CS246 Project Design Document: Chess

Ethan Zhang, Danyang Wang, Arsen Cui

Overview

Whenever the program is run, a Game object is created. A real-life chess game requires two essential components: a chess board (with chess pieces), and two players. Just like the real-life scenario, the Game class consists of a Chessboard class, and a Player class for the white player and the black player, respectively.

First, we will look at how the Player class is implemented. We know that each player is either a human, which makes its moves from standard input, or a computer, which makes its moves based on written algorithms (explained in the Design section). Thus, the Player class acts as a template for the Human class and the Computer class. We know that a player must have the ability to make a move, thus the virtual function “turn” is used to ensure that player makes a specific play. For the Human class, we know “turn” needs to read user input to perform the play. Thus, “turn” is overridden with processing the input commands, checking the validity of that play, and making that play on the Chessboard only if it is valid. On the other hand, a Computer does not require any user input. The function “turn” is then overridden with the AI’s play, compiled based on the level of the bot.

Next, we analyze the Chessboard class. We know that a chessboard must consist of black and white pieces. Thus, we use two vectors of Piece pointer type, one representing all white pieces and the other representing all black pieces. Each Piece consists of all the essential information that a piece can have: its location on the board, its colour, its value (king, queen, etc.). Furthermore, for special rules like castling, where the rook and the king must not have been moved, we use a boolean to keep track of that piece’s movement. With the Piece class established, we may include a class for each specific piece, including King, Queen, Rook, Knight, Bishop and Pawn. Since each piece has its own rules of play, we have a virtual method “checkValidMove '' in Piece class to see if it’s possible to move the piece to the given location under the rules of that specific piece.

Now, we discuss the design patterns used in the structure of the project. Both Decorator and Observer were used. First, for the decorator, we know that a Chessboard consists of numerous Piece’s. Thus, the Piece acts as the component, with each specific piece class (King, Queen, etc.) acting as a concrete decorator. Finally, the Square class acts as the concrete component. As a result, when a specific Piece is called upon in Chessboard, calling Piece->checkValidMove()

would go through the virtual method, and to the specific overridden function of that piece type. Next, for the Observer, it was used to update both the text display and the graphical display as a move is made by a player. The Chessboard is the concrete subject that's being observed, as it always contains the up-to-date state of the current game, namely the locations of all pieces. Next, concrete observers include the text observer and the graphical observer, as we have both as available displays of the current game. The notify function in this case would then be rendering the display, as the state of the chessboard only changes every time a move is made, and thus, it should be updated on the display at that time. With respective Subject and Observer classes for tracking the observers and getting the state of the chessboard, we have a complete Observer model that connects the chessboard and the text & graphical displays.

Design

Piece Move Validity Design

There are a total of six piece types, and thus, six sets of rules for the movement of pieces. It's an essential part of the project, as the rules of chess govern how the game may be played. Furthermore, when a computer AI makes a move, it must be guaranteed to be a valid move. We will explain how exactly each rule is implemented in the program.

<u>King</u>

A king can move to any of its eight adjacent locations. However, the special rule of castling is called to make the king move two units towards one of the rooks. Thus, checkValidMove checks whether or not the target position fits in one of those cases. Castling requires a few additional verifications, including that the king and the rook must not have been moved, no pieces can be in the path, and the king must not be in check while in cross.

<u>Queen</u>

A queen can move in any direction, and move any number of units following a straight line. The implementation was relatively simple, as it is a combination of the move rules for rook and the move rules for bishop. No additional rules affect the queen.

Rook

A rook can move vertically or horizontally any number of units. Also, no piece must be blocking the path. In our implementation, we go through every position in between the original position of the rook and the target position, and ensure that no piece must be on any of those spots.

Bishop

A bishop can move diagonally any number of units. Similar to a rook, no piece must be blocking the path. In our implementation, similar to the rook implementation, we go through every position in between the original position of the rook and the target position, and ensure that no piece must be on any of those spots.

Knight

A knight move follows an L-shape, where it moves 1 or 2 units in one axis and 2 or 1 units in the other axis, respectively. With a total of eight possible locations, we check if the given location is at one of those destinations.

Pawn

Being the piece with the least point value, pawn was in fact the hardest to implement. When in initial state, represented by the “move” boolean in Piece, a pawn can move one or two units towards the opposite side. Otherwise, it may move one unit toward the opposite side if the target location is empty. The capture moves are different: a pawn can only capture diagonally by one unit. Thus, whether or not the target location contains an enemy piece is essential when checking the pawn move. Finally, promotion is achieved when the pawn’s location equals the opposite side’s first row, and en passant uses the “enPassant” boolean to see if the diagonal empty capture is valid.

Computer Player Levels Design

For the sake of clarity and simplicity, from here on, the term “move” will refer to moving a piece to an unoccupied spot, rather than a general move. Thus, in every play, the AI either makes a move, or makes a capture.

Level 1 - Random Legal Decisions

For the easiest level, the computer tries a random move or capture. It goes through every piece that it has, and collects all the locations it can relocate to through the findMoves() and findCaptures() methods in the Piece class. Since every legal move specifies the starting location and the ending location, we use two vectors, each of type pair<int, int>, to store the starting location (startX, startY) and ending location (targetX, targetY), respectively. Finally, choose a random integer in the range [0, vector.size()) as the index, and perform the move at that index. As the decision is entirely random, the move or capture by the computer does not consider its consequences at all. This would theoretically lead to more giveaways, and ultimately easier to beat.

Level 2 - Prefers Captures and Checks

Level 2 can be described as an aggressive player. First, to consider the concept of prioritizing some decisions over others, we will use two pairs of vectors instead of one pair in level 1. The first pair, nicknamed VIP, stores all the decisions preferred by the algorithm. The second pair, denoted by LOW, then stores the remaining legal decisions. After all decisions are analyzed, the algorithm first attempts to choose a random turn from the VIP vectors. If there are no preferred decisions available, then the algorithm takes a random turn from the LOW vectors. This way, whenever a capture or a check is available, the level 2 bot will certainly gear up for a full-on attack.

As we mentioned earlier, every turn the computer takes must be either a move or a capture. In order for a check to be performed, it must also be the direct result from either a move or a capture as well. However, a capture already qualifies to be a preferred decision, then any check that results from a capture would automatically be included. With the leftover checks available after a move, the algorithm goes through all the computer's pieces and finds all legal moves with findMoves(). For each move, it performs that move first (using the setPiece() method from the Piece class), and verifies whether or not the opponent would be in check. If it qualifies as a check, the algorithm adds that move to the vector of prioritized decisions. Otherwise, it's a non-prioritized move, thus it's categorized as a low priority decision.

Compared to level 1, which is entirely random, this level has one and only purpose: attack at all costs. As a result, the level 2 algorithm would theoretically perform better than level 1. However, as level 2 does not consider any consequences, and only thinks one move ahead, it will not perform well against humans, or other bots that have a counter to this strategy (coming up!).

Level 3 - Prefers Captures, Checks, & Avoiding Captures

Consider the algorithm for level 2 again, if you will. One major problem about the algorithm was already brought up: it does not examine how beneficial the decision is in future rounds. For example, the algorithm may take its own king to capture an enemy pawn, but doesn't consider that it may get immediately captured when it's the opponent's turn. Thus, the algorithm uses a new concept: a favorable decision. The standard points system for evaluating chess moves states that pawns are worth 1 point, knight 3 points, bishop 3 points, rook 5 points, and queen 9 points. In the example above, if our king was lost after that round, the amount of points lost would be 9 - 1 = 8, as we lose a king and the opponent loses a pawn in the worst case scenario.

Next, we return to the concept of dividing moves and captures into prioritized decisions and non-prioritized decisions. First, a check that results from a move is always prioritized, just like in level 2. Next, consider the non-check moves. After that move has been made, it would either be targetable by the opponent or non-targetable. Recall the idea of a favorable decision: the amount of points the computer loses should be less than the amount of points the opponent loses in the worst case scenario. If this (non-capturing) move is made, and it may get immediately captured by the opponent, we earn no points, and the opponent gains points. This is not ideal at all, thus it qualifies as a non-prioritized move. Otherwise, we know that the current move does not put that piece in danger. With the move avoiding being captured, and likely to take more control of the chess board over the opponent, it's considered to be a prioritized move.

On the other hand, the captures are divided into two cases as well: the capture does not put the piece in danger (best case scenario), and the capture puts the piece in danger. For the first case, it would qualify as a prioritized capture, as it performs a capture and avoids being captured at the same time. For the second case, we compare the point values of our piece and the enemy piece captured. If the points the computer gains is greater than or equal to the points the opponent would gain if they choose to take out our piece in the following move, then with a net gain, it's considered a favorable capture. Otherwise, we have a net loss in points, thus it's not favorable.

Overall, the level 3 algorithm further improves the concept of the level 2 algorithm. Instead of being blindly confident and taking out every possible piece, the level 3 algorithm thinks two steps ahead, and evaluates the move in terms of piece point values. This prevents the computer from performing giveaways, such as putting its own piece at a location threatened by enemy pieces, and using high-valued pieces to attack low-values enemy pieces. Conclusively, it would perform better than level 2.

Level 4 - Nicknamed “Global Defensive”

In the level 3 algorithm, we proposed the concept of a favorable decision. However, the points gained/lost that the algorithm considers is only on the piece used to perform the capture, and the enemy piece being captured. In fact, realistically, any move/capture performed not only affects the immediate pieces involved in that turn, but may also have impacts on the entire chessboard. For example, moving a pawn to an unoccupied space does not seem to have any effects, but it may free up a space for its own bishop to move/capture, or block off the path of an enemy piece’s threat.

To ensure that every decision that the algorithm makes is optimal, we would perform the move/capture first, then analyze the state of the entire chessboard. Have a variable “score” that tracks the sum of the scores of all the computer’s own pieces that would be immediately threatened after that turn. This can be done by going through all the enemy pieces, and finding the point value of the piece at all locations that may be captured using findCaptures(). As the computer tries to keep as little number of pieces in danger of being captured by the opponent, then this score should be as low as possible. For example, having three pawns and a rook in danger would lead to a score of 1x3+5=8.

Next, we handle the different cases for a move and a capture. For a move, the score would have been obtained from the computer performing that move (onto un-occupied space). As the move itself does not immediately gain or lose any points for the computer, the score is kept as is. However, in the case of a capture, we use a similar argument as that of the level 3 algorithm, where the piece we use for the capture and the piece being captured should be considered. Thus, subtract the value of the piece being captured to the score (as we gain points from that capture, thus reducing the points lost from pieces in danger). Note that we do not need to add the value of our piece used for the capture, as it is already accounted for when adding the points of all the computer’s pieces in danger in the following move. Finally, a check is also beneficial. However, with the computer being a safe player, it usually wouldn’t sacrifice its queen or a rook just for a check. Thus, a check is given a point value of 4. If a move/capture gives the computer a check, then 4 is subtracted from the score.

With the score system described above, we may evaluate the value of every possible move. Finally, we find the lowest score possible, which would represent the optimal decision. If there are multiple decisions that all give the optimal score, we would choose a random decision and perform it. Interestingly, with the score preventing the computer from performing moves that would lose its pieces more, the actual algorithm barely performs risky captures. A test game was performed between two level 4 computers, and when black won with a checkmate, there

was only one piece on the entire chessboard that was captured, thus the nickname “Global Defensive”. Overall, the level 4 algorithm tries to make the safest available play to keep its pieces alive. This is a great method of countering aggressive players such as level 2, and players that only think one step ahead like level 3.

Resilience to Change

Ways in which our design accommodates new features and changes to existing features:

1. *Suppose we decide to introduce a new type of chess piece (we call the new chess piece “M”) with its own unique ways to move and capture pieces around the chessboard.* In our case, we would create a new class and header file for M and make it inherit from the Piece class, since M is a Piece. Inside the M class, we would initialize a constructor for it, for example M(int row, int col, char name); and we will override the parent’s (the parent is the Piece class) virtual checkValidMove(int targetX, int targetY, Chessboard *component) method and implement M’s version of checkValidMove. Note that the checkValidMove method basically validates whether the move and/or capture was a legal move. After the implementation, we can now just include M’s header file in our Chessboard.cc file and then we will be able to use M in our game. For example, in setup mode we will be able to place M onto the chessboard. In addition, when there are inputs such as move e4 e6, if there was a M piece on e4, then the program will automatically know to call M’s checkValidMove method and not some other chess pieces’ checkValidMove since we have inherited and overridden the checkValidMove method in Piece class.

2. *Suppose we decide to add a new capturing rule for an existing chess piece.* In this case, we will talk about how we incorporated En Passant for the pawn class in our project as an example. This special rule only applies to pawns. Furthermore, it only applies when the given conditions are met, and would be invalid otherwise. For example, en passant would result in a pawn moving diagonally by one space onto an empty location. However, without this rule holding, moving a pawn diagonally to an empty space would be an invalid move. As a result, the special conditions for en passant is incorporated into the virtual checkValidMove function, and overridden in the Pawn class only. Similarly, if we were to add new capturing rules, we can simply add those conditions into the appropriate checkValidMove function for the corresponding piece.

3. *Suppose we decide to add a new level of AI.* In the current structure of the program, the Computer class summarizes all the AI levels. More specifically, a level 3 AI would classify as a Computer object, with the “level” field equal to 3. Then, when the AI is required to make a move, it checks the level stored in that player, and calls the appropriate method “level3()” to

make the move. Thus, if we were to add a level 5 to the game, only a few classes need to be modified. First, the input should be changed to accept “computer5” as a valid input (for player). This can simply be done by adding “computer5” to the vector validPlayers in the Game class. Next, create a new method “level5()” in Computer.cc (and defined in the header file), which contains the algorithm for level 5. Finally, when in-game and it’s level 5 computer’s turn, run the virtual method “turn” from the Player class, which would lead to the overridden “turn” function in the Computer class. Finally, adding a simply “if” check for level 5 in that function would complete the process of adding the new level. In fact, all four levels were adding to the program this way. It was very efficient and simple to achieve, and it avoided conflicts with the commits by other team members.

4. *Suppose we want to change the size of the chessboard.* For example, instead of having the normal 8x8 chessboard, let’s say we wanted to expand the chessboard to 9x9. Then, when we call the constructor for the text observer and graphical observer, we just need to update the row and column value from 8 to 9. After this change, the text observers and graphical observers will display a 9x9 chessboard. In addition, inside Game.cc, we will need to set the max row and column number from 8 to 9, so that the game accepts something like i9 as a valid location on the chessboard. For the chess pieces and the levels of AI, nothing will be affected by changing the size of the chessboard; we won’t need to change how they move and capture pieces. There’s just one small exception for the pawn piece. For the pawn piece, we will need to check that the row number is 9 instead of 8 to allow for promotion. These are all the changes necessary for the program to play a chess game with a 9x9 chessboard as opposed to a 8x8 chessboard.

Answers to Questions

Question 1

On Due Date 1, we misinterpreted this question, and didn’t actually explain how this feature would be implemented in actual code, and rather we just talked about the logic behind it. To actually implement this, we could import sequences of opening moves from an online source. We could have a separate input command than “move”, called “opening”, which outputs a list of singular moves, where each of these moves transitions into a standard opening sequence from our list of sequences we imported. The user can pick one of these sequence moves to use if they so wish. Then, every time it is the user’s turn, calling “opening” will display a list of moves that could follow after the user’s previous movements in this sequence, essentially displaying the possible “next steps” in the opening sequence the user is using from our imported sequences.

Question 2

Instead of storing moves in a vector like we said on DD1, in our function that checks for king in check, the user makes a move, and if their king is in check after their move, the move is considered invalid, and the move is undone. We can use this same logic for an undo function, where we store the last location of the piece before it is moved, and then undo the move by moving the piece back to its original position. For undoing an unlimited amount of moves, we can use a std::stack, storing starting and ending coordinates of moves for all pieces in the order they were made, as well as a description of each move in the form of a string. By adding a description of the move, we can tell whether the move was a normal move, a capture, castling, promotion, en passant. In addition, the string will provide information such as which piece was captured, and which piece the pawn promoted to. Then we can undo the moves by popping the top object of the stack.

Question 3

To make a 4 handed chess game, we would have to change a few of the original classes and functions, as well as add in some new ones. The changes we make to the original code is to implement a different sized board to play on, as 4 players' pieces need to be fit onto the board. We would also have to make the program support up to 4 players instead of only 2. As for new things we need to introduce, we must implement a Points system. 4-player chess works by calculating the total amount of points of each player at the end of the game to determine the standings. This would require a new class, we can call Points, which would have a generalization relationship with the parent class Player (refer to the DD2 UML diagram). In 4-player chess, when a player is checkmated or stalemated, their pieces become grayed out and nulled. Capturing those pieces provides no points. Therefore, we have to implement a method that causes a player's pieces to be worth 0 points once they are out of the game. This is because their pieces are still obstacles on the board, so we cannot just remove them entirely.

Extra Credit Features

- A level 4 AI is written with an algorithm proven to perform better than lower level bots
- In setup mode, there is the option to allow/ban the rules of castling and en passant
- When the game is launched, the command "help" may be called both in the main menu, in game, and in setup mode to check out all available commands and the proper command arguments

Final Questions

a) This project taught us how to work on the same application, but on different files that can work with each other. Lots of communication was required to integrate all the parts together. We used github effectively to store our code in a repository to push and pull contributions from the team. We learned that it's important to start by planning out a structure for the project, as all the features and functionalities will have to follow the structure. If there is no set structure, some things may not be able to be integrated well. Having a structure makes things easy to modify and improve as well, since the basic layout is already in place, and doesn't need changing. This makes working more efficient, without major conflicts as different parts are worked on. We learned the importance of timing, how we needed specific features to be finished in the same time frame in order to fit them together and start on the next step.

b) We would have changed the way we did our graphics observer, as our current program has to redraw every square in the chessboard every time a move is made, making it slow to display and very inefficient. Instead, we could just redraw the squares that are affected by the move made by a player. We could do this by creating an "undraw" function, where it removes the piece at the square that is passed into undraw, and then create a draw function for drawing specific squares, and drawing the piece into the new location. This would make it so only the affected squares have to be redisplayed.