Ibrahim Khan, Mohamed Helwa, Muhammed Umer December 12, 2021

CS 246: Final Project Chess Design Outline

Overview (describe the overall structure of your project)

Our projects take a methodical approach to implementing the game Chess.Starting with our I/O implementation we present the player with an interactive and informative UI. After receiving input from the user the program invokes calls to the game class which deals with initializing the board and is the main control center for all operations in the game. When it is time to make moves the game class calls the board class which is responsible for updating and managing the vectors storing the chessboard information. The chess pieces are individual decorator classes with their own attributes as outlined in the rules of chess for the types of pieces. They are used to determine legal moves based on the situation of the board. Finally, the computer player class is an amalgamation of patterns used to make appropriate chess moves based on the computer skill level.

Design

1. Chess piece logic
1.1 Pieces
In chess, there are six different pieces, pawn, rook, knight, bishop, queen, and king, each with their unique movement patterns. In chess however there are certain restrictions, rules, or even unique moves that certain pieces can do, we will discuss these here. First, we discuss how each piece moves and we will make a vector called possibleMoves encapsulating all the possible moves.
1.1.1 Rook
Rooks move horizontally and vertically; they attack any square that is in their path. We find their possibleMoves by getting a rook's coordinates, say (i,j). then we check (i+k, j), for (k=1; k < 8; k++) if the square is empty, we add that to possible moves, if the square has a piece, of the same color, we break, if the square has a piece of the opposite color, we add that square to possibleMoves and break finally if (i+k,j) is outside the boards range we break. We repeat the above steps for the tuples (i-k, j), (i, j+k), (i, j-k).
1.1.2 Bishop

Bishops move diagonally and they attack any square that is in their path. We find their possibleMoves by getting a bishop's coordinates, say (i,j). then we check (i+k, j+k), for (k=1; k < 8; k++) if the square is empty, we add that to possible moves, if the square has a piece, of the same color, we break, if the square has a piece of the opposite color, we add that square to possibleMoves and break finally if (i+k,j+k) is outside the boards range we break. We repeat the above steps for the tuples (i+k, j-k), (i-k, j+k), (i-k, j-k).

1.1.3 Queen

Queens are an amalgamation of a rook and a bishop. So, for a queen we undergo the same algorithm for a bishop and a rook.

1.1.4 Pawn

If a pawn never moved. It can move two squares, else it moves one square. Before moving however, we must check that a piece, friendly or otherwise, doesn't block its path. Furthermore if a pawn's coordinates are (i,j), if a piece exists at (i+1, j+1), (i+1, j-1) then a pawn can attack the piece and move to said piece ((i-1, j+1), (i-1, j-1) if the piece is black)

1.1.5 King

The King can move to any square next to it. The king is also a special piece as any legal moves (We'll discuss that at the bottom) is tied to whether a king is being attacked by an enemy piece. So if an enemy piece is found to attack a king, it will call a function called checkKing().

1.2 Checking for a check.

To check if a piece is checking a king, we go through all the possible moves it can undergo and if the coordinate for the final position for said move lies on the king, then that means that this piece is attacking, and thus, checking the king will trigger checkKing() for the enemy king.

1.3 Special Moves.

All the executions done for special pieces will be handled by the board or game class.

1.3.1 En Passant

En passant can only happen if a pawn moves twice, so if a pawn moves twice to (i,j), the game will check if a piece is to it's right or left. If an enemy pawn is to its right or left, then add (i, j-1) to the possible moves for the enemy pawn. Furthermore if a pawn moves vertically but captures nothing, we can assume it's performing an peasant and capture the piece behind the pawns final position

1.3.2 Castling

If a king and rook have never moved, the king is not in check, no piece is between the king and the rook and no enemy piece is attacking the king, then the king moves two squares towards the specified rook and that rook moves to the square (i,j-1) to the king if its king side castle and to the square (i,j+1) if it's a queen side castle.

1.3.3 Promotions

If a pawn reaches the other side of the board, then it can be promoted to a rook, knight bishop or queen. So, a function will be made that replaces the pawn object with one of the 4 specified pieces (rook, knight, bishop or queen).

1.4 Legal Moves.

All the moves in the vector possible moves are playable, but they're not always legal, if a move leaves your king in check, then that is an illegal move (If you can make no legal moves, then we check if the king is in check, if so then that is checkmate, else that's a stalemate.) Let's generate a vector of moves, called legalMoves. To check if a move is legal, we generate a deep copy of the board, let's call it tmpBoard, we will play tmp board and generate all the possible moves for the enemy. If any move toggles the checkKing() for our king we know this is an invalid move, else, it's valid and we add that move to legalMoves. If legalMoves is an empty vector, we check if our king is in check, if so we lose, else we tie. If there are legalMoves then the client can only pick from that list.

1.5 Polymorphism

Since each chess piece has similar behavior, we will make a class called ChessPiece from which each piece can be derived and have properties specific to them (the way they move, tracking if a piece moved etc…)

2 Computer Player

The document specifies 3 computer levels and an optional 4+. The computer class will have 4 functions that specify each level and return the move based on the algorithm. (Level 4+ in Extra Credit Feature Section)

2.1 Level 1

For level 1, we find at most 100 moves and pick a random move from our vector each time.

2.2 Level 2

For level 2, it loops through every piece's legal move, if any piece captures an enemy piece, it returns said move, else, it stimulates a move, and checks if playing said move results in the enemy king getting checked, if so it returns said move. Else it returns a level 1 move.

2.3 Level 3

For level 3, it finds all enemy legal moves, if any enemy moves results in the capturing of a piece, it finds all legal moves for that piece and stimulates it, if that legal move is played and results in that piece avoiding the risk of capture, it’s returned. Else if all moves result in the piece being captured, we check if we can preserve any of the pieces by undergoing the same methodology. If none of these conditions are satisfied, a level 2 move is returned.

3. I/O

3.1 Input

The input deals with interpreting three types of commands, before the game, during the game, and after the game.

3.1.1

Initially, the program deals with input that helps initialize and set up a game. It takes in the initiating command from the user denoted by "game white-player black-player" where white-player/black-player can be any type of play, human or computer (1-4; where 1 is the least competent computer difficulty and 4 being the most competent). The command for playing as a human computer is "human", while the command for a computer player is interpreted as "compX" where X ∈ [1,4]. The I/O interpreter (main.cc) parses through the input read in from cin using getline, and determines whether the user wants to begin a new game expressed as above, or to quit the game and exit the program. Once the user begins a game they are prompted to enter whether they would like to use a default board and piece configuration or a special configuration. They can express which configuration they'd like by typing into cin "default" or "special" respectively. If the command read in is "default", main.cc calls initGame() from the game class which creates a default, traditional chess configuration and outputs it both on the text based output (cout) and the graphical output (Xwindows). If the command read in is special, the game then enters the setupGame section of main.cc in which the user has the option to Add pieces in specific positions, subtract pieces from specific positions, and change the initial turn of the game (whether black or white goes first). Once the user is done making their selections they type in "done" to cin it then verifies that the configuration is indeed valid and that it meets the requirements set in the project guide. Once it verifies the correctness, we output both the text based output (cout) and the graphical output (Xwindows).

3.1.2
During the game the user has the ability to either make a move using the command "move initial destination" where initial and destination are the coordinates for the initial and destination positions. These commands are entered as xy where x is the col and y is the row. An example command would be "move a1 a2" which moves the piece in col one row one, to col one row two. It ensures that the move is legal and valid which is determined in the makeMove(initial, destination) method of the game class. Moreover, the validity of a move is checked in multiple ways. One, the player making the move must only be able to move their own piece. Two, the initial position must have a piece present, you can't move something that doesn't exist. Three, the more sophisticated way of checking if a move is valid is by determining the possible moves and directions of each individual piece based on the chess rules. All in all, these procedures are in place to make sure the move being made is valid. The other command the user can make during the game is interpreted as "resign". If a user is to input "resign" the game will be awarded as a win to whoever's turn it was next (i.e. a loss to whatever user typed in resign).

3.1.3 The program interprets the current round of the game being over when the user enters the command "resign", after this, the program will prompt the user back to the initial menu where they can re-begin a new game.

3.2 Output

Similarly the output deals with interpreting three types of commands, before the game, during the game, and after the game. It also supports both a graphical and text based board that are updated and output after every move.

3.2.1

Initially the program outputs a welcome message for the user, and informs them of the command they need to type to begin a game. After this, it outputs a text based message prompting them to enter the type of game configuration they'd like. Based off their choice, it outputs both a text-based board (cout) and a graphical-based (Xwindow). With the graphical based board "Red" tiles replace the traditional black tiles, while each type of piece is labelled with their respective first letter (r,n,b,q,k, note that knights are represented with the letter n). For the text based output spaces (" ") represent white tiles, while underlines ("_") represent black/red tiles. Moreover, black pieces are represented with lower case letters, while white pieces are uppercased. After a piece is placed on a specific tile, it is drawn onto that tile. Once the setup or default initializing is complete, the program enters into the actual game

3.2.2

During the game the program is responsible for outputting the messages related to stalemates, invalid moves, and checkmates. It also outputs a message when either king is in check. After every move, the program is responsible for outputting the updated board on the text based output, while updating the appropriate tiles on the graphics based output.

3.2.3

Lastly, after each game is over, the program loops to the beginning where it outputs a welcome message. However, once the user presses ctrl d (EOF), the program then is responsible to output the game scores (i.e. how many points each player has when EOF is encountered).

4. Board Logic

4.1 Game

4.1.1 Initializing the game

4.1.1.1

As the program retrieves input for setting up the chess game, this class works in the background initializing all 32 pieces for the default game by calling upon the createPiece method and the respective decorator classes. Afterwards the pieces are assigned to their locations on the 2-D vector called board, and the vectors for the teams. Finally, the graphical interface is rendered as well as the text based interface by calling their helper functions.

4.1.1.2

The setup phase is different from the default setup, because the game class has to consider the various operations in the setup phase such as +,-,=, and ‘done’. The method setupGame adds pieces to the board by calling upon the helper function to create a piece and assigns that value to the board depending on the availability of the destination. If it is occupied the program has to remove the existing piece from the board and its team vector, place the new piece in that position and in the case of the king piece the program assigns the piece to the board class attribute Black/WhiteKing. When removing a piece the program deletes it from the team vector. Next the king is set to nullptr, and then the program searches through the friendly pieces for a King piece and updates the class attribute. Finally the piece is removed from the board followed by the update of graphics and the text based interface. The validateConfig function is called in the case of ‘done’ which checks the number of kings on the board by iterating through the friendly pieces and keeping count with an integer. It returns output if there are multiple black kings or none at all. The method also checks for pawns in the first and last rows of the board and lastly checks if either king is in check. If they are in check the program reacts accordingly by restricting the user from exiting the setup phase.

4.1.2 Making a move

4.1.2.1

The make move function determines if the provided coordinates are within the range of the board and calls upon the board functions to set all possible moves for friendly pieces and retrieve all legal moves. A move to an empty position is similar to adding a piece on an empty location in setup, the piece is created with respect to its parameters and assigned to the destination location as long as it is a legal move. Trying to move a piece that doesn’t exist returns a number informing the input class that the provided input is invalid. Next the method deals with promotions for computer players and replaces the pawn with the specified piece, in cases where a piece is not explicitly identified the program promotes the pawn to a queen. Following this step the makeMove function in board.cc is called to deal with allocating the pointers and updating all the appropriate vectors. The rest of the method deals with updating graphics for castling and en passant.

4.2 Board

4.2.1

Board hosts many useful variables that are available for other classes to have access to through its getters and setters. Variables like whiteKing, whitePieces, whiteKingCheck are all useful for determining the current state of the game. These getters and setters cast the chess piece pointers into the required types to access specific piece type methods. An example that is used quite frequently is casting ChessPiece * into King * to access the method getCheck().

4.2.2
The program deals with pointers for making moves within the bounds of the board by iterating through friendly pieces and getting the legal moves by invoking chess piece methods. For en passant the program checks if the current piece has any pawn to either side of it at which point it continues to determine if en-passant is possible by checking the conditions for the move. In the innermost if statement the program removes the enemy pawn from the pawn and places the friendly pawn to the destination position located one row behind the old enemy pawn. Since en-passant must be done immediately after the enemy pawn makes its first move the program has to set the variable used to determine if en-passant is a legal move. This step is invoked prior to the actually making the move and is reset shortly after it is made.

4.2.3
Castling is an important move to deal with since checking for the opportunity to castle requires a number of extensive checks. The board class calls the King class to deal with the job of determining if castling is possible and the board updates the pointers of the king and rook within the same move call. This way we ensure that the moves are being executed properly and without any issues.

4.2.4
Capturing a piece is similar to replacing a piece in setup which is done by removing the destination piece, updating the pointers and graphics, assigning the new piece to the destination position and finally removing the piece from the initial position on the board. Finally all the team vectors are updated to ensure that it is up to date with the board's current status.

5. Game status

In any chess match after every move we must check if the game ends in a stalemate, a checkmate or we play on.

5.1 Play on

If a legal move is found for the enemy pieces, this always means that the game hasn't ended.

5.2 Stalemate

We handled 3 cases of stalemate. If the enemy has no legal moves and is not in check, there is at most one knight each and no other pieces except the king and there is at most one bishop each and no other pieces except the king. If any of these conditions are satisfied the game ends in a stalemate.

5.3 Checkmate

If we find no legal moves for the enemy piece and their king is in check we trigger a checkmate.

Resilience to Change (Describe how your design supports the possibility of various changes to the program specification)

1.0 Chess Pieces

If any piece's movement logic changes for any reason, a module is created to deal with each case separately while not affecting the other pieces and the overall chess piece logic.

2.0 Board

The board is generalized to fulfill the role of calling the required methods and appropriate classes. In the instance of change, the changes on the board class will be very minimal due to its versatility and generality.

Extra Credit Features

1.0 Computer player Level 4+

For level 4+ we undergo a recursive min-max algorithm, we have a depth of d = level specified - 3. We stimulate each move, if the depth = 0, we add up the weight of all the friendly pieces (Queen = 11, Rook = 5, Bishop = 3, Knight = 3, Pawn = 1). We then return the move with the highest weight. If depth != 0, we stimulate all moves and run the min-max function with depth = depth -1 and turns = turns + 1. This will result in the min-max algorithm returning the piece with the highest yield for the computer player in terms of piece utility. If all moves yield the same weight, we return a level 3 move. The higher the depth the better the result, however, this function has exponential blowup. Therefore it's time complexity is O(2^n) and so it's highly recommended to not run this with large numbers.

Final Questions (the last questions in this document)

What lessons did this project teach you about developing software in teams?

Working on this project as a group was overall quite enjoyable, our group benefited from having complementary strengths, alleviated workloads, improved efficiency and productivity and we learned key project management skills. Starting from the group formation process, it was important to look for individuals who could fill in for places other members were not as confident which helped delegate tasks effectively. Our group's biggest asset was our experience using git which made development run much smoother and ensured that we had the most recent updates at all times. This wouldn't have been possible without a constant stream of communication which we established through Discord during our scheduled meetings. We set goals for each meeting so we could establish the next steps in development and ask questions about our respective tasks. Working as a team made the project more manageable since we no longer needed to create a whole chess game but rather smaller components which we could focus on developing, testing and improving. Finally, one thing we all walked away from this project with was a new set of project management skills from sprint planning to resolving issues that came up in production.

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

If we had to start over again, we would spend more time developing the design of our solution to improve in areas that would increase the efficiency of our program. Furthermore, we would focus on applying design patterns as much as possible to decrease coupling and increase cohesion. Specifically in areas where we could apply the observer pattern after making a move to call the allocated observers and render the updated displays. Although our group finished a completed solution to the Chess game, it would be better to spend a bit more time testing out components of our code as we developed them to reduce the testing load at the end of our timeline. This would ensure that corner cases have been thoroughly checked and reduce the likelihood of potential errors. Finally, we tried to clean up our code as much as possible within the time constraints and constraints of our program. However, creating meaningful helper functions and reusing code would make the code much more accessible and clearer to understand for programmers looking at the codebase for the first time. Furthermore one condition for stalemate that was found in research is that a repetition of move sequences leads to a stalemate. Thus if board positions were tracked and compared, this edge case could be dealt with.