Final Design for Chess Game

Philip Chen, Celine Chung, Charles Liu

University of Waterloo

CS246, Object-Oriented Software Development

Dr. Ayub, Dr. Hackman, Dr. Kierstead, Dr. Lushman, Dr. Richards, Dr. Roh

December 15, 2021

Overview

Our program can be broken down into two main pieces: game logic and display.

From the top down, the game logic begins by instantiating the Game class and executing the setup function or the start function in main. Afterwards, Game handles the rest of the user input. Accordingly, Game instantiates an 8x8 board which then instantiates 64 squares for each cell. Each square is then assigned a piece if appropriate during setup or at the beginning of the game.

Calculating which moves are valid can also be broken down into two pieces: the "main" moves and the "pseudo" moves. The main moves refer to the moves each individual piece calculates based on its coordinates and the board dimensions, since each piece knows how it can move. This is later flattened and filtered by the board to remove "main" moves that can't be executed since they may be blocked by other pieces. This is also flattened even more if the king is being checked, so only the moves that can stop the check remain. As only the board knows of all the pieces on it, it does all the flattening and filtering of the moves calculated by each piece. It also calculates the "pseudo" move where pawns can take pieces diagonally.

To account for win conditions and its edge cases, given a move the player wants to make, we calculate all possible piece moves one step forward to check for any checks, checkmates, by both players (since moving your own piece can actually open your king to a checkmate). This was also necessary to do in order to calculate the next best move our AI can perform against its opponent.

Finally, to handle displaying the game, we used an observer design pattern between the board and the text and graphics display. This allowed for the simple exchange of information and rendering of the squares when appropriate.

Design

The first challenge we faced was communicating the various piece types, colour types, and player types across the application. We wanted a consistent symbol across each file, so we decided to create separate enumeration files to handle this issue. This came with the benefit of having a single source of truth in terms of how our types are interpreted. However, this also came at the cost of requiring separate maps and switch/case statements when casting from something like a string to our enumeration types.

Another challenge we faced was deciding how to code our board in our program. We thought about using 2-d arrays and vectors, but ultimately decided on using a map with the keys being string representations of the coordinate (ex: "a1") and the value being the pointers to Square Objects representing the squares on the board. Using a map, we can see if the coordinate exists on the board by seeing if anything is returned after using the coordinate as a key. If a nullptr is returned, then that coordinate does not exist on our board. This also lets us make any board we want as long as we have the coordinates representing the board.

Another problem we faced was how we were going to efficiently calculate valid moves for our chess pieces. Ultimately, we decided that we'd have to calculate every single move possible on the board in order to handle cases like check, checkmate, pins, and more. To solve this problem, we broke it down into pieces. First, since each piece only knows how it can move, each piece is able to calculate all of the "regular" moves given its board coordinates. The data structure we used as our return type is a vector of a vector of coordinates, with each vector representing a direction the piece can go. For example, a rook can move in 4 directions (up, down, left and right), so the rook will return a vector with 4 vectors of coordinates. The reason for having a nested vector is to filter out the "regular" moves that can't be executed since they're blocked by the existence of other pieces on the board, or the "regular" moves aren't legal since the king is being checked.

Since the board knows where all the pieces are on the board, the board will be able to filter all the "regular" moves made by the pieces. So, the board will go through each nested vector returned by the pieces and combine all the possible moves into a single vector of strings, where the strings represent possible moves in the form of "[starting coordinate] [end coordinate]", similar to the move command. The board will also calculate any pseudo moves that are possible, such as pawns capturing diagonally. The reason is, it's impossible for a pawn to know that it can take a piece diagonally without knowing where the other pieces exist on the board.

Lastly, we leveraged the observer pattern to communicate between the board (subject) and our text display and graphics display (observers). This presented two benefits:

1. *Low coupling between our classes.*
   The interface between the board, the squares, and the displays all have public methods to either retrieve state or notify others of changes without having to worry about the implementation of other classes.
2. *Simple object attach and detach process.*
   Adding an additional observer to the board is as easy as instantiating another observer and attaching it to our program. Same with removing the observers. Such a design pattern gives us the flexibility to add or remove however many observers we want with only a few lines, which is very helpful when it comes to development. We can disable graphics with only a couple of line comments when working locally.

Resilience to Change

Due to our frequent use of enumerations, adding additional options to our piece types/colours/players only requires adding another enumeration. Though each file may need additional implementation depending on which enumeration we're adding, our program is resilient to breaking while implementing new enumerations.

Additionally, our program handles errors very well. Internally, given unexpected inputs, our program will either throw an error, catch it, and output a message, or return an empty or failed response. Therefore, if we were to reuse our functions and accidentally pass invalid arguments, our program will still be usable despite the unexpected occurrences.

Rather than assuming certain characteristics of chess to be true, we made our implementation very flexible. For example, we use a map to represent the board shape. So to determine if a piece wants to go to a coordinate on the board, we can simply just check if the coordinate is a valid key in the map. This makes it easy for us to determine valid regular moves for any board shape, and makes our code less prone to logic errors. Furthermore, rather than using a boolean everywhere to determine if a piece if black or white, we used a colour enumeration which can be expanded to allow for more pieces with colours other than black or white, which may be necessary in games like four player chess.

Lastly, our program follows important OOP guidelines. Classes are modularized by setting helper functions private to prevent outside classes from accidentally calling them and changing state. All displays inherit the same methods from the Observer class, making it easy to build and attach a different display; and each piece polymorphs member functions like getPieceType and getValidMoves from the parent class Piece, making it easy to add new, custom pieces to our chess game. Every class has low coupling. For instance, our pieces are only able to calculate moves based on passed in parameters and not what other pieces are on the board. Thus, adding additional, non-complex features are straightforward and require little modification to previously existing code.

Answer 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.

We'll first create an StandardOpeningTree class where the nodes have a string representing the moves made in the standard opening. The root of this tree will be the first move of the standard opening and each response for a given move will be represented as a child node. Therefore, each layer of the tree represents a turn in the game.

The book of standard opening will then be a map with the name of the opening as the key, and a StandardOpeningTree object pointer as the value. We will then traverse the tree depending on what we want to accomplish.

For example, if the player wants the computer to use a specific opening, then our program would access the StandardOpeningTree object. The computer will traverse then traverse through the tree and perform the move commands using the values returned from the nodes.

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

In our program, the Board class is responsible for knowing the state of the game, which is essentially the state of the board. So, it should also keep track of all the moves made during the game, which can be tracked in a stack of strings. Each move will be represented as a string in the the form of:
"[*initial_position (ex: e4)*]-[*final_position (ex: e7)*]
-[*chess_piece_taken, or null if no piece was taken*]".

We "move" our pieces by creating a new Piece by using a unique pointer on the new square, and assigning nullptr on the old square, since there is no longer a piece there. So to undo the move, we can pop the string out of the stack, which tells us the initial coordinate, final coordinate and any captured pieces on the final coordinate. We know that the piece that moved will be at the final coordinate, so we just create a new Piece of the same type, and create another new Piece at the square at the final coordinate, since we have the information about it from the popped string. Doing so will make our board back into the previous state.

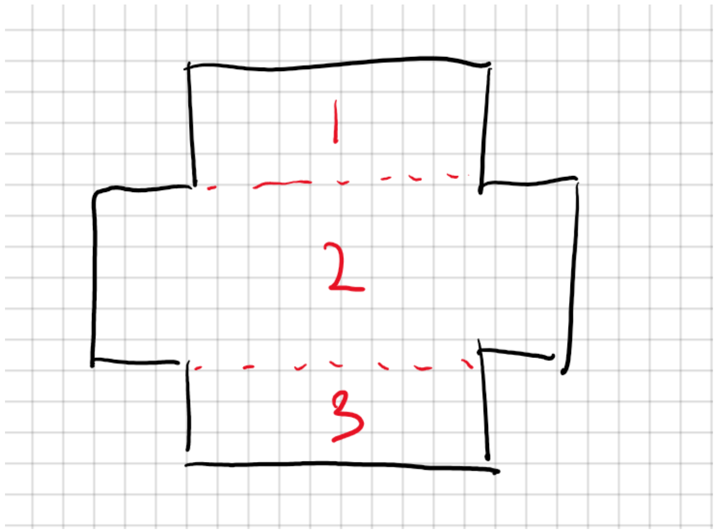
Variations on chess abound. For example, four-handed chess is a variant that is played by four players (search for it!). Outline the changes that would be necessary to make your program into a four-handed chess game.

We first have to change:

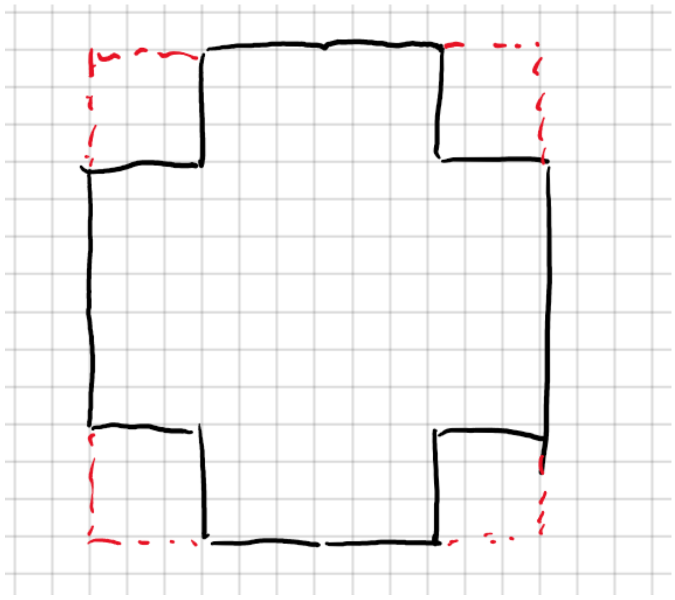
1. Change player array into a vector, so that the number of players can change dynamically, thus accounting for both 2 player and 4 player games (and maybe more in the future)
2. Increase colour enumeration to account for 2 more players. Since our piece's state only accounts for colours and not which player it belongs to, each colour is indirectly tied to a player.
3. Our Board class uses a Map with strings (coordinates) as keys and pointers to Square Objects to keep track of the shape of the chess board. Each square on the chess board will be represented by a key, value pair in the map. So as long as we have the coordinates, we can make any shape for the chess board. Since 4-handed chess also has coordinates, we

can represent each square on the 4-handed board with a key, value pair in our map, and thus we can represent the board in our program. Since the 4-handed chess board is in the shape of a cross, we would either use multiple nested for-loops (ii) or one big nested for-loop (ii) to fill the map with the key, value pairs

i. Multiple for loops: We would use 3 nested for-loops where each for-loop will initialize a section of the board as shown below. The benefits of initializing in this manner is that it is a lot easier to code and understand, but the code will not be concise/ compact.



ii. One big nested for-loop: We can treat the board like a big rectangle with squares cut off from the corner, so we can use 1 big nested for-loop to initialize the entire map. This map will make our code a lot shorter and concise, but it will be harder to code the algorithm for creating the kay, value pairs, which also makes the code harder to understand.



4. Add a direction field to the Piece class since valid moves will be calculated differently depending if the player is facing up/down vs left/right. This mainly only affects Pawns because pawns can only move in one direction.
5. Since 4-player chess most pieces all move the same way in normal chess and all the rules for which moves are considered legal are the same, therefore the only main difference that will affect valid moves is the shape of the board. The different board shape shouldn't matter because we use a map to keep track of the board's shape, so we just check if there is an existing key, value pair for a square's coordinate to see if the square exists or not. Everything else will be the same since our code already can already determine if a move is valid or not, then it should work for the new program since the rules are essentially identical.

Extra Credit Features

Our program does not use the new or delete keyword. As such, our program is free of memory leaks. We actually found this to be a lot less challenging than expected. In fact, we thought using raw pointers and manually allocating them in the heap would've been harder, since we'd have to remember to delete them later on. Smart pointers and the RAII principle made development a lot more straightforward since a lot of annoying work was automatically done for us.

Final Questions

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

While developing software in teams, our group realized how important it was to actively communicate with one another. During our meetings, we shared updates on our progress and what we were planning to work on next. The meetings also helped us clarify any misunderstandings early and share our inputs on how to solve any issues that arose. Active communication helped our team prevent any conflicts by building mutual understanding and involving everyone.

Moreover, we learned how powerful it was to be able to seek help and ask questions while developing as a group. We had numerous situations where a member was stuck trying to debug their code and they would ask another team member for help. In many scenarios, the other member knew how to fix the issue quickly because they were previously in a similar situation. This boosted productivity because the other members may have spent hours trying to solve the issue if they didn't request for help. Our group also found it helpful to ask questions when we were unsure about a certain concept because explanations from another team member were usually easier to understand than research from the internet. Therefore, asking questions and seeking help from other members increased our group's efficiency.

During the last 2 weeks, our group learned a lot about software development in teams. Overall, we learned about the importance of active communication and asking questions for groups that are coding.

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

The first noticeable issue we encountered was handling merge conflicts. Due to time constraints, we decided it would be best that we all worked on the same branch. However, as we often worked at the same time, we would get annoying merge conflicts when modifying the same files. Additionally, this made it really difficult to remove commits if they introduced new bugs, since our commits were so entangled. Therefore, it would've been much better to use pull requests. Not only would we have been able to clearly see the intentions and code updates made by other people, we'd be able to give code suggestions to each other during the development process. Furthermore, it keeps the team updated on the coding process. Often we would find ourselves having to read a huge chunk of new code after taking a day or two off from coding. This was especially annoying when the new code diverged from our original plan drafted in the UML. By creating pull requests, and squashing our commits, we would be able to easily keep up with commit history.

Additionally, following a simple project management tool would've been helpful, rather than stating what we wanted to work on in our group chat. By properly creating issues on a kanban board and delegating ourselves tasks there, we would've had less overlap in our work and a more organized way to track our progress.

Though better commit messages and pull requests help with communication, nothing beats good documentation. Unfortunately, to speed up development, we left documentation to the end, just in order to meet course requirements. However, it turned out that leaving out documentation slowed down our development process as a team. We had difficulty following along each other's code, especially if the variable names were not great. Functions like "getValidMoves" and "getAllowedMoves" sounded similar, yet they have different purposes. Creating additional comments here and there to explain the intent behind our code would've saved us a lot of time and effort trying to guess what others were doing vs what they were actually doing.

<u>Conclusion</u>

The project was a great learning experience to develop our coding and soft skills. During the past few weeks, we learned about the importance of active communication and asking questions when developing software as a team. We worked more efficiently when we had a common understanding and clearly clarified our goals. With the obstacles we encountered in this project, we learned how to overcome them and prevent similar situations from occurring again. In the future, we will use pull requests to avoid unnecessary merge conflicts and easily add suggestions to other people’s code. Furthermore, we realized how good documentation can improve a group’s understanding of the code, which boosts productivity. Overall, we learned many lessons about developing software in a team through this project.