Overview

For our design we implemented the Architecture Pattern as well as using the Observer Pattern to regulate the different views.

Observer Pattern:
In our program, the View part of the Architecture pattern is called Observer. Graphic, Text and EndGameService, HistoryService inherit from Observer. Text prints the board to the standard out stream and Graphic prints graphics using Window and its methods. The EndGameService checks if the game has ended after every move. The HistoryService stores the most recent move. Each observer is notified after every move, and the views are notified after every change in setup.

MVC Architecture Pattern:
As mentioned before, the View part of the pattern is called Observer in our program, Board is Model and Controller is named the same as in the pattern. Controller keeps track of who's turn it is, switches who's turn it as after every move, makes moves, checks for valid moves, checks for certain moves that happened, empties and sets the board, and calls Observer's notify(Move move) method when needed. Other than calling notify(Move move), most of its other methods rely on the methods in Board.

The Board Class:
Board contains all methods and fields needed to make up the chess Board which is composed of Squares (white or black), each of which is empty or contains a piece. All the individual Pieces in the game inherit from Piece (white or black).

Player:
Player is the base class for the different players available in the game: Human and Bot. Human is the user and Bot is a computer that plays the game and there are 4 levels. These four levels of Bot inherit from Bot.

Design

Observer Pattern:
Observer has an additional function than the pattern, initNotify(). We added this function in addition to notify(Move move) because notify(Move move) only updates the squares that changed in Graphic. Thus we needed a function that prints the entire Board when a game starts. This is what initNotify() does. We added the move parameter to notify() because we needed a way to keep track of what squares were changed. Move has two fields, start and end which each have an x and y field. This is how we know the coordinates of the two Squares that changed. notify(Move move) prints the piece or empty square of just the start square of move and the end square of move so that only the necessary pieces are reprinted. This is different from text where text rerenders the entire board. notify(Move move) is called after a move is made. With this order of calling notify(Move move), we ran into a problem with en passant and castling (more than just two squares need to be taken into account and we need to check for this before the move is made. We fixed this by checking for en passant and castling in the move function before the move is made and setting the corresponding

isEnpassent or isCastle field (in Controller) to true or false. Then in notify(Move move) in Graphic, at the end of the function after updating the two squares in move, we updated extra squares as needed if the move was enpassent or castle. Then we reset these fields back to false after printing. The way Graphic sets up the Board in an optimal way is by first printing the entire square (white or black) starting top left down to bottom right, with one call of fillRectangle(...) which is fast, and then prints just the pixels of the pieces on top of the square which is a bit slower. Each individual piece has its own print method since each piece needs specific lines of code to print the right shape of the piece. The pieces are pixels in a 10 by 10 grid.

Text is fairly simple. It gets the character needed to print for each square using the getState function which traverses through Controller to Board to Square to Piece to finally get the resulting character (function calls charAt(...) further in the chain).

Architecture Pattern:

Controller inherits from Subject. Subject contains the observers field which is a vector of all the current Observer shared_ptrs. It contains the public methods attach(...) and detach(...) which adds an Observer to observer or takes an Observer away. notifyObservers(Move move, bool onlyNotifyView) calls notify(Move move) on all observers. We added the extra field bool onlyNotifyView which uses dynamic casting to call notify(...) on only the Text and Graphic subclass Observer during setup because the EndGameService is not equipped to Handle the initial empty Board. printInit() calls initNotify() on all Observers and prints the initial Board (more important for Graphic as mentioned in the Observer Pattern section above).

Controller has the private fields whitePlayer blackPlayer which contain shared_ptrs each to a Player (Human or Bot). It also contains Color playerTurn which contains the Color of the players who's turn it currently is. It contains board which is a shared_ptr to the Board. Mode mode contains the state of the game (SETUP, GAME, or START). Finally we have the last private fields bool isInGame, bool isEnpassent, bool isCastle. isEnpassent and isCastle is used as mentioned in Observer Pattern section above. Controller has a map field called score that keeps track of the score of each Color for each game played.

The Board Class:

We implemented the game board using a vector of Squares called board. The way we access the Square at [letter][number] is by converting the letter into the corresponding integer value (0 to 7, a to h) by subtracting the character 'a' from the letter string, and getting the corresponding integer value (7 to 0 starting from the top of the board to the bottom) of the number by subtracting '0' and 1 from the number string (this is done by a Player subclass) and then getSquare is called in Controller and then Board indexes board at index [yDimension - 1 - y] * xDimension + x]. This is how we work with the one-dimensional array.

Square has an int x and int y field as well as the color of the Square, bool black, and a pointer to its Piece, piece. A square is empty (has no Piece) if piece is nullptr and this information can be found out with the public method isEmpty(). The color can also be found with the function isBlack(). The piece can be accessed with the public function getPiece() and the piece can be set with setPiece(...). The coordinates of the Square can be access either by calling getX() and getY() separately or getting a vector of the coordinates by calling getCoords();

Piece has a field PieceType pieceType (one of ROOK, QUEEN, KING, PAWN, BISHOP, KNIGHT), Color color (one of WHITE or BLACK) and bool hasMoved. Piece has pure virtual functions name() which gets the character that should be displayed to standard out for that Piece, canMove(...) and canCapture(...) returns true if the move is possible and true if the move is a capture, getPieceType() which returns the PieceType of the piece and copy() returns a shared_ptr to that Piece. The other method are getColor() which returns the piece Color and getHasMoved() and setHasMoved which returns and updated the bool field hasMoved. All the pieces (ROOK, KNIGHT, BISHOP, QUEEN, KING, PAWN) inherit from Piece.

Player:
The player class handles what moves are to be played. This is done by creating 2 concrete classes, Human and Bot which are implementing getNextMove(). For the human class, getNext move takes in arguments from the command line to parse the moves (e2 e4) to a Move class which is composed of a starting square and end square (so e2 would be converted to (4, 1) and e4 would converted to (4, 3)). For bots, where there is an additional 4 classes botlevel[1-4] which inherits from a bots class, there are several methods such as findCaptureMoves, findCheckMoves… These methods are then arranged in different priorities depending on the level of the bot. For example bot level 2 will priortize capturing moves over random moves.

Resilience to Change

The observer patterns and the way we have configured the services allows us to add more features to the program. The abstract base classes means that we can have different implementations and also add different observers to listen to the subject. For example if we wanted to add another service that checks if this is a known position in a database, then this could be done by creating a DatabaseService that inherits from observable, which has a notify method that is notified every move which queries for the position and see if it has a name.

The Piece abstract class allows us to implement different pieces, so a potential extension to the game could be adding different pieces. For example a new piece would just need to implement the canCapture and canMove methods to determine what squares it could attack and what squares it can move to.

We could also extend the game to a square to be more than 8x8 board. The board method has a xDim and a yDim attribute, which is used to allocate space for the vector which defines

the board. Therefore, changing these attributes will allow a larger board. Furthermore, the parsing of the squares will still stay consistent for example, I9, a square normally not on the board would still be correctly be parsed as (8, 8). There is also a squareDim field in Graphic that changes how big the board graphics are displayed so you can make the board bigger or smaller by changing this field.

Cohesion and Coupling:
We have high cohesion because each module has one purpose, which is why we have many modules. Piece is only to implement the specific functions relevant to each individual Piece. Square is just to store the piece, the color of the square and set the piece. Graphic only displays the graphical display and text only displays the text display. Bot only implements the functions needed to make each level of the computer player. Player's job is only to implement functions needed different from Human and Bot. Board is the Model that implements all the functions needed to make all the moves and checks needed to actually play the game. And finally the Controller sets the game in motion and communicates between the View (Observer) and the Model (Board) to play the game.

We have loose coupling because the Observer pattern ensures that more observers can be added and we could take them away and their would not be a ripple effect so classes within the Observer Pattern do not highly depend on each other. They do have high cohesion thought because there is lots of composition between these modules and the subject calls notifyObservers(...) which calls notify(...) of each observer, thus working together. Also, the pieces are all subclasses of Piece so changing one piece will not have a ripple effect on the program. Also, Controller and Board work together a lot to change the state of the Board and keep the game going as well as Controller and Player which gives high cohesion.

Answers to Questions

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. See for example https://www.chess. com/explorer which lists starting moves, possible responses, and historic win/draw/loss percentages. Although you are not required to support this, discuss how you would implement a book of standard openings if required.

A book of standard opening moves can be implemented by supplying a json file, where the keys are the names of opening moves, nested inside would be more keys such as "moves", which outlines the series of moves in string to input to get to that position, and win/loss. Then a class can be made called PositionService which inherits Observers and is part of the Observer pattern where the subject is the controller and notify is called after every move. The notify method then scans this json file to see if the current position is a match which can then be displayed. This pattern is similar to how the EndGameService and the HistoryService.

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?

This has been implemented in the project. The way this works is that we have a HistoryService that handles previous previous moves. When a move is made it is notified and the move is pushed back to the a vector called boardHistory, which stores copies of the class Board. There is also an attribute called currIndex which is meant to keep track of the current board.Then when undo or redo is sent in to the cin, we can decrement or increment the attribute currIndex. Then we can set the controller to the new copy of the board indicated by the currIndex. Finally notify is called but with the onlyCallView arg is called with true, so it does not call the EndGameService and HistoryService. This allows the views to rerender the undo and redo. As a note since the text and graphic views render the board in different ways (text rerenders based on the board and graphic renders the squares that are passed in via Move), calling rendering the view from undo and redo, will cause a full rerender to the graphic view.

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. (If it's important to your answer, state whether you're assuming free-for-all or team rules and then answer the question. You don't need to get too specific into the rule set changes in answering the question though; your focus should be more on what would need to be altered at the high level of the design?)

In order to implement 4 player chess. A lot of changes need to be made in the current state of the game. For example, when implementing pawns a notion of direction needed to establish. In 4 player chess there would be changes to accommodate the additional 2 directions. Furthermore, the board is not a square. One way to accommodate for this is to find a bounding square for the board and then if a move is made that goes to a place that is not supposed to be on the 4 player board but is valid on the square then render that as an invalid move. Only a small change is required to accommodate for 4 players, currently the main handler takes in arguments from game and assigns the corresponding player to the right color based on the order that the args were entered. This same approach would work for 4 players.

Extra Credit Features

A major extra credit feature we implemented was the undo and redo feature. The undo and redo is implemented via the HistoryService, described in one of the answers above.

We also only used smart pointers which means that we never had to use any deletes to manually manage memory without having any memory leaks.

Level 4 takes in the values of the pieces and can look for if it can do checkmate in one move.

Final Questions

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

Allison:

This project has taught me that every programmer has a different approach to solving a problem. Sometimes we think of similar ways to tackle it, for example my team and I all thought to do the Observer Pattern for the Text and Graphic views and make a Piece superclass with the individual pieces subclasses. Other times, we would both have ideas about how to fix a problem and would discuss the best way. It was really interesting to me, to do this project because I have never used git before and worked on code with other people. I have only ever known how I would implement things, so it was interesting to hear how my team members would approach each problem. It taught me that working as a team - each of use doing our own subprojects (me the graphics and text, DB the underlying implementation of Controller and Board and Quang doing the Bots) - causes us to run into a lot of problems if we don't understand each other's implementation fully. For example, the way I implemented Graphic was to just rerender the start and end square of the move that was passed in. However we ran into problems for Enpassent and Castling since the functions getIsEnpassent() and getIsCastle() checks for Enpassent and Castling when a move is being made and in notify(...) after the move is made. So then the notify(...) function could not properly check for getIsEnpassant() and getIsCastling since the move was already made. I wasn't aware this how was DB's code was implemented and thus I used these functions incorrectly. We fixed the issue by updating fields in Controller IsEnpassant and IsCastle before the move is made so that Graphic can access them and find out if more squares need to be rerendered to accommodate these moves. It also taught me that we need to git pull and git commit as much as possible. I learned git stash and git stash apply after pulling. We ran into some conflicts where my changes didn't save and we needed to go back to my other commits and find my changes. It was frustrating at times but it really reinforces the point that we need to git pull and git commit as much as possible and be careful when we merge with other's changes.

DB:
Throughout this project, I learned alot more about using the gdb debugger. Especially when managing memory manually using the gdb debugger is especially important to see any segfaults. I learned many more gdb commands such bt to see the stack trace, and p to print a specific piece of code. C++ implicitly calls many things so using the gdb to see the stack trace and the flow of the program was especially important.

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

Allison:
I would have read through DB's code more thoroughly to see how he implemented some of the functions so that I would run into less errors when I called some of controller's functions. I also would have let DB do setup instead of me since he knew the underlying workings of his code better than I do. I would have implemented the underlying code alongside DB so that we could discuss and implement together and both understand the underlying code. Also it would help our learning for both of us if we bounced ideas off each other.

DB:
One thing I would've changed is to change the composition of the Move class works. Currently the Move class has 2 attributes a Square called start and a Square called end. However, since these aren't references this is misleading, since Square has a getPiece()

method to get the pieces on them. Calling getPieces on the Squares contained by move does is not the same as calling getPiece by squares created by Board. Therefore, this caused some bugs due to miscommunication and forgetting about this fact. If I were to do this again, Move would store 4 integers, instead of Square since Square has a different job than what Move is using it for.

‘