CS246 Chess Design

By Inqiyad Patwary, Mahzabin Rashid Fariha and Sadman Ahmed

Overview

When we run the chess program, we can start the game against either a computer or human or enter setup mode.

For a normal game, we build the chess board by calling default_board() and initialise the players using initialise_players. The pieces are then stored in a vector of vector of Squares which is called when printing out the board.

Our major class is the Board class which includes the implementation of the is_valid() function, which checks if the input by the user is valid and then makes the necessary implementations.

When we make a move, we check whether the move is valid by using is_valid(), empty_square() to check the initial Square is not empty, correct_player() to check whether the piece moved is the same colour as the player and correct_command() to check whether the move is inside the grid.

In setup mode, we call put_piece to insert pieces in the board, delete_piece to remove pieces from the board, and change_current_player() to change the current player.

We use is_check(), is_checkmate() and is_stalemate() to see if the King is in check, checkmate or if the game is in stalemate. The game ends if we reach a checkmate situation or stalemate.

We have a Player class which keeps track of if it's the white or black player's turn.

Design

Our new UML is different from the previous UML we designed. This had to be implemented in order for our chess program to run and to accommodate various new cases such as checkmate, stalemate, check, setup mode and to check valid moves.

We created a Player class. It keeps track of a player's colour, whether they are the human or computer and their level. The level for a human is 0, and the level for a computer is 1,2 or 3.

We have a player class to keep track of the current player and a vector of players to keep track of the two players.

We have a Piece class which stores the name and colour of pieces.

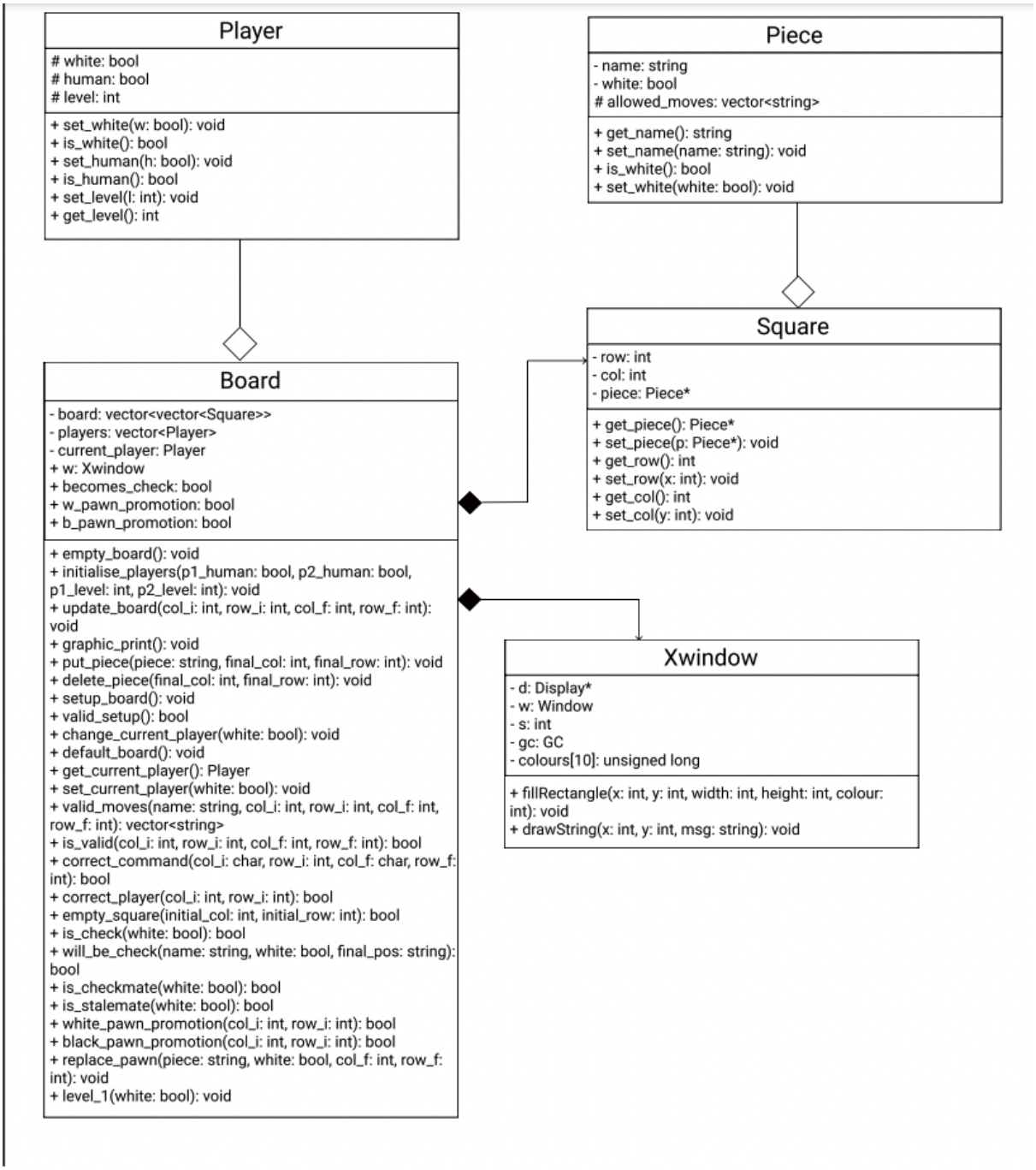
We created a Square class. It stores ints for row, column and a piece pointer to a piece. We use a vector of vectors of Squares to create and update the board.

A lot more boolean expressions were added in the board class to accommodate the usage of pawn promotion and to observe checks.

To improve our design patterns we initially decided to implement an observer pattern to display the text display and the graphical display. We wanted to use our piece class as an abstract base class for all the other pieces. We also wanted to use our player class as an abstract base class and have human and computer as two concrete classes.

However, due to a lack of time we were not able to achieve this. But we made the pieces using the piece class and added the graphics functionality inside the board class which works perfectly.

Updated UML



Computer Players

We were asked to implement 4 levels of AI to play against the human player. However, due to a lack of time, we could only finish AI level 1.

Level 1:

The computer makes random moves in level 1. We store the Squares of the pieces that belong to the computer in a vector and then use a random number generator within the range of the array's length to choose a random piece's Square.

We find all the available_moves of that particular piece and store it in a vector of strings. We check whether the piece we chose has no valid moves. If it has no valid moves, we generate another random number and choose another random piece's Square. We keep doing this until we get a piece which has at least 1 valid move.

Then, we generate a new random number within the range of the available_moves array's length and make the computer play that random move.

Resilience to Change

We tried our best to implement our code in a way that would require minimal work if changes need to be implemented.

If we need to add a new Piece or some new functionality for a piece, we just need to make changes in the valid_moves() function for the specific piece. This will make sure the new piece or it's functionality will work properly and that the user can use it.

Answers to Questions

Question 1.
Chess programs usually come with a book of standard opening move sequences,which list accepted opening moves and responses to opponents' moves, for the first dozen or so moves of the game. Although you are not required to support this, discuss how you would implement a book of standard openings if required.

We can use a map structure to store a series of opening moves that can be implemented. The opening moves will be immutable and the computer will iterate through it to decide the best move possible. We will also make a function that checks the respective response that the opponent can take from our map structure for the given move and then implement that. To check the

next move, we will use the values of the moves stored in the map structure in our conditionality statements.

Question 2.
How would you implement a feature that would allow a player to undo their last move? What about an unlimited number of undos?

We will use a vector to store move history. This will keep track of whichever piece is moved where on the board. Each index will contain the previous coordinates (row, column) of the piece and its name. If a player wants to undo their last move, the first piece from our vector will go to its previous coordinates again and the first move history entry is popped off our vector. This will work for an unlimited number of undos since we store move history from the beginning of our game.

Question 3.
Variations on chess abound. For example, four-handed chess is a variant that is played by four players (search for it!). Outline the changes that would be necessary to make your program into a four-handed chess game.

We will first need to change the display of our board to accommodate the addition of two additional players. We will need two new sides and colors and also allow them to make moves. We will allow players to choose whether they want to play as a team or as a free-for-all. The conditions which lead to check, checkmate, and the game-ending will change accordingly, depending on which format the players choose to play.

Extra Credit Features

For extra credit features we just implemented the graphics.

Final Questions

a) What lessons did this project teach you about developing software in teams? If you worked alone, what lessons did you learn about writing large programs?

We learned to communicate, collaborate and work within a team effectively. We learned that it is easier to tackle a problem when you talk about it with another person and brainstorm ideas to tackle it. We also learned to use git

and github effectively since the majority of us did not have extensive experience using the mentioned technologies. We also learned about patience and hearing out other's opinions since everyone had a different approach to tackling problems.

b) What would you have done differently if you had the chance to start over?

We would have managed time better and started working on the project earlier. We also would have made some implementation of move history and undo which would have made it easier to implement en passant, check, checkmate and castling. We also would have made the pieces reference to each individual class of the piece instead of referencing it to a new piece class and it became too late to overhaul the entire project.