# Chess Report

By: Jessica Ding, Jiwon Kim, Lillian Mo

## Overview

Our team decided to make chess for the final project. The game works just as specified on the project guidelines, with one stipulation - as it is quite difficult to tell the difference between the uppercase and lowercase pieces in the graphic display, we decided to duplicate the lowercase letters as well, for ease on the eyes. Furthermore, we decided to add a unique touch to our graphic display by making the border red. Since the dark squares were green, we dubbed this special design “Christmas Mode.” A castle can only be called by moving the king two squares left or right—in other words, a rook cannot initiate a castle.

We primarily used an observer pattern to update the text and graphic displays, with our piece class being the abstract subject and the individual piece types (bishop, knight, etc.) being concrete subjects. Then, we created a Player class, where the Computer inherited from, and in turn Levels One, Two, Three, and Four inherited from the Computer. A general Player would be a human player, while different computers are objects of their respective levels. The movement of the game is handled in the Board class, which has a board that is a 2-D vector of Piece pointers (and null pointers where there is no piece), a Text Display (a 2-D Vector of chars), and a Graphics Display. The Board also keeps track of the White Player, the Black Player, the White King, and the Black King. All the commands are handled in main, to the standards demanded in the project guidelines. The more sophisticated Level Four algorithm prioritizes captures of pieces with higher “value” (we assigned Queen = 4, Rook = 3, Bishop = 2, Knight = 1), then captures of pawns/checks, and then evasion moves, before doing a random move if none of those moves exist.

## Design

Some design challenges that we faced throughout the duration of this project included that we initially tried to fit our model entirely into the Observer pattern. However, we realized that making all pieces observers of one another would result in clunky and inefficient code. Thus, we pivoted to making the pieces just objects. We kept the Observer pattern, but we only used it to notify the Text Display and the Graphic Display.

Further, in the original UML, we designed an aggregation relationship between the concrete observers Graphic Display and Text Display to "have" each of the concrete subjects, which are the pieces. However, when designing our concrete observer classes, we quickly realized that it was unnecessary to have a dedicated field for each piece, as we were passing in the piece as a parameter in the notify function anyways. This approach also allows each concrete Observer class to be more succinct, as an aggregation relationship would require the Observer to "have" up to 64 pieces at a time.

Our new approach also includes a concrete Player class from which the abstract Computer class inherits from. The old design assumed that a human player would be making their moves using the overridden versions of the "makeMove" functions in each piece, and a computer player would have separate "makeMove" functions entirely. However, we quickly realized that since both players must use a "makeMove" function, it would be reasonable to put the function in a superclass that each type of player would have access to. Thus, we created a concrete Player class. Since a computer is a player, we made the abstract Computer class inherit from the Player class. Inheriting from the Computer class are the four levels of the Computer player based on intelligence. We implemented an aggregation relationship so that the Board object "has" 2 Players. This allowed us to simply call the virtual function "makeMove" in the

Board class, which used the correct overridden version of the function based on the dynamic type: a Player, or a Computer level.

In addition, we initially had the Level One, Level Two, Level Three, and Level Four objects handle making a move separately in their own class. We soon realized that this would cause an issue with the code, because every successive level wanted to use the functionality of the level below it, should it not be able to find moves in its current level. One of our group members came up with a rather ingenious solution to this problem—in the Computer Class, implement levelOne, levelTwo, levelThree, and levelFour functions that each individual Computer object could call in their makeMove function. These functions would return bool values to indicate whether or not they were successful at making a move.

Another design challenge worth noting was the issue of "special" moves, such as castling, *en passant*, and pawn promotion. These moves often required special functionality that only pertained to specific piece types, such as pawns, kings, or rooks. Intuitively, we thought we should use static casting, but after going to office hours, we thought that, to make our code more resilient to change, we should prioritize making virtual methods where we can. We ended up only using one static cast throughout the whole program, to check if the Rook was on its first move. Otherwise, everything else was handled entirely through virtual functions that were overridden in the pieces that needed them.

Ultimately, the main way we chose to tackle design challenges was to first pinpoint where any problems may arise (high coupling, inefficiency, etc.) and then brainstorm multiple solutions for those challenges. We would then discuss the pros and cons of each solution, sometimes consulting an expert such as a professor. Then, we would come to a consensus and implement our improved design.

## Resilience to Change

Our code is fairly resilient to change, as there is very little static-casting. For the most part, the pieces are moved using virtual methods. There are functions such as “capturePiece” that allow the pieces to handle other pieces on the board, which could lead to some very interesting versions of chess. One that a professor had mentioned during office hours was that perhaps in one version of chess, if a certain piece gets onto a row, it captures every piece on that row. This is a change that could be fairly easily implemented with our current code.

Other changes that could be easily implemented with our code are adding more levels, because we separated the levels into different functions within Computer that can then be called in successive order if the highest level does not have any possible moves. To add a level, one would simply create a class inheriting from Computer and implement the virtual “makeMove” function. Since the levels inherit from a Computer class which inherits from the Player class, the function in Board that calls “makeMove” on each Player would not need to be changed at all, as it would use the correct overridden version based on the dynamic type of the Player. In fact, none of the other classes would need to be changed at all, indicating high resilience to this change.

In addition, creating pieces beyond the standard six chess pieces could be easily implemented, as each piece inherits from an abstract Piece class. This class outlines a template for the implementation of the new piece, offering virtual methods such as “checkMovementValid” and “checkPossibleMoves”, as well as concrete methods such as “checkAttackingMe” that the classes share. To add a customized piece to the game, the only change required is to create an additional class and fill in the implementations of the virtual methods.

## Answers to Questions

1. **Chess programs usually come with a book of standard opening move sequences, which list accepted opening moves and responses to opponents' moves, for the first dozen or so moves of the game. 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.**

We would hard code the standard moves from the book for both players in 2 different files respectively up to a specific point and then the computer would take over with generating its own moves based on its level for computer vs computer/player. When a move has been made, the opponent would compare the computer's moves with the standard moves in the text file and see if they are the same. If so, check that the standard opening for the current player has a higher win percentage historically. If it is, make/suggest its move based on the corresponding opposing move from the text file to continue with the standard opening. Otherwise, suggest a different move based on the level of the computer. The main function would need to be adjusted to read files as well as differentiate between end of file in the input and end of standard moves. Since main calls the makeMove function for all Players, none of the other functions would require a change.

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

If we only wanted to undo one move, we would simply have a field that recorded previous location in the piece. Furthermore, we would create a vector field in the piece that stores any pieces that this piece has captured. Whenever move is made, simply change that field

to store the current move. Then, if a previous piece was where the Piece moved was before, return that piece to the board and remove it from the vector field. Then, we would access the previous location. However, if we wanted to have an unlimited amount of undo moves, we would implement a stack by creating a linked list of moves, and then traverse through that linked list. This would allow for an unlimited amount of undos, because we could keep going through the linked list until we reached the desired undo. If I wanted to implement a redo, we would just traverse forwards.

*Note post-project:*

After having completed the project, our answer to this question changes slightly. First of all, we would have to not only store the "last move", but also the piece that was on the square that the "last move" moved to. Furthermore, for situations such as en passant and castling that are more intricate, the last move would have to find a way to restore both pieces involved in the move.

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 are following this version in our response: https://www.chess.com/terms/4-player-chess. First, we would have to modify the init function for the Board class to accommodate for setting up a four-handed chess board, which is not a perfect-square. Instead, we would have to implement a board that extends the standard 8 x 8 board to have 3 x 8 sections on each side. This would change our vector of vectors for the grid. The coordinates would also expand to include a-n and 1-14. Instead of having a boolean variable such as "whiteTurn" which would keep track of the truth value of the first player (white)'s turn to move, we would need to keep track of which

player's turn it is using 4 boolean variables for each player. Alternatively, we could keep track of the total number of moves that has been made using an integer variable and take the modulus of 4 to indicate which player's turn it is. Additionally, we would need to keep track of points based on the number of captures for each player. Another major difference is that kings can also be captured. Also, we must keep track of pawns that have been promoted into different pieces to update score properly.

## Extra Credit Features

N/A

## Final Questions

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?**

We learned that to effectively work in a team, the most important rule is consistent communication. Working with tools such as Gitlab allows for effective version control, but if teammates do not communicate about who is working on what in the main branch, it can often lead to nightmarish merge errors. Furthermore, while it is helpful to divide the work prior to starting the project, it is just as crucial to be flexible to changes throughout the process of actually working on it. As the UML changed, it became evident that some parts of the code that were split between team members were more closely related than we'd previously thought so we rearranged the tasks to appropriately reflect that. If we hadn't done that, it would've meant a lot of overlap in the code, which would have caused various merge errors. Managing a large amount of code in a team means that the entire group must be on the same page at all times. We found that, throughout the project, as we developed and enhanced our communication skills, progress on our code was streamlined and our coding experience was far less stressful and confusing.

In addition, we learned that when making big decisions, there is no room for egos. When we wanted to take a big step or change our code in a significant way, we always made sure that everyone was involved in the discussion and that everyone was able to express their point of view. Many times after hearing opposing perspectives, one of us would change our opinion and we would come to a consensus. When working with a group, emotional regulation is just as important as code regulation, and we took many steps to make sure that no one was unhappy or felt like their contribution to the group was being ignored. We also prioritized everyone's schedules outside of this project as well—we understood that everyone was busy in different ways, and we were able to find ways to not always work at the same time, but work in sync nonetheless.

One example of us prioritizing our mental health over the code was near the middle of the project, when we were experiencing many errors and segmentation faults. Debugging was heavily demoralizing for the entire team, and so we had the idea to make our graphic display have a red border, and write "Christmas Mode" on top. While it may have seemed like a waste of time, we found that the more visually engaging display boosted morale, letting us finish our tasks more efficiently.

Developing software in teams is a little like making a meal with your friends—everyone is working on something different, but there are many opportunities for clashing, and it is important to keep up constant communication. The only difference in this scenario is that the clashing is not putting things that require different baking temperatures into the same oven, or bringing the same ingredient multiple times—it's overloading a function that your teammate already overloaded, or forgetting to git pull before you edit a module. Ultimately, though, if everyone priorities teamwork and collaboration, the result will be a beauty to behold.

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

We probably would have started with using unique and smart pointers instead of beginning with delete statements and trying to replace everything with unique pointers. Going back through hundreds of lines of code made it quite difficult to know what should and shouldn't be changed. That being said, the reason we decided to start with using delete statements was because our group was still fairly unfamiliar with using shared or unique pointers when we started the project, and we were afraid that if we started off with those pointers, we would end up being snakebitten. Ultimately, after we resolved the memory leak errors that we were getting, we came to the conclusion that our overall progress was helped by the fact that we started with code structures and tools that we were familiar with.

Intuitively, one may infer that we would've preferred to have started with a correct UML instead of changing it as we went along. However, we believe that it was impossible for us to reach the UML we had without getting our hands into the code. As mentioned in the previous response, flexibility is important. In this way, our project was an exercise in Agile software development, as we were continuously testing and designing our code as we were developing. Thus, while our end result deviated from our original plan of attack, we cannot sincerely say that we have many regrets for this project.