CSC 323: Object Oriented Design

Chess Game

Project Report

Submitted by:

Student ID	Name	Major
20198045	**Karim Hamadi**	**CCE**
20208022	**Firas Dimashki**	**CCE**

Project Description

This project is a complete chess game implemented using object-oriented programming principles. It features a full user interface (UI) and an AI player, making it a comprehensive and engaging experience for players of all skill levels.

The objective of the game is to defeat your opponent by capturing their king. This can be achieved through strategic maneuvering of your pieces, capturing enemy pieces, and ultimately putting the enemy king in checkmate, a situation where they are under immediate threat of capture and have no legal escape.

Game Flow: The game starts by setting up the pieces on the board in their starting positions. The players then take turns making moves. A player selects a piece and a destination square, and the game checks if the move is valid according to the rules of chess. If the move is valid, the piece is moved to the new square, and the turn ends. The game continues until one player checkmates the other, or a stalemate occurs where neither player can make a legal move.

Key Features:

- Object-oriented design: The project utilizes object-oriented principles to create a modular and maintainable code base.
- Complete UI: The game features a user-friendly UI that allows players to easily interact with the game.
- AI Player: The AI opponent provides a challenging and engaging experience for players of all skill levels.
- Turn-based gameplay: The game follows the traditional turn-based structure of chess, allowing players to think strategically about their next move.
- Sound effects: The game incorporates sound effects to add to the overall gameplay experience.

Object-Oriented Design Principles:

- Encapsulation: Data and functionality within each object are hidden and accessed only through public methods, promoting data integrity and modularity.
- Inheritance: Piece class serves as a base class for specific piece types like pawn, rook, knight, etc., inheriting common properties and behaviors while overriding specific move logic.
- Polymorphism: Each piece type responds differently to the same method, demonstrating polymorphism in action.
- Abstraction: The Board class manages the underlying structure and logic of the game, abstracting away the details from other classes.

Graphical User Interface (GUI): The project utilizes C#'s GUI to create a visually appealing and interactive chessboard. It allows users to:

- View the current state of the game.
- Select and move pieces by clicking on squares.
- Receive visual feedback for valid and invalid moves.
- View captured pieces.
- Acknowledge check/checkmate scenarios.

AI Player Implementation: The AI player utilizes an AI model called Stockfish, to analyze the board state and choose optimal moves. This algorithm considers various factors like piece value, position, and potential check/checkmate scenarios. The AI difficulty level can be adjusted by varying the search depth or implementing different evaluation functions. However, the implementation of the AI player is not included in this report because it is outside the scope of this course.

This chess game project demonstrates a successful implementation of object-oriented programming principles in a game development context. It provides a fun and engaging experience for players of all levels and offers a strong foundation for further development and enhancements.

Class Diagram (UML)

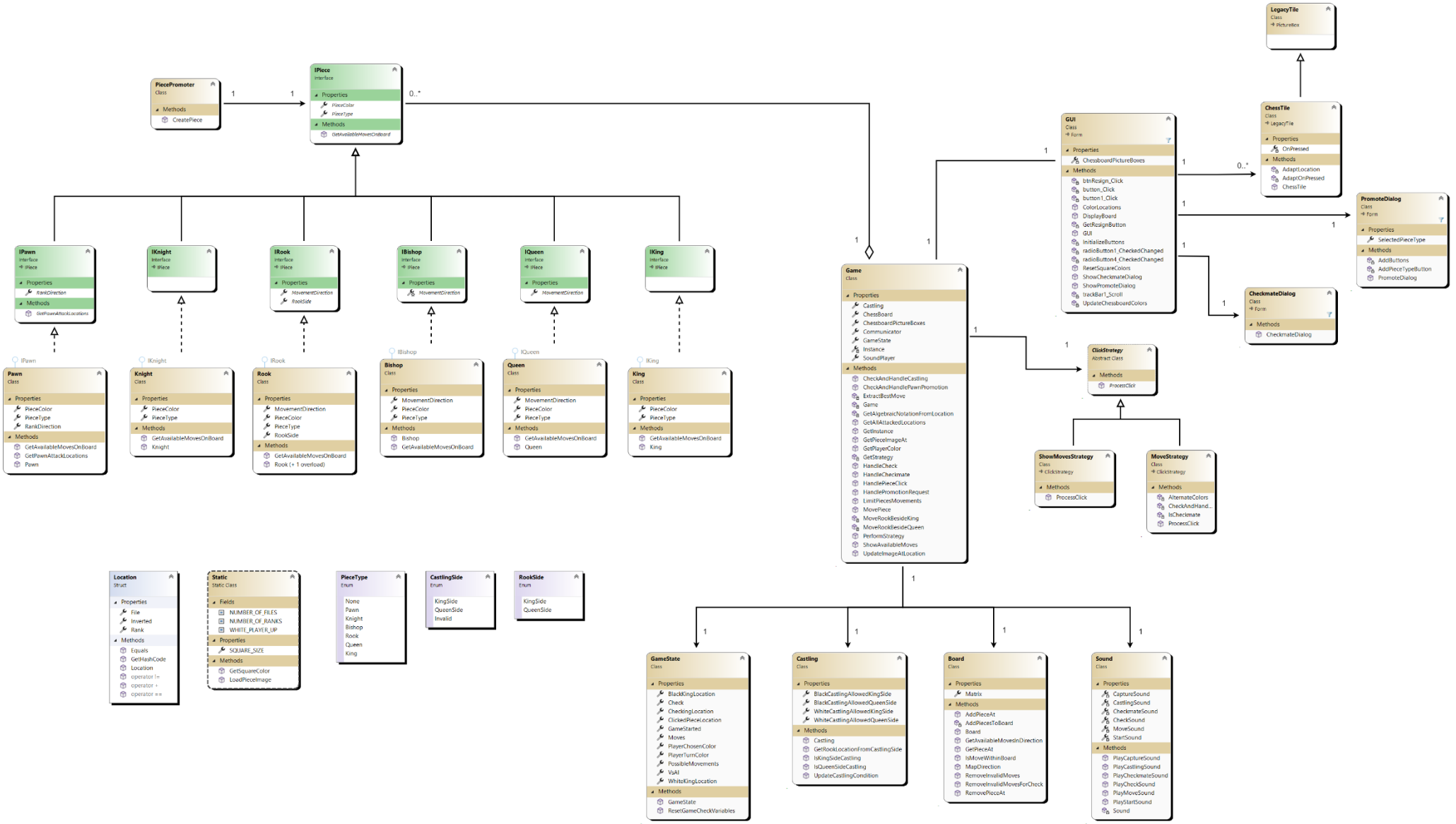
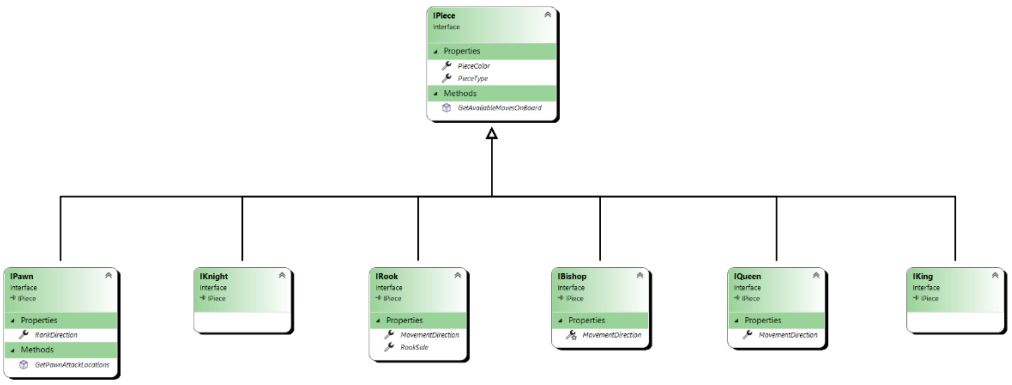
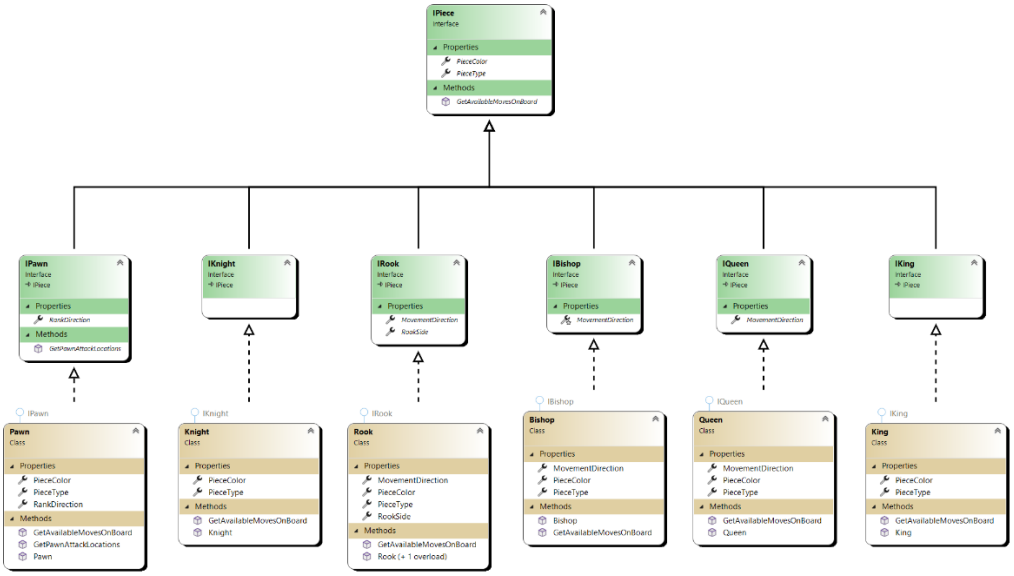


Figure 1. Class Diagram

Descriptions of how and where you have used key concepts

Concept	Description
Abstraction (abstract classes and/or interfaces)	 Here, we have a base interface called iPiece, and 6 other interfaces inheriting from it. In addition, there is the ClickStrategy abstract class which abstracts the different kinds of strategies.
Concrete classes	 Each of the piece's interfaces, as well as the base interface IPiece, is implemented in a concrete class which is the actual chess piece. In addition, the ClickStrategy abstract class is inherited in two concrete classes: MoveStrategy and ShowMovesStrategy.

Namespaces	The project contains a total of 5 namespaces:
	`namespace ChessGame`
	`namespace ChessGame.GameNamespace`
	`namespace ChessGame.Global`
	`namespace ChessGame.Strategy`
	`namespace ChessGame.Subsystems`

Methods	The project contains a large amount of methods, for example, the interface IPiece contains the method GetAvailableMovesOnBoard(), which is implemented in all 6 concrete classes of the pieces. Another example is the ProcessClick() method which is implemented in the strategy classes.

Properties	The project contains a large amount of properties, for example, the interface IPiece contains the properties PieceColor and PieceType, which are implemented in all 6 concrete classes of the pieces. Another example is the GameState class which contains 7 properties related to the state of the game including the kings' locations, player turn, etc.
Overriding of methods and/or properties	An example of overriding methods is the ProcessClick() method of the ClickStrategy class, which is overridden in the inherting classes, MoveStrategy and ShowMovesStrategy.

Constructor overloading	`public Rook(Color color)`
	`{`
	`  PieceType = PieceType.Rook;`
	`  PieceColor = color;`
	`  RookSide = RookSide.KingSide;`
	`  MovementDirection = new() { new Location(0, -1), new Location(0, 1), new Location(-1, 0), new Location(1, 0) };`
	`}`
	`public Rook(Color color, RookSide side)`
	`{`
	`  PieceType = PieceType.Rook;`
	`  PieceColor = color;`
	`  MovementDirection = new() { new Location(0, -1), new Location(0, 1), new Location(-1, 0), new Location(1, 0) };`
	`  RookSide = side;`
	`}`
	In this example, the Rook piece has 2 constructor overloads, one which takes the RookSide and another which does not.

Enumeration (Enum)	`public enum RookSide`
	`{`
	`  KingSide,`
	`  QueenSide`
	`}`
	`public enum CastlingSide`
	`{`
	`  KingSide,`
	`  QueenSide,`
	`  Invalid`
	`}`
	These are two enumeration examples implemented in this project.

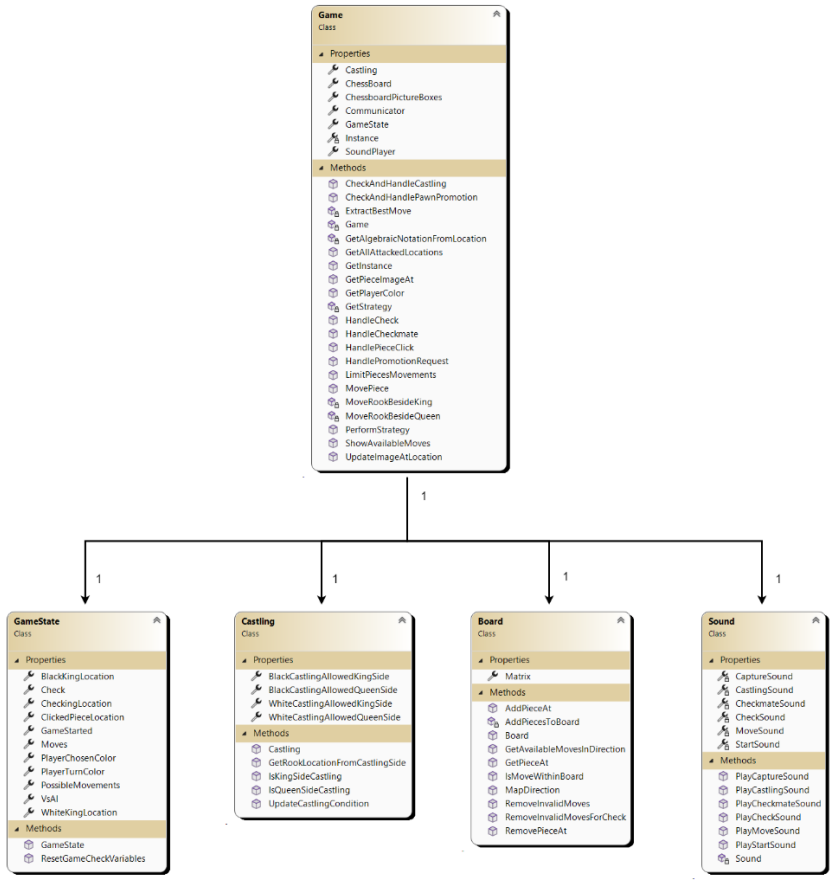
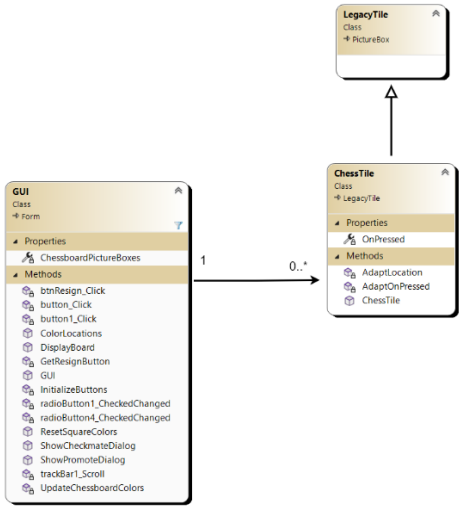
Collections (e.g., Lists)	`public List<Location> PossibleMovements { get; set; }` The GameState class contains a property called PossibleMovements that is a list of locations. There are other examples of collections as return types for methods.

Loops (e.g., foreach)	`public List<Location> GetAvailableMovesOnBoard(Location currentLocation)`
	`{`
	`  List<Location> pieceMovements = new();`
	`  foreach (Location movement in MovementDirection)`
	`  pieceMovements.AddRange(Game.GetInstance().ChessBoard.GetAvailableMovesInDirection(currentLocation, movement.Rank, movement.File, Game.GetInstance().GetPlayerColor()));`
	`  return pieceMovements;`
	`}`
	Each of the pieces in the project implements the GetAvailableMovesOnBoard() method of the IPiece interface. Most of these implemented methods contain foreach loops. The example provided is for the Bishop class.

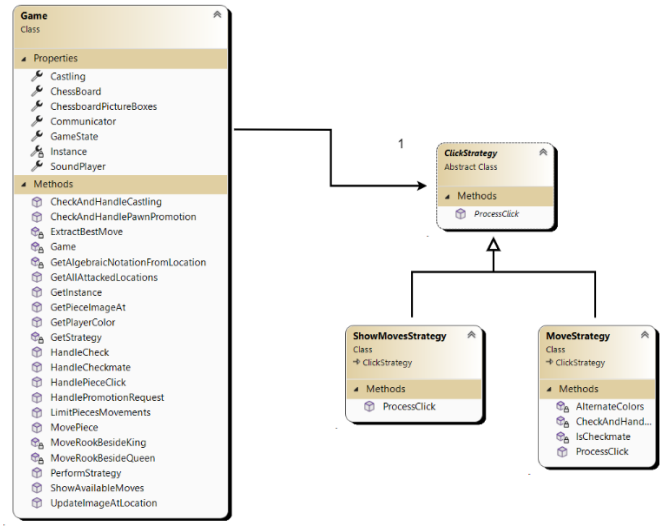
SOLID Principle	Description of where the principle is realized and for what purpose

1) Single Responsibility	`public void UpdateImageAtLocation(Location location)`
	`{`
	`  ChessboardPictureBoxes[location.Rank, location.File].Image = Static.LoadPieceImage(ChessBoard.Matrix[location.Rank, location.File]);`
	`}`
	`private void MoveRookBesideKing(Location kingLocation)`
	`{`
	`  MovePiece(Castling.GetRookLocationFromCastlingSide(GameState.PlayerTurnColor, CastlingSide.KingSide), new Location(kingLocation.Rank, kingLocation.File + 1));`
	`}`
	`public Image? GetPieceImageAt(int rank, int file)`
	`{`
	`  IPiece? piece = ChessBoard.GetPieceAt(new Location(rank, file));`
	`  return piece != null ? Static.LoadPieceImage(piece) : null;`
	`}`
	These 3 methods can be found in the Game class, and each has only one responsibility.

2) Open-Closed	Instead of using one class for all the chess pieces and specify extra properties and methods each time we need to add a piece to the game, we extended the IPiece class to new interfaces and concrete classes. This way, we ensured that the IPiece interface is open to extension and closed to modification.
3) Liskov Substitution	Because of the layer of abstraction for the chess pieces, any concrete class (Rook, Bishop, Pawn, etc.) can be substituted in place of IPiece. For example, calling GetAvailableMovesOnBoard() on any chess piece works without causing any problems.
4) Interface Segregation	Since each piece has its own special methods, an interface has been created for every piece (e.g. IPiece←IKnight←Knight) so that no piece has to have any methods or properties that it does not use, which is the main idea behind interface segregation.
5) Dependency Inversion	Instead of using a list of different strategies (MoveStrategy and ShowMovesStrategy) in the Game class, a layer of abstraction has been added (the abstract class ClickStrategy) so that the Game class does not know which concrete class it is collaborating with, which makes it easier to add other strategies.

GOF Design Pattern	Description of where the pattern is realized and for what purpose
1) Façade pattern	 The Facade pattern is a structural design pattern that simplifies the interface to a complex system by providing a single, unified access point. It acts like a "front door" that hides the underlying complexities of the system and makes it easier for users to interact with it. In this case, the façade class is the Game class, and the other classes are the subsystems. These subsystems cannot be accessed without going through the façade.
2) Adapter pattern	 The Adapter Pattern is a structural design pattern that allows incompatible interfaces to work together. It acts as a bridge between two interfaces, allowing them to communicate even though they have different methods or parameters. In this case, the Game class which can be considered a client uses the adapter class ChessTile to access the adaptee class LegacyTile (which is

	essentially a PictureBox), where the adapter can adapt the input of the client to the adaptee. One simple example of the importance of the adapter is the following:
	`private static Point AdaptLocation(Location location)`
	`{`
	`    return new Point(location.File * Static.SQUARE_SIZE, location.Rank * Static.SQUARE_SIZE);`
	`}`
	Here, the adapter is adapting the provided location into a Point object which the LegacyTile is compatible with.

3) Strategy	 The Strategy Pattern is a behavioral design pattern that allows to encapsulate different algorithms and switch between them at runtime. It promotes modularity, flexibility, and reusability in one's code. Here, the Game object can switch between different strategies during runtime and do different actions according to the chosen strategy, all whilst not being aware of the actual code that is being run.

4) Singleton	`private static Game? Instance { get; set; }`
	`public static Game GetInstance()`
	`{`
	`    Instance ??= new Game();`
	`    return Instance;`
	`}`
	`private Game()`
	`{`
	`    ChessBoard = new Board();`
	`    GameState = new GameState();`
	`    Castling = new Castling();`
	`    SoundPlayer = new Sound();`
	`    ChessboardPictureBoxes = new PictureBox[Static.NUMBER_OF_RANKS, Static.NUMBER_OF_FILES];`
	`    Communicator = new StockfishCommunicator();`
	`}`
	The Game class is set to be a singleton because there is no situation in our project where we need more than one instance at a time.

User Interface (Windows Forms)

Figure 2. User Interface

Description of UI

This project features a visually appealing and interactive chessboard built using C#'s GUI capabilities. Players can enjoy a rich experience through:

- Real-time visualization: Witness the game's current state unfolding before your eyes.
- Intuitive piece movement: Simply click on squares to select and move your pieces.
- Clear feedback: Get immediate visual cues for valid and invalid moves.
- Captured pieces in focus: Keep track of captured pieces for a comprehensive overview.
- Engaging check/checkmate alerts: Receive clear notifications about check and checkmate scenarios.
- Interactive pawn promotion: When a pawn reaches the other side, a dialog appears allowing you to choose the piece it promotes to, adding a layer of strategic decision-making.

This user-friendly interface enhances the chess experience, making it both visually engaging with different game colors and strategically stimulating.