CS246 - Fall 2023
December 4th, 2023
Harry Lam, Gen Ichihasi, William Xiao

Final Design - Chess

Introduction

As part of the CS246 final project in Fall 2023 semester, our group attempted to design and implement a full chess game engine, which consists of all the standard rules in chess, a text and graphics display, and different player modes (human and computer).

In this document, we will highlight some notable changes to our design of the chess game in comparison to the original UML we have declared at the beginning of the project. We will also give a high level detail in our chess implementation, discuss the design pattern that was chosen for this chess game and how the whole design can accommodate change in the long run. At the end, we will reanswer some of the questions in the project specification, and make a final reflection on what could be done better in future projects.

Overview

While there are many changes to the number of member functions included in each class, function signature and how objects in each class are interacted with each other in comparison to our original UML, in general, our class design still remains intact throughout the project. That is, our final design still based on a Model, Controller and View (MVC) model, where we include all the necessary pieces (king, rook, knight, queen, pawn, bishop, queen, empty) with differnt colour that inherit from chess pieces as models, an 8 x 8 chessboard that consist of different type of chess pieces, a chess game interface where user control the game through standard input, a text display, graphic display and a scoreboard for view, and a combination of human player and computer player class that inherit from abstract player class.

In terms of our original plan of attack, there are not many changes, and we are able to work on different components at the same time according to plan: Gen was responsible for the implementation of chess pieces and graphic display, Harry was responsible for all the chessboard and chess game logic and main function, also debugging for the computer players, and William was responsible for test cases for modules and the game itself, and also developing the algorithms for computer player. However, since the ChessBoard/ChessGame is deeply integrated with the text display, it is eventually decided that Harry would do the TextDisplay instead of Gen. The main function, which was developed early on to test for main functionality, was modified by other members to accommodate more features later on, such as the scoreboard. All members have also contributed to testing and debugging instead of one person doing everything. That being said, while we were able to work separately on the functionality most of the time, in some cases there are merge conflicts due to some members working, editing, or debugging on the same file at the same time, which slows down the progress a little bit.

Program Design: Before and After

Compared to our original UML, there are many notable differences. In particular, in ChessPiece, the array of observer pointers, which was originally included inside individual chess pieces to notify changes to display when necessary, was removed, and the responsibility of notifying observers when a particular piece was changed was instead hand over to ChessBoard, which make more sense since both text display and graphic display is associated with the board rather than individual chess pieces. There is also the addition of enum class ChessType, which is a list of all types of pieces in chess (including Empty, which means no piece). This will become important when checking for valid move and path, pawn promotion, castling and en passant.

The biggest change would be the design of ChessBoard. One such change is that the pointer used for both the kings, which originally stored as unique pointers of King, is now replaced with a ChessPiece raw pointer, which stores the address of the kings' whereabouts on the board (which is on the stack). As mentioned before, the array of observer pointers have now moved here, along with typical observers member functions such as attach and notifyObservers, so that the chess game can notify the board whenever a move has been made, and change state according to where the piece is originally located and where the piece has moved to. In terms of member functions, the original ChessBoard include setting the board, evaluate valid move, valid path (no obstacle) and checking whether a chess piece is under attack, which only act as a way to check standard move by the player, and does not take into account of the setup mode and special valid move according to chess rules. The new ChessBoard also have these functions, but now with the additional functions to aid in setup mode (such as addPiece, removePiece and isValidBoard), check for checks/checkmate/stalemate (such as kingIsUnderAttack, isUnderAttack and validMoveExist), and functions to evaluate or make special valid move (such as isCastlingPossible, isEnPassantPossible, and pawnPromotion). There are also the addition of functions that are useful for AI such as PossibleMoveGenerator, isChecking and isCapturing for evaluating moves that can be made. The chessMove function, which simply moves a piece to a new square on the board, now accepts two ChessSquare (ChessSquare simply stores the row and column of a piece, which is also a field of ChessPiece), instead of none, so that the board knows which piece to move and where to move to. The isValidMove function now also accepts another parameter, ChessColour colour, to know which turn is it and what piece is allowed to move. There are also small changes to parameters of other functions.

Next, the ChessGame now has four extra fields: a text display unique pointer and a graphics display unique pointer (attach to the chess board and created upon starting the game), a boolean variable that checks for stalemate (isStalemate), and a boolean that denote whether white win or black win upon a game is won (isWhiteWin), which we will use to output accordingly in TextDisplay. Other than that, the member functions remain mostly unchanged, although the function that accepts player input to move a piece in the chess game (makeAMove) now also accepts two string inputs that correspond to what piece and where to move the piece to, and we also add some other private helper function to help it easier to monitor control flow.

The TextDisplay for the chess board remains the same, although we include additional functions to print certain output according to game state (such as outputInvalidRow, outputInvalidColumn, outputInvalidMove, outputCheckmate, outputStalemate, outputTurn, etc.). Other than that, little changes have been made to the classes Player, HumanPlayer, ComputerPlayer, Scoreboard and abstract class Observer.

The GraphicsDisplay module enhances visibility in chess games by displaying pieces in their respective side colours with contrasting highlights; black pieces appear in black with white highlights, and white pieces are the inverse. This feature addresses the issue of pieces blending into cells of the same colour. Private fields displayColour and display store the colour and displayed character of each piece, akin to theDisplay in TextDisplay. The setBoard method initializes these fields based on row and column positions. At game start, drawGrid activates, recursively calling drawCell to render each cell with the appropriate character and colour. To mirror TextDisplay's functionality, we added labelOffset to allocate space for row and column labels on the board's edges. Text size on row and column labels increased from 6x13 to 10x20 for better readability. drawTitle introduces a game-enhancing feature by displaying the title "CHESS" and available commands, using ASCII graphics sourced from patrjk.com's Test to Art generator. Each row employs drawString for title rendering, which is conditionally displayed only when the height exceeds the width by at least 220 pixels, accommodating the title's size requirements. The module's width and height are privately set at 560 and 780 pixels, respectively, to ensure an optimal viewing experience.

High Level Implementation & Design Pattern

1. **ChessPieces, King, Queen, Pawn, Bishop, Rook and Knight, and Empty**

We will first start with individual pieces. The King, Queen, Pawn, Bishop, Rook and Knight all inherit from ChessPiece, which possess two main functions: isValidMove and generatePath, which are used to evaluate whether the move is valid on the ChessBoard according to the rule, and whether the path that take from point A to point B, if valid and exclude the endpoints, contain any piece on it (which will be evaluated in ChessBoard), which will be an invalid move if true. Note that for knight and king, there is no path that takes between them, so the move only needs to be valid.

2. **ChessBoard**

The ChessBoard consists of an 8 x 8 array that consists of ChessPiece objects stored in it. The board will be initially empty (only containing Empty pieces), and the function init() is used to initialize the board with standard chess pieces and their locations. An important thing to note is that the ChessBoard use 2D array of ChessPiece objects, not a 2D array of pointers to ChessPiece, since it offers a much simpler way to move piece to another location on the board, does not deal with heap memory, and smart pointers of prove to be difficult to implement as a board, as there is an issue of who should own what. However, a disadvantage is that since the type of object does not change after assigning to a specific piece, this would mean that we would have to copy the content of the ChessPiece to the specific piece on the stack in order to calculate isValidMove and use generatePath functions correctly. The two pointer fields, whiteKing and blackKing, are used to keep track of where both kings are on the chessBoard by assigning them to the specific piece address on the stack.

When evaluating whether a chess move is valid, and assuming that it is not a special move, the ChessBoard will mainly use three functions: isValidMove, isValidPath, and kingIsUnderAttack (3 conditions), where the first two must be true and the third one must be false (cannot make a move that cause the player's king in check), in order to qualify the whole move to be valid. The function kingIsUnderAttack works by finding every single opponent piece on the board given by the colour of the player, and evaluating whether a valid move (similar to the 3 conditions) can be made from at least one opponent piece directly to the player's king (whiteKing or blackKing depend on turn). After checking the requirements, the piece can then be moved on the board by using the chessMove function. Getters and setters, such as getPiece, addPiece and removePiece are there to support setup mode.

Checking checkmate and stalemate on the board can be determined by evaluating validMoveExist function, which returns whether a player can make a move that satisfies the 3 conditions. If not, and kingIsUnderAttack is true, then it is checkmate. Otherwise, if kingIsUnderAttack is false, it is a stalemate.

Special moves, such as castling, en passant, and pawn promotion have their own respective function to check for it. Castling (isCastlingPossible) can be done if the player makes a specific type of move (ex. e1 g1), the king and the rook in the direction of the move have not moved yet, and the king is not under attack on the original square and any of the two squares towards the rook. En passant (isEnPassantPossible) can be done if the player's pawn makes a pawn capturing move to an empty space, the opponent pawn is on the row of the pawn and the column of the pawn destination, and that opponent pawn have made two square forward moves on the last turn (using chessMove, which store the initial piece and destination piece before a move is made in chessGame). Pawn promotion (pawnPromotion) conditions are checked in chessGame, whether the white pawn is in the first row, or the black pawn is in the last row. Human player can choose the type of piece it can promote to, but for the computer it is decided that the pawn always promotes to the Queen.

The ChessBoard is where the Observer Pattern is implemented, where it keeps track of an array of observer pointers, user can subscribe using attach, and notify the observers (notifyObservers) for changes with respect to specific piece on the board)

3. **ChessGame**

The ChessGame consist of the chessboard, two subclass of Observers, TextDisplay and GraphicsDisplay, two player pointers (can be either human or computer), a moveLog that store all the chess moves that have been achieved throughout the game, and booleans that handle whether a game is won or stalemate, and who won. Beside the getters and setters, the two main functions of the chessGame are makeAMove, which have two versions (overloadings) that will be used depending on whether a move is made by a human (two strings correspond to initial and destination), or a computer (no parameters, the computer will generate move instead). If a move is a success, the chessGame will simply call the ChessBoard's notifyObservers() to notify changes to both displays, log the corresponding move to the moveLog, and switch to the next player's turn, while checking check/checkmate/stalemate at the beginning of each turn. The second function is resign(), which makes the player on the current turn resign and the opponent wins.

4. **TextDisplay, GraphicsDisplay and ScoreBoard**

Whenever in a setup mode, a game begin or a player attempt to make a move, the TextDisplay will either output an error to the output stream upon invalid move, or display the board to the screen, along with information about the turn, whether the current player is in check, or the game is won through its member functions and output overloading. GraphicsDisplay can do the same, although it can only output change to the board, and cannot notify about invalid move, check or winnings.

The ScoreBoard is the class that lives separately from other classes and only exists in the main function. Whenever a game is won or stalemate, which is given by the getters, the scoreboard will add score accordingly. The score can also be resetted through command.

5. **Player, HumanPlayer and ComputerPlayer**

The Player is an abstract class that has a colour field corresponding to the colour of the chessPiece that it will be assigned to, and a virtual function generateMove that takes in the state of the board as a parameter. For HumanPlayer, this will be empty since the move input will be done in main and ChessGame already. For ComputerPlayer (and all of its levels), the function will generate a ChessMove inside the ChessGame's makeAMove() function based on what is known from the state of the board, and the chessGame will use that to make a move.

There are four levels in our AI. A level 1 computer player is essentially choosing a move from a list of possible moves (PossibleMoveGenerator) at random. A level 2 computer player can do what level 1 does, but can also check for check and capture, and prioritize these moves instead (using isChecking and isCapturing functions inside ChessBoard). For level 3 computer players, it combines all the possible moves of level 2, but it can avoid capture, and therefore is smarter than level 2. Lastly for level 4, we assign each piece a weighted point based on the official piece value described at chess.com. In the game of chess, the King is not assigned a conventional piece value since its capture signifies the end of the game (checkmate). However, for the purposes of artificial intelligence (AI) strategy, we have assigned the King a value of 20 points. This encourages the AI to prioritize moves that lead to checkmate or avoid being checked. We set the worst possible outcome, losing the King without any compensation, at a score of -20. This score is used as a benchmark for evaluating all possible moves. Initially, until level 3, the AI tended to sacrifice valuable pieces like the Queen to capture less valuable pieces like pawns, which is not an optimal strategy. Additionally, when the best score is zero, there are two cases which are either when the piece does not capture anything and will not be captured in the next move or the piece captures its same type and will be captured in the next move. Since we want to avoid AI losing its own piece, we prioritize the first case to output. At level 4, the AI improved by predicting the opponent's next move and evaluating the worthiness of its own moves. To further enhance the AI's sophistication, we aim to predict additional future moves by both the AI and its opponent, allowing the AI to select the best possible strategy from the available options.

6. Other Helper Classes

- ChessSquare: keep track of row and column
- ChessMove: keep track of the original configurations of the piece before a move occur

Accommodate Changes

The use of MVC and Observer Patterns mean that our program can accommodate changes in various parts of the program. For example, in situations where the client wants to output to a specific file instead of standard output, all of the changes can be done by changing where to store output in the TextDisplay with minimal amount of effort. A similar argument can also be done for GraphicsDisplay if a client requests a change in how it should be displayed. The use of Observer Patterns here means that we can add as many observers as we want to notify changes to any specific component that we need to. If required, the program can also accommodate for different implementations of AI upon request, since most of the work is done in ComputerPlayer. If more rules of chess are added, or some part of the rules are modified, or the client want to change how each piece is represented in the textDisplay or graphicsDisplay, most of the work can be done in ChessBoard and ChessPiece (if necessary) only, and little to no modifications are needed for ChessGame or other places. That being said, due to the issues of ChessBoard using ChessPiece 2D array objects, adding more pieces and rules to the board may be quite challenging due to an additional modification of isValidMove and isValidPath functions upon that happen.

Coupling and Cohesion

The relationship between ChessGame, TextDisplay, and GraphicsDisplay are considered to be low coupling, as they only communicate via function calls, and the ChessGame acts according to their result without having access to the board or any of the other's implementation. However, the relationship between the ChessBoard, ChessGame and ChessPiece tend more towards mid coupling, as the ChessPiece would often need to pass arrays of ChessSquare (generatePath) to the ChessBoard to perform operations on it, the ChessGame would often need to know about the type of ChessPiece in ChessBoard in order to perform operations on it, the modifications of rules in each chessPieces or addition of special valid move would often require some changes to the ChessBoard and potentially the ChessGame itself. The relationship of ChessGame and Player, specifically if one of the players is a computer player, is also mid coupling, as the ComputerPlayer would need to return a chessMove object for the ChessGame to perform a move.

For cohesion, almost all of the modules that we implement have mid to high cohesion, since all of the elements cooperate to perform one or a few task - ChessPieces and ChessSquare are models of pieces, ChessBoard store the ChessPieces objects and is able to evaluate move based on rules and notify observers, ChessGame act on this information to make a move, control the turn and evaluate winning conditions. The TextDisplay and GraphicsDisplay are there only for display, ScoreBoard are also only for display, and the ComputerPlayer's purpose is only to generate moves for chess games to act on.

Answer to Program Specifications

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. 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.*

Since we have the class ChessMove, we can collect loads of them in a container like vector. To get the standard opening moves, we can prepare for some of them, for example, French Defence. In class ChessGame, we can implement a function to determine if the white player plays a e2 e4, then if a function detects that, it can generate a move of e7 e6 to play the opening French Defence. Involving a data structure like linked list would also work, since each move can be in a linked list and points to a highly possible corresponding move (we can get the information from the chess.com)

The difference between this and the answer in DDL1 would be the final design of out program, there will be a chessMove class and checking through the whole board function inside of class ChessBoard would lead to a high cohesion code reuse, which makes this change become easier than applying a much more complicated algorithm.

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?*

As mentioned before, we have a class ChessMove which indicates a single move on the board. When a player plays a move, we have a vector of it to store each of them. Since the latest move will be emplaced back to the vector, we can pop back to get the latest one undo, replace the piece to the board by adding a chess piece, which is a high cohesion for the class chessBoard. For unlimited undo, just simple recursion through the vector of the past move would work.

That being said, it has come to our attention this would only work for standard moves, and some moves like castling and en passant may not work for this type of structure. This is because castling is essentially a two move into one, and en passant requires us to have at least three pieces of information - the original pawn, the empty space it moves to, and the captured pawn, while our original structure only has two pieces. This prompts us to suggest a new ChessMove, but now with four variables stored in it - the original piece, the square where it moved to, the captured piece, and a boolean that indicates castling. If the captured piece and the piece that the player moves to is the same, then it will be the same as our original ChessMove. However, if it is not, then we know that it is en passant and we assign accordingly. For castling, we know where the king was originally at and the direction it moved thanks to the original and after location, so all we need now is to reassign accordingly.

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.*

We need to expand the constant that holds the board dimension. And also, the board initialization part will be modified too. Main function and chessPlayer will be changed to be able to hold four players. The answer for this question holds the same as in DDL1, since we don't have much time working on this higher level question.

Final Reflection/Conclusion

Harry: "I have come to realize that developing software in teams is difficult - there are many ways that miscommunications can happen, and it is not easy to manage a project under a tight deadline. Dividing work evenly between individual members is also a very challenging issue that presents in this project, as some members are more comfortable at working at a specific part, but in some cases the part that they work on requires other parts that they do not want to work on. I feel like I could have done better, and what I would have done differently from this project is to establish a clearer communication between project members, have a better understanding of my daily schedule so that I could spend time more effectively in future project, and maybe try not to do too much work and spend a little bit more time on distribution of work between all members."

Gen: “Reflecting on the project, I’ve gained valuable insights into the importance of communication and flexibility. Initially, I found myself overly focused on the design aspects, somewhat at the expense of the core functionalities. This approach seemed effective in the early stages allowing me to capture the bigger picture of the project. However, as challenges and errors arose, I realized my attention was to finish other classes first rather than collaborating with team members to resolve these issues. This lack of focus resulted in miscommunications, particularly when significant changes were made to the parent class without corresponding updates to the UML diagram. On a positive note, I was genuinely impressed by the adaptability of my peers. Their ability to reevaluate and creatively rethink fields and methods for enhanced functionality was not only effective but also inspiring, teaching me the value of maintaining a balanced approach and the importance of being open to different perspectives and solutions.”

William: “It was my first time doing a project like this (involving teamwork for several weeks in a language that he had no prior experience about), I found out the hardest part of doing object-oriented programming is designing a correct, efficient UML and useful methods. After breaking a huge problem into small pieces, working on each class with abstractions of other classes is a lot easier.”