Design Document: Chess Project

Group: Harsh Patel (h334pate), Jaimil Dalwadi (jdalwadi), Mayank Mehra (m4mehra)

Introduction

As a group, this was our first project working together and it gave us an opportunity to explore various aspects of software design while developing Chess, a game that we all love.

Overview

Planning the structure of a software project is essential before going on to write the actual code. We aimed to structure our project such that we can incorporate code reusability and abstractions. Through this overview, we will give a brief breakdown of the different functions and overall design of our project.

Board

ChessBoard is the main class that stores all the pieces and does all of the move validation for moves each player makes. As such, it is the central class or model that our program relies on to progress through the chess game and determine move logic. The board class also stores pointers to the players, pieces, and scoreboard and is the subject that our text observers and graphical observers rely on. This is a design flaw that we recognized after learning about the Model View Controller (MVC) design pattern, however, it was too late to refactor our program due to time constraints. It holds a vector of vectors of piece pointers which make up each tile on a board (nullptr representing blank tiles). It has all our major validation and input methods which are used by each player, the computer to generate valid moves, and main.cc.

Player -> NPC (Levels) and Human

The Player class was also abstract as it was the superclass with the computer (NPC) and human (Human) inheriting from it. It consisted of a Colour enum to distinguish the player, a reference to the chess board, and a purely virtual method which allows a player to make their move on the board when it is their turn. The NPC class also had a reference to a Level, which could be any of the four computer levels outlined in the project specification. Furthermore, the Level is an abstract class itself with the four specific levels inheriting its properties. Each level has a makeMove() method which takes in a chessBoard, the colour of the current player, and generates a specific move depending on the specification of that Level. In our case, the levels follow the required logic as mentioned in the project guidelines. Level 4 however has specific checks for checkmate, and weighted captures. In this case, it will only make captures if the piece that it captures is of a lower weight than the piece being moved. For each of the levels, the computer (which is a specific Level subclass) would utilize the logic written in the chess board to make a move. Specifically, the makeMove() function would call chess board methods to determine the moving piece, the current state of the board, and most importantly, all of the possible moves that the piece is capable of making by calling the targetedTiles() method in the chess board. The specific level would process the possible moves and determine which one is the most appropriate depending on the level, returning that move. In terms of the human player, the makeMove function would simply parse the string given by the input and generate a Move pointer for the board to use. By abstracting the Player into the two respective players, and then further separating the Levels at which the computer (NPC) can be at, we are easily able to make modifications to the Players and Levels.

Pieces

We created an abstract piece class that we used as a base for every piece on the board to inherit from. This superclass contains various functions that are relevant to each of the individual pieces. As such, each piece would then override these functions to implement their own logic to validate various moves the user inputs in. For example, our Pieces have a validMove method which determines the validity of a given move from one point on the chess board to another. Each individual Piece subclass (ex. King, Rook, etc) had its own implementations. This validMove function checks if a move is a valid move for the piece but does not check if that move causes checks on the board. As such, we called moves that returned true from pieces validMoves a "pseudo-valid move". A pseudo-valid move would only become a legal move if applying it didn't cause any checks to the user (this was handled inside the chessboard, applying a chain of responsibility style methodology). Inside the subclasses, we added additional functionality for the respective Piece. Specifically, a Rook and King must be able to engage in castling so we added additional functions and fields in those respective classes. The pieces played a crucial role in determining if a move that the user input is even possible to do for the piece they try to move. However, the piece class is never the one to apply the move. Movement is handled by the actual board class.

Applying Moves

Applying moves to move pieces around was handled by the Chessboard class itself. Initially, we had thought that after our piece returned true after calling ValidMove we could immediately apply the move onto the board. However, this logic was flawed as we forgot to take into account the fact that if a user makes a move on the board it can cause their own king to go into check. As such this caused us to rethink our implementation of how we would determine when to move a piece. We created a method called canApplyMove() in which we passed the responsibility of checking if the move was valid for each specific piece. Then we checked if applying the move resulted in a check on the same player. If it did not, it was a valid move and was applied. To resolve the check validation we had two ideas. One was to create undo functionality so that we could apply a move check if the board was in check and then undo if it was. We thought that this would cause a lot of issues for special move types like en passant or castling. So we just created a clone of the board, applied the move to the clone and then checked if the player on the clone was in check. This logic was easier for us to program and think through and as such was the one we choose to implement even though it is less efficient than implementing undo's.

Check

We hardcoded our implementation for the check validation. To see if a player was in check, we found the corresponding king on the board from the pieces vector, and checked through all diagonals, horizontals, verticals, and knight positions, to see if there was a piece of the opposite colour and the correct type that was targeting the king. This seemed like the most time-efficient method of implementing check validations and it worked out well for us.

Move Generators

Before we had implemented checkmate we knew we needed to be able to generate all possible moves for colour based on a given board state. This would be beneficial for both our level CPU implementations and checkmate. As such for each piece, we decided to add a targeted tiles function that generates all the possible moves a piece can make based on its row and column on the board by going through and calling valid moves on any move the piece can make. These targeted tiles functions would return a vector of moves that a piece could potentially make on the board. As such, the move generators play a crucial role in our checkmate, stalemate, and levels implementation. This is also a very important function that needs to be implemented properly for any new pieces added in the future. Whenever adding pieces, it saves a lot of time to simply generate these moves instead of adding specific validations in checkmate and others.

Checkmate and Stalemate

After the move generators were done the logic for both checkmate and stalemate was simplified. As all we had to do was generate all possible moves that a colour can make on a board state. From these possible moves, we find all the legal moves that don't result in the user being in check themselves. That is what was handled by our getLegalMoves method which returns a vector of all possible moves a colour can make without being in check based on the current board state. As such checkmate was as simple as checking if the enemy king was in check and if it was calling getLegalMoves on the enemy colour returned a vector of size 0. If so, then a checkmate occurred. Similarly, if the enemy is not in check and getLegalMoves returns a vector of size 0 then a stalemate has occurred.

Reflection on Plan of Attack

Our initial Plan of Attack was organized into three phases with respective timelines for each and tasks dedicated to each group member. The three phases were; setup, testing and submission, and lastly additional features. As soon as we started the project, we noticed that there was a lot of overlap between who worked on which task, and by the end, we had not adhered to the tasks we were assigned. While it may seem poor, this was instead helpful as it allowed us to cooperate with each other and tackle tasks that were better suited for each of us. Eventually, this increased our efficiency individually and as a team. Furthermore, we aimed to follow the hard deadlines for the three different phases but soon realized that we were unable to reach phase three due to the challenges encountered in implementing, debugging, refactoring and testing our code in the previous two pages. However, creating a schedule allowed us to follow a plan and facilitate the progression of our progress as a whole. The original time allocated for phase 3 allowed us some buffer time to fix things last minute and was a really helpful scheduling aspect.

Updated UML

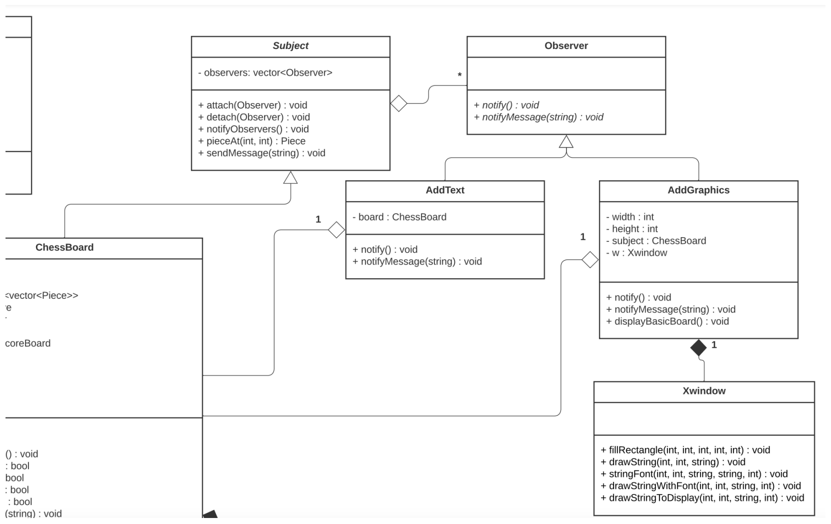
The initial and final UML for our project was mostly similar in terms of structure and organization but different in terms of the logic flow between classes and the implementation details of each of the classes. After comparing the two, we recognized that we had clearly underestimated the number of functions we would require to develop the game. However, this was expected because we encountered various edge cases and new scenarios that required us to create new methods. Furthermore, we noticed that we often rewrote the same code for the move and check validity, hence we constructed separate functions which carried out one single task, aiding in code reusability and clean design. For example, we added the canApplyMove() function which handles all of the logic related to making a move, calling specific functions like a piece's validMove(), the boards checkIfInCheck(), canCapture() and such. Additionally, we needed functionality for starting and ending the game, which was carried out by new functions such as initBasicBoard(), resetBoard() and more. Furthermore, in terms of the logic of the program, we completely changed our move validation, check validation and checkmate validation, resulting in various clone functions such as SoftClone() for the chess board, and clonePiece() for each of the respective pieces. These changes required various controller methods to be added in the chessboard class, many of which were discussed above. In terms of relationships among classes, the interactions among classes remained the same. Initially, we had expected the Pieces to have a pointer to the chessBoard as a protected field, indicating an aggregation relationship, but we soon realized that was unnecessary as the Pieces do not require the entire chessboard object, but simply the vector of pieces. Furthermore, we maintained and implemented the design patterns that we had on our initial UML diagram. Overall, we had planned the structure of our program fairly well, as exemplified by the limited changes in design between the initial and final UML. However, we had numerous changes in the logical flow of the game, which accounted for the updates in the final version, as mentioned above.

Design

Design Methods Used

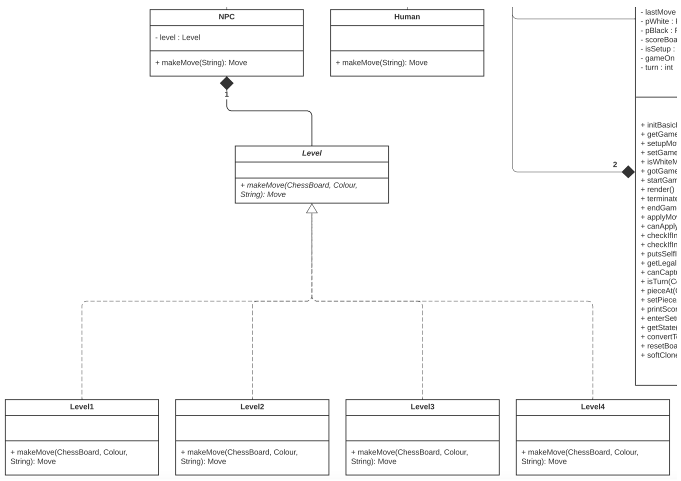
1. *Observer Pattern*

One of the main design patterns we used was the observer pattern. By creating our chessboard class as a child class of the subject class provided in lectures, we abstracted the need to handle output streams. The observer class has multiple output handlers such as a text-based observer and a graphics-based observer to display the game in the terminal and on an XWindow respectively.


Subject
- observers: vector<Observer>
+ attach(Observer) : void
+ detach(Observer) : void
+ notifyObservers() : void
+ pieceAt(int, int) : Piece
+ sendMessage(string) : void
Observer
+ notify() : void
+ notifyMessage(string) : void
*
AddText
- board : ChessBoard
+ notify() : void
+ notifyMessage(string) : void
1
AddGraphics
- width : int
- height : int
- subject : ChessBoard
- w : Xwindow
+ notify() : void
+ notifyMessage(string) : void
+ displayBasicBoard() : void
1
1
Xwindow
+ fillRectangle(int, int, int, int, int) : void
+ drawString(int, int, string) : void
+ stringFont(int, int, string, string, int) : void
+ drawStringWithFont(int, int, string, int) : void
+ drawStringToDisplay(int, int, string, int) : void
ChessBoard
<vector<Piece>>
coreBoard
() : void
: bool
bool
: bool
: bool
(string) : void

2. *Strategy Pattern*

If there is a need to add multiple ways to perform the same task, a strategy pattern is useful. The NPC is supposed to generate a move and so, each level is created to generate better or worse moves. The strategy pattern was implemented to accommodate different types of computer-level moves in the game. It allows us to create new algorithms for generating better moves easily and then abstracts it to the NPC class which is a type of player in the game. All the client then has to do is set the level and everything else is handled.



3. *Chain of Responsibility Pattern*

Applying moves is a very tedious task to program when creating chess. We had to check if the move was valid on multiple levels. We leveraged this idea in our validation methods, instead of implementing it through entire classes. So our workflow has a chain of responsibility for validating moves. The player class checks if the move input is valid from the command line. It then generates a move and passes it to the ChessBoard. The ChessBoard passes this down to each individual piece to check if it is a valid move based on the type of piece that is being moved. If it is valid, it returns to the board and the ChessBoard checks if it is a valid move on the board relative to other pieces. It checks if this move puts the player in check if it cancels a check that is currently in place and more. This chain of responsibility pattern is very helpful as it allows us to easily add new piece validations and add new types of moves because everything else in the board will likely remain the same and it adds a better structure to the flow of the game.

4. *Flawed MVC*

We attempted to apply the MVC pattern but it is currently flawed. The ChessBoard acts as a controller and model, whereas the terminal and graphic act as a view generated by the text and graphics observers. This mixing of handling input and also making changes to the data is the flaw of the MVC pattern we implemented. If we had time, we would make the ChessBoard the model object that would strictly handle the data changes, and add a controller which would be called in main to handle input parsing. Our method currently still benefits us as it allows us to keep the main function very abstract. However, by abstracting the parsing of input to the controller, we can make it much easier to add different types of inputs (ie; graphics-based input handling).

Design Improvements and Future Implementations

If we were to go back and add more design patterns or make enhancements, we would improve the MVC design, reduce coupling, implement RAII, and possibly create better copy constructors.

1. As mentioned earlier, we would abstract the controller part in MVC to it's own class so we can allow different types of input instead of mixing the handling of input with chessboard data changes. This would allow the main function to only see the controller method and not directly access the chessboard in a game. This would improve security, improve resilience to change and more.
2. The chain of responsibility pattern we implemented is a great way to handle move validation, however, there is still a lot of coupling. For certain validations, different parts of the board require other parts (ie; each piece requires a pointer to other pieces when checking for validation). We would create a more robust way to handle moves, by maybe changing the way each piece validates the moves or creating more functions to check for obstructions.
3. We would try to implement RAII to handle memory much better. Currently, we use classic deletes and had a tough time handling memory leaks. We didn't have time to refactor to RAII as we learned it after we created the game. We tried implementing RAII, but it was too late to fully validate and we hit many seg faults. So we decided to use destructors and raw pointers.
4. To perform the check and checkmate validations, we make a copy of the board and apply the move to see if it is valid there. It would be better to perhaps implement an undo move function that just undo the move if it is invalid. Copies led us to more memory leaks and were tedious to deal with.
5. The ability to castle and a flag for whether castling is allowed should only be included in the King and Rook, instead of there being a canCastle() and setCastle() mutator and accessor methods in all of the pieces. This was a last minute resort we used to allow castling and was not properly implemented as these methods are unnecessary in all pieces. A way this could have been fixed is typecasting.

<u>Resilience to Change</u>

We aimed to incorporate numerous object-oriented programming principles to create flexible and reusable modules (classes) in our program. Initially, we had abstract classes such as the Piece, Player, and Level classes. This allowed us to add and remove a piece (or player or level), and make modifications to existing individual pieces without altering the functionality of any other piece. This is crucial as each piece has different logic associated with movement and some even have unique moves such as a pawn en passant and a King castle. By abstracting the pieces, and similarly with the different computer Levels and the Players, we can prevent copying the same code in multiple areas, and distinguish between the respective subclasses.

Cohesion and Coupling

We aimed to design our program such that it maximized cohesion while having low coupling. Our current program implements low coupling for most parts, but not all as we do not have any modules (classes) altering each other's implementations. Furthermore, we do not have any friend classes which give the class access to the private/protected fields of another class and incorporate accessor and mutator methods in our prominent classes such as the Move, ChessBoard, Player and Piece classes. However, we do have tight coupling between the ChessBoard and the Player as both classes have references to each other. We would change the overall logical flow and design of our program to remove this if we were to make changes in the future. Furthermore, as a result of the chessboard being the controller, it is coupled with quite a few other classes individually, but can be modified to remove such interactions in the future. In terms of cohesion, we believe we achieved high cohesion in our design as the program is easily organized, and readable and contains modules that interact to perform one function. For example, if we need to change the functionality of a piece or add special features to a piece, we can make changes to the respective piece class with minimal impact to the rest of the program. Likewise, the Player, and Level classes work in a similar manner. However, our ChessBoard class can be further modified to have a higher level of cohesion. Currently, it handles the move logistics, starts the game and endgame handling, checks validation and much more. This can be simplified such that it works as a whole for one general function such that there is very little impact to the functionality of the game if it were to be altered and modified in the future. Overall, for the most part, our program employs low coupling and high cohesion, but it can be vastly improved and would require a new design or numerous modifications to the current one.

Accommodating Change

We kept in mind the ability to accommodate change by being able to add new pieces to the board, new commands and new rulesets. If we want to add new pieces to the board all we have to do is create a new child of the piece class and implement the respective functions like targetedTiles() and validMoves(). One difficult or important note for adding new pieces is that the targetedTiles() function logic must be robust and generate all possible moves for the new piece based on its desired movement logic. This is great because if we catch errors in the program, we can track them down to this function. If the piece has a new special Movetype enum aside from the ones already part of chess (ie; en passant, castling). This would need a bit more care as then this new move also has to have additional logic added to the chessboard class to actually apply the special move. New commands are also not difficult to add as this functionality would be added in our command loop in main.cc. Based on the command it would have been more beneficial to have a controller as then we would just add the command logic to that. But since our chessboard acts as both a controller and model it would need to be implemented there. As mentioned before this is not ideal as it would be more streamlined to have a dedicated controller class.

Difficulties arise if we want to change the board size or board layout to something like 4-way chess. Currently, much of the board dimension logic is hard coded for an 8x8 board. As such this would require a refactor on the code making board dimension and layout a difficult change to accommodate in our current implementation. This is where our design is flawed and not so resilient to change. The reason we were unable to make this more robust was due to the lack of time we had left and sprinting to finish the game.

Additional Features:

There were several additional features we planned on creating in our initial plan of attack, however, we did not have time to implement these. We added a bash script to make our lives easier and thought it was noteworthy. We created a bash script to create tests based on FEN strings which made our life a lot easier and has been submitted as “createboardtest.sh”.

We didn’t have enough time to implement additional features and realized that we had not allocated enough time for leaks and error handling in our initial plan of attack. Something we learned from the process. We tried implementing smart pointers and RAII as well, but it took too long to refactor and caused many errors. If we had started off using smart pointers, we would have saved a lot of time, too.

Answers to Questions

1) Chess programs usually come with a book of standard opening move sequences, which list accepted opening moves and responses to opponents’ moves, for the first dozen or so moves of the game. Although you are not required to support this, discuss how you would implement a book of standard openings if required.

After implementing our project we still believe that our initial idea for implementing a book of standard openings still applies. The book of standard opening move sequences can be implemented by using the map (standard template library) where each element would contain a key, which in this case stores the current state of the board or a 2d vector of pieces. The corresponding value for each of these keys would be an array of possible responding moves associated with the current board state.

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

Currently, in our final UML, we have a Move class and a reference to the last move in the Chess Board class. This is to handle the case of en passant as well as other undo functionalities, however, it would only support a single instance of undo. In order to facilitate an unlimited number of undos, we would retain our proposed method before creating the program, which was to store a history of moves in a vector (standard template library). Furthermore, we can create an undoMove() method in the chess board class that can take in an integer specifying the number of moves to undo and execute that. It would pop off moves from the move history vector, and change the 2D vector of pieces (the actual board) as it applies the undo of the last move. There would be additional logic and validation logic which checks for the size of the move history vector and also additional memory handling to ensure that pointers to pieces on the board are changed accurately without any leaks.

3) Variations of chess abound. For example, four-handed chess is a variant that is played by four players. Outline the changes that would be necessary to make your program into a four-handed chess game.

The process to implement this version of chess remains the same as our previous discussion in the plan of attack. We would require more space on the chess board and instead of having 64 tiles, it would have 96 tiles as seen below. We would have to add two more player instances/classes, two more colours, and additional logic to check the validity of moves (ie; making sure moves are in the correct boundary). This is due to the addition of two more players and changes to the size of the board (since there are 4 corners (3x3 squares) cut off on the whole board). Furthermore, the input handler (command interpreter) would have to account for the two new players when starting a new game with the “game” command and setting up the game with “setup”. When implementing the chess board, we can expand the 2D vector from 8x8 to 14x14, but set pointers to corner tiles to be “nullptr” and add logic to ensure that they are not accessed. Since our current implementation is hardcoded for an 8x8 board it would require us to refactor a lot of the logic to account for the changed size in the board. There would also have to be a mode where the players can play in a team (2 v 2), in addition to the option of playing individually. The final checks for checkmate and end-game results would also have to be altered to account for the two new players. For example, if one player runs out of possible legal moves they can make then they must be banned from making moves and their pieces should be discarded from the board.

<u>Bonus Features</u>

We didn’t have enough time to test out our extra features as such we decided not to include them in our final product. Things that we were looking to include in was creating hint functionality that would leverage our getLegalMoves function to help the user decide on a move. Having various starting positions for the user to choose from for example having both white and black start with an army of pawns or having default board setups that the user could start with instead of having to call setup every time and setting pieces themselves. This would be an additional command that would leverage our setPieceAt function.

<u>Final Questions</u>

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

This project was an amazing opportunity to learn about the whole project design flow from conceptualizing the design and architecture of a program to actually developing features alongside a team. Overall as a team, we learned a lot through all the merge conflicts and design meetings that we had throughout the development cycle. One of the main things we got to experience was utilizing version control software like GitHub to efficiently work together as a team. For example, we created branches for each new feature. We also divided up the game features to reduce the number of possible merge conflicts down the line. This is a mindset that we never experience when working on assignments or projects alone in other courses. One thing we found beneficial was getting together and breaking down the project into smaller features that we then could tackle individually in our own separate branches. This allowed us to work simultaneously together while reducing merge conflicts and being efficient. Thinking like a project manager was very important in dividing up the project and being able to finish on time.

Another thing we found beneficial was Pair Programming for key features in the game. Functions like targeted tiles were very important and crucial as such we decided to tackle them in groups together to make sure they were done right. Two sets of eyes are better at spotting bugs in logic than one set. This also encouraged a lot of discussion in the group about how we could implement logic which was a crucial part of working as a team. It saved time that would have been spent later on individually on debugging.

Good communication and writing easy-to-understand clean code was also something that we learned along the way. Making sure that variables or function names were descriptive enough for teammates to easily understand was very important. Furthermore, commenting on the logic that might be harder to read quickly was also an important strategy.

Great software programs require readable code bases, frequent meetings, maintenance, and efficient teamwork. This project really showed us what the real world looks like when working as a software developer. Teamwork is crucial and a concept that is not really focused on in most courses.

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

We discussed the several design changes we would have made if we were to start over earlier. We definitely would have allocated more time for debugging and underestimated how long such tasks would take. RAII was a big part of this project that we hoped to have implemented from the start as we also spent a lot of time fixing memory leaks which could have been spent on improving our MVC design or also adding additional features. One thing we made a mistake with was having RAII as a second thought in our initial design. As such if we were to start fresh RAII was definitely something we would take into account for our design. For example, making sure that our pieces were stored as unique pointers in the 2d vector as they should solely be owned by the chessboard. These types of additions would have made our memory management significantly easier when it came time to debug. Furthermore, having a more explicit MVC design so that the chessboard class didn't have as much responsibility as it currently does on the board. The chessboard should only function as a model that applies, calculates, and validates moves. This design pattern would have better streamlined the program and made it easier to accommodate more changes to our program in the future. Overall, we believe spending a lot more time on defining the correct architecture, and design pattern used, and thinking about the RAII idiom would have benefited us in the long term. Taking as long as possible to flesh out a well thought-out design before programming would result in less memory or logic-based issues when implementing features in the game.

Conclusion

Overall, this project was very memorable to complete and we learned a lot that can actually be applied to the real world as opposed to other courses.