CS 246 Final Project Chess - Design

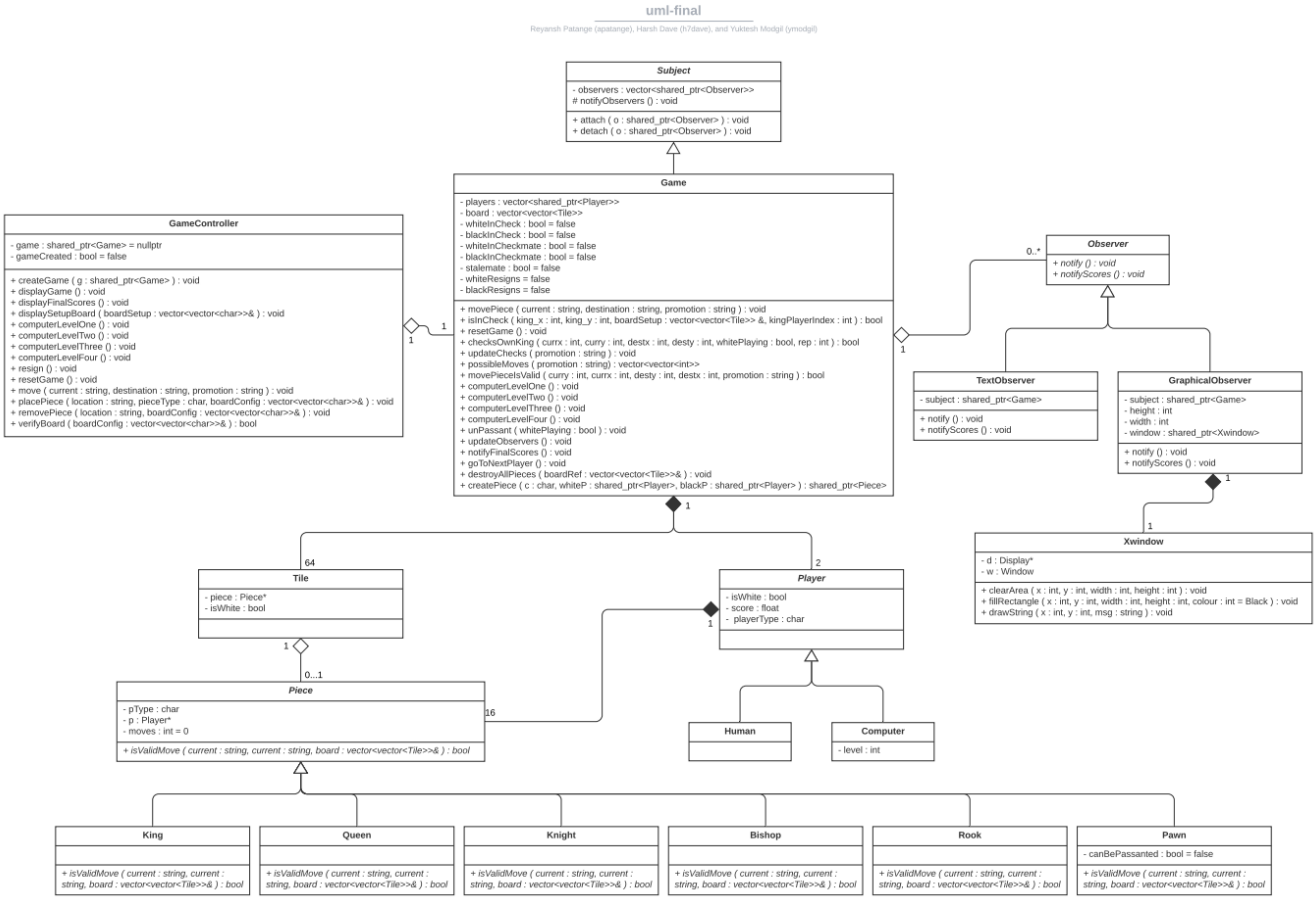
Yuktesh Modgil (ymodgil), Reyansh Patange (apatange), Harsh Dave (h7dave)

Introduction

For the CS 246 Final Project, our group has chosen to recreate the game of Chess using C++ with OOP principles to demonstrate the knowledge we have gained from this course. This design document will provide an overview of all aspects of our project, including how, at a high level, they were implemented.

Overview

We have chosen to use the Model-View-Controller (MVC) architecture along with the Observer design pattern to allow us to separate the logic of the game from the presentation through the User Interfaces. In our project, the Model is the Game class, the View is the Observer class with its Concrete Observers (TextObserver and GraphicalObserver), and the Controller is the GameController class. Looking at our project from an Observer design pattern perspective, the Subject class is the Abstract Subject, Observer class is the Abstract Observer, the Game class is the ConcreteSubject, and the TextObserver and GraphicalObserver are the ConcreteObservers. The Final UML can be found below or in the submitted *uml-final.pdf* file.

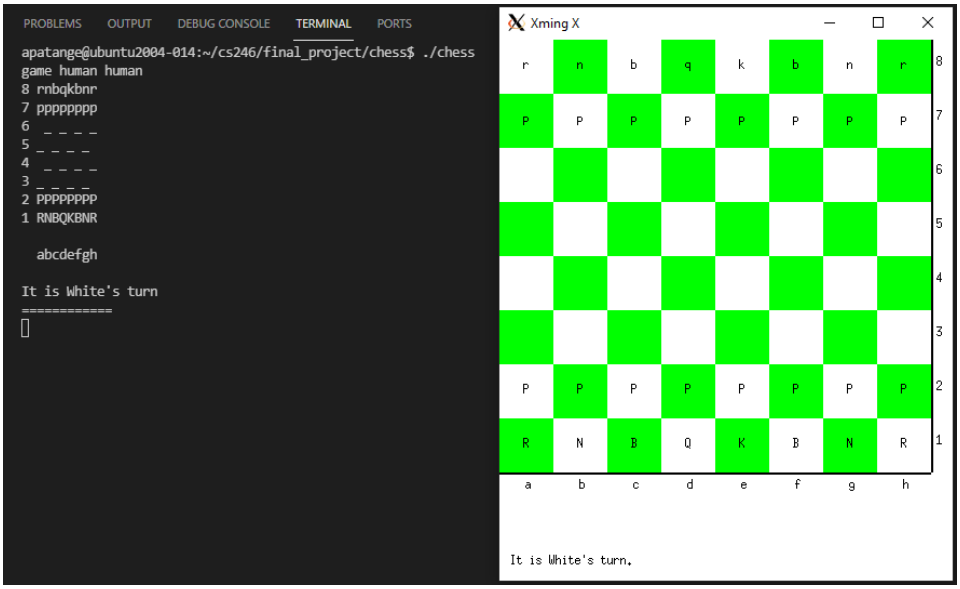


As seen in our UML, the Game class is central to our program's design. The Game class inherits from the abstract Subject class, it owns the Tile and Player classes, and it has a GameController and an Observer. The Game class stores the chessboard in a private field called *board*, which contains all of the Tiles of the chessboard in a 2D vector of Tiles (vector of vectors of Tiles), and it keeps track of the black and white players using a vector of Players. Furthermore, the Game class contains many important methods such as movePiece, movePieceIsValid, and isInCheck; these handle moving pieces, validating those moves, and checking if it's in check.

The game class owns 64 Tiles (since there are 64 squares on a chessboard) and the purpose of the tile is to store a pointer to a piece, which may be nullptr if there is no piece currently on the tile. The Piece class is an abstract base class for the 6 different types of game pieces in chess (King, Queen, Knight, Bishop, Rook, and Pawn). Piece stores a char corresponding to the type of piece it is holding (k,q,n,b,r,p) and also a pointer to the player who owns the given piece. The King, Queen, Knight, Bishop, Rook, and Pawn classes all override the isValidMove function in the Piece class according to their unique move set. The Player class keeps track of the player's colour (black/white), their score, and the player's type (human/computer).

The GameController class is important because it can interpret user input and accordingly access the model to produce the desired effect. GameController has a shared pointer to a game, and GameController can create a game using the createGame function. It can also handle resigning and resetting through the resign and resetGame functions respectively. This class also has the displayGame and displayFinalScores methods to display the board and final score to the user.

The Observer class is an abstract base class that TextObserver and GraphicalObserver (ConcreteObservers) inherit from. The text observer displays the chessboard in the terminal when notify is called, whereas the GraphicalObserver class owns an Xwindow object and uses it to draw the chessboard graphically when its notify function is called. Shown below are the Text and Graphical Observers.



Design

Our group opted to implement the Model-View-Controller (MVC) architecture since it allows for separating the logic of the game, the UI that displays the state of the game, and the command-line interpreter that the user utilizes to play the game. This has allowed us to significantly reduce coupling in our code, as the purpose of each module is clearly defined and there are minimal dependencies between our modules. Also, we have ensured that there is high cohesion between all of our modules by organizing our program into classes that contain very closely related code which serves a single shared purpose. For example, the gamecontroller.cc file contains the implementation of all methods that interpret user input and make the corresponding changes in our game. This makes it easy to add new commands because we only need to modify very few files.

In addition to the MVC architecture, we have also implemented the Observer design pattern; this allows us to display the state of the game using the Text and Graphical Observers of the game. As mentioned previously, the Game is a ConcreteSubject, and it inherits from the abstract class named Subject. The Game class has the Observer abstract class and TextObserver and GraphicalObserver are ConcreteObservers that inherit from Observer. This design pattern worked quite well because it was easy to attach or detach observers as needed, and the notifyObservers( ) function made it quite straightforward to have both observers display the board simultaneously.

Compared to DD1, our plan changed in a few ways; this was to be expected, and we were able to adapt as the program continued to come together. We removed the GameBoard class, and instead stored the chessboard as a private field within the Game class itself. Initially, we had move functions for each different piece, but we ended up removing that from Piece since we realized that we only needed to validate a move using a piece's isValidMove function before allowing Game to handle moving the piece.

Resilience to Change

The game of chess described by the program specification can be changed for the purposes of refactoring, adding new features, or improving the way users interact with our program. We have made our program resilient to change by making as many of our methods as general as possible and modular. For most parts of our code, we are not directly accessing internal fields of classes; instead, we have created methods for operations such as isInCheck which allows us to pass the coordinates of the king (or can be any other piece if we want to check if that piece is in danger), and a reference to a board.

An example of a situation where we can easily modify our program to accommodate new features is a change in the input syntax. For this, the strategy that we followed was that the command interpreter should never be able to interact with our model, the Game, at any point, so all we had to do was create methods in our GameController class that handles input, and does the appropriate operations in Game, thus maintaining class invariants at all time. New methods are very easy to add for corresponding input commands and allow us to abstract Game details.

Additionally, we have designed our code to have low coupling which means that the purpose of each class has been made clear, and that class handles everything relating to itself with only members that belong to it. This reduces the amount of dependency there is between different classes. If a new feature

needs to be added, there are very few places where the code needs to be updated. For example, if we wanted to change the rules of chess and allow a Pawn to move any number of steps forward, we would only have to update the overridden isValidMove( ) method that belongs to Pawn that is inherited by all the concrete pieces, from the Piece abstract class.

Finally, another example is the change of creating different ways for allowing the user to view the game. Since we have employed the Observer design pattern, all of our observers can be updated using the same method within our model, notifyObservers( ). This allows us to effortlessly create more observers and implement them by just overriding the notify( ) virtual method that is declared in the Observer superclass. This allowed us to create the GraphicalObserver in less than an hour once the Observer class had been defined and we had the TextObserver working.

We put a lot of thought into making decisions wherever we came to a crossroads during development. It helped us to make better decisions that allow us to generalize our algorithms as much as possible, make our code reusable, and improve the readability of our code.

Answers to Questions

Question: 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.

Most openings are two moves - white's move and black's response - but some are as long as 5 (such as the Ruy López Opening). Considering this, a book of standard opening moves could be created with a vector of vectors of GameBoard pointers, where each element in the outer vector would be the unique opening moves and each inner vector would be the sequence of black and white moves that form or represent the opening sequence. This would be shown as pieces on a GameBoard. These inner vectors would, of course, be of varying length depending on the number of moves in each opening as mentioned above. A possible way to support this during a chess game is that for the first N moves, where N represents the max length of opening moves that exist in the book of standard openings, the chessboard could compare the current board's position to the positions of each element in the opening book's (AKA vector's) position and instead of iterating over all the moves in each opening in the vector, we would only have to check the gameboard at the index representing the current number of moves so far - if it exists of course.

For time efficiency's sake, if the number of moves the players have made in-game is currently greater than any move's max number of moves or if the number of moves is equal but the pieces differ at that point in time, the corresponding opening move (represented by a vector) could be removed from the outer vector representing the current possible opening moves. This way, the program doesn't need to unnecessarily iterate over and compute moves it already deemed irrelevant after each step. To finally output this to the user, so that the person playing could see the suggested opening moves, the chessboard could be colored in a way that shows where to move a specific piece. For example, considering the tileColor() method from the Tile class for each piece, the piece's current tile position could be changed to green and the recommended position's tile color could also be colored green to show the player where to

move which piece. This would be applicable for the GraphicalObserver, and something similar could be done with the TextObserver but instead of coloring tile's (which obviously wouldn't be possible with TextObservers), the current coordinate and recommended coordinates could be outputted for the player to see which piece is recommended to move according to the specific opening being considered in the book of standard openings.

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

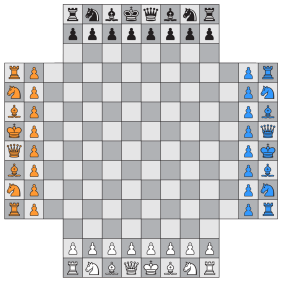
A straightforward and space-efficient solution to the first part of the question would simply be to store a single GameBoard object that creates a deep copy of the current setting of the board (before the player makes their move) in their corresponding Player object. This would allow either player to undo their past move by simply reverting the version of the GameBoard to what is stored in their corresponding Player object.

To give each player an unlimited number of undos is a bit different and will occupy more space. Since we need to remember the state of the GameBoard after every move made, a stack would be the ideal data structure for storing this. The last in first out (LIFO) design of a stack makes it only possible to undo the most recent move. Therefore, to maintain invariants of the game, we would pop moves from the stack until the desired game state that either player wants to revert the board to is encountered. Since every single move is stored in the stack, this would allow us to have an unlimited number of undos. This stack can be stored within the Game object and each item in the stack can be a GameState object. This GameState object would store a deep copy of the GameBoard at the corresponding previous move(s) similar to when simply undoing a player's last move, however, it would also store the corresponding player whose turn it was at that point (ex. in the form of a char). So while popping through the stack, we look for the player who wants to undo their last move. Of course, adding to the stack would also have to accommodate this change, where the GameState object created would also include who the current player moving is alongside the deep copy of the GameBoard.

Question: 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 the four-handed chess variant, a significant difference between traditional chess is the board. The board is now a 14x14 board with 3x3 cutouts from each of the corners. So this can be represented as a 14x14 2D vector. Of course, this means that pieces should not be allowed to access the indices which lie in the corners of the game board as these are not valid locations for pieces to land. This can be checked in the isValidMove() method from the Piece class. Another difference is, instead of the Game object storing 2 different players, it must now own 4 players which can all either be humans, or computers, we have stored the players in our current implementation in a vector, so adding more players is possible by scaling up the vector.

Furthermore, the TextObserver and the GraphicalObserver will have to be updated to now render a board with 4 different players, which looks similar to the image below. This will involve adding new colors aside from just the simple black and white colors offered in standard chess which can be controlled through the Player class, by storing the colour of the Player's pieces within the Player object itself.



https://en.wikipedia.org/wiki/Four-player_chess

For the TextObserver, we must also find a way of displaying the pieces of the 4 different players, we can no longer just use uppercase and lowercase letters as specified in the Chess game description for this project. One way, which may not be the most user-friendly, is to prefix the letter of the game piece, with a numerical ID assigned to each player, ranging from 1 to 4.

Next, there will now be 4 players whose turns are rotating in order, instead of switching back and forth between 2 players. This rotation of players can be stored as a vector containing each of the players and an integer that keeps track of whose turn it is in the vector.

Pawns can also reach left and right ends, instead of just across to change pieces. So now instead of just checking if it has reached a coordinate that is at the last row of the opposite side, we will also have to check if the pawn has reached a coordinate on the first or last column. This can be checked within the move method of the pawn, which after moving, checks if the aforementioned conditions have been met.

Extra Credit Features

The first notable extra credit feature we implemented was smart pointers. Though we actually added them at the end, we were still able to get them functioning in time. We definitely learned a lot about these types of pointers through this project and ultimately found it very convenient to not have to delete our pointers every time with smart pointers unlike with the raw pointers.

The next extra credit feature we added was the 50 move stalemate. We found that when putting a computer against another computer, oftentimes, one (or both) of the computer's pieces would be reduced to just a king and from there, the game would go on forever. To fix this, we added the 50 move stalemate draw which checks if there is only one piece left on the board (which would obviously be the king - ensured by board movement logic) and then increment the number of moves for that piece each time a move is made. If the number of moves hits 50, the game ends in a stalemate.

The next feature we thought would be very convenient for the user was to ask the user if they would like to play again upon the completion of a game instead of waiting for EOF input. They would be prompted to simply type *"y"* or *"n"* to restart the game or end it respectively.

The next extra credit feature we added was assigning points to chess pieces which would be used by computer4 to make its move. Since we wanted a level 4 computer to make more optimal moves than the levels below it, we added a feature where it would consider the points of its pieces in danger, and the

pieces it can attack and compare the maximum points of the piece gain or loss of these two types of moves to determine an optimal move that would give the most points.

King	Queen	Rook	Knight	Bishop	Pawn
1000 pts	9 pts	5 pts	3 pts	3 pts	1 pt

To further differentiate between the various computer levels and their corresponding difficulties, we decided to have pawn promotion reflect that by promoting level 1 pawns to bishops, level 2 pawns to rooks, and level 3 & level 4 pawns to queens.

Lastly, we also added a feature where level4 would be given priorities regarding its moves instead of selecting a random preferred valid move like levels 2 and 3 or an entirely random valid move like in level 1. The order of the priority we decided to go with is: checkmate, check opponent without getting killed, and capture or avoid capture depending on piece points as mentioned earlier. Though we would have liked to add more features, we believe that all these features add value to the game for increased user experience or realism.

Final Questions

(a) What lessons did this project teach you about developing software in teams? If you worked alone, what lessons did you learn about writing large programs?

This project has provided us with countless opportunities to grow our skills as developers using the most widely used programming paradigm in the world. Working in a team is a very different situation than working on code that is entirely your own, and it comes with its own set of challenges. In a group, everyone puts their best foot forward in terms of their skills and what they can contribute, so it is much easier to think about problems, and have people to bounce ideas off of.

For this project, our group used Git as our version control system for our code, and we hosted our repository on GitLab. This was an excellent way of learning how to manage our code and allowing multiple people to work on the project simultaneously, thus increasing our productivity as a team. Another important skill that we developed during this project is the ability to effectively communicate our thoughts and ideas. This is an integral part of the software development process as it is much easier to share ideas verbally rather than expecting everyone to understand what the code does by looking at it. We ended up having a large program, however, it remained relatively easy to navigate and understand the logic because of effective communication and design.

Good documentation is also a form of communication and when done right, can save a lot of time debugging and understanding code. In this project, we learned how to write good comments, and code that is self-documenting, so it saved us many hours in debugging time which was better used in quickly implementing features to satisfy the program's specification.

Furthermore, we had many experiences when we realized that our code was not meeting all the program specifications and criteria, so we had to learn how to adapt to such situations. We had

circumstances where we had to rewrite entire functions because halfway through, we realized that some assumptions we made were false or that there was an easier or cleaner solution that lets us accomplish the same goal. A good example of this was when we were trying to implement setup mode, which allows the user to create a custom board. Our Game constructor was already constructing the game in a way that did not allow users to modify the board beforehand, so we had to completely change that up. However, we learned from this oversight and started to pay more attention to what the implications of our current design decisions could be for future development and maintenance.

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

If we had the chance to start over, we would have first started by thinking of a clean, space-efficient and time-efficient algorithm for checking for checks and for checkmates. These two features are very important; otherwise, the game of chess would be incomplete and unplayable. If we had started off by thinking about these algorithms, we could have thought of an efficient data structure that would better allow us to check for checks. What we ended up doing was passing in a reference to the board to our isInCheck method so that it could also test different scenarios before actually allowing a player to make the move. This proved to be a very expensive task, as we would have to copy the current board into a new one, simulate all the possible moves, and test if the king is in check or not. We decided to also use this function for checking for checkmates and this is an even more expensive task as it has to verify that there are inescapable threats before deciding whether it is a checkmate or not.

Furthermore, we would have started writing the program by only using smart pointers, to begin with. We initially started with regular heap-allocated pointers with the idea that we would be able to change it to smart pointers once we had the functionality working. And although we did get smart pointers working, the codebase became quite large, at which point it became very tedious to change every instance of a normal pointer to a smart pointer. Doing this from the beginning would have helped us save a lot of time, which held us back for around a day from actual development on other features.

We have also found that using the GraphicalObserver by redisplaying the whole board becomes very slow. So, in retrospect, it also would have benefited us to think about how to only redraw affected portions of the board and not the whole thing. This alteration would have made using the GraphicalObserver faster and more user-friendly. Another thing we would have changed about the GraphicalObserver is to use a graphics library with more capabilities such as drawing pieces perhaps from images of the pieces.

Conclusion

In conclusion, this project was an important learning experience that tied together with the valuable topics that we learned in class and let us apply Object-Oriented principles in a real scenario. Ultimately, we were able to create a Chess game that followed all of the required specifications and we also had the opportunity to implement some of the bonus features mentioned previously. Using OOP techniques learned in this class, we were able to speed up the development process significantly and stay organized as our codebase grew larger and larger.