CS 246 A5 Chess Design Document

By: Nathan Kwon

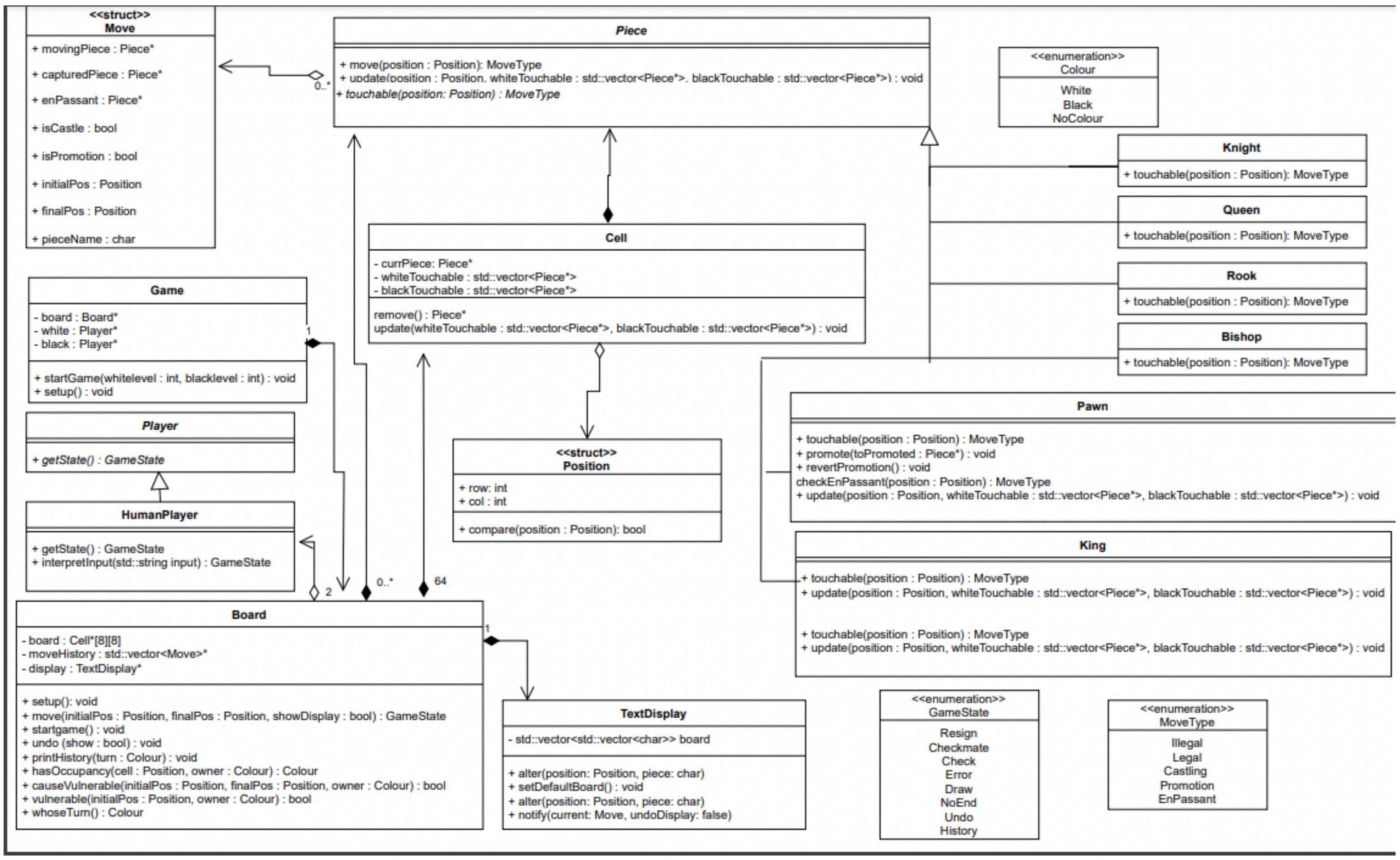
Overview:

As my UML shows, I have a Board class that oversees the state of the board and the way that pieces are set up, while my Game and HumanPlayer classes mostly manage the input and of the output of the program. Namely, when a new Game object is created, it calls the Board default constructor, which then constructs my chessboard as a 2D array of Cell pointers. Each Cell keeps track of its own position, which is represented by a Position struct, a pointer to the current piece that it may or may not contain, and two vectors that represent the white and black pieces that can reach this cell. I use the startgame() function to construct a board with pieces in their respective default positions. The Board class also has various functions to verify its current state like check, checkmate, and stalemate alongside a few getters and setters for determining whose turn it is.

The most important methods defined in my board class are move() and update(). Specifically move() relies on the various piece subclasses like pawn or rook (which will be further explained later). These subclasses determine whether the respective piece can carry out this move and returns its decision as a MoveType enumeration. This enum identifies whether the move is legal or not and distinguishes between the various moves that can take place on a chessboard. Move() handles the legal, illegal, castling, promotion, and en passant moves accordingly and will return a GameState enumeration to notify other classes that manage input/output. For instance, if GameState::Error results from the move() method due to a user's invalid input, the input/output classes will send a message for the user to try again. Furthermore, update() will iterate through the various cells and pieces on the board to determine the white and black pieces that can reach a respective cell. Each cell is updated with this information and passes it along using the piece pointer to the virtual Piece function update(), which is either inherited from the Piece superclass or is overridden by a specific subclass like Pawn. These Cell and Piece update() functions generally update the piece to their new position.

Finally, I have a TextDisplay class that represents the board as a vector of vectors of chars. This class has setDefaultBoard() to initialize the default chessboard and can alter() to change specific single cells during setup mode. When the board's state is changed, I also have a notify() method in the TextDisplay class that receives a Move structure as a parameter. This structure contains information about a specific move such as initial position, final position, and the piece that is moving, which is then used to notify which cells have changed and need to be changed accordingly.

Updated UML



<<struct>>
Move
+ movingPiece : Piece*
+ capturedPiece : Piece*
+ enPassant : Piece*
+ isCastle : bool
+ isPromotion : bool
+ initialPos : Position
+ finalPos : Position
+ pieceName : char
Piece
+ move(position : Position): MoveType
+ update(position : Position, whiteTouchable : std::vector<Piece*>, blackTouchable : std::vector<Piece*>) : void
+ touchable(position: Position) : MoveType
0..*
<<enumeration>>
Colour
White
Black
NoColour
Knight
+ touchable(position : Position): MoveType
Queen
+ touchable(position : Position): MoveType
Rook
+ touchable(position : Position): MoveType
Bishop
+ touchable(position : Position): MoveType
Cell
- currPiece: Piece*
- whiteTouchable : std::vector<Piece*>
- blackTouchable : std::vector<Piece*>
remove() : Piece*
update(whiteTouchable : std::vector<Piece*>, blackTouchable : std::vector<Piece*>) : void
Game
- board : Board*
- white : Player*
- black : Player*
+ startGame(whitelevel : int, blacklevel : int) : void
+ setup() : void
1
Player
+ getState() : GameState
HumanPlayer
+ getState() : GameState
+ interpretInput(std::string input) : GameState
<<struct>>
Position
+ row: int
+ col : int
+ compare(position : Position): bool
Pawn
+ touchable(position : Position) : MoveType
+ promote(toPromoted : Piece*) : void
+ revertPromotion() : void
checkEnPassant(position : Position) : MoveType
+ update(position : Position, whiteTouchable : std::vector<Piece*>, blackTouchable : std::vector<Piece*>) : void
King
+ touchable(position : Position) : MoveType
+ update(position : Position, whiteTouchable : std::vector<Piece*>, blackTouchable : std::vector<Piece*>) : void
+ touchable(position : Position) : MoveType
+ update(position : Position, whiteTouchable : std::vector<Piece*>, blackTouchable : std::vector<Piece*>) : void
2
0..*
64
1
Board
- board : Cell*[8][8]
- moveHistory : std::vector<Move>*
- display : TextDisplay*
+ setup(): void
+ move(initialPos : Position, finalPos : Position, showDisplay : bool) : GameState
+ startgame() : void
+ undo (show : bool) : void
+ printHistory(turn : Colour) : void
+ hasOccupancy(cell : Position, owner : Colour) : Colour
+ causeVulnerable(initialPos : Position, finalPos : Position, owner : Colour) : bool
+ vulnerable(initialPos : Position, owner : Colour) : bool
+ whoseTurn() : Colour
TextDisplay
- std::vector<std::vector<char>> board
+ alter(position: Position, piece: char)
+ setDefaultBoard() : void
+ alter(position: Position, piece: char)
+ notify(current: Move, undoDisplay: false)
<<enumeration>>
GameState
Resign
Checkmate
Check
Error
Draw
NoEnd
Undo
History
<<enumeration>>
MoveType
Illegal
Legal
Castling
Promotion
EnPassant

Design

While the general idea remains, my final design somewhat differs from the old UML. As briefly mentioned in the Overview section, I still employ general Object-Oriented Principles like Inheritance and Polymorphism. This can be seen in a crucial part of my design through the Piece superclass and its subclasses. The Piece class has six children, which each represent a unique piece in chess: King, Queen, Rook, Bishop, Knight, and Pawn. They have specific name and owner fields to distinguish one from another and, while I was unsuccessful in implementing my ComputerAi, specific getPieceName() and getVal() getters were intended to help prioritize the computer's decisions while getOwner() is inherited from the Piece superclass and helps with clarification since game logic differs depending on the piece's colour.

The pure virtual function touchable() also plays an integral role in this implementation. As mentioned in the board class, the move() function receives a MoveType enum to determine whether a move is valid. While specific implementation varies from subclass to subclass, the overridden touchable() receives its potential new position as its parameter, and it considers every legal move the piece can make and returns a MoveType corresponding to its decision. For instance, rooks will override this function to check if its new position will be in the same row or column and return MoveType::Valid if it is. This was an implementation of the template method pattern as my virtual function had no implementation and subclasses had their own touchable() functions to specialize the function for each piece.

In addition to this, I tried to adhere to as many programming techniques and principles as I could. Since this chess program involves significant amounts of information, I decided to encapsulate my data. Each module of the program tries to serve its own unique purpose with pointers to the associated classes. For instance, the Board class has a vector of vectors of pointers to the class Cell. Correspondingly, each Cell has a field currPiece that is a pointer to a Piece object and each Move struct has pointers to Piece objects. I also generally adhere to the Model-View-Controller structure as well: my model is the board class, which handles all game state and logic, my view is the graphic display, and my controllers are the Game and HumanPlayer classes. Admittedly, my Board::setup() method does violate the Single Responsibility Principle (SRP) since it communicates with the user, a responsibility that should be dealt with by another class, and something I would have redesigned if I had more time. Aside from this however, the Board class provides ways to identify information pertaining to gamestate as it has various check, checkmate, and stalemate methods. These methods are all private since the gamestate should only be changed in my board object since my other objects like Piece and Cell are working towards their own unique goals.

Resilience to Change

I tried to design my modules in a manner that accommodates change and so that a change in rules, input, or features would not require a complete redesign of my program.

If the command format were to change, then I would have to change my driver file and my Game and HumanPlayer classes. While it appears that I'm changing multiple modules of code, I believe that it's significantly easier to maintain and modify decently short pieces of code rather than one long one. Additionally, because of the way I have allocated specific input/output code to each file, I can easily locate where changes need to be made. My driver file handles all input/output before the game starts, HumanPlayer during the game, and Game as the game ends.

If board position input changes, depending on the exact change, I may be able to override my convertRowAndColumn() with an extra parameter so I can support both formats of input in my program. I also may need to modify one of the files that handles input/output accordingly. If the board needs to be displayed in a different manner, I only need to alter my TextDisplay and how it communicates with the user.

My design also promotes simple change when needing to replace a type of piece or changing the rules for movement of a piece. If adding a functionality to a specific piece, I can add or alter a function in the specific subclass. If adding to many, I can take advantage of the Inheritance my program implements; I can define a virtual method in the Piece superclass and override it wherever needed. Additionally, since each piece manages its own movement, I can simply change specific piece's overridden touchable() methods as the Board class employs Polymorphism when getting the state of a piece.

Furthermore, I have a Player superclass that the HumanPlayer subclass inherits from. I initially designed it this way because I believed it was a simple solution that allowed me to add ComputerAi subclasses for each level and if necessary, it can accommodate adding extra computer levels since I just need to create a new ComputerAi subclass and an extra command case. Finally, different variants of chess can be supported using our program with minimal change. The answer to question 3 in the next section briefly describes how I would go about implementing four-player chess on top of my existing program. If I were to implement variants like Fischer Random Chess, I believe it would be even simpler as I only have to add a field to check if the user would want to play this mode and can use rand() from the standard C <cstdlib> library to determine which pieces would be setup differently in Board::startgame(). This would allow me to make a completely new game with different rules without causing much disruption to the pre-existing program.

In terms of cohesion, by segregating classes according to their unique purpose, my program achieved high cohesiveness. Each of my classes work towards their own goal: the Piece class stores information about chess pieces through private fields like name and colour while the Cell class stores information about each cell on the chessboard through private fields like position and currPiece. These fields provide the program with the minimum necessary information possible and solely belong in their respective classes. My use of enumerations and structures further encourages high cohesion as my code is well organized and significantly easier to follow. One can specifically differentiate between my return statements and what they represent rather than having to decipher integer outputs via documentation in my header files.

My program also has decently low coupling as, with the exception of one or two functions, modules communicate through function calls that return simple results. As an example, my Board class solely returns enumerations and pointers. Furthermore, changes in one module do not affect other modules. For example, if I have to add an extra field to my Piece class, I can do so in my piece header file and I do not have to make changes (or if so extremely minimal) to any other header files.

Answers to Questions

Answers to Question 1 and 3 remain the same as in my DD1 response but Question 2 is altered as I managed to implement it in a different manner as previously described.

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.

I believe this could be implemented through using a Map structure, which consists of a key and value part. The "key" would refer to the opening sequence of moves such as the "Sicilian Defense". It could either store the name of the opening or the name of the opening move itself. The "value" part of the map would also have another Map structure with some representation of the board as the key and the value as the next move based on the current formation of the board state. We also may need to implement a boardCompare() function so that we can compare our current board with the opening move board. After each move from the opposing player, we can check our opening sequence Map structure. Comparing this to the current state of the board, we get the coordinates of our next move from the value part.

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?

Implementing the undo feature proved to be rather challenging as multiple types of chess moves had to be accounted for. For instance, castling involves the movement of two moves, promotion would change a piece entirely, and capturing moves would remove a piece from the board. This meant that I had to make changes to my Move structure in order to support this feature. I decided to create a capturedPiece field that would be a pointer to a Piece and would represent the "captured" piece in a given chess move. I specifically wanted this to be a pointer as this would allow me to easily distinguish between a normal move, where this would be a nullpointer, and a capturing one. Similarly, I added a Piece pointer enPassant to distinguish between a regular pawn capture or an enPassant one and had various fields like isCastle and isPromotion to specify castling and promotion as well. Since the Move structure also keeps track of the moving piece's initial and final positions, my program can figure out the rook's initial and final positions. This is because castling results in a set position for the rook so I can remove the king and rook from their castled positions and set them to their previous positions. A similar logic is used for both the en passant and promotion cases where I remove the pieces affected and put them back in their original positions. Using the aforementioned, I had a vector of Moves that I would build up using push_back() in my Board::move() function, would pop_back() the previous move, and set the piece to their previous positions on both the board and text display. I would also add an undo input case in the driver, and since I am simply iterating through the MoveHistory vector, the undo command could be performed multiple times until the start of the game is reached.

This answer differs from my DD1 response as I did not consider the fact that these fields would be better implemented as pointers, and I also did not require the pieceSwapped field since castling results in a fixed position for a rook. I can determine this fixed position by checking the colour of the king move and the direction, which is done by comparing the initial position field to the final one.

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.

Starting from our Board class, the first feature that we will need to implement is for 2-players to team up. We will also need to modify our pre-existing functions to accommodate a four-player chess board as opposed to the original 8-by-8 board. Specifically, one function that we will need to alter is the setDefaultBoard() function as not only the chess board would change but we would also need to add two additional sets of pieces for the two new players. Some features of the game like promotion would also have to be changed; pawns now promote on the middle of the board so our check for this would be slightly altered. Finally, we will have to determine the winner and if the game is over differently. When declaring GameState::Checkmate from my getState() method, I will now have to check that three Kings are checkmated before the final King/player is victorious or that two Kings of the same team are checkmated before the opposing team is victorious (This solely depends on whether the team or free-for-all was chosen at the beginning of the game). The way I display check and checkmate may have to be individualized and checked for each player, and the requirements for the way we display our game board through text and graphics (X11) would also have to be changed to represent this new board.

Extra Credit Features

- Undo: As described in the Question 2 Response
- MoveHistory: The way in which I implemented the undo feature allowed me to implement a history feature rather easily. I first added an interpreter to check if the "history" command was read from the input stream. Because I initially decided to represent MoveHistory as a pointer to a vector of my Move structure, I could create a printHistory() method in my Board class that would simply iterate over the moveHistory vector until moveHistory->size = 0 and format/print out each move since the beginning of the game.

Final Questions

Question 1:
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?

The most important lesson I learned about writing large programs was effective version control. Since I worked alone, I did not make a GitHub repository and decided to work solely on the CS servers, which was my biggest mistake. When I was implementing bonus features or made changes while trying to debug my code, a previous part of my code often failed. Because of this, I was forced to search extensively for any possible errors and had to figure out whether it was my original code failing some test case or it was from the changes I made. Another lesson I learned was that writing some form of pseudocode is extremely effective, especially when dealing with a topic like chess that can be very prone to logic errors. I wasted many hours debugging just to change an enumeration or swap some values around and believe that I could have made significantly better progress if I were more careful around these areas.

Question 2:

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

As mentioned in the previous response, I would definitely implement better version control. I also wish I implemented my functions in a different order. I started working on my input/output because I figured it would be the easiest to implement, but I wish I started on the board class, where most of the game logic is, as input/output is significantly more malleable than how the logic of the game is programmed.