Chess: Plan of Attack

May Ly & Raj Saragadam

University of Waterloo

CS246

December 6, 2022

Overview and Design of Project

Our project is very similar to our original plan and UML. Our general structure takes on the MVC design pattern. We have the model which is the logic of the program. This includes move, board, square, piece, player and all children branching off those listed. Then we have the view component which is the display class, which is responsible for printing out anything for the players to see. Lastly, we have the controller which is the game class, who facilitates all of the users' commands. Breaking the program into these three sections allowed us to keep our code organized, and reusable and made it easy to develop the project separately.

Game:
Our game class is fairly similar to our original idea with most of the same instance variables. We restructured how we wanted to interact with the class through our variables. The game class is responsible for setting up the game board, managing the display, setting up the players and their turns, keeping track of players' scores and enforcing the rules of the game. It manages any sort of input coming from either the command line or the move generator coming from the Computer Class. Initially, we thought that the game would only be in charge of initializing and running the user side of the same with managing setup, resign, and move functions. But after going through the structure again and seeing how the back-end of the game interacts with the user/computer inputs we decided to still keep the basic ideas of initializing, playing, and setting up the game but included more complex functions like moving a piece and adding/removing a piece from the board.

Board:
Our Board class is almost the same as what we had previously. It was generally created to implement the 8x8 chess grid consisting of vectors of the Square class that the game would be played on and keep track of the current state of the game (stalemate, check, checkmate). One problem we encountered later while trying to implement the state checkers was going through and keeping track of all the possible moves for each active Piece on the board. As such, we implemented functions isMovePossible() and getMovePossible() to keep track of possible moves on the board which we can then use and reference in order to check if the Board is in check, stalemate, and checkmate.

Square:
Our square piece is the same as we planned. A game always initializes 64 squares that will belong in the board class. In the square class specifically, it will hold a pointer to a Piece object which will be one of the children rook, pawn, king, queen etc. and it will hold a Position. We defined Position to be a struct with int x, and y values ranging from 0 to 7 to represent the position in the board nested vector.

Display:
We encountered very few changes in the Display class as opposed to what we had before. The Display class was implemented to securely print out the current active Board, the active Squares on the Board, any messages/errors to the user, and the help/documentation for the game in case the user gets stuck. One implementation we did add to the Display class when we encountered problems with printing the entire board at once instead we implemented printSquare() and to then print the Board we printed each of the squares that made up the active Board. By simplifying the function and breaking down the printing into easy individual steps we created high cohesion.

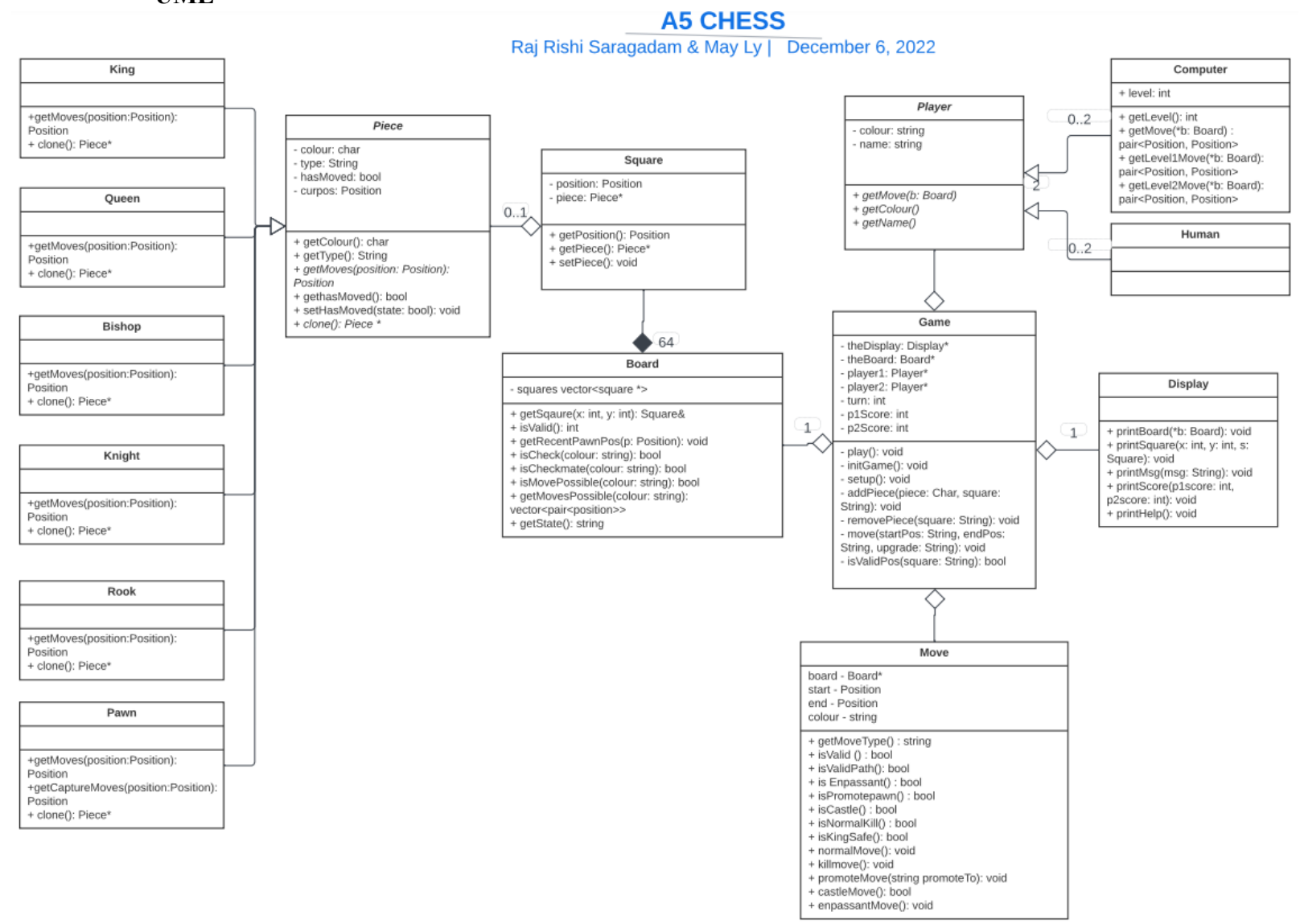
Piece:
Our Piece class is implemented separately from the Square and Board class, with no connection other than each Square pointing at a Piece there was a very loose coupling between the Piece, Square and Board class. We originally stuck to fragmenting the Piece class into different child classes made up of each unique type of piece that exists on a chess board (i.e. King, Queen, Bishop, Rook, Knight, and Pawn). Each subclass of Piece would have a type, colour, a bool check hasMoved(), and getter and setter functions for those instance variables. We did add a getCaputreMoves() function that created a list of capture moves that was unique to the Pawn subclass as pawns move differently when capturing versus moving around the board. Other than that our final Piece class did not deviate from our original plan.

Player:
Our player class was almost identical to the one we planned for originally in a sense that it held the same children - human and computer - and had the same attributes such as name, colour, and level (if it is a computer). Our player class is abstract so our program only creates human and computer players when the game is initialized. Since we have a general player object (Player currplayer) in the game class to hold the current player who is allowed to move, we realized that the human and computer steps in orienting a move were different, so we needed to differentiate if “currplayer” was of type human or computer. Our solution to this was to dynamically cast the general player into one of the two types allowing for less code in the children classes. In addition, although our player class was similar to the original plan, our computer class was much longer than we anticipated as we had to write functions defining how to get moves for each level of the computer - getLevel1Move(), getLevel2Move(), and so forth.

Move:

At the start, our move class was very small and only had implementation to make a move on the board and to check for valid moves. The idea was then to further break down the move class into the subclasses for each of the different types of moves; en passant, castling, promotion, kill/capture move, and then normal move. We then faced trouble in implementing all the different subclasses and directing user move inputs towards those classes and felt that we made the structure too complicated. Instead, we condensed the subclasses into individual functions which would instead all exist within the Move class. To break down the implementation further we created different error-checking and validation functions that made sure that the “move” was valid. Then we created a function which takes in the “move” and outputs which one of the five move types it is. As a result, our move type functions (i.e. killMove(), enpassantMove(), etc.) would only be in charge of moving the piece. This minimizes the coupling and maximizes the cohesion between the functions of the Move module.

UML


A5 CHESS
Raj Rishi Saragadam & May Ly | December 6, 2022
King
+getMoves(position:Position): Position
+ clone(): Piece*
Queen
+getMoves(position:Position): Position
+ clone(): Piece*
Bishop
+getMoves(position:Position): Position
+ clone(): Piece*
Knight
+getMoves(position:Position): Position
+ clone(): Piece*
Rook
+getMoves(position:Position): Position
+ clone(): Piece*
Pawn
+getMoves(position:Position): Position
+getCaptureMoves(position:Position): Position
+ clone(): Piece*
Piece
- colour: char
- type: String
- hasMoved: bool
- curpos: Position
+ getColour(): char
+ getType(): String
+ getMoves(position: Position): Position
+ gethasMoved(): bool
+ setHasMoved(state: bool): void
+ clone(): Piece *
0..1
Square
- position: Position
- piece: Piece*
+ getPosition(): Position
+ getPiece(): Piece*
+ setPiece(): void
64
Board
- squares vector<square *>
+ getSqaure(x: int, y: int): Square&
+ isValid(): int
+ getRecentPawnPos(p: Position): void
+ isCheck(colour: string): bool
+ isCheckmate(colour: string): bool
+ isMovePossible(colour: string): bool
+ getMovesPossible(colour: string): vector<pair<position>>
+ getState(): string
1
Player
- colour: string
- name: string
+ getMove(b: Board)
+ getColour()
+ getName()
0..2
2
0..2
Computer
+ level: int
+ getLevel(): int
+ getMove(*b: Board) : pair<Position, Position>
+ getLevel1Move(*b: Board): pair<Position, Position>
+ getLevel2Move(*b: Board): pair<Position, Position>
Human
Game
- theDisplay: Display*
- theBoard: Board*
- player1: Player*
- player2: Player*
- turn: int
- p1Score: int
- p2Score: int
- play(): void
- initGame(): void
- setup(): void
- addPiece(piece: Char, square: String): void
- removePiece(square: String): void
- move(startPos: String, endPos: String, upgrade: String): void
- isValidPos(square: String): bool
1
Display
+ printBoard(*b: Board): void
+ printSquare(x: int, y: int, s: Square): void
+ printMsg(msg: String): void
+ printScore(p1score: int, p2score: int): void
+ printHelp(): void
Move
board - Board*
start - Position
end - Position
colour - string
+ getMoveType() : string
+ isValid () : bool
+ isValidPath(): bool
+ is Enpassant() : bool
+ isPromotepawn() : bool
+ isCastle() : bool
+ isNormalKill() : bool
+ isKingSafe(): bool
+ normalMove(): void
+ killmove(): void
+ promoteMove(string promoteTo): void
+ castleMove(): bool
+ enpassantMove(): void

Resilience to Change

Being flexible is an important part of the design of a coding project. That is why when creating our implementation of the chess game we looked to create low coupling and high cohesion. A good example of this concept is our Piece class. Instead of creating multiple instances for each separate game piece (i.e. rook, queen, bishop, etc.) within the same class, we instead used children classes to represent each of the game pieces (i.e. rook, queen, bishop, etc.). Which then used override on virtual functions defined in the Piece class to create a unique implementation for each of the classes. This allows for flexibility on any of the Pieces such as adding a new piece, adding a new virtual/non-virtual function, or creating a specific Piece implementation that also allows for better error handling. Another example would be our Move class, within the program specification it is outlined that there are five different types of moves we can have in chess, en passant, castling, promotion, kill, and normal move. Instead of trying to jam all of the different move types into one move function and overcomplicate things we instead broke down the move function into the five types. Creating a function per each of the five different move types, taking it a step further we created a different function getMoveType() which takes in an input of a user move and returns the specific type of move the user is making. This allows our move classes to only focus on moving the piece and managing the board while the getMoveType() function will focus on managing the specific type of move and any invalid moves. Once again we gain the flexibility of being able to identify errors for specific moves, add or change the implementation of a type of move, and give us the freedom to add more "fun" moves in the future too.

Answers to 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.*

In order to implement a book of standard opening moves my partner and I would implement a map structure, **OpenmoveBook** in our Board class. The key is the name of the opening move while the value would contain another map structure.The "value" map structure's key would be the possible starting state of the board as a Text_Display and the value is the subsequent move that should take place based on the composition of the board. The computer would read from **OpenmoveBook** and determine the possible moves from there. As we have implemented the opening moves this way they are now immutable and are fixed structures for the moves. In the case where we need to counter-move, we would implement a boardCompare function that will compare two given board representations. After every "move" we will iterate through the map **OpenmoveBook** and compare the move board and the current state of the board and use the "value" map to pull a corresponding move. Hypothetically this would only work for the first

dozen moves or so as if the map was longer this will cause the computer to be noticeably slower.

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 allow a player to undo their last move or as many moves as they want we will implement a vector of vectors, **undoMove** in the Board class. The internal vector will be of Square class, as such, it can first store two values of struct Position, where a Position value stores a set of coordinates. This way we can record the starting and ending coordinates, the type of the moved piece, and the original piece that was originally in the final position. The code itself will locate at the final position, look at the last addition to **undoMove**, move the piece to its prior position and if needed replace the piece that was initially at the final position otherwise if no piece was taken and thus did not need to be placed at the final position then an empty piece will be placed there. As our vector can store multiple vectors of class Square we can include as many moves as we want and if undo is used we just need to use pop_back().

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.*

To implement the four-handed chess game we first need to create two more players, two more sets of pieces, and expand the grid. As we are expanding the grid we would need to implement checks to make sure that the nine squares at each corner are deemed invalid moves then we need to alter the Display/Xwindow classes to reflect this change. As part of the setup, we first need to give the players a choice to play in teams or alone. Next, we need to implement code that will end the game if both kings are in check for team play or in case of individual play if three kings are in check while one is not. In addition, we would have to modify the existing functions to check for check and checkmate to accommodate the added kings.

Final Questions

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?

This assignment taught us many things about programming in a team. From the initial planning to adding the finishing touches, we had to do many things differently than we normally would. The largest difference about working in a team in which we encountered was using GitHub. This was the first time we created a repository that had multiple developers branch off the main, so we learned the many ways to correctly push and pull code, protect the main repo and prevent any merge conflicts. In doing this, we learned that it was extremely important to assign tasks accordingly, be clear with your commit messages, and be communicative with all the pushes you facilitate. It is important to be clear about what is finished and what is not, and any dependencies your code may have if the other person would like to work on the project. In addition to using GitHub, this project taught us - the more developer brains working on an issue at hand, the better! At times when one of us was stuck on a bug, we would ‘pair program’ and try to find a solution together. This was extremely efficient and helpful, and is definitely something we look forward to in the future when working in teams.

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

If we had the chance to do this project over, we would have allocated our tasks much differently than we did in the plan of attack. We realized that when we started the program, the tasks assigned to each other had many dependencies and therefore, it was difficult for one of us to continue to move on to the next task or compile our code without other pieces of the project being completed first. Upon doing it again, we would have not worked class by class ie. trying to implement all functions within the class before moving on to the next. Instead, we would initialize all classes with their basic constructors, and work function by function, testing as we go. Starting with the init() function, play() and so forth. This way, we get all the basic preliminary functionalities first, and we can have a strong foundation to build off of. If we programmed this way, we could have the ability to complete other tasks such as the Xwindows or the higher level computer classes.