Deep, Nguyen, Vansh

CS 246 Chess Project - Plan

We have chosen to create a Chess game for the CS246 Fall 2021 Project.

Project Breakdown:

Main function and commands
It is always beneficial to start with an interactable program, thus our first step will be to write the command interpreter (main file) which will provide access to all text-based commands supported by the project. As we then add support for each command, we will individually add the required code inside the main file to support the syntax to run that command. For example, the resign function would automatically give the other player a point and reset the board.

Board and Piece Implementation
Our next step will be to implement the Board and Piece objects. We first implement the board with the default setup, then we'll add the ability to custom setup after we have implemented the basics of the Board and Piece objects. In order to achieve this, we'll also implement a temporary overloaded friend operator<< until we can implement the TextDisplay observer. At first, we plan to begin by printing a blank 8x8 chess board where spaces represent white squares and underscore represents black squares.

Then we will start working on our piece classes. We will start by implementing the easiest piece, the queen, as it will make the implementation of other pieces easier, then bishop, knight, rook, then king and pawn. Kings and pawns would be hard because of castling, pawn captures, pawn's first moves, pawn promotion and en passant. So, we plan on writing the basics of King and pawn movement, leaving castling, en passant, and pawn promotion to the very end.

Move Implementation
After implementing the Board and Piece objects we will create the move command for manual moves (not computer). The move command will also consist of capturing other pieces. Our initial implementation of the move command will not check for illegal moves due to checks or invalid direction, for now.

Setup Implementation
After the basics of the Board and Piece objects are implemented and we can now move pieces, to help us with testing we will implement the setup command which will allow us to define a custom setup before a game starts. This command will be extremely useful for testing since we can move the pieces around the board and specifically test each scenario (Ex. en passant, castling, check, checkmate).

Check Valid Move Implementation
First, we will need to check if the piece can move in that direction, for example a rook moving diagonally is an invalid move. We also need to check for a piece of the same color in the square the piece wants to move to, as that will be an invalid move. Furthermore, we check if a player is in check, in checkmate and in stalemate. We will use this function in the move method to check if the move is illegal, and not perform the move if so.

Observers Implementation
Our third step will be to implement the Observer design patterns. First, we'll implement the TextDisplay observer first, which'll essentially take over the temporary operator<<. Then we'll implement the GraphicsDisplay with the XWindows display. In both displays, we will represent the black pieces with lower case letters and white pieces with capital letters.

Castling, En Passant, Pawn Promotion
After thorough testing of the basics of the chess game, we'll revisit castling, en passant, and pawn promotion or any hurdles we may have skipped.

Computer Player Implementation
Lastly, we will implement the computer algorithm for the computer vs computer and player vs computer games. We will start by creating an algorithm for the level 1 difficulty, making the subsequent levels as we progress.

Bonus
If time permits, we will try to implement the following bonus features:
1. We implement using smart pointers.
2. We implement the undo command.
3. We implement Computer's level 4 or higher.
4. We implement four-handed chess.

Estimated Completion Dates (ECD) & Project Division:

We plan on starting the coding part of the project on Monday November 29th.

We have designed our workflow such that each step has to be completed before moving on to the next task, we will collaboratively work on each task. For example, if a person completes their task before the ECD, then he will help the other people working the next task before moving too many steps in front. This will ensure we still follow our plan, and we are all on the same page before moving on to more challenging steps in our plan.

Main function and commands - ECD: Nov 29th - Deep
This will be primarily done by Deep.

Board and Piece Implementation - ECD: Dec 1st - Nguyen (Board), Vansh (Piece)
There are 2 steps in this task: Board Object and Piece Object. Nguyen will work on the Board object, while Vansh will work on the Piece object. After Deep finishes the Main function and commands, he will help Nguyen and Vansh.

Move Implementation - ECD: Dec 1st - Deep
This will be primarily done by Deep.

Setup Implementation - ECD: Dec 2nd - Nguyen and Vansh
This will be primarily done by Nguyen and Vansh, and after Vansh finishes his above tasks he will help out here before moving on. .

Check Valid Move Implementation - ECD: Dec 5th - all three
All 3 of us will work on this task at the same time. Each piece will have an invald move method, which will be called by the move function before moving a piece. So, Nguyen will work on King and Rook, Vansh will work on Pawn and Bishop, and Deep will work on Knight and Queen.

Nguyen will work on checks, Vansh will work on checkmates, Deep will work on stalemates.

Observers Implementation - ECD: Dec 6th - Nguyen (TextDisplay), Deep (Graphical Display)
There are 2 observers that need to be implemented in this step: TextDisplay and GraphicalDisplay. Nguyen will work on the TextDisplay and Deep on the GraphicalDisplay.

Castling, En Passant, Pawn Promotion - ECD: Dec 7th - Vansh
While Nguyen and Deep are working on the observers, Vansh will be working on some of the toughest parts of the chess game. If Nguyen and Deep are done, they can go help out Vansh.

Computer Player Implementation - ECD: Dec 12th - all 3 of us
All 3 of us will work on this task at the same time. This is one of the most challenging steps in the project, and we will work on each level together.

Testing and Bug Fixes - ECD: Dec 13th - all 3 of us
All 3 of us will work on this task at the same time. Thorough testing is crucial, and so we will independently test our program.

Bonus - ECD: Dec 15th - Nguyen (Smart Pointers), Deep and Vansh (Steps 2-4 in Bonus section above)
Nguyen will work on implementing smart pointers (step 1), while Vansh and Deep will work together on implementing as many steps from 2-4 defined above. Nguyen will join Vansh and Deep when he finishes.

Questions:

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 store a vector of all moves inside a board object, or maybe we can store a tree of all possible opening moves, because a player's recommended opening moves change depending on what the opponent does. In our tree, each node, representing a player's move, has two to three children nodes which represent the other player's possible responding moves. For example, if White plays e4, Black can play e5 or c5, in which White can respond in different ways depending on what Black plays.

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

To undo their last move only, we can use a field to hold the previous Move (a class in our UML) in the board object, and this field will be updated after a new move is performed. There will be an undo method in the board object, which will be called to undo the move which will simply get the previous move from the field and reset the board accordingly (captures are replaced, pieces are put back to previous location).

To undo an unlimited amount of moves, we can use a vector (stack) of all moves inside the board object. If the player wishes to undo, we pop the last move out of the stack and resets the board based on that. The undo method must be connected to the Board class.

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.

Firstly, we will have to change the size of the Piece 2D array in our Board Class to handle 4 players. So now our board will be 14x14 from 8x8. Also, the corners squares of size 3x3 are cut out in the 4-player chess board, so we will add a new Piece called "Invalid", and the Piece 2D array inside Board will have Invalid Pieces initialized at the 4 corner squares, so pieces cannot move there. We have to add new Piece colours as well. Next, we have to slightly change the implementation of the Pawn class, because the direction they travel does depend on the colour of the piece. The rest of the Piece subclasses should be fine because the directions they travel should be independent of board orientation. Then, we slightly change the implementation of the TextDisplay and GraphicalDisplay. We change the number of rows and columns iterated and we have to deal with the corners cut out.

The other methods will not be affected since they will consider all Pieces on the board to determine for example if it is an invalid move, is in check, or if there is a winner.