University of Waterloo
CS 246

CS246 Fall 2023 Project – Chess (Plan of Attack DD1)

Prepared By:
Jerry Chen
Dev Desai
Anirudh Goel
Nov 24, 2023

PROJECT BREAKDOWN

In order to create the game of chess, we have structured the program into eleven main classes, outlined as follows. As a brief overview, a Board class has been created to contain a vector of Piece objects. Employing the observer design pattern, both the TextDisplay and GraphicsDisplay classes will be observers of the Piece objects in order to update the game state that is shown to the user. Additionally, distinct classes are implemented for the various levels of computer difficulty. The main function serves as the central controller for the entire program. Below, you'll find a detailed description of each class.

Board

The Board class will handle all the pieces through a 2D Pieces vector. It will also handle moving and setting up the pieces with set(), initialize(), and move() functions. The Board class will also have a function, gameState(), to determine the state of the game; checkmate, stalemate, or none of those.

Piece (Abstract)

Piece is an abstract base class that defines the functionality of a chess piece. It implements an isValid() function to check if the location the piece will move to is valid or not. For example, a rook cannot move diagonally but a bishop can. It will also have a location field in order to determine the current location of the piece, and a colour field in order to make sure that a piece does not capture another piece of the same colour. In order to determine if a piece is “empty”, the Piece class also implements an isEmpty() function. Observers will observe the Piece class. For example, when a piece moves, it will notify observers (TextDisplay and GraphicsDisplay), and observers can update accordingly based on the move.

The following classes are the concrete classes that inherit the Piece class:

- Pawn
- Knight
- Bishop
- Rook
- Queen

- King

<u>AbstractComputer</u>

The AbstractComputer class is an abstract base class that defines a nextMove() function that the computer calculates the next move in. This function will be overridden by the LevelOne, LevelTwo and LevelThree concrete subclasses that each use a different strategy to calculate the next move.

<u>LevelOne</u>

The LevelOne computer will calculate its next move by choosing any random legal move.

<u>LevelTwo</u>

The LevelTwo computer will calculate its next move by prioritizing capturing moves and checks over other moves.

<u>LevelThree</u>

The LevelThree computer will calculate its next move by prioritizing capturing moves and checks over other moves, as well as avoiding the capture of its own pieces.

<u>Observer (abstract)</u>

The observer class is an abstract base class that defines a notify method which is how an observer would react to a change in state of a Piece. This notify method will be overridden by the TextDisplay and GraphicsDisplay classes.

<u>TextDisplay</u>

Concrete implementation of the Observer class for displaying the chessboard in text format. It observes changes in the Pieces in the Board class and updates the text representation accordingly.

<u>GraphicsDisplay</u>

Concrete implementation of the Observer class for displaying the chessboard in graphical format. It observes changes in the Pieces in the Board class and updates the graphical representation accordingly.

Player

The Player class is a base class representing a participant in the chess game. It includes essential properties such as colour (white or black, and possibly more for other chess variants).

XWindow

The XWindow class is responsible for managing the graphical user interface using the X Window system. It handles the creation of graphical elements such as the chessboard and chess pieces.

We plan to handle turns, player input, and score, as well as the general flow of the chess game in the main function.

SCHEDULE

Task	Members	Estimated Completion Date
Setup Gitlab	Dev	Nov 20
Create UML	Jerry, Anirudh, Dev	Nov 21
Finish Plan of Attack	Jerry, Anirudh, Dev	Nov 22
Create all header files	Jerry, Dev	Nov 22
Implement Board class	Jerry	Nov 25
Implement all Piece classes	Anirudh	Nov 25
Implement Observer classes (TextDisplay, GraphicsDisplay)	Dev	Nov 26
Test basic board (setup, initialize, basic moves)	Jerry, Anirudh, Dev	Nov 26
Implement complex logic (castling, checks, en passant, pawn promotion)	Jerry, Anirudh, Dev	Nov 29
Complete main function (player vs player, command interpreter, scoring)	Jerry, Anirudh, Dev	Nov 29
Implement Computer (Level One, Level Two, Level Three)	Dev	Dec 1
Add Player vs. Computer in	Jerry, Anirudh	Dec 2

Complete Main Function		
Add bonus features (undecided)	Jerry, Anirudh, Dev	Dec 3
Prepare demo	Jerry, Anirudh, Dev	Dec 3

ANSWERS TO QUESTIONS

Question: 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. See for example https://www.chess.com/explorer which lists starting moves, possible responses, and historic win/draw/loss percentages. Although you are not required to support this, discuss how you would implement a book of standard openings if required.

Answer:
Chess openings are on average about 10 moves. To implement a book of standard opening move sequences, we propose using a dictionary structure. The keys in this dictionary would consist of strings representing board positions, while the corresponding values would be string vectors. These vectors should include information such as the name of the opening, historical win/draw/loss percentages, starting moves, and possible responses. Within the Board class, we would incorporate a string variable dedicated to tracking the current board position. This ensures that each unique board position is associated with a distinctive string representation. At every move, we would index the dictionary with the current board position string. If the indexing operation is successful, we can output pertinent information about the opening, such as its name and relevant statistics. In the event of an unsuccessful index, we can handle the situation appropriately. This approach allows for an efficient and organized way to manage and retrieve information about standard opening move sequences based on the current board position.

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

Answer:
To implement a feature enabling a player to undo their last move, we propose augmenting the Board class with a private string vector to preserve the move history. Each move, represented as a string like "E7E5," is appended to the end of this vector. To facilitate undoing the last move, the most recent move is removed from the vector, and the board is reinitialized by replaying all moves stored in the vector. For unlimited undos, this operation can be repeated.

Recognizing the inefficiency of this approach, we suggest an optimization by incorporating a Piece vector alongside the string vector. At each move, if the move involves capturing a piece, the captured piece is appended to the end of the Piece vector. For non-capturing moves, an "empty" piece is added. When undoing a move, the string is split to obtain the original and destination positions. The moved piece is returned to its original position, and the Piece vector is checked to determine if a piece was captured during the move. If so, the captured piece is reinstated at the destination position; otherwise, no action is taken.

For clarity, let's consider an example: a pawn on d3 captures a pawn on e4. The string for this move in the string vector is "e3d4," and the corresponding Piece in the Piece vector is the captured pawn from d4. To undo this move, we split the string into "e3" and "d4." The capturing pawn currently at d4 is moved back to e3, and the Piece vector indicates a non-empty piece (the captured pawn). Consequently, this pawn is restored to the board at position "d4" using the set() function of the Board.

Handling edge cases like Castling and En Passant requires additional logic, which can be implemented separately.

Question: 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.

Answer:
To transform our program into a four-handed chess game, several modifications are required. First, a new four-player Board class should be created, inheriting from the existing Board class. Adjustments to the logic for determining the validity of moves are essential, considering the unique rules of four-handed chess. Notably, movement into the 3x3 corners of the board is restricted, and crossing over these areas is prohibited. For instance, a bishop cannot traverse the 3x3 corner to reach the opposite side.

Expansion of the Colour Enum is necessary to accommodate two additional colours, representing the four players in the game. The main function must also be updated to include two more players. The logic for checking and checkmating requires modification since there are now three opposing players to consider. Checking for checks and checkmates from each of the three opponents becomes imperative. The winning conditions are altered as well, as the game must continue until three out of four players are checkmated.

A crucial addition involves introducing a property to the Piece class to determine whether a piece is active or not. After a player is checkmated, their pieces remain on the board but become non-functional. Therefore, a mechanism to track the activity status of each piece is essential. The pawn promotion process may also require slight modifications to align with the unique dynamics of the four-player variant.