https://drive.google.com/file/d/1aWH9EwNgLM3pcvmNsAgNzeIKarY_QTeU/view?usp=sharing
Link to UML ^^

CS 246- Chess: Chiara Alcantara, Franklin Ramirez, Helena Xu

Overview

Our project is a recreation of the 8 by 8 game Chess played through the command line, designed using Object Oriented Programming (OOP) principles and integrated with overarching design patterns such as the Model Controller View (MVC). Starting with the curation of the game board itself, the user is first able to choose between either a default chessboard or to set up a custom game board by placing and removing pieces as long as they meet certain requirements (e.g. only one king on each team, kings aren't in check, etc). Then, the user has options of setting the two players as combinations of Human or Computer players, with the Computer having further options of specifying a level of difficulty (prefers a certain type of move over others), in which we have implemented this logic using a Factory design pattern. Additionally, we employ validation within our game that ensures that each player's pieces can only make valid moves based on chess rules. This is managed through two Observer design patterns and occurs both for Human players (in which we cross-reference the move the user wants to make), and is also used as standard practice within each of our generated Computer moves. We also support special moves such as pawn promotion, castling, and en passant. Then, all of this information and logic is updated within our displays, that being the TextDisplay and GraphicsDisplay, through another Observer design pattern. This is a general structure of our implementation of our Chess game, in which we will detail specifics of our design, techniques, and challenges in the next sections.

Design

We employ various OOP concepts within our Chess design, from the MVC architecture to Observer and Factory Design Patterns, all with a central goal of decreasing coupling and increasing cohesion. Describing the overall structure of our project, we utilize a MVC template with the ChessBoard as a controller, TextDisplay and GraphicsDisplay as the viewers, and the rest of our classes as models that store the structure and data. This process started as initially our first draft UML was made without directly following MVC, resulting in more of our logic being widespread in several classes, instead of focused in one controller. For example, our original implementation of the Piece class was created such that every piece would conduct control logic for specific moves by receiving input from the user, validating the move, and then altering the gameboard through an observer pattern where each piece observed each other on the gameboard. However, during the revision of our structure and throughout the subsequent implementation of our classes following this UML, we realized that our design would greatly benefit from a properly structured MVC template.

Note, here are some useful terminology that will be discussed in the following:

- Possible Moves are moves created by our *Pieces* subclasses for different piece types (pawn, king, knight, etc) in which these moves have only gone through the first stage of validations. This validation is checking that the end coordinate of a move does not already have a piece at that coordinate, or that if it does, it is a piece from the enemy team (for capturing).
- Legal Moves are moves validated within functions of the *ChessBoard* class that check if the move is legal in chess rules (eg. a move that doesn't put your own king in check).
- Then, remaining Capture Moves, CheckMate Moves, etc have further filtering and functions to check and categorize the move into special types if necessary.

By reducing our code to follow the MVC template, our group was able to better understand the connections between each class, subclasses, and different pointer relationships. Beginning with our controller, the ChessBoard class is mainly composed of three tasks, constructing and initializing the gameboard and its components, the logic in regards to translating user input to simulate and make actual moves in the board (taking into account features of castling, en passant, etc), and finally, logic within functions conveying or creating this information. As demonstrated, ChessBoard handles request flow and logic behind function and game controls, processing and passing this information to the viewers and receiving required data from our models, all through Observer design patterns. Through functions such as defaultBoard(), etc we are able to initialize the standard chessboard, with additional functions for custom setup like setupWithChar() and boardIsValid(). Then, the bulk of the chess game logic is focused on functions such as makeMove() which makes the actual move on the board, or testMove() which simulates possible moves of an opponent in response to the current user's move and categorizes them as legalMoves, captureMoves, checkMateMoves, etc[1]. Alongside this purpose are functions that specify features of chess moves like upgradePawn() and castleMove(). Finally, there are several remaining control logic functions that control the passing of information to and from our viewers and models, such as chessboard's overwritten notify() for updating the viewers' chessboard outputs (handled through the ChessBoardObserver observer design pattern) or forfeit() and updatePawnMoved() for updating required game logic stored in the model.

Next, our viewers describe the use of TextDisplay and GraphicsDisplay classes to display command line output in TextDisplay and using xWindow for a graphical interface in GraphicsDisplay. Our viewers simply use its given data to display for the user, and this is done through the use of an Observer pattern between the viewers and our Chessboard controller as per MVC template. By utilizing class DisplayObserver, the Text and Graphics Displays are able to override functions such as notifyMoves() which is called inside of Chessboard when a move is made to notify our observers (the viewers) that a move has been made and so needs to be shown accordingly on the text and graphics display. Or, for example, the singular notify() takes in a boolean *white* field, which is used to determine which user's turn it is (white's turn versus black's turn). The use of observer design pattern here solves multiple challenges in terms of allowing for the board game displays to update accordingly whenever a user makes a move or needs to display information like scores and turns accordingly. By having more than one pure virtual notify function inside of the DisplayObserver we are able to have different notifies for

different purposes as outlined above, but also the flexibility for one subclass to not use a notify such as how the GraphicsDisplay didn't have an implementation of the boolean notify() function as it didn't need to display the turn graphically on screen. By using the Observer pattern we were able to apply low coupling and high cohesion principles without creating a dependency between the chessboard and the displays being notified in a one-to-many relationship. It is also how we were able to separate the two displays into viewer classes of the MVC template. Additionally it is worth mentioning that our main.cc file, which although is not directly a viewer model, interacts with our user interface as it handles communication between the user and the controller through command line input.

Finally, our model part of MVC pattern is focused primarily on three classes, *Game*, *Pieces* and its subclasses, and *Player* and its subclasses. Beginning with the simplest, we have a *Game* class that keeps track of the overarching scores through each of the chess matches, and hence is a model as it holds data. The chessboard instantiates a *Game* class and since the chessboard is reset to the emptyBoard but never created again in a single running of the program, it is able to keep "global" values of the scores throughout the entire program duration. Next is the *Pieces* class, in which is another model as it stores information and data about its subclasses, that being the different types of chess pieces. It also has the required information specific to each move of the piece, such as getPossibleMoves() in which it uses the chessboard coordinates to find all the possible moves for that particular piece in the current state of the board. Finally, the *Player* class is one of the more complicated models, in which not only does it store general player information like the player's categorized moves (legalMoves, captureMoves, checkMateMoves, etc) but it uses an Observer design pattern with the Observer abstract class in which we have overwritten notifies that chessboard calls to categorize these moves in the player fields. Adding onto this however, there is another Observer pattern between the *ChessBoard* and the *Player* in which the chessboard also observes the player through the ChessBoardObserver abstract class. This is because it is in the *Player* class that either validates that the move can be made (for humans) or selects a random move from its categorized/legal moves (for computers) and thus calls *ChessBoard's* notify() to then actually make this move. Hence, this double Observer pattern allows us to ensure that the validation in choosing and making a move from both types of players is properly encapsulated inside of the *Player* class only as it is related to either the human or computer's moves, and not within the *ChessBoard* class.

Moving onto the derived classes of the *Player* class, we have two different subclasses, Human and Computer, with purposes as explained above. This brings us to the last class of Level, specific to the *Computer* class, in which we have invoked a factory design pattern to categorize and construct all of the legalMoves for a computer player in a specific level. By utilizing a factory design pattern, we are able to effectively create different levels of difficulties using the categories of moves without explicitly knowing what level is required until the user selects one. Through the factory design pattern, we are able to use the createMoves() function to customize the type of moves according to the levels, which properly allows for low coupling and high cohesion principles. This factory design pattern also allows us to support scalability in adding more levels as described in the next section.

However, our current implementation still isn't a perfect replication of the MVC architecture. There exists limitations such as the *Piece* class still conducting some business logic of validating moves. In order to provide the *ChessBoard* with the function returnPossibleMoves() we still have some logic in *Piece* class such as ensuring that the end coordinate of that move does not already have a piece at that coordinate, or that if it does, it is a piece from the enemy team. However, this implementation decision was made in mind as the pieces don't directly control the chessboard, it solely views the gameboard to make decisions about moves and so can also be argued as not directly controlling the game. Additionally, another improvement could have been for *Player* to directly have access to the simulation and testing of the moves from the *Pieces* class to categorize into its own legalMoves, captureMoves, etc fields. Currently, we still have *ChessBoard* containing this logic, and so to better encapsulate and separate the *Player* class logic, we could have used an observer pattern or to pass *Player* the vector of vector of shared pointer *Pieces* for it to categorize the moves itself. Although, due to our current time restraints and structure, some of these limitations from not completely adhering to MVC is reasonable while still ensuring well-thought out code organization or low coupling and high cohesion concepts.

For the most part however, by utilizing these different design patterns to overcome our design challenges, we were able to avoid many reiterations of our structure which involved circular dependency. Additionally, by following the MVC template we were able to better separate our classes and functions such that not all of our classes were dependent on *ChessBoard* and required access to it, which avoids these classes to be implemented as controllers as well.

Resilience to Change

Whether it be through the use of various design patterns or the general structure of our classes and subclasses, our implementation supports the possibilities of various changes. Starting off, by employing MVC architecture one of the features that we allow for is easy modification and scalability of the entire program. For example, if we were to add a new type of view then this would be a change to the separated viewers of MVC and thus will not affect the entire architecture. The same can be applied to different changes in which only the *ChessBoard* logic needs to be altered or a new Model type needs to be added.

Following the idea of adding new views, by implementing the Observer design pattern this also allows us to support adding more displays by just creating more concrete subclasses of the abstract superclass and then overriding the notify functions accordingly. In the case of changing the displays, an example can be expanding the game board, in which we can also easily do this within our Text and Graphics Display classes. An example implementation of such is depicted in our answer to question 3 of 4-player chess.

Next, we consider if we were to add new pieces entirely, this is easily accomplishable within our current implementation, in which through our *Piece* superclass we can simply create a new subclass of it and then populate virtual functions in the subclass such as getPossibleMoves()

accordingly to how this new piece moves on the board and set its starting coordinate on the standard gameboard. If we need to add any new fields to the subclass itself we can also easily do so without requiring any changes to the Pieces superclass, and so hence this design allows us to add new pieces with various logic or functionalities.

Additionally, if there were changes to program specifications in terms of the actual rules or game theory behind chess itself (eg. special moves) then this can be easily implemented within both our Player and ChessBoard classes. For special moves that require a second component outside of moving a piece to a new position, like passantMove() ,we would need to add additional logic to ChessBoard to update the gameboard according to that second component, and any fields like booleans to keep track. Otherwise, special moves that don't require that "second component" or new pieces that don't have special moves can be implemented using the Piece superclass itself. Finally, in the case that we want to add a new rule (eg. an alternate type of checkmate) that requires the knowledge of an opponent's move to respond to that change, we can also just add more respective logic to the testMove() function in ChessBoard, allowing for easy scalability.

For any changes in the levels of the game, our current Factory design pattern supports changes such as by adding more levels. This requires a new subclass of the abstract *Level* class that follows factory design, where we can mix and match already existing categories (eg. captureMoves and staleMateMoves preferred) just as we have already implemented in our current levels. Even if we were to add new "categories" of moves (such as through new chess rules/specifications illustrated above) we can easily implement this category through first the ChessBoardObserver design pattern between the Players and ChessBoard classes, in which the chessboard will notify the *Player* to add a certain move to this new category and then the levels unique_pointer inside the player field can utilize the new categorized moves for this added level through createMoves().

Answers to Questions

Q1:
Currently in our implementation, we have something similar to a Book of Standard Opening Moves already accounted for. This is done through our vector of Vecs, legalMoves in the *Player* class. The way that the current code works is that within our testMoves() function in the *ChessBoard* class, a new set of legalMoves is redetermined for the opponent player after the board has changed (eg, the current player has made a move). This allows us to first create a vector of all the opponent's moves to then use in checking if the next move this opponent wants to make is valid if it's a human player (by cross referencing it with the vector of legalMoves), and in the case of a computer player, this vector of legalMoves is used to provide a random move for it to play based on the selected level in the *Computer* class. If assuming a direct implementation of this into our code, (eg. assume that this Book of Standard Opening Moves is something that we are required to generate), then we already have such simulation implemented through the specific *Piece* subclasses (Pawn, King, etc) and simulating their moves in *ChessBoard's* testMove(). Hence, under this assumption, this is already implemented in our current game and

all we have to do is to output this information to the user. On the other hand, if we take the assumption that this “book” of standard opening move sequences is provided externally (eg. as a .txt file), then in our main.cc we will import the fstream library and then create a testing harness/command interpreter to read in each line of the starting move and the lines of its subsequent responses, to populate our legalMoves vector accordingly instead of using our own generated moves.

Q2:

There are two possible solutions for implementing singular and unlimited undos. The first is more memory-heavy in that the idea involves creating a vector of ChessBoards like vector<previousBoards> that consists of deep copies of each ChessBoard state throughout the game. After every turn, a deep copy of the current state of the ChessBoard is added with emplace_back() into the vector of the previous boards. Then, the undo function would be called with this vector as well as an integer value (in the case of unlimited undos, eg. 3 or 23 undos back), which allows us to traverse backwards in the previousBoards vector and modify it to delete the ones we’ve traversed back and skipped over (destroyed as it moves out of scope), then returning the requested previous ChessBoard state. The alternative solution is to implement a linked list that stores the moves of both players directly inside the *Players* class. This works as we already pass the start and end vectors for human move validation and in the case of computer players, we randomly select the start and end vectors for its move. Now, all we need to know is if the move was a capturing move, in which we can establish this by setting a new field in *Player* class, something like isCaptureMove, and then giving the *Player* class the current gameboard state such as through a vector of vector of shared *Piece* pointers (similar to what is already implemented in *Piece*) so that we can look at the current state to see what piece has been captured. Then, to undo, we would just have to pop the end of the list (the most recent node added), make a copy of this and set its head pointer to the next node, and then deconstruct this “undo-ed” move. This can be repeated with an argument for the case of unlimited undos, so long as we keep track of the number of “undo-ed” moves made so far. In regards to which player’s undo we change in this linked list, we have also rationalized that in the case you make the first move, the opponent makes a countermove, and then you decide you want to undo your move, then we simply have to “undo” with our linked list twice to go back to your original move since there will always be an opponent’s countermove in between. Hence, this allows us to implement a feature for a player to undo both their most recent, and unlimited moves.

Q3:

These changes are under the assumptions that four-handed chess follows the rules of Teams variation. The first change we’d have to implement is to add 2 additional Player instantiations and two more players in the *ChessBoard* class for a total of four players, and then implement changes to the *Player* class like having a string field to indicate different colors to distinguish them, or to add the Observer pattern between the new *Players* and *ChessBoard*. Additionally, we have to re-arrange the ChessBoard's gameBoard and emptyBoard itself instead of being an 8x8 board, it needs to support a 14x14 board, but with 3x3 squares removed in the corners to accommodate this arrangement. We can make these changes by creating a new *Board* abstract

class in which we create new subclass instances of StandardChessBoard and FourPlayerBoard. Alternatively, we can make a new type of the *Piece* that is a NonExistentPiece (like our empty *Piece),* in which will be placed in the 3x3 corners. Then, our *TextDisplay* and *GraphicsDisplay* classes will have to support these changes as well (eg. by extending the board, adding colors, etc). For the bulk of it, most of the rules will remain the same (eg. castling, en passant) except now since there are two players working together to checkmate the king of the opposing team, this means that the logic in functions such as legalMoves() in which we check the legal moves for the opponent (and hence the successive filterings of which moves are preferred in the different levels) will need to change as well. This is due to the fact that we now have to check that the move the user/computer wants to make is valid, ensuring that it doesn't put either its own or its teammate's king in check. With this as well, we need to consider that check/checkmate moves are now checking either two kings of the other team. We will also need to verify that we're not capturing pieces from our team, and only opposing team's pieces. Finally, in our command line we will need to support the option for players to choose which of the two versions they would like to play, the original chess or this four-handed chess.

Extra Credit Features

One of our extra credit features involved attempting the described enhancement in the project guidelines, that being completing the entire project without leaks, and without explicitly managing your own memory. We were able to successfully complete the first half of handling all memory management via STL containers and smart pointers, with no delete or new statements in our program at all. However, our program still leaks memory and thus we were not able to complete the full requirements. We started off initially using regular raw pointers, and then later on realized that it would be more effective for our learning and to effectively manage memory by using such smart pointers. Hence, we implemented the rest of our code following this specification, but the challenge was that despite only using smart pointers, our memory still leaks somewhere. We tried running valgrind on our program and were able to see how many bytes of memory was leaked directly and indirectly, but were unable to find out or debug how or where it leaked. Unfortunately due to our time constraints, we prioritized getting our code working and tested and so were unable to fix this issue.

We have also implemented a skip feature. Since we found that all of our functions keep track of the user's turn throughout the game (eg. within deciding possible moves we need to keep track of the turn to determine the opponent's next moves, or when keeping track of bCheck and wCheck variables), we would be able to take advantage of this value and skip a user's turn. Hence, this enhancement was completed by just implementing a setter for the current turn of the chessboard's player, and this can be called upon from command line using "skip".

Next, another feature that we added is the ability for the computer to resign a game, which as per Piazza post 1897 is an allowed enhancement. Much like the ability for a human to resign, for a computer to resign, the user just has to type "resign" in the command line rather than "move" and so allows the computer to resign and gives a point to the opponent.

Finally, we also added a move output feature, in which we output to the command line the coordinates of the move that the computer. This is only done for computer players as in human players the user enters the coordinates and so already knows which move is selected.

Final Questions

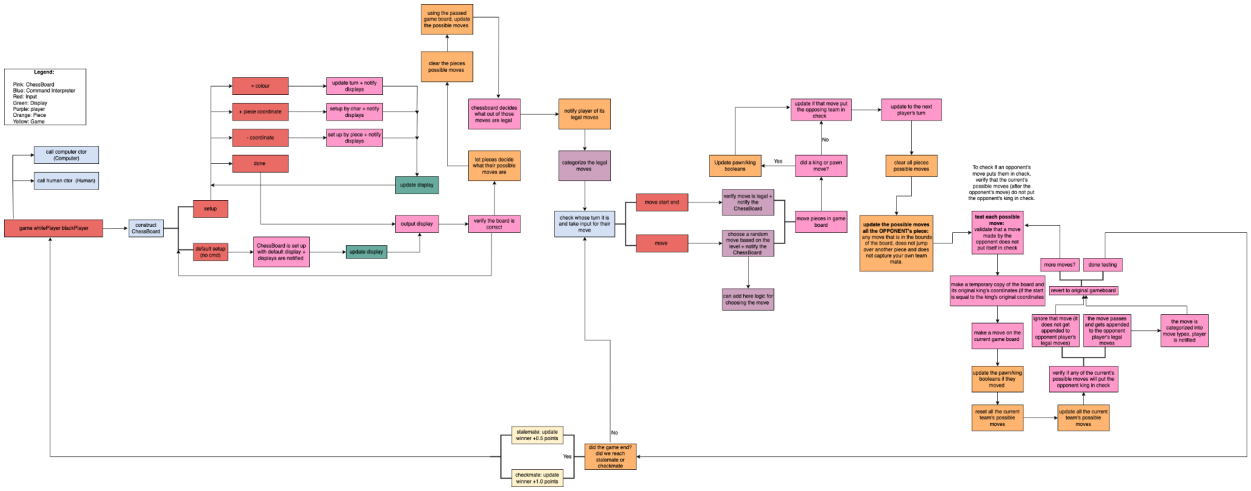
What lessons did this project teach you about developing software in teams? If you worked alone, what lessons did you learn about writing large programs?

The main takeaway from this project regarding developing software in teams is the importance of good planning, communication, and teamwork. From the initial UML and design stages it was already crucial for our team to brainstorm and work through the structure together. There were often times when one of us would forget an edge case to consider and another teammate would remind us or we would need to bounce ideas off one another. These brainstorming processes helped a lot and was a great summary as our group personally found it most beneficial to have all of us be aware of the logic of classes and functions, even if we weren't assigned that specific part of the code. This was particularly useful during the compiling and debugging stages as we wouldn't have to completely rely on whoever coded that component to fix an error, but rather each of us would have a general idea of where to start and then could work on debugging together. However, with this too is a learning experience regarding the importance of proper documentation and communication. In our case, since all of us worked on parts of *ChessBoard*, this often resulted in having to ask if another team member already coded a function that another team member might need for their logic, which resulted in issues such as having multiple functions that served similar, but slightly different purposes which we may have benefited by sitting together and creating a singular function. Additionally to the point of documentation and commenting, it's important to establish beforehand some general coding practices like commenting styles or naming conventions, just so that our code is easier to follow. Nonetheless, we believe that we were able to work well together as a team in developing this large program, in which through proper planning and communication throughout splitting tasks and combining, we have been able to successfully work together.

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

Although our group believes this to be a rather successful chess game project, given the opportunity to restart there are nonetheless a couple things we would take note of. Firstly, in terms of structure as mentioned in the design section, we'd like to refactor some of the design to reduce coupling, such as by putting the logic for categorizing and checking move types inside *Player*, or to have player directly notify the *TextDisplay*. Otherwise, in terms of the process of planning to implement this project, we would have benefitted from thinking chronological-wise earlier on, particularly during creating our UML. We believe that we got too ahead of ourselves during the start of creating our UML, and were only focused on what classes we needed and the inheritance and subclasses of them, which often made us confused as we would jump back and forth between the game process. Instead, we should have started from the beginning directly from command line inputs and what we'd require to take in these input, and then considerations

of human or computer players/different levels, as that would have helped us to conceptualize all different variations of the game, which would've naturally brought us to the creation of our subsequent classes. Closely following this point is that if we were to start over, our team would have benefitted from drawing a flowchart to help us visualize and organize the control flow between our input and respective functions. After several iterations of the game flow done on whiteboards, we actually did create a brief flowchart to ensure that we understood the process and wasn't missing anything, but this was already halfway through the coding process and so would've been more time-effective if we had done this earlier. Finally, just as a minor point to the design section, it would also have helped if our group was able to gain a better understanding of design patterns such as MVC and factory design patterns earlier on during our planning process as that would've helped us create our first UML more effectively and the overall structure. However, this was an expected limitation as at the start of the project we also didn't have a lot of turnaround time from learning the design patterns in class to being able to properly implement it in our code. Elsewise, our team would say that we dedicated our time pretty well for this project, and after over two weeks of hard work and plenty of design iterations and debugging, we are proud of our final result!



Sample flowchart that we created and whiteboard iterations of structure