A5 Chess Final Design Document

John, Tim, and Ricky

Introduction

For our CS246 final project, we decided to code Chess. We agreed on this topic because we were all very familiar with the basic rules of chess, and understood the testing scenarios before beginning the programming portion of the assignment. In this final design document, we will be going over the basic flow of our program, how it was structured/designed, responses to some preliminary questions, what this project taught us as a software development team, and some final remarks and thoughts about this assignment.

Overview

To begin with, we were able to fully implement all required features of chess in our program. Throughout implementation, we also found ways to integrate bonus features that will be outlined later in the document. Before we begin discussing the design behind our code, it is important to understand the flow and nomenclature of our game. When the user compiles the program, they are given two options. They can either enter setup state, by calling “setup”, or go straight into a game between two players by calling “game.”

To begin with setup, we assume that the user can create their own board from scratch, and thus, an empty board is given to the user. The user can add pieces or replace existing pieces (+ piece coordinate), take away pieces (- coordinate), or change the player's turn (= color) before entering game state. All of these commands can be issued before the input “done”, in which the program verifies that there is a king for both players, no pawns for either player are in row 1 or row 8, and that both kings are neither in check or checkmate. Once these conditions are passed, the game now has a valid board that can be passed in to start a new game. If the setup phase is never entered, then the program assumes any game that is started begins with a default board.

To start a game, the user must issue the command “game”, followed by the two players. Both players can vary between a human and computer player, and to call a human player, the user should call “human”, and for a computer player, “computer[*level*]” where level varies from 1 to 4. Once these commands are issued, the players alternate in commands by either calling “move” followed by two coordinates, (e.g move e2 e3) or calling resign to forfeit and end the game. Once the game ends, the winning player gains a point. If the result of a game is a draw, both players gain half a point.

Once the user decides that enough games have been played, they can end the program by signaling EOF (ctrl-D) and a scoreboard is shown displaying the scores of each player.

Updated UML

This is attached at the end of this document.

Design

While it may be very tempting to code our chess game all in one file, the goal of our design is to maximize cohesion, and minimize coupling. In order to do so, we incorporated the fundamentals of Object-Oriented programming as well as design patterns to ultimately make our code more efficient.

1. **Display / Text Class:**
   An important design element to our project was using inheritance to create multiple subclasses of abstract base classes. For instance, all levels to the computer player and the human player were subclasses to an abstract base class titled Player, and similarly, the text display and graphic display classes were subclasses to ChessDisplay. The ChessDisplay class took in a single virtual method titled “Display” where given a board, it would display the contents of the board, specifically, the pieces and the color of the tiles. This way, we were able to override this method and output our board based on which screen we wanted to display the contents to (either the terminal or on XMing).

2. **Board Class:**
   The Board class internally uses the observer pattern. We store an 8x8 array of Square objects, each of which maintains a list of moves and a list of observers. When a Square changes, for example via a piece move in or away, it notifies its observers, who then update their list of moves. This makes Board updates efficient, as we do not need to loop through every piece on the Board in order to update the set of legal moves, only those pieces whose movement was blocked or is now blocked by a changed Square.

   The same notify mechanism keeps track of the number of pieces attacking each player’s king, allowing us to easily recognize a move leaving the mover in check as an illegal move.

3. **Player Class:**
   In order to effectively implement object oriented principles, we made an abstract base class called Player that was responsible for returning the appropriate action based on the 5 derived classes (HumanPlayer, Computer1, Computer2, Computer3, Computer4).

   The action class then uses the visitor pattern. Now, we could simply pass a reference to a Board to the Player and ask them to make their move. But that is giving too much power to the players. Currently all subclasses of Player are guaranteed to only make legal moves, but who’s to say in the future? We could have a Player subclass that represents a remote player. Even if the remote player acts in good faith, it is still possible that their move data becomes corrupted during transmission. Instead, the ultimate authority to actually modify the Board lies in the ActionPerformer class, which is used by the Game class. This way, the Game class has full control over the Board, instead of allowing a Player to potentially make malicious changes to the Board.

   1. Human Player Class:
   The human player class was responsible for returning the action based on user input. If the user were to resign from a game, our board would take that resigned action, and perform the necessary steps (vice versa for inputted move, undo).

2. Computer Player Classes:

1. Computer1 simply uses a random number generator to choose a random move from our board's list of legal moves.

2. Computer2 first creates a temporary board using the copy constructor, loops through the list of legal moves, and after each move, either 1) analyzes the new board state to prioritize checkmates and checks or 2) utilizes the evaluation function to find the move that minimizes the enemies points (hence getting the best capture possible).

3. Computer3 builds upon the previous level by not only prioritizing checks / checkmates / captures, but also avoiding captures. We do so by looking 2 moves ahead of the current board state (predicts the enemies optimal move), and utilizes the evaluation function, which is tuned to evaluate the maximum difference between the current players points and the enemies points.

4. For our Computer4 implementation, we decided to implement the minimax algorithm to evaluate what moves our computer player would make. It is important to understand that Chess is a Zero-sum game, which is a mathematical representation of a two-player game, where one player results in victory and the other results in a loss. Therefore, the minimax algorithm is appropriate for this game because it revolves around the idea that each player will make the optimal move. This is much more sophisticated than the level 3 implementation because based on the guidelines of a player who “prefers avoiding capture, capturing moves, and checks,” the computer player is restricted to rules where the strategy revolves around playing not to lose, rather than our 4+ implementation which plays to win!

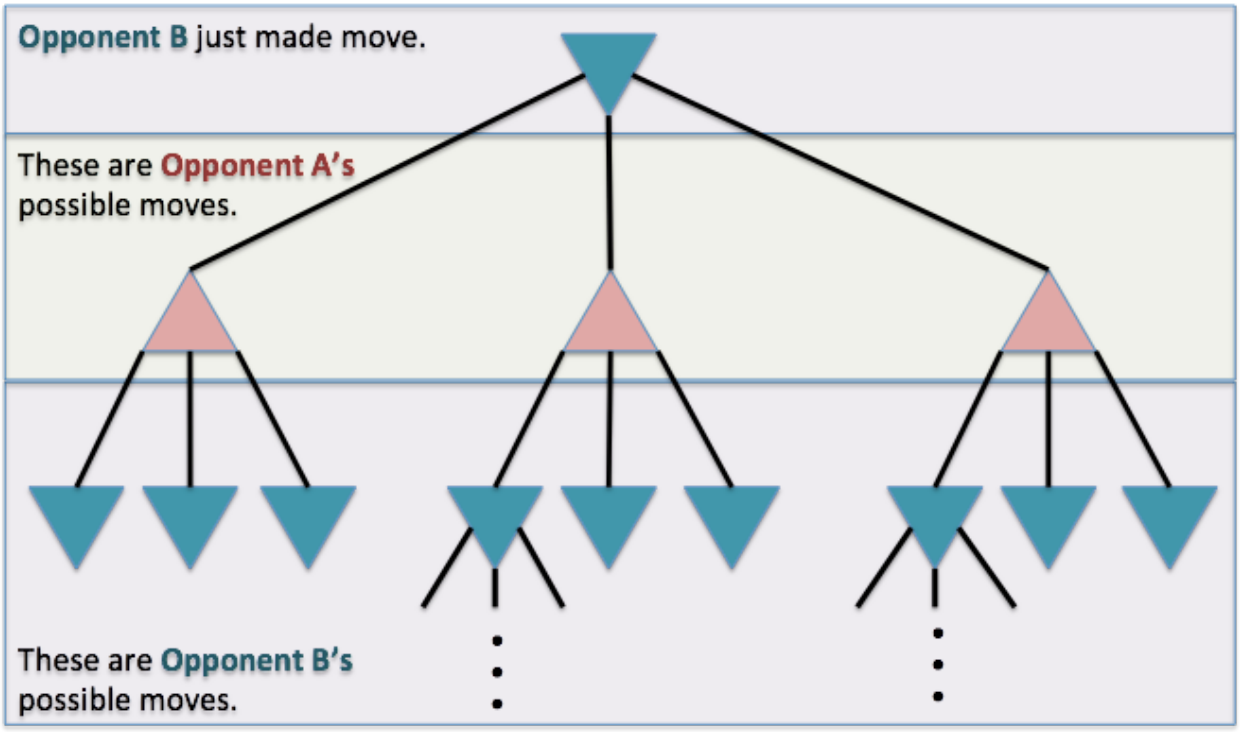


Figure 1: Tree diagram of the minimax algorithm

Figure 1 shows the visual representation of this recursive algorithm. Each node represents a state of board after a specific move, and it splits into n-branches, where n represents the length of our list of legal moves. Each node is associated with an evaluation weight, which denotes the player's board points as a result of performing that move, and the goal is to find the move which results in the most points. The deeper the tree (parameter depth adjustable), the more moves the algorithm can look at, learn, and choose from. The tradeoff to a deeper tree however is the runtime of our program, because more nodes results in a larger search space. We solved this inefficiency by implementing alpha-beta pruning, which selectively removes redundant parts of the tree. It is important to note however that computer4 takes significantly longer than any other computer player.

Coupling and Cohesion

In this project we use a number of plain-old datatypes to aggregate data, for example, Coord holds two integers row and col, and Move holds two Coords, from and to, as well as a PieceType for promotion moves. These types exist for convenience and serve only to group related data. The main modules of the project are Board, Game, TextDisplay, GraphicDisplay, HumanPlayer, and ComputerPlayer[1-4]. These modules are loosely coupled as they communicate with each other through method calls passing either plain-old structs or const reference to a Board. No two modules are friends, the only communication is through public methods. And since we are programming to the interfaces Player and ChessDisplay, the coupling is further reduced since we can swap out the underlying Player or ChessDisplay without any change at all to the code that use them.

Each of these modules also have high cohesion. Board has a single responsibility: to maintain a chess board that can only be mutated through legal chess moves. All of the public methods exposed by Board are to this end. Each Display module performs the singular task of drawing the Board in some way, and each Player module performs the singular task of coming up with a move given a certain board position.

We are quite happy with our design, as it has low coupling and high cohesion. We are certain that the value of such a design only increases as time goes on, in the software maintenance and update cycle. Even during our short development cycle, we have already reaped some of the benefits of this design, in the form of different team members being able to independently work on different parts of the program based on the pre-established interfaces, with the confidence that by the time we are done, our code will simply work together.

Resilience to Change

Programming to interfaces as opposed to implementations pays off in spades when it comes time to change or extend the program. Below we outline ways in which the interfaces through which the different parts of our program communicate allows for changes.

Player Interface
Suppose we wish to enable playing against remote opponents. This can be added as simply as creating another subclass of Player that performs HTTP requests when asked for an Action. We could also add new types of computer players that use completely different algorithms to make their decisions, or perhaps employ inter-process communication mechanisms to delegate its decision to an external chess playing program like Stockfish. These are but a few examples of the vast flexibility afforded by the Player interface.

ChessDisplay Interface
Just like the Player interface, the ChessDisplay abstraction allows us to easily add new ways to display the chess board. For example, we could add a display that draws the pieces with fancy graphics instead of just text, a display that communicates over the network to show the board to a remote opponent, or a display that communicates via a serial bus to draw the chess board to a dot matrix display peripheral.

Answers to Questions

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.

First, in our Board class, we need to change our method of determining whether a move is legal or not, to only allow for standard opening moves. We can create an abstract base class (BookOfOpenings) that takes in its own board, and given a move, makes the move on the board to determine if it is a standard opening. One way to implement this logic is by using a Tree. The root of the tree can represent the initial board configuration (8x8 board with pieces in their starting positions), and each node within the Tree represents a new board configuration, based on the edge that represents a specific move. The current node would represent the current board configuration, and we can continuously check if the given move from a standard book of openings is an edge of the current node to its child. If it is, then we can update our current node, and continue traversing through our Tree. If at any point we cannot match the standard opening with an edge of the current node to its children, then we know that this is not a valid opening. If we have reached a node that does not have any children, then our opening phase is complete and we are no longer restricted by the book of openings. Therefore, our opening set of moves is approved and the player/computer can freely make any move they want.

How would you implement a feature that would allow a player to undo his/her last move? What about an unlimited number of undos?

To begin with, we would need to change the move class to allow undo as a possibility. This would allow either a Human or Computer to undo their last move, although we assume that a computer will never use this option. To allow the player to undo a single move, we can simply store the previous move made by the player. Based on this, whenever a player requests an undo, we can reverse the effect of the previous move. Keep in mind, we must also undo the opponent's move as moves are made on an alternating basis. Some special moves like castling and en passant should not be a problem as it directly follows the logic stated above, but for pawn promotion, we would need to add special logic to change the type of the piece back to a pawn.

When implementing the option for a player to make unlimited undos, we must store every move made by the Player in a vector, along with additional Board metadata needed to restore the previous state perfectly, so that we have access to the entire history of the game. Essentially, we treat each undo similarly to the above, but have the option to do so any turn we would like, as many times as we like. We can perform this action using standard vector methods like push back and pop back.

If we were to implement a standard book of openings in addition to this, then we must inform our abstract base class (BookOfOpenings) that an undo has been made, so that we can move back to the previous node within our tree.

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.

For starters, four-handed chess has a board of a different shape and size from regular chess. It has a 3x8 rectangle protruding from each side of a standard 8x8 chess board. We could account for this by changing the Board class to use a 14x14 array to represent the grid instead of an 8x8 one. Now, this includes extra squares that are not actually part of the four-handed chess board, but we can simply disallow movement into these squares via changing the isLegalMove method. Since the 2D array storing the squares is central to the Board class, all methods of the Board class would need to be updated to account for the new shape and size of the board.

The Player interface would not need to change. Although it deals with Board, it is not concerned with the underlying implementation of Board. However, the concrete subclasses of Player would need to change. HumanPlayer would need to change to take input corresponding to four-handed chess moves instead of regular chess moves, and computer players would need to change to play by a different set of rules.

The ChessDisplay interface does not need to change for the same reason as the Player interface, however again concrete implementations of ChessDisplay do need to change to draw a different kind of board.

In the implementation of Board, for checking if a move is legal and enumerating legal moves, the logic for checking if a move goes outside the bounds of the board can be separated from the logic for checking if the move obeys the movement pattern of the piece being moved. Going to four-handed chess, the movement pattern of each piece remains the same, but there would need to be different logic for checking if the move goes out of bounds.

There would also be significant changes to the logic in the Board for reporting the state of the game. In regular chess, the states are normal, white check, black check, white checkmate, black checkmate, white resign, black resign, and stalemate. However, in four-handed chess, each player could be under check independently. Furthermore, even when a player is faced with a checkmate, the game could go on as normal for other players. This is best captured by changing the representation of the state of the game to a 4-tuple, each element of which corresponds to the state of one of the players.

Finally, there would be a small change to the Game class. The game loop would now consist of rotating between the turns of 4 players as opposed to 2 players.

Extra Credit Features

- **Unlimited undo.** The challenge of this feature is that we would like to perfectly restore previous states of the board, however it is very inefficient to store the entire board every time we make a move. Initially we thought we could get away with only storing a vector of moves. This turned out not to be enough, as there are other pieces of information that are required to restore the previous board position: was there an en passant move? Which castling rights did each player have? What piece was captured, if there was a capture? Once we know all of these pieces of information, the undo features really pays off. Not only can the player undo moves, but it facilitates the development of computer players as well, as they can make a copy of the board, try out different moves, and then undo them.

- **Only using smart pointers.** In some ways, doing this extra credit challenge is easier than not doing it. Smart pointers made it easy to manage memory and avoid common pitfalls of manual memory management. However, unique pointers do take some getting used to. There were quite a few times where cryptic error messages left us scratching our heads, which ended up being caused by trying to copy an object holding a unique pointer.

Final Questions

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

Upon developing software in a team, we learned how important it is to stay communicated with one another, especially for a project that has to be delivered in a short timeline. We found that because of this, a lot of us were encountering conflicts when merging our code together as it was inevitable to work on many of these classes simultaneously. Of course, these conflicts became more difficult to resolve the longer one of us had gone without committing our changes, and so it taught us to frequently commit our changes and notify our team members when we did.

It also taught us how important it is to listen to one another when discussing how to approach a specific task to the project. While we may individually have an idea of how to implement something, it is important to be open minded to solutions that may not come to our minds during the planning phase that may make our design more flexible or simplify our code complexity.

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

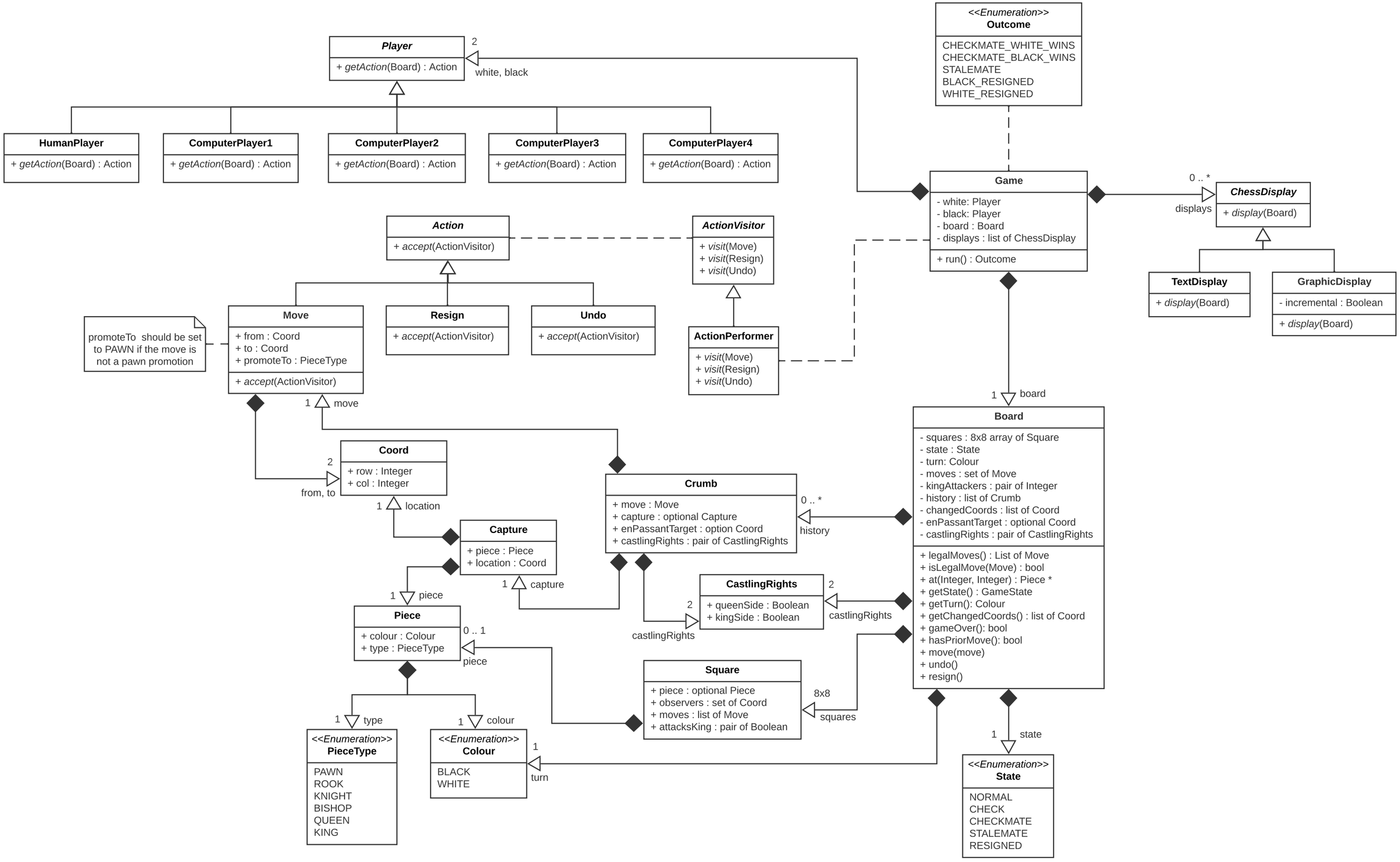
For starters, our Board class was quite detailed, and shared many dependencies with other classes in the program. Thus, if we had a chance to start over, simplifying the board class into more subclasses whilst still maintaining object oriented principles would allow for more simultaneous work. Additionally, we also agreed that the computer player algorithms were challenging to understand/implement (especially levels 3 and 4+), and so before beginning actual implementation of code (during planning phase), doing research on game theory and listing possible solutions would have most likely been very beneficial. Finally, while it is popularly known that git is the most common way for multiple people to work on a single project at once, for a project that has to be delivered in a short period of time with limited developers, we could have possibly explored IDE extensions like VSCode LiveShare, for us to work on specific classes at once.

Conclusion

Assignment 5 taught us many phases of a full software development life cycle that is very important and applicable to real life working scenarios. From using good object oriented coding principles like classes, inheritance, encapsulation, and design patterns, to interacting and communicating with developers through version control and Github, this project challenged us to apply what we learned throughout CS246 into a real-time application. While chess is a game introduced to us at a young age, there is much logic and game theory knowledge needed to understand how to make the right move on a board, and how AI can be developed is a complicated, challenging task. On a final note, while implementing chess may have been quite a difficult project, it has definitely made us more skilled players!

Works Cited

Eppes, M. (2019, October 6). *How a computerized chess opponent "thinks"-the Minimax algorithm*. Medium. Retrieved April 3, 2022, from https://towardsdatascience.com/how-a-chess-playing-computer-thinks-about-its-next-move-8f028bd0e7b1


Player
+ getAction(Board) : Action
2
white, black
<<Enumeration>>
Outcome
CHECKMATE_WHITE_WINS
CHECKMATE_BLACK_WINS
STALEMATE
BLACK_RESIGNED
WHITE_RESIGNED
HumanPlayer
+ getAction(Board) : Action
ComputerPlayer1
+ getAction(Board) : Action
ComputerPlayer2
+ getAction(Board) : Action
ComputerPlayer3
+ getAction(Board) : Action
ComputerPlayer4
+ getAction(Board) : Action
Game
- white: Player
- black: Player
- board : Board
- displays : list of ChessDisplay
+ run() : Outcome
0 .. *
displays
ChessDisplay
+ display(Board)
TextDisplay
+ display(Board)
GraphicDisplay
- incremental : Boolean
+ display(Board)
Action
+ accept(ActionVisitor)
ActionVisitor
+ visit(Move)
+ visit(Resign)
+ visit(Undo)
promoteTo should be set to PAWN if the move is not a pawn promotion
Move
+ from : Coord
+ to : Coord
+ promoteTo : PieceType
+ accept(ActionVisitor)
Resign
+ accept(ActionVisitor)
Undo
+ accept(ActionVisitor)
ActionPerformer
+ visit(Move)
+ visit(Resign)
+ visit(Undo)
1
move
1
board
Board
- squares : 8x8 array of Square
- state : State
- turn: Colour
- moves : set of Move
- kingAttackers : pair of Integer
- history : list of Crumb
- changedCoords : list of Coord
- enPassantTarget : optional Coord
- castlingRights : pair of CastlingRights
+ legalMoves() : List of Move
+ isLegalMove(Move) : bool
+ at(Integer, Integer) : Piece *
+ getState() : GameState
+ getTurn(): Colour
+ getChangedCoords() : list of Coord
+ gameOver(): bool
+ hasPriorMove(): bool
+ move(move)
+ undo()
+ resign()
Coord
+ row : Integer
+ col : Integer
2
from, to
1
location
Crumb
+ move : Move
+ capture : optional Capture
+ enPassantTarget : option Coord
+ castlingRights : pair of CastlingRights
0 .. *
history
Capture
+ piece : Piece
+ location : Coord
1
capture
1
piece
CastlingRights
+ queenSide : Boolean
+ kingSide : Boolean
2
castlingRights
2
castlingRights
Piece
+ colour : Colour
+ type : PieceType
0 .. 1
piece
Square
+ piece : optional Piece
+ observers : set of Coord
+ moves : list of Move
+ attacksKing : pair of Boolean
8x8
squares
1
type
1
colour
<<Enumeration>>
PieceType
PAWN
ROOK
KNIGHT
BISHOP
QUEEN
KING
<<Enumeration>>
Colour
BLACK
WHITE
1
turn
1
state
<<Enumeration>>
State
NORMAL
CHECK
CHECKMATE
STALEMATE
RESIGNED