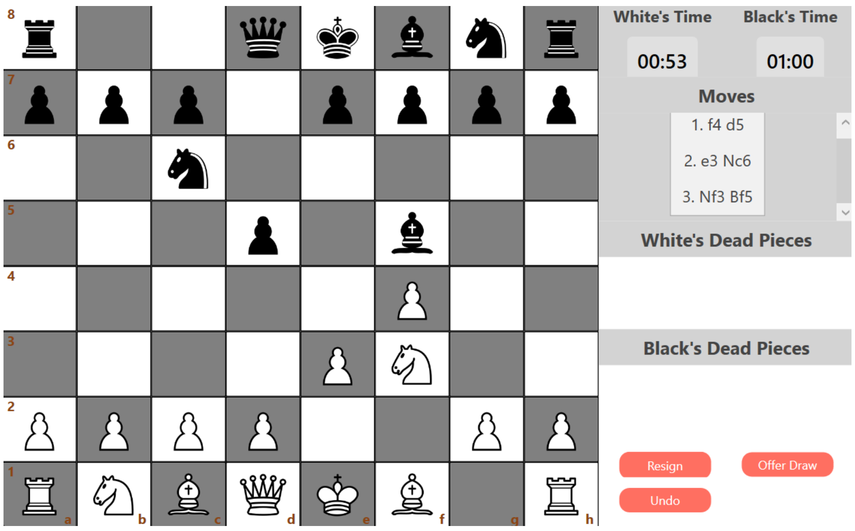
Choose Difficulty Level: Medium

Figure 7: Vs Computer Select Options

2.8 Vs Computer Progress Page

Table 10: Vs Computer Progress Page

Name	Vs Computer Progress page
Actor	Player
Description	It displays current progress of the game. We can clearly see that the time of Black(Computer) is not used at all which shows it's fast move selection.



White's Time: 00:53
Black's Time: 01:00

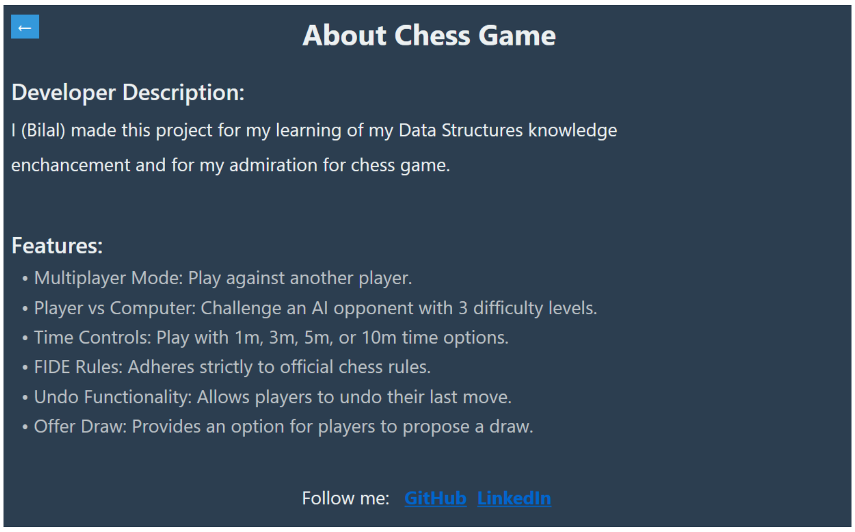
1. f4 d5
2. e3 Nc6
3. Nf3 Bf5

Figure 8: Vs Computer Progress

2.9 About Page

Table 11: About Page

Name	About page
Actor	Player
Description	It displays the developer info, project's features and developer's social links.



About Chess Game

Developer Description:

I (Bilal) made this project for my learning of my Data Structures knowledge enchancement and for my admiration for chess game.

Features:

- Multiplayer Mode: Play against another player.
- Player vs Computer: Challenge an AI opponent with 3 difficulty levels.
- Time Controls: Play with 1m, 3m, 5m, or 10m time options.
- FIDE Rules: Adheres strictly to official chess rules.
- Undo Functionality: Allows players to undo their last move.
- Offer Draw: Provides an option for players to propose a draw.

Follow me: GitHub LinkedIn

Figure 9: About Page

3 Test Cases

3.1 Test Case 1: Knight Valid Moves

Description: Ensure that White's Queenside Knight can move to valid squares.
Expected Outcome: White's Queenside Knight can move to a3 and c3 from b1.
Result: Requirement Satisfied.

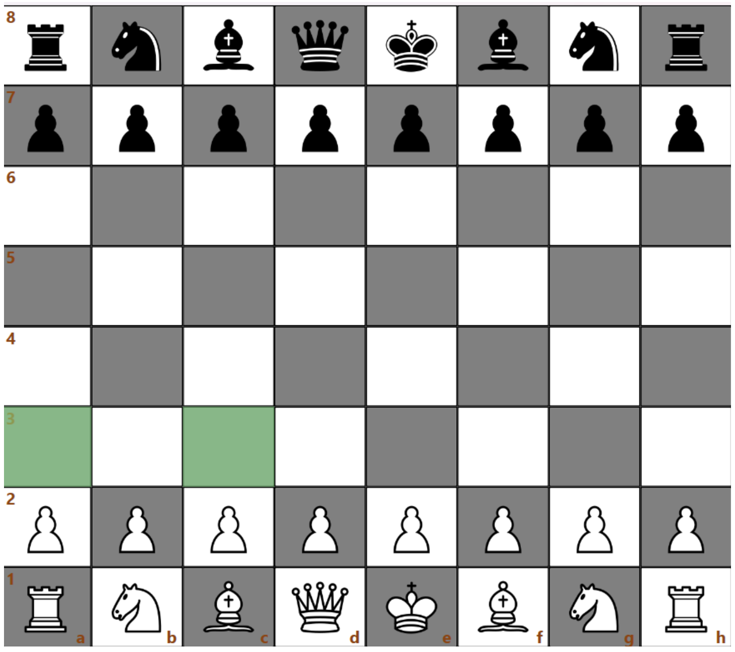


Figure 10: On Clicking on the Knight

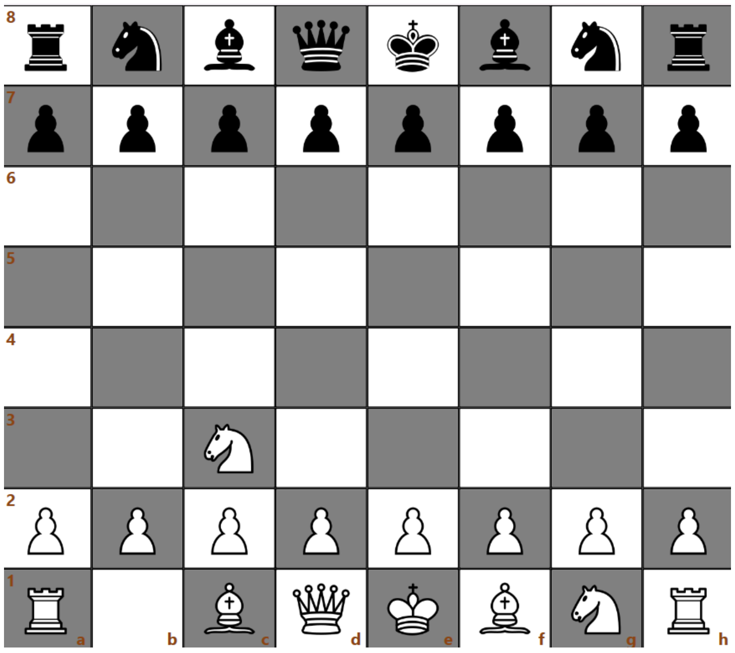


Figure 11: On Clicking on the c3 square

3.2 Test Case 2: En Passant

Description: Ensure that Black can perform En Passant capture on White's pawn.
Expected Outcome: Black can capture White's pawn on g3 using En Passant.
Result: Requirement Satisfied.

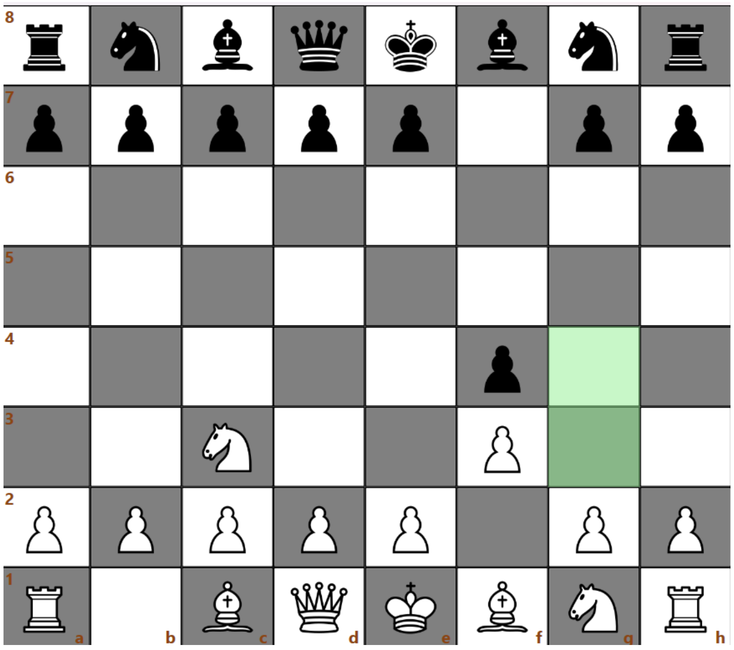


Figure 12: Moving White's Pawn

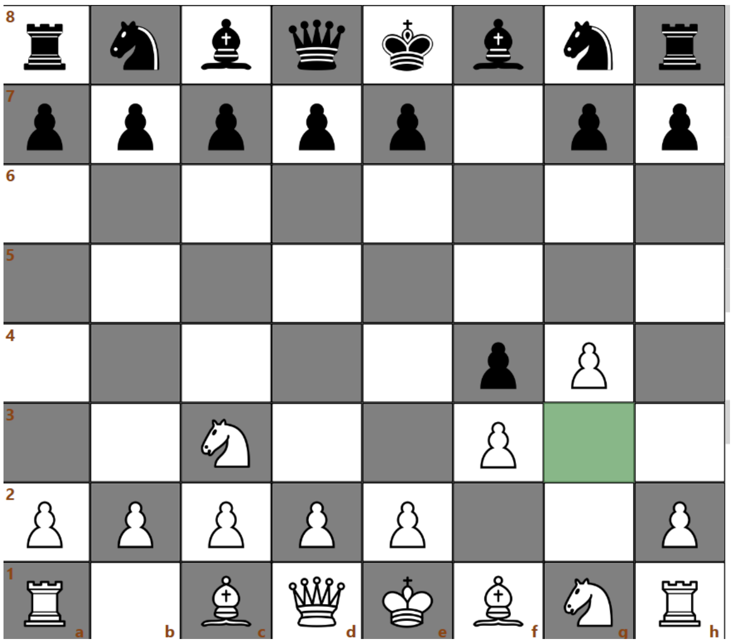


Figure 13: Before En Passant Capture

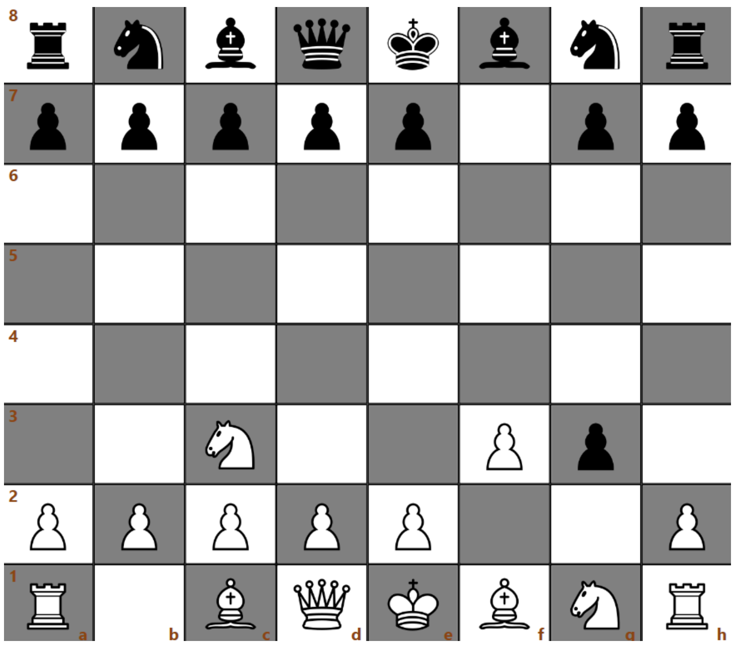


Figure 14: After En Passant Capture

3.3 Test Case 3: Check

Description: Ensure that the game detects when a king is in check.
Expected Outcome: The game correctly highlights the check condition when the White King is threatened by the Black's Dark-Square Bishop.
Result: Requirement Satisfied.

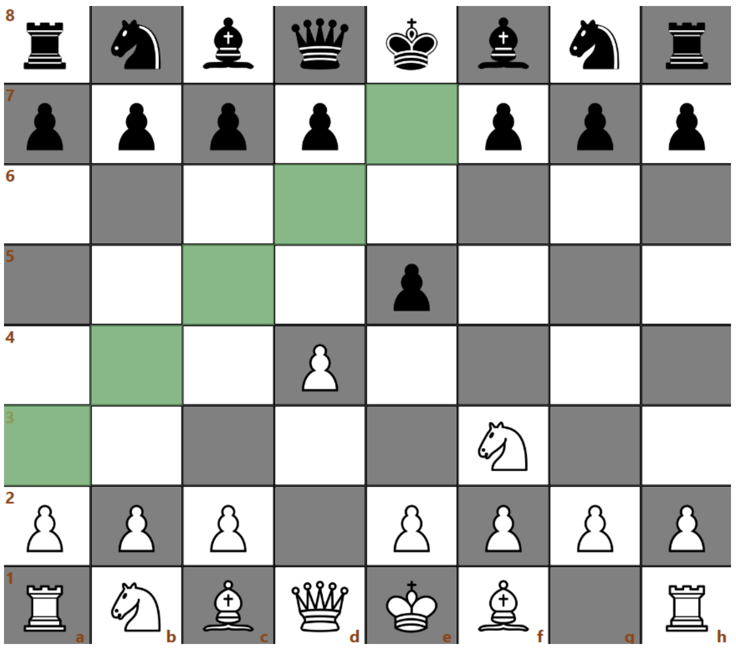


Figure 15: Before Check

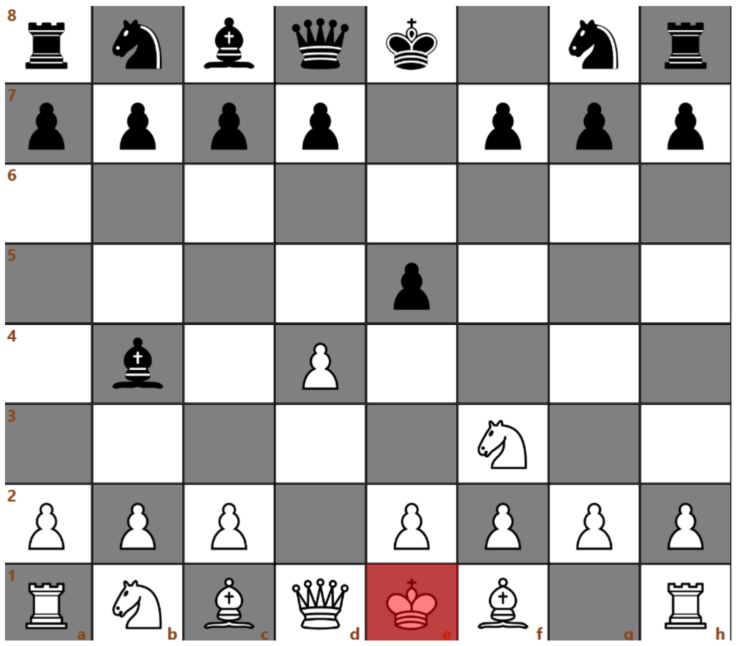


Figure 16: After Check

3.4 Test Case 4: En Passant Undo

Description: Ensure that an En Passant move can be undone correctly.
Expected Outcome: En Passant capture can be undone, and the pawn returns to its original position.
Result: Requirement Satisfied.

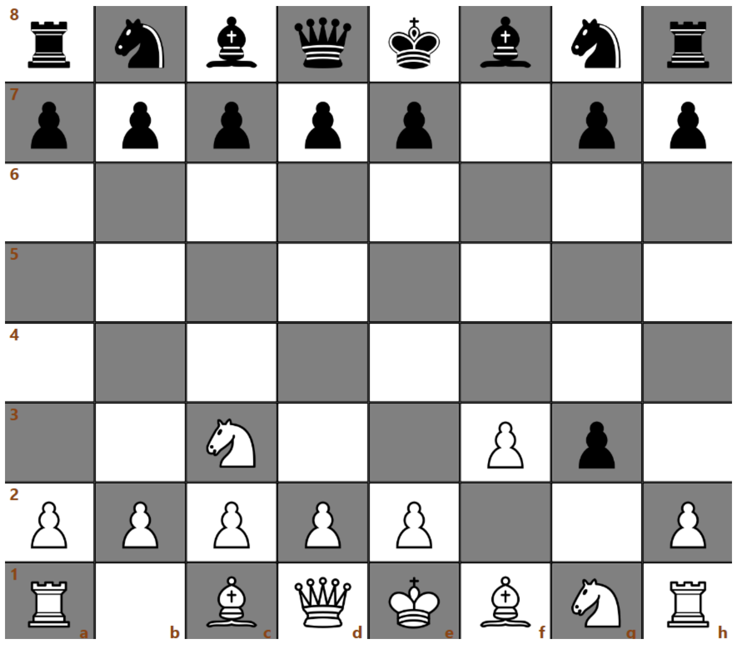


Figure 17: Before Undoing En Passant

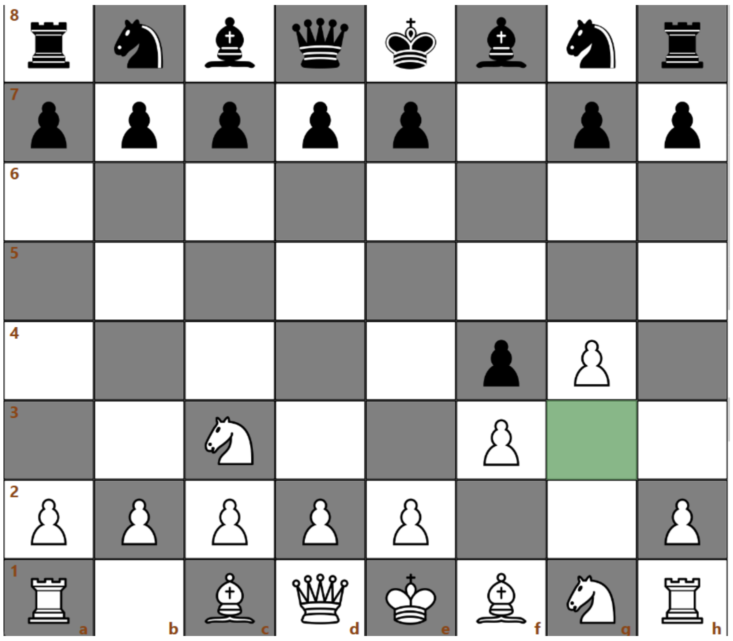
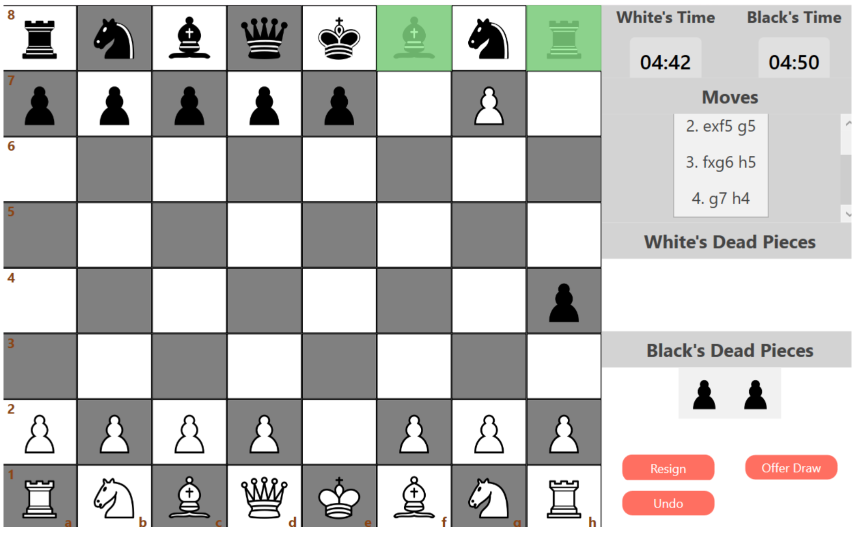


Figure 18: After Undoing En Passant

3.5 Test Case 5(Part 1): Notation for Promotion

Description: Ensure that the promotion notation is displayed correctly when a pawn reaches the 8th rank.
Expected Outcome: The game displays the correct notation for the promotion of a White pawn to a Queen.
Result: Requirement Satisfied.



2. exf5 g5
3. fxg6 h5
4. g7 h4

Figure 19: Before Promotion Notation

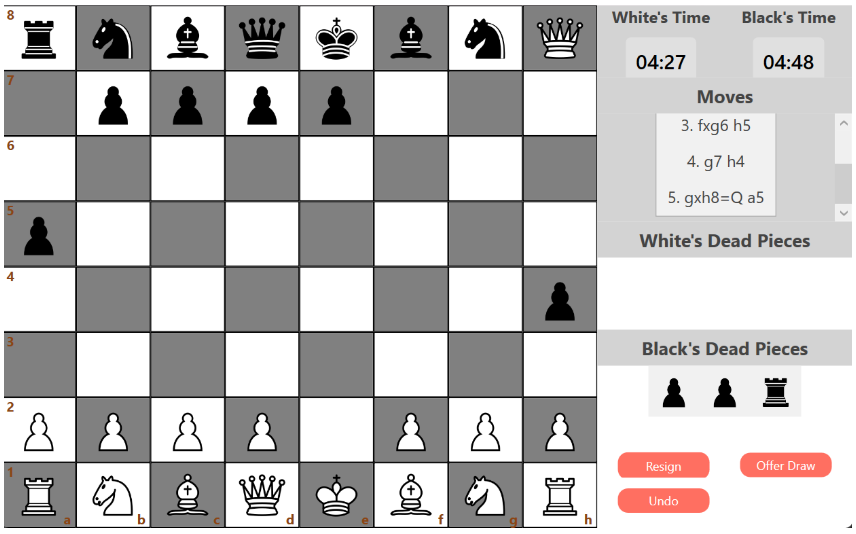


Figure 20: After Promotion Notation

3.6 Test Case 5(Part 2): Notation for Pieces Attacking Same Block

Description: Ensure that pieces attacking the same block are highlighted correctly.
Expected Outcome: The game correctly highlights the squares where multiple pieces are attacking.
Result: Requirement Satisfied.

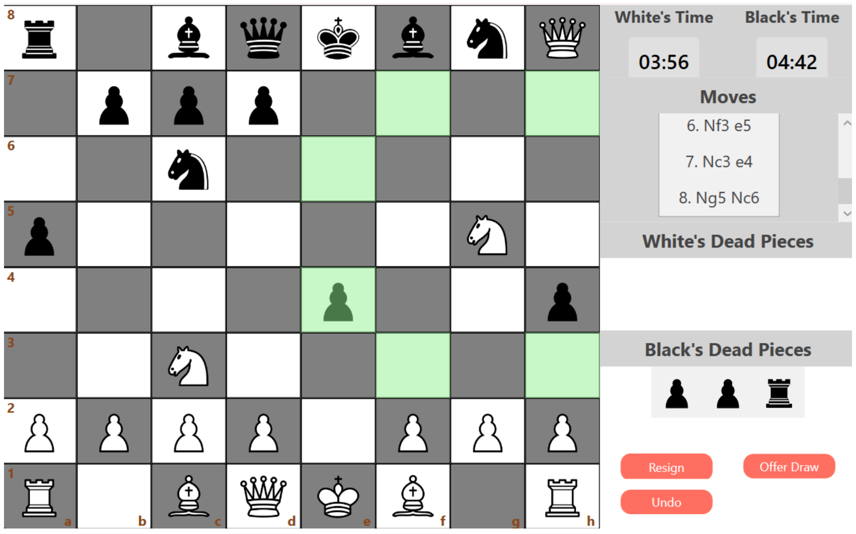


Figure 21: Before Pieces Attacking Same Block

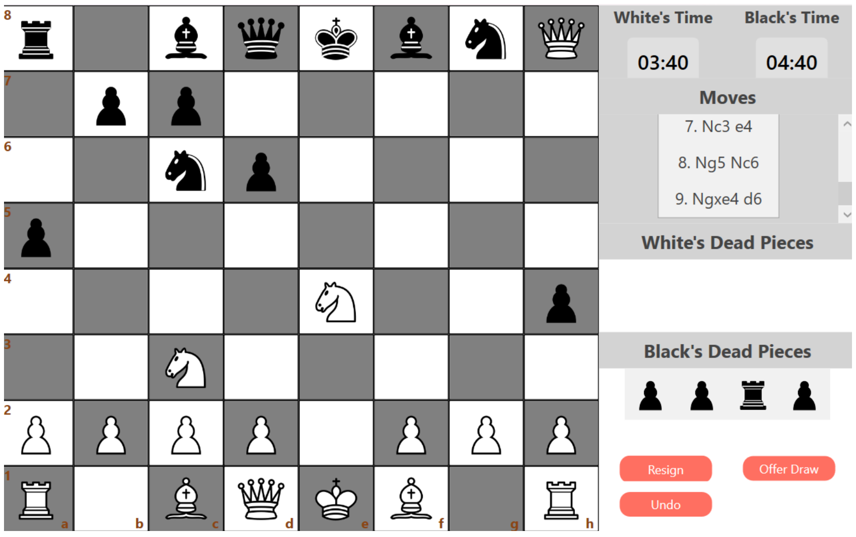


Figure 22: After Pieces Attacking Same Block

3.7 Test Case 6: Castling

Description: Ensure that Black can perform short castle with the king and rook.
Expected Outcome: Black can castle by moving the King from e8 to g8 and the Rook from h8 to f8.
Result: Requirement Satisfied.

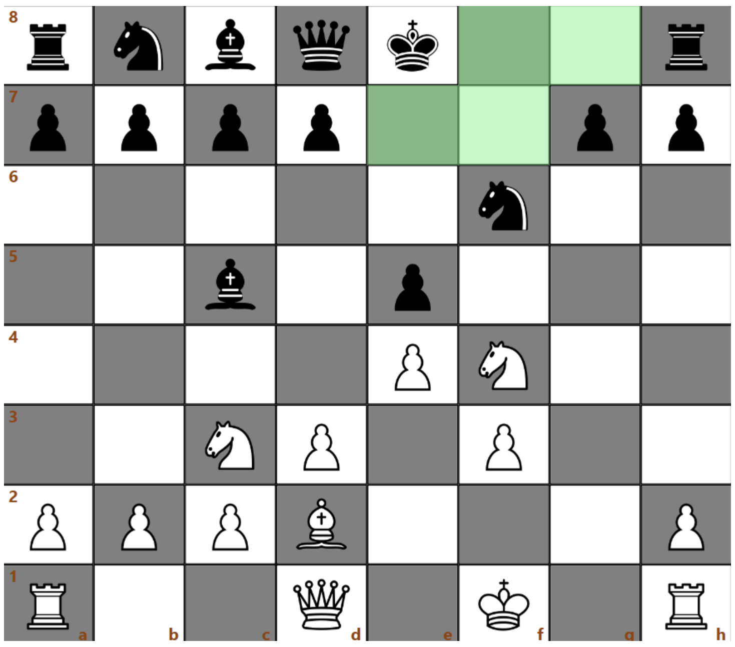


Figure 23: Before Castling

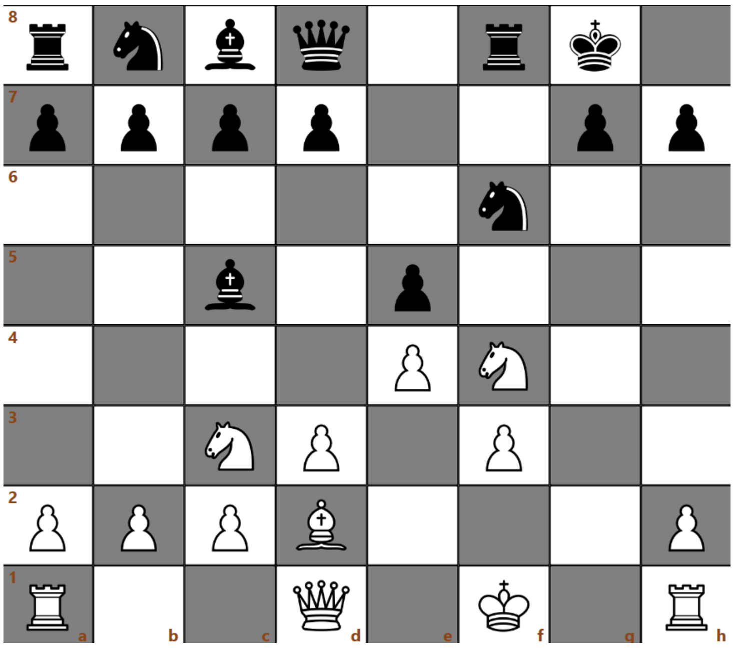


Figure 24: After Castling

3.8 Test Case 7: Scholar's Mate

Description: Ensure that Scholar's Mate can be achieved by White in 4 moves.
Expected Outcome: White delivers Scholar's Mate by moving the Queen to f3 and checkmates the Black King.
Result: Requirement Satisfied.

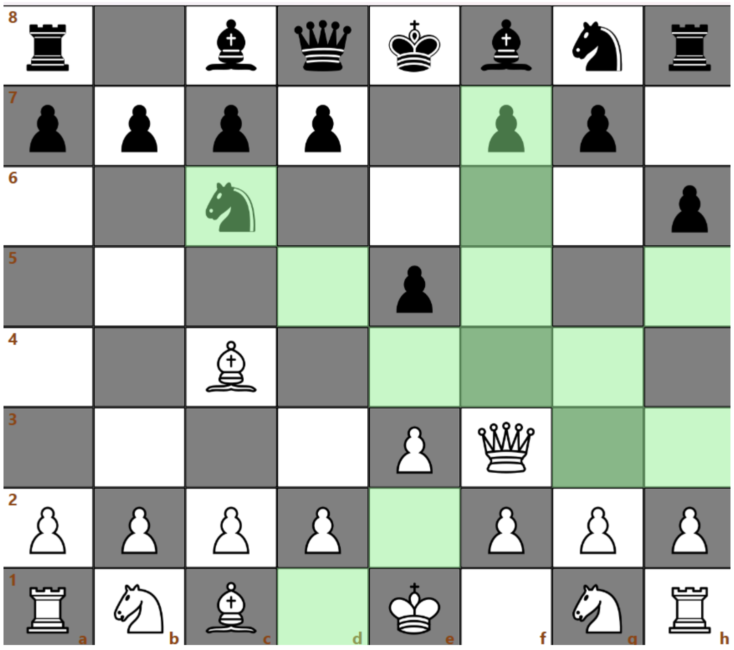
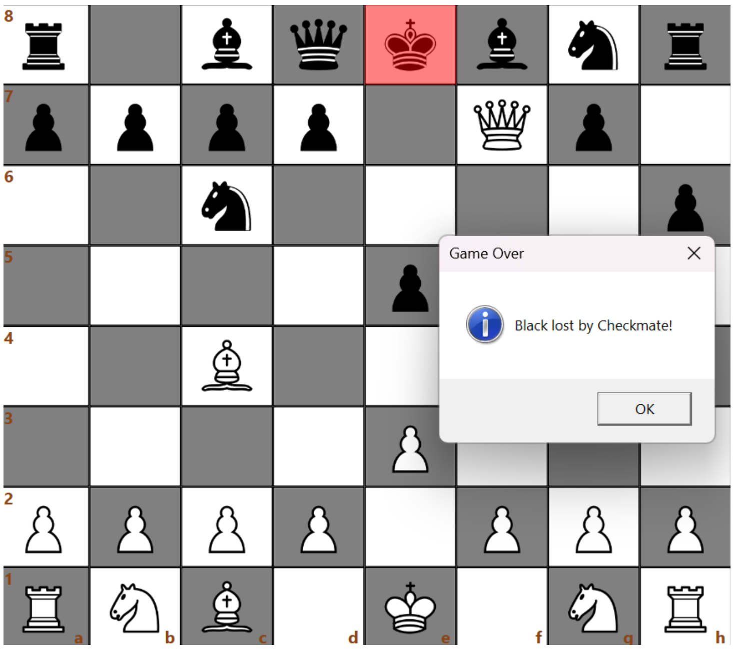


Figure 25: Before Scholar's Mate



Game Over

Black lost by Checkmate!

Figure 26: After Scholar's Mate

3.9 Test Case 8: Check by Castling

Description: Ensure that Check by castling is possible.
Expected Outcome: The game detects that castling for an open d file leads to check by rook of opponent king.
Result: Requirement Satisfied.

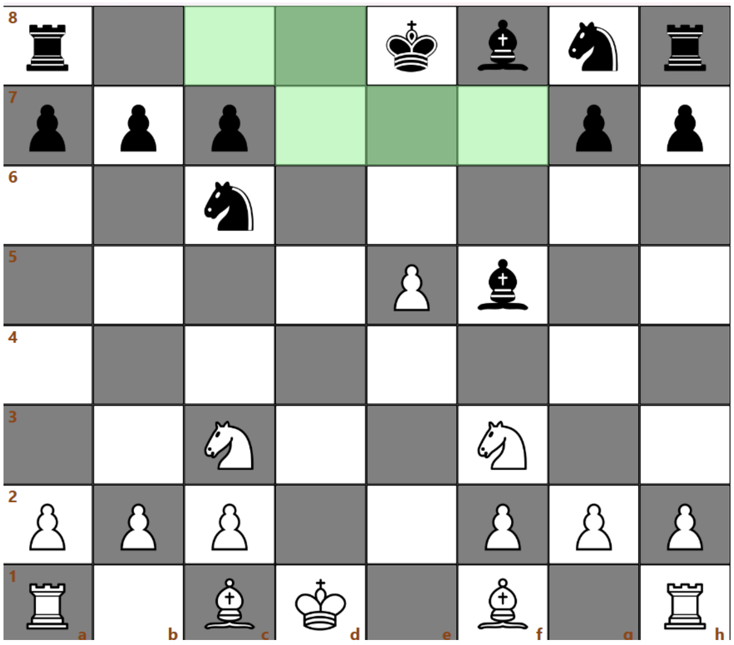


Figure 27: Before Check by Castling

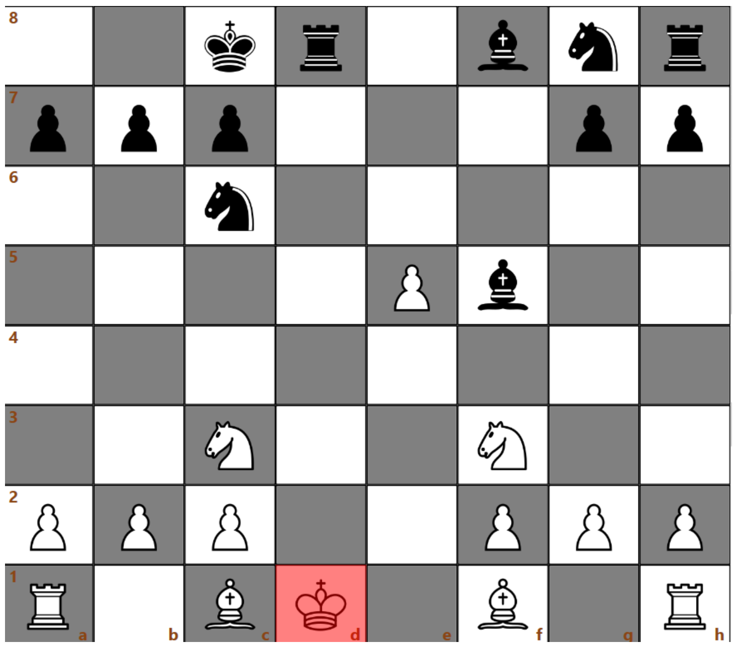


Figure 28: After Check by Castling

3.10 Test Case 9: Promotion Check

Description: Ensure that a promotion results in check if the opponent's king is in danger.
Expected Outcome: The promotion results in check if the opponent's king is in check.
Result: Requirement Satisfied.

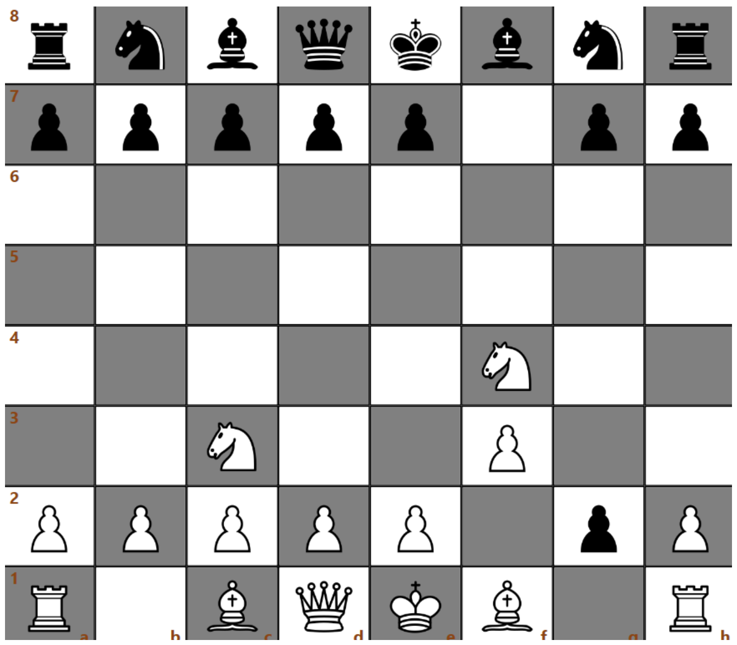


Figure 29: Before Promotion Check

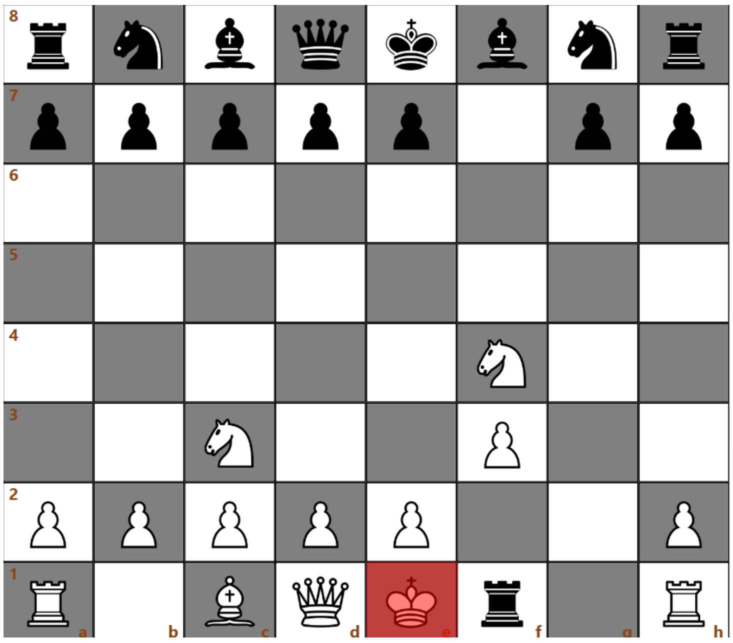


Figure 30: After Promotion Check

3.11 Test Case 10: Computer Move

Description: Ensure that the computer makes a valid move according to the chosen difficulty.
Expected Outcome: The computer makes a valid move on the board for each difficulty level.
Result: Requirement Satisfied.

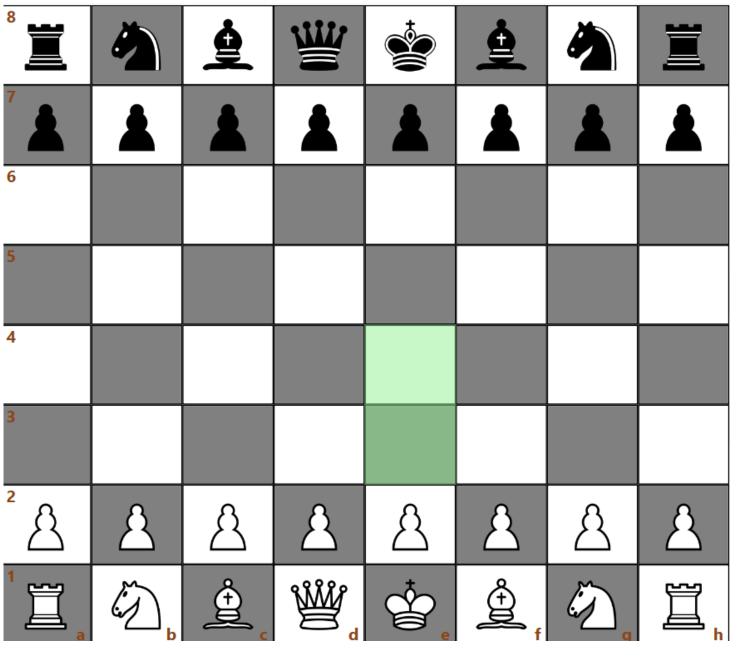


Figure 31: Before Computer Move

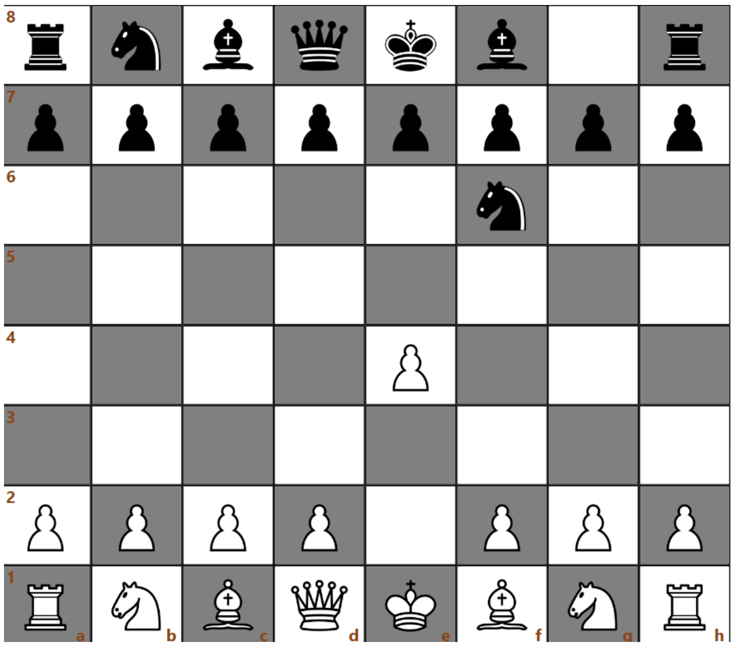


Figure 32: After Computer Move

4 ER Diagram

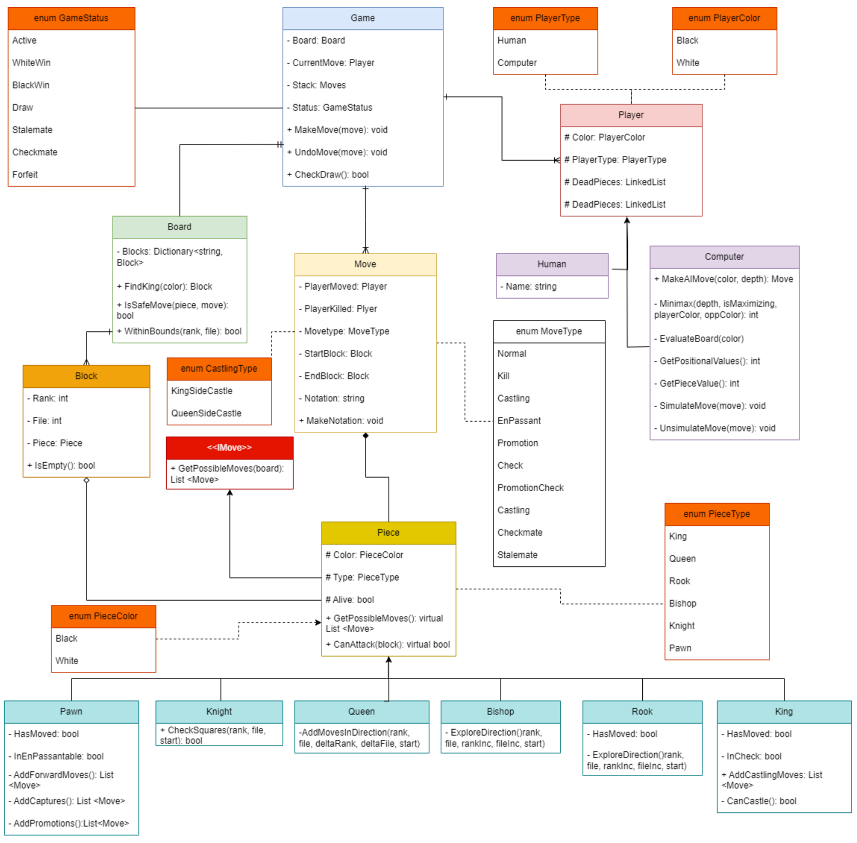


Figure 33: ER Diagram

5 Implementation

5.1 Utility Classes

The following utility classes provide the functionality required to manage the game's state, players, and pieces:

- **Game**: The Game class orchestrates the entire game flow, including managing the turns, checking for win conditions, and initializing the board. It handles player interactions and keeps track of the game state.
- **Player**: This is the base class for both human and computer players. It defines attributes common to all players and includes methods for making moves and interacting with the board.
- **Board**: The Board class represents the chessboard, holding the blocks and pieces on the board. It is responsible for initializing the board, placing pieces in their starting positions, and updating the board state.
- **Block**: A Block represents an individual square on the chessboard, storing information about the piece occupying the square and its rank and file.
- **Move**: The Move class represents a single move made by a player. It stores the starting and ending positions, along with any special move actions like castling or en passant.
- **Piece**: The base class for all chess pieces. It defines common attributes and movement rules for the pieces, including color, type, and position.
- **King, Pawn, Queen, Bishop, Knight, Rook**: These classes represent specific types of chess pieces, inheriting from the Piece class and implementing their unique movement rules.
- **Human**: The Human class represents a human player, responsible for making moves through user input. It interacts with the game board and manages human-specific gameplay logic.
- **Computer**: The Computer class represents an AI player. It uses algorithms to evaluate moves and make decisions based on the state of the game.

5.2 Data Structures Used

The data structures utilized in the Chess game are designed to support efficient gameplay, move management, and game state tracking:

- **Array**: Arrays are used for small, fixed-size tasks such as storing specific piece attributes or temporarily holding data during computations.
- **List**: A List is used to store the possible moves for a player. It allows efficient iteration over all possible moves and provides an easy way to manage and evaluate them.

- **LinkedList**: The LinkedList is used to store dead pieces, keeping track of captured pieces. This allows easy modification and retrieval of captured pieces throughout the game.
- **Stack**: The Stack data structure is employed for move management and undo functionality. It stores a history of moves, allowing players to undo actions using the last-in, first-out (LIFO) principle.
- **Adjacency List (Graph)**: The adjacency list represents the connections between blocks on the board as a graph. It treats each block on the screen as node and it's possible moves are edges. It helps efficiently manage the possible movements of pieces by storing neighboring blocks.

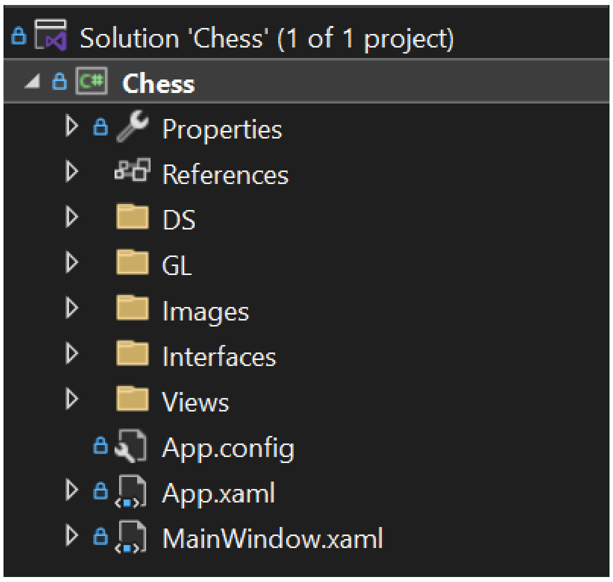
5.3 Algorithms Used

The following algorithms are used to enhance decision-making and evaluate game positions:

- **Minimax**: The Minimax algorithm is a decision-making strategy used by the AI to determine the best possible move by exploring all possible moves and selecting the one that maximizes the AI's chances of winning while minimizing the opponent's advantage.
- **Alpha-Beta Pruning**: Alpha-Beta pruning is an optimization of the Minimax algorithm. It reduces the number of nodes evaluated in the game tree by pruning branches that will not affect the outcome, thus improving performance and making the AI faster.

5.4 Directory Structure

This figure shows the directories of the project:



```
Solution 'Chess' (1 of 1 project)
  Chess
    Properties
    References
    DS
    GL
    Images
    Interfaces
    Views
    App.config
    App.xaml
    MainWindow.xaml
```

Figure 34: Directory Structure

- **DS**: It stores the class files for the data structures that are custom implemented for this project.
- **GL**: Game Logic (GL) directory stores all the files that are necessary for game logic such as Game.cs, Player.cs, Board.cs, etc. It further contains two subdirectories:
  - **Players**: Stores the implementation for Human and Computer classes.
  - **Pieces**: Contains all the pieces such as Pawn, King, Knight, etc.
- **Images**: Contains the images of all the chess pieces.
- **Interface**: Stores the interface IMove.cs that is used in this project to get the possible moves of each piece.
- **Views**: This folder contains all the User Interface files. All of the xaml and xaml.cs code is stored here.

5.5 Code

- **FindKing(PieceColor pieceColor)**: This function searches for the King piece of the specified color on the board. It iterates through all the blocks and checks if a block contains a piece of type King and the specified color. If found, it returns the block that contains the King.

```
public Block FindKing(PieceColor pieceColor)
{
    return Blocks.Values.FirstOrDefault(block =>
    {
        Piece piece = block.GetPiece();
        return piece?.GetPieceType() == PieceType.King && piece.GetColor() ==
    });
}
```

- **IsKingInCheck(PieceColor pieceColor)**: This function checks if the King of the specified color is in check. It first finds the King using the `FindKing` method. Then, it checks all blocks on the board to see if any opponent piece can attack the King, considering both pawns and other pieces.

```
public bool IsKingInCheck(PieceColor pieceColor)
{
    Block kingBlock = FindKing(pieceColor);
    if (kingBlock == null)
        return false;

    foreach (var block in Blocks.Values)
    {
```

```
                Piece attackingPiece = block.GetPiece();
                if (attackingPiece == null || attackingPiece.GetColor() == pieceColor
                    continue;

                if (attackingPiece.GetPieceType() == PieceType.Pawn && attackingPiece
                    return true;

                if (attackingPiece.CanAttack(kingBlock, this))
                    return true;
            }
            return false;
        }
```

- **WithinBounds(int rank, int file)**: This function checks if the given rank and file are within the bounds of the chessboard (0 to 7). It returns `true` if the coordinates are valid, and `false` otherwise.

```
        public bool WithinBounds(int rank, int file)
        {
            return rank >= 0 && rank < 8 && file >= 0 && file < 8;
        }
```

- **AddMovesInDirection(int rank, int file, int deltaRank, int deltaFile, Block startBlock, Board board)**: This method adds valid moves in a specified direction to the list of possible moves for a piece. It moves step by step in a given direction (specified by `deltaRank` and `deltaFile`) and checks if the destination block is empty or contains an opponent's piece. It stops further movement when an obstacle is encountered.

```
        private void AddMovesInDirection(int rank, int file, int deltaRank, int delta
        {
            for (int i = 1; i < 8; i++)
            {
                int newRank = rank + i * deltaRank;
                int newFile = file + i * deltaFile;

                if (!board.WithinBounds(newRank, newFile))
                    break;

                Block endBlock = board.GetBlock(newRank, newFile);

                if (endBlock.IsEmpty() && board.IsSafeMove(this, endBlock))
                {
                    possibleMoves.Add(new Move(startBlock, endBlock, this, null));
                }
                else
                {
                    if (endBlock?.GetPiece() != null) // stop further movement in thi
                    {
                        if (endBlock.GetPiece().GetColor() != this.GetColor() && boar
                        {
                            possibleMoves.Add(new Move(startBlock, endBlock, this, en
                        }
                        break;
                    }
                    if ((endBlock.GetPiece() != null && endBlock.GetPiece().GetColor(
                    {
                        possibleMoves.Add(new Move(startBlock, endBlock, this, endBlo
                    }
                }
            }
        }
```

- **CanAttack(Block targetBlock, Board board)**: This method determines if a piece can attack a target block. It checks if the piece can move to the target block either diagonally, horizontally, or vertically. The function returns `true` if the path is clear, allowing the piece to attack the target block.

```
    public override bool CanAttack(Block targetBlock, Board board)
    {
        Block currentBlock = board.GetBlock(this);
        int rank = currentBlock.GetRank();
        int file = currentBlock.GetFile();

        int targetRank = targetBlock.GetRank();
        int targetFile = targetBlock.GetFile();

        if (Math.Abs(rank - targetRank) == Math.Abs(file - targetFile)) // Diagon
        {
            return board.IsPathClear(rank, file, targetRank, targetFile);
        }
        else if (rank == targetRank || file == targetFile) // Horizontal or verti
        {
            return board.IsPathClear(rank, file, targetRank, targetFile);
        }

        return false;
    }
```

- **EnPassant(PieceColor pieceColor, Block currentBlock, int rank, int file, int direction)**: This function checks for the possibility of an en passant capture. It checks if the current rank is the rank where en passant can occur (based on the piece color). If the rank matches, it checks adjacent blocks for opponent pawns that are eligible for en passant and adds possible moves for the current piece to capture them.

```
public void EnPassant(PieceColor pieceColor, Block currentBlock, int rank, in
{
    int enPassantRank = (GetColor() == PieceColor.White) ? 3 : 4;

    if (board.GetFirstPlayerColor() == PlayerColor.Black)
        enPassantRank = (GetColor() == PieceColor.White) ? 4 : 3;

    if (rank == enPassantRank)
    {
        foreach (int offset in new[] { -1, 1 })
        {
            if (!board.WithinBounds(rank, file + offset))
            {
                continue;
            }

            Block sideBlock = board.GetBlock(rank, file + offset);
            if (sideBlock != null && !sideBlock.IsEmpty() &&
                sideBlock.GetPiece().GetPieceType() == PieceType.Pawn &&
                ((Pawn)sideBlock.GetPiece()).GetEnPassantable() &&
                sideBlock.GetPiece().GetColor() != GetColor())
            {
                Block endBlock = board.GetBlock(rank + direction, file + offs
                if (board.IsSafeMove(this, endBlock))
                    moves.Add(new Move(currentBlock, endBlock, this, sideBloc
            }
        }
    }
}
```

- **MakeNotation()**: This function generates the algebraic notation for the move made by the piece. It handles different types of moves such as castling, capturing, and promotion. It also takes into account check and checkmate scenarios, appending appropriate symbols to the notation.

```
public void MakeNotation()
{
    if (MoveType == MoveType.Draw)
```

```
                return;

            if (MoveType == MoveType.Castling && CastlingType == CastlingType.KingSic
            {
                Notation = "O-O";
                return;
            }
            if (MoveType == MoveType.Castling && CastlingType == CastlingType.QueenSi
            {
                Notation = "O-O-O";
                return;
            }

            if (PieceMoved.GetPieceType() != PieceType.Pawn)
            {
                Notation = GetPieceMovedString(PieceMoved.GetPieceType());
                // Now we check if any any piece can move to same square
                Piece otherPiece = Board.OtherPieceCanMove(PieceMoved, EndBlock);
                if (otherPiece != null)
                {
                    if (Board.GetBlock(otherPiece).GetFile() != StartBlock.GetFile())
                    {
                        Notation += GetFileString(StartBlock.GetFile());
                    }
                    else if (Board.GetBlock(otherPiece).GetFile() == StartBlock.GetFi
                         && Board.GetBlock(otherPiece).GetRank() != StartBlock.GetRa
                    {
                        Notation += Board.TranslateRank(StartBlock.GetRank());
                    }
                    else // if both are same write rank and file both
                    {
                        Notation += GetFileString(StartBlock.GetFile()) + Board.Trans
                    }
                }
                if (PieceKilled == null)
                {
                    Notation += GetFileString(EndBlock.GetFile());
                }
            }
            else
            {
                Notation = GetPieceMovedString(PieceMoved.GetPieceType()) + GetFileSt
            }

            if (PieceKilled != null)
                Notation += "x" + GetFileString(EndBlock.GetFile());
```

```
        Notation += Board.TranslateRank(EndBlock.GetRank());

        if (MoveType == MoveType.Promotion)
            Notation += "=" + GetPieceMovedString(PromotedPieceType);

        if (MoveType == MoveType.Checkmate)
            Notation += "#";
        else if (MoveType == MoveType.Check)
            Notation += "+";

        if (MoveType == MoveType.PromotionCheck)
            Notation += "=" + GetPieceMovedString(PromotedPieceType) + "+";
    }
```

- **EvaluateBoard(PlayerColor aiColor)**: This function evaluates the board state from the perspective of the AI's color. It sums up the values of all pieces on the board, factoring in the piece's type and its positional value on the board. The score is positive for the AI's color and negative for the opponent's color.

```
    private int EvaluateBoard(PlayerColor aiColor)
    {
        int score = 0;

        foreach (var block in Board.GetBlocks().Values)
        {
            if (!block.IsEmpty())
            {
                Piece piece = block.GetPiece();
                int pieceValue = GetPieceValue(piece.GetPieceType());
                int positionalBonus = GetPositionalValue(piece, block);

                score += (piece.GetColor().ToString() == aiColor.ToString())
                         ? (pieceValue + positionalBonus)
                         : -(pieceValue + positionalBonus);
            }
        }
        return score;
    }
```

- **GetPositionalValue(Piece piece, Block block)**: This function calculates the positional value of a piece based on its location on the board. The value rewards control of the center and provides bonuses for pieces in strategic positions.

```
private int GetPositionalValue(Piece piece, Block block)
{
    // Reward control of the center
    int[,] positionalValues = new int[8, 8]
    {
        { 1, 1,  1,  1,  1,  1, 1, 1 },
        { 1, 5,  5,  5,  5,  5, 5, 1 },
        { 1, 5, 10, 10, 10, 10, 5, 1 },
        { 1, 5, 10, 20, 20, 10, 5, 1 },
        { 1, 5, 10, 20, 20, 10, 5, 1 },
        { 1, 5, 10, 10, 10, 10, 5, 1 },
        { 1, 5,  5,  5,  5,  5, 5, 1 },
        { 1, 1,  1,  1,  1,  1, 1, 1 }
    };

    return positionalValues[block.GetRank(), block.GetFile()];
}
```

- **GetPieceValue(PieceType pieceType)**: This function assigns a value to a piece based on its type. The values are commonly used in chess AI for evaluating material advantage.

```
private int GetPieceValue(PieceType pieceType)
{
    switch (pieceType)
    {
        case PieceType.Pawn:
            return 1;
        case PieceType.Knight:
            return 3;
        case PieceType.Bishop:
            return 3;
        case PieceType.Rook:
            return 5;
        case PieceType.Queen:
            return 9;
        case PieceType.King:
            return 90;
        default:
            return 0;
    }
}
```

6 Repository Link

The repository for this project on GitHub: https://github.com/mb0227/Chess.
You can also just run the setup file and enjoy the game. Setup. It is also available in github repository of this project as a zip file named as Setup.zip.

7 Conclusion

7.1 Summary

This Chess Game Project is designed to understand the practical application of data structures in real-world projects. The game was developed using WPF and the .NET Framework, specifically .NETFramework, Version=v4.8. It supports both Multiplayer (regular and Chess960) and Player Vs Computer modes, with three levels of AI difficulty: Easy, Medium, and Hard. The project also features time controls, allowing users to choose between 1-minute, 3-minute, 5-minute, or 10-minute time limits. The game follows the FIDE rules, including specialized moves like En Passant and Castling, and offers Undo functionality along with the ability to draw.

This project helped me deepen my understanding of implementing data structures, like stacks, linked lists, and graphs, in complex scenarios. It also provided a comprehensive view of how to manage game logic and UI components, and how various design patterns can be applied to create a smooth gaming experience.

7.2 Challenges Faced

- Developing the game using WPF for the first time, as I had no prior experience with it before.
- Working solo on the project, which meant managing all aspects of development by myself.
- Debugging special moves like En Passant and Castling, especially when ensuring that the first player could select either White or Black.
- Ensuring that the board accurately reflected moves and changes in dead pieces in the UI. This required seamless integration of game logic and the user interface.
- Another big challenge was while undoing a move, especially in the moves in which a piece was killed, removing it from the UI was quite a challenge but with a lot of research I overcame it.
- But the challenge which was really hard to overcome was implementing Chess960. Initial setup of Fischer Random was pretty straight-forward as most of the code was already in good structure, but coding the castling part in Chess960 was really hectic and challenging.

7.3 Project's Weaknesses

- Sometimes a logical error occurs when trying castle in Chess960, but mostly it works fine.
- The AI functionality is somewhat weak, as it only evaluates moves based on three factors: king safety, positional value and piece value. This limited evaluation results in a less challenging AI opponent.
- Introducing additional evaluation conditions causes stack overflow errors, indicating the AI's evaluation logic needs to be improved.

7.4 Future Enhancements

- Add Redo move functionality, allowing players to redo moves after undoing them.
- Implement Pre-Move functionality using Queue data structure, enabling players to plan their moves in advance.
- Integrate the game with the internet, as it is currently set up locally.
- Improve the UI, including displaying move notations simultaneously for both players, instead of after both players have made their moves which is the current implementation.
- Enhance the AI by evaluating the board in greater depth, considering more factors for better move determination.
- Use of data structures like Trees to do best move analysis.

8 References

- **WPF Documentation - Microsoft:** `https://docs.microsoft.com/en-us/dotnet/desktop/wpf/`.
- **Introduction to AI in Chess - ChessBase:** `https://www.chessbase.com/`.
- **C# Programming Guide - Microsoft:** `https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/`.
- **Chess.com:** `https://www.chess.com`.
- **Lichess**: `https://lichess.org`.
- **GeeksforGeeks**: `https://www.geeksforgeeks.org`.
- **FIDE (Fédération Internationale des Échecs)**: `https://www.fide.com`.
- **Chess Talk - Chess Notations**: `https://www.youtube.com/watch?v=b6PR885Rgb8`.
- **Kampa Plays - WPF Tutorial**: `https://www.youtube.com/watch?v=t9ivUosw_iI&list=PLih2KERbY1HHOOJ2C6FOrVXIwg4AZ-hk1`.

- **Green Chess - Piece Images**: `https://greenchess.net/info.php?item=downloads`.
- **ER Diagram**: `https://app.diagrams.net/`.