Introduction

Over the course of 2 weeks, the group attacked the “Chess” problem as their final project for the course CS246. We have built an amazing game of chess through lots of collaboration and communication. It was truly an eye opening experience for us, especially the new concept of MVC design pattern used in this project. It was painful at times, especially with final exams for other courses going on, but we had fun building the project, and the 2 weeks of bonding has made our friendship stronger than ever. Now we will be showing you the details of the project, hope you enjoy :)

Overview

After some careful consideration, we have decided to split this problem up into 5 different parts.

The first part, which is the heart of the program, is main.cc. Main manages most of the interactions the program does with the user, including taking in commands, giving out error messages to users, etc. Some of the commands main is able to handle are “game player1 player2”, “resign”, “move” and “setup”, as required in the project guideline. We tried to make the program as interactive as possible by giving specific error messages, asking users for their names, and if they want to play again after each round has ended. Hence for the best user experience of our program, it is recommended to test with command line inputs, rather than .in test files. Optimizations were done at times to avoid repeating code.

The second part of the project is our Game class, it is the body of our program. In Game, we store our gameBoard, which is an 8 x 8 array with the type Square, a class we have defined. We also have game-winning decision variables including inCheck, checkmate and stalemate in game, these variables are critical in deciding if a game should continue or end. To verify en passant, we have variables including moveNum and twoMovePawn[3], which records a pawn's movement and at which step it moved. Next we have whiteKingLoc, blackKingLoc, whiteKingSetup and blackKingSetup, these variables are used to keep track of where the king of each colour is located. This is important because we need to be constantly checking if one side is in check. The whiteKingLoc and blackKingLoc variables are of type String, this is the best choice as there will always only be 1 black king and 1 white king present on the board. The whiteKingSetup and blackKingSetup are vectors, they are mainly used while the user is configuring the board. The reason why we chose vector as the type is because even though the user cannot leave setup mode having more than 1 king of each colour, they are allowed to put as many kings as they wish, and remove them before they exit setup mode, leaving only 1 king of each colour. A char variable named “turn” is also stored in game, where we keep track of which colour’s turn it is, to prevent the user from moving the same colour continuously.

Now we will explain the functions in class Game. There are 2 private functions, checkCheckmate and checkStalemate.. A default constructor is created for the class Game where we initialize the variables at the beginning of the program. Then we have a few getter functions including getCheck, getStalemate, getCheckmate, getSquare, getKingLoc, getTurn, getMoveNum and getTwoMove. These functions are used to get the private fields of the class. Next important function we have is move and movePawnPromo, these 2 functions are called respectively when the program receives command “move” for a normal move and pawn promotion. They are in charge of checking boundaries, making sure that it is a valid move and is following the rules, not putting the king in check, and some other invalid cases. genComputerMove is also a function we have in Game, it is used to generate the moves made by the computer players. It intakes the colour of the computer’s side and the level of smartness, then generates a move based on the level of difficulty stated in the project requirements. newGameParam in the class Game is used to reset some of the parameters for the class Game when the user wants to start a new game. Some of the variables being reset include “inCheck”, “checkmate”, “stalemate” and “moveNum”. initiliazeBoard, clearPieces and defaultBoard are functions used to set up the board or clear the board if the user decides to enter setup mode. displayText, displayGraphics, showBoard, endDisplay are functions needed by the MVC for views, as the name suggests. Then we have a score related function namely printScore, which prints out the current score of the 2 players. Furthermore, we have 4 setup related functions: placePiece, removePiece, changeTurn and verifyBoard. They are called respectively when the 4 commands +, -, = and “done” are received. Lastly, we have a endOfTurnDisplay function where we displays “colour’s turn” and sometimes “colour’s in check!” If the player is a human, their name would also be displayed. Moreover, if stalemate / checkmate was reached by the last move, the function will display the correct messages respectively.

The third part of the project is the Player class, where we keep track of the type of the player (human or computer), their score that they have accumulated over games, level if the player is computer, and some other fields that are personalized for each player. In the class Player, we have private fields colour, score, type, level and name, with their getter functions getScore, getColour, getType, getLevel and getName. For setter functions, we have setName to be used at the beginning of the program, setScore which is used to update the score at the end of each round, and setTypeToNull which is used to reset the game if requested.

The fourth part of the project is squares and all the pieces. Squares are the cells of the game board, which is the location we place the pieces. To us, square is the connection between the game board and the chess pieces. A Square has a pointer to a piece so we can point to different pieces throughout the game. It also stores information about the location of itself within the board, so the observer can find the specific square that needs to be updated. Square is also inheriting from Subject to attach the observers to the board. We created a default constructor to initialize a default board, and created a copy constructor and a copy assignment function to copy the chess board if needed. There are two ways to replace a piece that a square is pointing at: replacePiece and swapPiece. swapPiece notifies the observers while replacePiece does not. There are getter and setter functions for the row and column of the square. There’s also a getter function for the piece, and you can think of replacePiece or swapPiece as the setter functions of the piece field.

All 6 types of chess pieces, King, Queen, Rook, Knight, Pawn, Bishop, represented as classes in this program. They inherit from a general class called Piece. In the base class Piece, it stores the piece’s colour and its name. There are many functions that are overridden in its subclasses, such as getCastling, setFalseMove, setCastling, checkValid, and availableMoves. For these functions, the Piece class returns a general return type: bool -> false, int -> 0, and vector -> and empty vector. Since the Piece class stores the colour and the name of a piece, there are getter functions for those: getColour, and getName. The most important function of the Piece class is checkCapture. It looks in the 8 directions and determines if there are any pieces that could potentially capture the piece located in that space. Since we need to know the location of other pieces, we sent the current game board into the function. Only King and Rook classes have a field called castling, to check if the move called castling is legal during the game. To change or get this field, we overrode appropriate setter and getter functions. Only the Pawn class has a field called firstMove, and this is to check whether or not the pawn can move two squares forward in the game, which is only legal if the move is its first move. We overrode the appropriate setter function to later set the firstMove as false. Each of the subclasses override functions called checkValid, and availableMoves. checkValid function consumes the beginning location, the destination, the game board, the current location its king is in, the twoPawnMove (array to check for en passant), and moveNum (the number of current move). Then, it returns if it is legal to make the move from the first location to the destination, by checking it abides by the subclass's rules and if the king is not in check. availableMoves takes in the same input except that it receives a beginning location but not the destination, and returns a vector of all the moves that are legal to make by the piece of the same type in the current given location. checkValid is used in availableMoves, and to check if the given input from a human player is valid. availableMoves is used to generate moves for a computer player.

The last part of the project is the views that the players get to see. This includes both the text view and the graphics view. Both text view and graphics views are inheriting from Observer, which is an abstract class, to update views whenever the observer is notified. The virtual function update(Subject &square) is overridden in text view and graphics view to update specific parts corresponding to the square in two views. Subject class, which is the class from where Square is inheriting, "has-a" Observer class.. The Subject class is a connection between Square and Observer so we can subscribe and unsubscribe views for each square. It includes abstract getPiece, getColumn and getRow which are overridden in Square class. Besides, it has notify() which notifies observers whenever the function is called.

For text view, we stored the pieces and their colour by specific characters as indicated in the project description in the separate 2D array of char, TextBoard. We created a default constructor for TextView to initialize a default TextBoard. We override the function update(Subject &square) to update a char in the TextBoard according to the piece in the square. We also overload the output operator so that we can print out the TextBoard with the output operator.

The graphics view has a pointer to the Xwindow class in a field, and GraphicsView "owns-a" xWindow class. We created a default constructor for GraphicsView to initialize a window corresponding to the default gameboard. We override update(Subject &square) function to update a square in the window.
We modified and added some functions to the given Xwindow class from the assignment 4. The functions fillPawn, fillRook, fillBishop, fillKing, fillQueen and fillKnight are the functions to draw pixels of each piece on a window. The functions fillFrame and fillKQFrame are to fill a frame of pieces and to remove the duplicated codes.

Design

For the first part of the project, main.cc, we have used lots of iostream and sstream, with string and vectors in order to interact with the players. Iostream and sstream are some of the techniques we learned at the very beginning stage of this class, but they are very handy tools to interact with the user through command line by intaking commands and outputting error messages. Main.cc is also where we initialize the game and the players using the Game class and Player class we have respectively for this project. We have very high cohesion and low coupling among our modules, following good design patterns. There was also a good amount of logic involved in main, especially with the setup feature and playing again after a round has ended, as well, some optimization was done in main.cc to avoid repeating code. As the heart of the project, main.cc connects all parts of the program and delivers the result, a game of chess, to the users. Interacts with the user and lets them enjoy the fun of chess.

The second part of the project is the Game class. It is in a "has a" relationship with the player class, which is an aggregation. The reason we decided to do it this way is that some of the functions in Game need the type Player, but to ensure high cohesion, we wanted to keep the class Player separate, following a good design pattern. At the same time, the class Game "owns" the class Square, which is a composition. We did it this way as the type Square is not used anywhere else outside of Game, but to have high cohesion, we want to keep different responsibilities to different modules, ensuring good design. Furthermore, all fields in the class Game are private, and can be accessed with the getter functions and setter functions we wrote in the class. This is important as it prevents unwanted modifications to the fields, making sure we are following good encapsulation.

We have also made quite a few changes to our code structure during these 2 weeks compared to what we planned during DD1. In the class Game, our game board changed from a <vector<vector<Square>> to an 8x8 array of Square. This is because since we know the size of the game board, it is easier to work with an array, and it prevents accidental modification to the size of the board, as an array does not allow a change in size. We have also added fields stalemate, which keeps track of the status of the game. 4 fields of king location related variables were added, which helps us with checking if the king is in check. A variable that keeps track of the turns is also added, which was not previously thought of. And after implementing the views, we have decided to add fields for the textview and graphics view to help display the board. Function wise, most of the functions remained the same with the addition of a few functions to help display the board, setting up the board to default, and clearing the pieces when starting a new game. Since we have increased the number of private fields, we have also made a few getter functions to get the value of these fields. For some of the functions, more parameters were passed in as we came to a realization that some fields are needed to perform certain action, this includes passing in the players for score printing, as we have decided to keep Player class separately for cohesion purposes. A few other functions were also added to help with graphics display and text view as a part of the MVC design pattern, which will be discussed later on.

The class Player is the third part of the project, they are an independent module that stores the information of the players, but is included in the Game class to suit our needs (as described above). Originally in DD1, we planned to have 2 other classes called Human and Computer, inheriting the Player class. However, we have decided that it's not needed, as "human" is pretty much the same thing as player, and the only difference for computer and player is the field level. Keeping them as different classes also caused big implementation problems and created lots of unnecessary work while trying to access fields, as it prevents us from playing the game with the same user settings more than once. Hence we have decided to combine them and only have 1 player class. To differentiate human and computer players, the variable "type" is added for each player.

The fourth part of the project is the Square class, Piece class, and its subclasses. To ensure that there is no memory leak, we used shared_ptrs that are accessible from the <memory> library. Smart pointers are a good method to ensure Resource Acquisition is Initialization, since they are destroyed automatically. For example, in Square class, the pointer to a piece is a shared_ptr, and not a regular pointer. Square is the model of the MVC design pattern. This is why it inherits from the Subject class, and functions like getPiece, getRow, and getColumn are overridden in square class. Square manages the data and notifies the observers without knowing how the data is presented to the users. In addition, we wanted to perform fake moves that will help us find if the king is or going to be in check, without notifying the observer. Hence, we created another function in DD2 called swapPieces that is considered as actually moving the pieces and updating the observers. We thought since we can keep all the Squares in an array, we can just not specify the row and the column of each Square in DD1. However, to notify the specific Square that was updated, we had to add its column and row in DD2. All the fields are private in Square, to promote encapsulation and to avoid accidentally modifying its fields.

Notice that all the pieces have similar properties, such as that they store a colour and they all have a name. Hence, we decided to use inheritance and create a general class called Pieces and make the different types of chess pieces as its subclasses. It does not matter what piece is in the location x and y, if we are looking at if that piece is going to be captured by another piece nearby. This is why we decided to keep the checkCapture function in Piece class, instead of repeating the same function. Also, this allows us to use polymorphism in our code. Square can just point to a Piece, but it can be any of its subclasses. Since it is a pointer, Square will have access to the specific type of subclass instead of the general class. Piece classes only serve one purpose: to represent the chess pieces and its properties such as its available moves, and the pieces do not depend on other classes to perform its functions. This promotes high cohesion and low coupling. In DD2, we realized that we need to give each piece a name so we would know what type of piece it is when we look for specific types on the board. To achieve polymorphism, we had to put the functions of its subclasses in the base class, so they could be overridden by each of the subclasses. We added necessary functions such as checkCapture and availableMoves in DD2, after realizing that we're missing them in DD1. availableMoves made use of another function in its class called checkValid, so there would be less errors made in the code and to optimize the code. More parameters are added to functions such as checkValid to accommodate the rules of chess.

The last part of the project is views, textview which manages a text-based interface and graphics view which manages a graphical interface to the users. We used the Model-View-Controller design pattern, to allow multiple views, text and graphics, to receive notification when Square(Model) changes. By making multiple views inheriting from Observer, it allows both of them to get updated when the their parent class get notified. In DD1, we have also added Controller class in the UML for the bonus feature by allowing the users to interact with graphics display. However, we decided to not implement controller since it is hard to implement interaction for xwindow, which supports pixels.

In the TextView, we used encapsulation by having the TextBoard which stores characters to print out in the private field and updating the TextBoard using the public method update(). Besides, in the GraphicsView we encapsulated the class by having the pointer to window private and have public method update() to modify the part of display. GraphicsView “owns-a” xWindow, which is composition, thus the xWindow will be destroyed automatically when the graphics is destructed. Also, we used make_shared when creating new text and graphics view to have better memory management.

In the xWindow class, every function that draw pieces takes three parameters, which are x-coordinate, y-coordinate and the colour of pieces. In the functions, we implemented the each drawing of pieces based on one square with the built in functions in Xlib/X11. Then, by passing in the location of the square which we can get from getColumn and getRow, we are able to show drawing of pieces in the right location in the graphics view.

In the Observer update() function, we gets the reference of subject as an parameter to update as little part as possible for graphics view, i.e. update only one square instead of updating the whole board. In the Subject class, when it gets notified, we pass in the pointer to its subject object as the parameter when the update function in Observer is called. Thus, since we know which subject is modified, we could be able to update the corresponding squares only instead of the whole board.

In the Subject class, the function notify() is protected because it should only be called by Square object. The Subject and Observer classes produces low coupling since Subject doesn’t need to know details of Square other than the public functions Subject has, and Observer doesn’t need to know details of TextView and GraphicsView classes other than the update function.

Resilience to Change

We have learned lots of techniques in this course, including composition, aggregation, and encapsulation. However, the most important thing that this course has taught us, is to ensure high cohesion and low coupling in our programs.

As described above in the design section, our program promotes high cohesion and low coupling throughout, with implementations everywhere to show good design patterns. Using the strengths of our design, we can support various changes to our program specification.

Our programs are split up into different classes, with each class focused on its own specific tasks. We have the Player class where it keeps track of all the customized fields for each player, the Game class where it stores the game board and functions to determine whether a game should continue or end, the Pieces classes where each individual piece has its own checkValid and availableMoves functions, following the different rules for each kind of pieces, then we have the textview, graphicsview, xwindow, observer classes whom each focuses on their own tasks to make sure the display is working. Our classes barely depend on each other, as they each have their own tasks and they are very clearly split up. With our way of design, the classes can finish the tasks mostly through fields and functions within itself, ensuring low coupling. For more details of how we ensured high cohesion and low coupling, please refer back to the design section.

High cohesion and low coupling is the key to make the program resilient to change. These two key points ensure that if the program specification were to change after the program has been completed, only a small part of the program, or maybe just one small part of a class, needs to be changed, rather than the whole code base.

For our program, if we wanted to make the players even more customizable with more fields like maybe their records, all we need to touch is the player class. All that's needed would just be an extra field to store it and maybe a function to take the input. If at any point, the program specifications have asked us to be able to handle more commands, we will just need to adjust the command intaking statements in main, and then implement the function in our game class. If the rules of chess were to change for a certain piece, since we have low coupling, our pieces functions are not dependent of each other. We could simply just change the functions of checkValid and availableMoves for that certain piece in its own class. Adding a new kind of piece would also not be a problem. Just like how all the pieces are inheriting the abstract base class Piece, we could just add a new class which inherits the class Piece, and add the functions required for the new piece based on the rules required. If program specifications were to change for displays, since we followed the MVC design pattern, it would be very easy to adjust the views.

Making the program resilient to change is one of the most important things to do when building a project, especially in the software consulting field, as program specifications could change at the very last minute. We deeply understand that, and with our setup that inherits high cohesion and low coupling, our program has very strong resilience to change.

Answers to Questions

Our answers remain the same from DD1.

Question 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.

We will choose to use a readily available database for chess openings found through research, and create a tree with all openings in the database. This can be done through having the individual steps as nodes, and looking through the next level children of the tree everytime the opponent makes a move. The tree should be programmed in a way that the top winning-rate openings will be on the left side, while the less winning-rate ones will be more on the right side. Every time when the program looks through the children of the tree, we would start from left, and settle with the first result (most left one) we find. If we happen to reach the end of the tree but the game still has not ended, we will go back to the default algorithm.

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

We will create two vectors of moves for player 1 and 2. For a new move we can use v.emplace_back() to add to the vector, and for undo we can erase the last move in the vector by using v.erase(v.end() - 1). Then we will prompt the player to redo their new move, and add to the vector as described previously. We would also have a boolean that will record if the current move has already been undone. This boolean will be reset every time when we move onto a new move. For an unlimited number of undos, we can erase the end of vectors using v.erase(v.end() - 1) alternatively, starting from the last move made until the players are satisfied. Compare the size of the vectors' length to determine the player's turn after undos.

Question 3: 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 would have to change the way main is collecting input from the users so we would take in four players. We would have to add more colours to differentiate different players. Also, the score function would have to display the other two player's scores and we would have to keep track of two more scores. The game board would now be a bigger array to place the additional player's pieces. We need to modify our if statements in main after a move is made, so that we would be checking if other players are in check or not. We can add team or individual mode by having two colours in team mode and having four colours in individual mode. We need to fix the invalid location checker to accommodate the larger game board.

Extra Credit Features

- Smart pointers were implemented throughout the program to manage memory
- The program asks for the player's name if they are registered as a human, and games are displayed in a variety of places throughout the game to increase interaction with the players, these places include:
    - After each move, (valid or invalid), the program will display "white (John)'s turn" or "black (Doe's) turn" as appropriate
    - When a game ends and if we have a winner, we will display "white (John) wins!" or "black (Doe's) wins!" as appropriate
    - When one side is in check, the name is displayed like "white (John)'s in check!" or "black (Doe's) in check!" as appropriate
    - When scores are printed, name is again displayed after the colours as appropriate.

  We believe that this is a great feature as it makes it more clear which user we are referring to, especially when both players are human. This also increases interaction with the user and makes them feel like they are more of a human than just a bot.
- Despite the assignment telling us to ignore en passant if the board is configured, we were able to implement the feature, enabling en passant under any circumstances.
- We have implemented a "white's turn" or "black's turn" display after each attempt of moving, reminding the users whose turn it is. During the process of testing, we found it hard to keep track of colour's turns. Hence the feature was implemented to remind the user.

- Though not required, after changing the structure of our player classes (as mentioned above), we are able to ask the players to play the game again with the same user settings, this ensures that the players can keep the scores they have accumulated, and see their final score when ending the game.
- The accumulated scores for the players are printed after each round, while the assignment only required us to print it when the user ends the program.
- Lots of error messages were implemented, and specific messages were given to remind the users of the problem. This includes:
    - removing an already empty cell when configuring the board
    - Out of bounds for placing or removing pieces in setup mode
    - Invalid piece names were put in at any time during the game under various scenarios
    - Specific out of bounds error messages when attempting to move, both begin and destination

Final Questions

Question 1: 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?

For this project, we decided to work as a group. With 3 team members, communication has become super important. From the beginning, we have always made sure to have smooth communication. We have created a group chat to update on everything about the project, including our progress, bugs that we have found, our thoughts of implementing a feature, etc. While working together, we also made sure to communicate in order to avoid merge conflicts as much as possible. After having both in person sessions and online sessions, each team member talked about their feedback for those sessions, including how productive they felt they were, and which one they liked better. This helped us to ensure that we were working at a good productivity level. Everyone in the group has felt the importance of communication through this project, and it will be a good habit for us to carry on in the future.

Another thing that we learned was to set specific goals and adjust them as needed. When you are working in a group, tasks might get messy. It is important to make sure that each person is clear of what they are supposed to be doing, to avoid unwanted situations. However, remember to always have an end of day review on everyone's progress, adjust goals and re-assign tasks as needed.

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

The group thinks that writing a pseudo code version of the program before implementing the actual code would be helpful in reducing the time needed to re-implement things. Oftentimes, especially with big projects, some things might not work as you imagined in your head, and re-implementing a function takes a lot of time, and also increases the risk of forgetting to change something. By having a pseudo code written beforehand, it helps the coder to think about the logic more thoroughly, rather than just imagining it in their heads, which could save time and maybe even have better design patterns.

Conclusion

Building this program is our first ever experience to work together as a group. Though things could get frustrating, we all had fun overall while learning about new things. This project has helped us bond and made our friendship stronger than ever.