Object Oriented Chess Application

22-04-2025

—

Syed Wahib Wali
N1324284
Bsc. CSC. AI, First Year
Student Group Code: CSAI_2

Overview

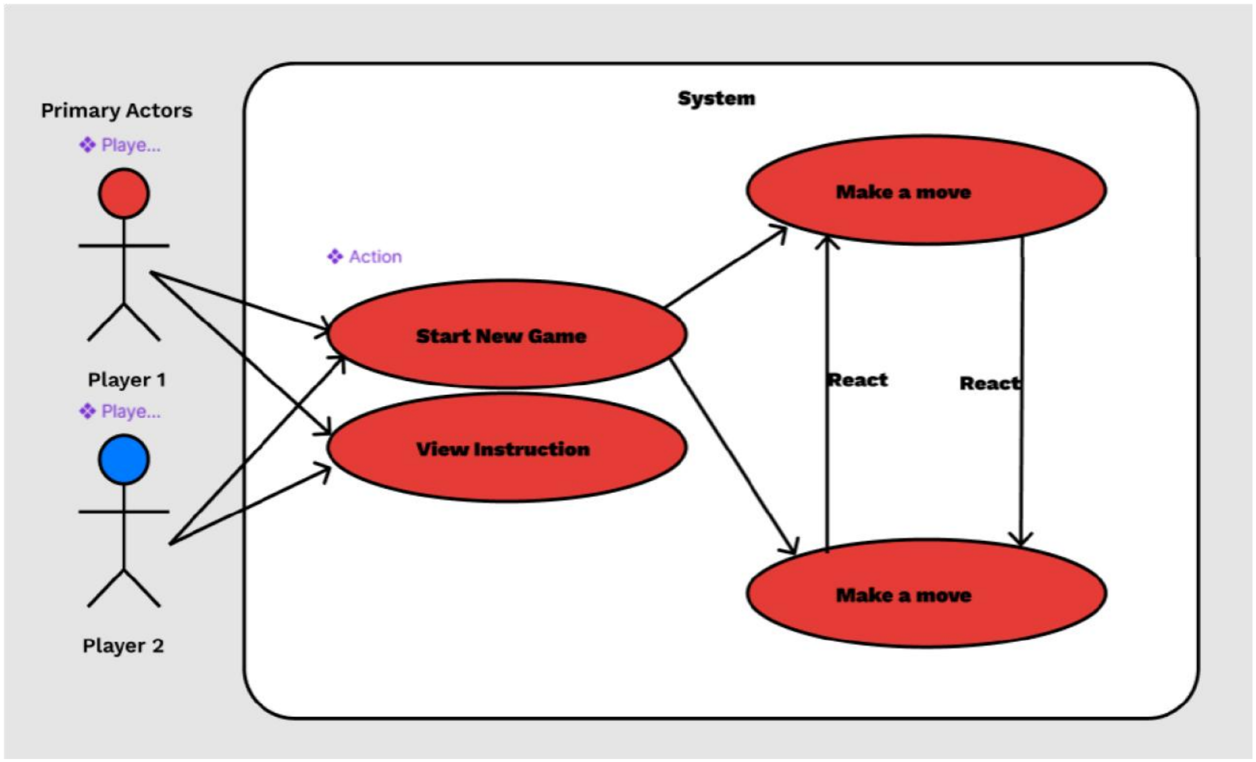
This is a simple CLI program which supports the functionality for 2 people to play chess against each other. The program runs in the Windows Console and does not use any additional libraries.

Features

1. When the program is first loaded the main menu is displayed which prompts the user to start the game or view instructions, or to Exit.
2. The user can view instructions on how to play the game.
3. During the game, all standard chess moves can be used, except for EnPassant.
4. The program handles invalid user input efficiently.
5. Error handling is implemented for ensuring proper chess format
6. Check, Checkmate and Stalemate will be displayed to the user when appropriate

Design

Use Case Diagram



The players can:

1) Start a New Game
2) View Instructions

Once the game is started, the players interact with each other by reaction to the other's move until a result is reached, either checkmate or stalemate.

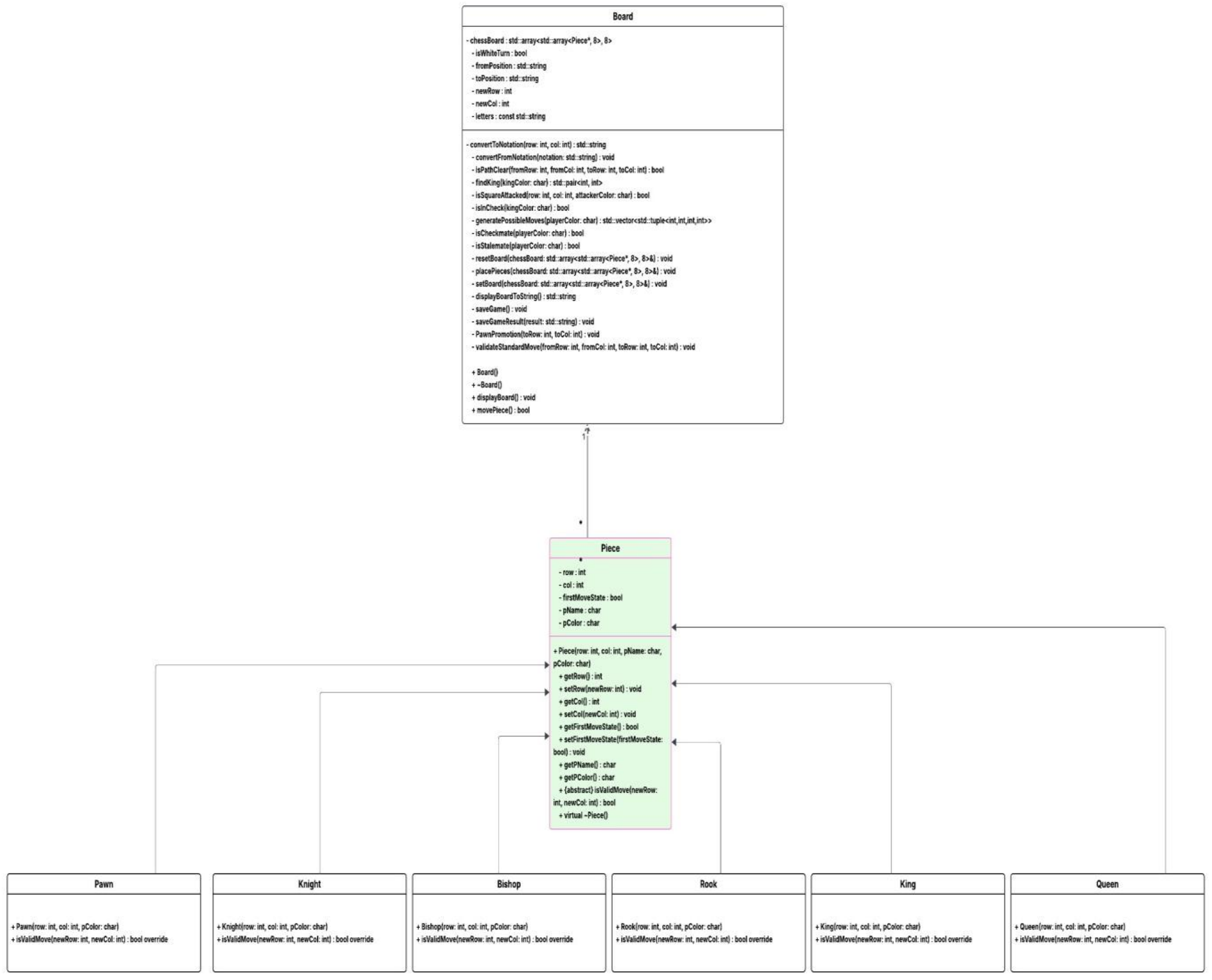
Class Diagrams and Structure

An object oriented approach has been taken for this project.

2 major parent classes have been used, the Piece class as an abstract for pieces and a Board class for the actions related to the game, such as taking and parsing input and moving the pieces, as well as error handling and ensuring valid rules are followed.

6 further classes have been defined as subclasses for Piece, each containing their own method for ValidMoves which will be called depending on the Piece subclass.

Proper encapsulation has been implemented wherever appropriate using getter and setter methods in all classes.


Board
- chessBoard : std::array<std::array<Piece*, 8>, 8>
- isWhiteTurn : bool
- fromPosition : std::string
- toPosition : std::string
- newRow : int
- newCol : int
- letters : const std::string
- convertToNotation(row: int, col: int) : std::string
- convertFromNotation(notation: std::string) : void
- isPathClear(fromRow: int, fromCol: int, toRow: int, toCol: int) : bool
- findKing(kingColor: char) : std::pair<int, int>
- isSquareAttacked(row: int, col: int, attackerColor: char) : bool
- isInCheck(kingColor: char) : bool
- generatePossibleMoves(playerColor: char) : std::vector<std::tuple<int,int,int,int>>
- isCheckmate(playerColor: char) : bool
- isStalemate(playerColor: char) : bool
- resetBoard(chessBoard: std::array<std::array<Piece*, 8>, 8>&) : void
- placePieces(chessBoard: std::array<std::array<Piece*, 8>, 8>&) : void
- setBoard(chessBoard: std::array<std::array<Piece*, 8>, 8>&) : void
- displayBoardToString() : std::string
- saveGame() : void
- saveGameResult(result: std::string) : void
- PawnPromotion(toRow: int, toCol: int) : void
- validateStandardMove(fromRow: int, fromCol: int, toRow: int, toCol: int) : void
+ Board()
+ ~Board()
+ displayBoard() : void
+ movePiece() : bool
1
*
Piece
- row : int
- col : int
- firstMoveState : bool
- pName : char
- pColor : char
+ Piece(row: int, col: int, pName: char, pColor: char)
+ getRow() : int
+ setRow(newRow: int) : void
+ getCol() : int
+ setCol(newCol: int) : void
+ getFirstMoveState() : bool
+ setFirstMoveState(firstMoveState: bool) : void
+ getPName() : char
+ getPColor() : char
+ {abstract} isValidMove(newRow: int, newCol: int) : bool
+ virtual ~Piece()
Pawn
+ Pawn(row: int, col: int, pColor: char)
+ isValidMove(newRow: int, newCol: int) : bool override
Knight
+ Knight(row: int, col: int, pColor: char)
+ isValidMove(newRow: int, newCol: int) : bool override
Bishop
+ Bishop(row: int, col: int, pColor: char)
+ isValidMove(newRow: int, newCol: int) : bool override
Rook
+ Rook(row: int, col: int, pColor: char)
+ isValidMove(newRow: int, newCol: int) : bool override
King
+ King(row: int, col: int, pColor: char)
+ isValidMove(newRow: int, newCol: int) : bool override
Queen
+ Queen(row: int, col: int, pColor: char)
+ isValidMove(newRow: int, newCol: int) : bool override

Other Design Approaches

A variety of Data Structures have been used for this project,

Array<array<Piece*, 8>, 8> chessBoard

This is an array of 8 arrays containing pointers to the subclasses of Piece depending on their position.

Firstly, pointers have been opted for to enable polymorphism

A 2d array has been used since the chessBoard is a fixed data structure, which is a better approach when compared to vectors.

Pair

This was used in the findKing() method of the board class to return the king's current row and column. This choice is perfect as a way to return two values.

Vector<tuple<int,int,int,int>>

This was used in the GeneratePossibleMoves method of the board class to return the possible moves after a move was performed.

This is a perfect choice as tuples are used to group the four coordinates together, fromRow, fromCol, toRow, toCol and then appended into the vector. Vectors were used as the proper size of the array would not be known until the moves are actually generated.

File Structure

Headers

This contains the header files

Board

Contains board.h, the declarations for the methods and attributes of the board class

Pieces

Contains Piece.h and its subclasses, the declarations for their methods and attributes

SourceFiles

Board

Contains the implementation files for the methods of the Board class

Main

Contains the main function, main.cpp

Pieces

Contains the implementation files for the declarations of the pieces and their subclass

- To run the game, go to x64->Debug->chess.exe

Test Plan

All the Requirements identified in the project proposal have been tested and successful

Test Case	Testing Strategy	Test Expectation	Test Result
The program should display the board whenever the game is run	Load the program Select 1	The user can see the board after running the program and selecting to start a game	Successful, can be seen
The program should inform the user which color makes the move next	Load the program Select 1	The user can see the program tell them when they load the program which color's move it is	Successful, can be seen
The program should inform the users of the format in which the move can be made	Load the program Select 2	Can see the program show them the Alphabet+Number format with example	Successful, can be seen
The program should include error handling for whenever the user makes an invalid move or out of turn	Load the program Select 1 Type "Random" When the move is White-, select the black pawn	Program says "Invalid format" and suggests correct format for invalid move, and Reminds that it's white's turn when black pawn is selected	Succesful, both cases return the expected result
The program should recognise when a move results in check and inform the users	Load the program Select 1 Input following moves: White: [e2-e4, d1-g4, g4-g5, g5-g6] Black: [f7-f5, f5-f4, f4-f3,]	Program says "BlackKing" is in check	Succesful, Program says BlackKing is in check

The program should recognise when a user gets checkmate or if it's a stalemate and end the program	Load the Program Select 1 Test for Checkmate: Scholar's Mate (4 move checkmate) White: [e2-e4, f1-c4, d1-h5, h5-f7] Black: [a7-a5, a5-a4, a4-a3] Test for Stalemate: 10 move Stalemate White: [c2-c4, h2-h4, d1-a4, a4-a5, a5-c7, c7-d7, d7-b7, b7-b8, b8-c8, c8-e6] Black: [h7-h5, a7-a5, a8-a6, a6-h6, f7-f6, e8-f7, d8-d3, d3-h7, f7-g6]	Program says Stalemate, the game is a draw, or checkmate, [Color] wins and ends the game, returning to main menu. All pieces perform as expected.	Succesful, Program performs as expected.
The program should save the result and how the board looks once checkmate is achieved.	Load the program. Select 1. Test for Checkmate: Scholar's Mate (4 move checkmate) White: [e2-e4, f1-c4, d1-h5, h5-f7] Black: [a7-a5, a5-a4, a4-a3]	The program saves the game and result in SavedGame.txt	Succesfully saves both in SavedGame.txt