Denise Goping, Kevin Lee, Sebastian Villate

Design

Introduction:

Our group decided to implement the game of chess. Along with the standard game rules and project specifications, we added extra features wherever we saw a good opportunity. The following is a documentation of our design, the problems we encountered, and our solutions.

Overview: (the overall structure of the project)

The main classes that we created for this program are:

- Game, Board, Cell (declared within board class), Piece, Player, Move
- Pawn, Knight, Rook, Bishop, Queen, King (all inherit from Piece),
- ComputerPlayer
- ComputerPlayerDiff1 (inherits from ComputerPlayer)
- ComputerPlayerDiff2 (inherits from ComputerPlayerDiff1)
- ComputerPlayerDiff3 (inherits from ComputerPlayerDiff2)
- ComputerPlayerDiff4 (inherits from ComputerPlayerDiff3)

Using our main.cc file, the program asks for user input indicating setup mode or game mode. For setup mode, the board adds the specified pieces accordingly by creating a new piece object (either Pawn, Knight, Rook, Bishop, Queen, or King depending on user input). It adds this piece to the designated player's vector of pieces, which is stored in a board attribute called playerMap. playerMap is a map, with the player's colour as the key and the vector of their pieces on the board as the value. Once the piece is created with its x and y coordinates in the desired spot, the associated Cell on the board is also updated to hold this piece. Before exiting setup mode, the game checks to see if any Kings are in check, if there is exactly one King for each player, and if there's no pawns on the first or last rows of the board.

For game mode, you can either have human players or computer players from levels 1-4 (or a combination of both). Once these are entered and initialized by the game, the main aspect of this part of the program is the moving of pieces. Below is the line of function calls done in order to perform a move:

- Game.makeMove()
    - Sends the start and end coordinates to the board, along with the colour of the current player
    - Board.move()
        - shouldRender will tell the function if the move is being shown to the players or not. This is because there are times where board.move() is used for testing the future conditions of the board after a hypothetical move, and then undo() is called to revert the state back to its original form, which also takes in a shouldRender boolean.
        - Finds the piece on the start coordinate of the board
        - piece.isPathValid()
            - Checks if the move is valid for the piece if the board is empty
        - If piece.isPathValid() returns true, board.validateMove()

- piece.generatePath()
    - Generates a vector of coordinates showing how the piece moves from the start coordinate to the end coordinate
  - Checks that there is no blocking piece on each coordinate in the path of the move
- If board.validateMove() throws an error, catch and rethrow back up the call stack to game.makeMove(). If not, board.movePiece() or board.moveAndRenderPiece(), depending on shouldRender
  - Moves the piece to the cell located at the new coordinate
  - Changes the piece's x and y attributes to match the new coordinate
  - If shouldRender, will also remove the render of the piece in the old position, then render the piece in its new position
- board.isInCheck()
  - Checks if the new move puts the current player's king in check
  - board.movePiece() puts the piece back to the start coordinate
    - cell.movePiece()
      - Moves the piece from and to the designated cells on the board, called by board.movePiece()
  - If true, return, if not, throw error that piece prevents check
- If shouldRender, for all opponent players, call board.isInCheck() on their pieces
  - If this returns true for a player, call board.isCheckmate()
    - If board.isCheckmate() returns true, call board.removePlayerPieces() to remove the player from the game
  - If not checkmate, call board.isStalemate()
    - If true, remove all players from game and end game
  - If neither checkmate or stalemate, call board.isInsufficientMaterial
    - If true, remove all players from game and end in a draw
- At the end of call, if move has been validated in all aspects, call board.moveAndRenderPiece() which will move the piece and then draw the changes on the graphical board

- Loops through the players still in the game, and if a player is in checkmate, removes them from the game. When one or no players are left in the game, the game ends.

Below is the typical line of function calls done in order to choose a computer player move (varies slightly per level):

- computerPlayer.chooseMove()
  - Loops through all of the player's pieces
  - piece.generatePossibleMoves() creates a list of that piece's possible moves when the board is empty
  - For each move in the list, call board.isMoveValid()
    - If this returns true, check criteria for the order of prioritizing moves based on the level of the computer player. If this move meets the criteria, return the move.

    - If no move can be found for this level, move down to the next level's chooseMove function (level 1 is random legal moves, so will always find a move that fits the criteria since the player will not be in a stalemate when this is called).

Computer player level 2 selection logic:

- Top priority is checkmate
- Prioritizes moves that take pieces + check (takes the highest value piece that allows this)
- Prioritizes piece takes and checks equally (always takes highest value piece possible)

Computer player level 3 selection logic:

- Top priority is checkmate
- Tries to save pieces in danger, starting with its highest value piece (saves best possible piece)
    - If only 1 piece is targeting the current piece it is trying to save, it tries to save the piece by taking it (if it is either safe to do so, you are taking a higher value piece from your enemy than the piece you are putting in danger, or you are sacrificing a lower value piece to save a higher value piece when the high value piece has no other way of escaping)
    - Tries to move the piece to a safe square that is not in range of the enemy, while maximizing the value of the piece it takes/ checking the enemy in the process
- If no piece that is in danger is savable, it tries to make a safe move with a piece that is not in danger
- If no safe move is possible, use Computer player level 2 logic to execute a purely offensive move.

*Note: Computer player level 3 will never make a move that puts a piece in unnecessary danger (moving a piece to a position where it is capturable, exposing a valuable piece while moving another one), unless it is to sacrifice a piece in order to save a better piece, or to make a positive trade on an enemy piece.

Design: (the specific techniques used to solve the various design challenges in the project)

During the course of this project, we came across many challenges with respect to making decisions about our design. In general, we prioritized high cohesion and low coupling, and tried to connect the different classes in a way that made sense.

Looking first at high cohesion, we made the classes only have attributes/methods that are related to the intention of the class. As a simple example, we have a Player class. The Player class holds all of the information that a player in real life would have: their current score, their pieces that are still on the board, and their King's current position. The player itself does not store information about things like moves, since that's related to the pieces and the board. As a more complicated example, the Piece class/subclasses only have attributes/methods related to the actual piece. Looking at their attributes, they keep track of their current x and y position, their colour, etc. Meanwhile, the board unites all aspects of a game and keeps track of the relations between pieces, cells and players, using the pieces' validation and adding to it by using the entire board state. Looking at the pieces' methods, they can generate a list of the piece's valid moves, generate the path that the piece would have to take given a valid move, check if a path is valid based on the piece's move rules, etc. These are all things that are related to the intention of the piece, since something like the board or the player should not keep track of each piece's valid move rules.

On another note, we aimed for low coupling, making it so that a change in one class should have no effect on the other classes. The classes interact with each other via function calls, containing basic

parameters and results. In other words, one class does not directly manipulate the attributes of another class. Whenever possible, attributes are kept as private or protected to avoid unwanted usage. Similarly, if a class uses a form of helper function that should not be accessible by outside users, the method is kept private as well. This maximizes the independence of the classes. Whenever an attribute of a class needs to be obtained or updated by an outside class, accessors and mutators are utilized. This enforces encapsulation and invariants. There are also a few cases where classes need access to another class's information. In these situations, classes are friended to each other.

In general, we tried to avoid repeated code as much as possible. If certain logic were to be carried out in more than one place, we would make a separate function for those steps. This way, if we needed to change part of that logic, we would only need to update it in one place. This minimizes many potential errors that can occur if we were to update most instances of the logic, but forget to change others.

Furthermore, inheritance was used for pieces and computer players. For pieces, there is an abstract parent class of Piece. We made this abstract since you can never initialize just a "piece," it will always fall into one of the subclasses. The subclasses override the virtual methods and cater towards the specific piece's rules. For computer players, there is an abstract parent class of ComputerPlayer. In a similar manner to pieces, this is abstract because you can never initialize just a "computer player," you must always specify the level as well. Each level inherits from the level below it (level 4 inherits from level 3, level 3 inherits from level 2, level 2 inherits from level 1). This is because each level uses the functionality written out in the level below it, but also adds more features. By having them inherit from each other in this way, repeated code is avoided.

Once we began coding, we had to make many small changes to our initial plan of action. Any additional attributes and functions are represented in the updated version of the UML diagram. One of the biggest differences in our design is the addition of a Move class, implemented mainly for undo functionality. The Move class contains pairs in the form of <startX, startY> and <endX, endY>, signifying the start and end positions of a single move for a given piece, as well as a multitude of boolean values to create a full image of what state the moved piece was in before the move was made. These values especially help with undoing special moves such as pawn promotions, castles, and pawn double forward moves, and ensure that the piece is returned to a state where those special moves can be made again. The Move class also holds a pointer to Piece object, which stores a taken piece on a given move if there is one. Every time a move is made, a new Move object gets created based on the properties of the move, and it is added to the Board's vector of moves.

Due to the fact that it is possible for a checkmate to occur and not end the game in 4 player chess, we also had to add a vector checkmatePieces in the Move class which would hold all pieces belonging to a checkmated player, so that we know which pieces to change back to active should a checkmate move be undone.

Resilience to Change: (how the design supports the possibility of various changes to the program specification)

For the organization of this project, we split up the logic into several classes. While the classes do interact with each other to connect the elements of the program, the main logic/heavy lifting for any class is within its own class file. Therefore, if a change is to be made for a specific aspect like the board, a piece, a computer player, etc, the scope of this change will mostly be within the functions/attributes of that singular class. This is due to the high cohesion of the program, as the classes only have methods that

are related to what the class should be doing, and the low coupling of the program, as a change in one class will usually not affect the other classes. It's also important to note that within each class, there are many functions that are each meant to carry out one specific task (specified by the name of the function). By splitting up the code this way and having the elements separated in a way that makes sense, it's easy to reuse and update code.

There are many ways in which our program can account for changes to game rules. Looking at the example of 4 player chess, the players are stored in a map with their associated colour as the key for lookup, as opposed to having hard coded variables for 2 players' information (more details on this described in the "Answers to Questions" section). This allows for the introduction of any number of players, since the board functions know which player to look at based on the colour of the current player. Thus, if we needed to implement a version of chess with a non-standard number of players, we can simply add that amount of players to the map and carry out the logic as it normally would.

Our design is also useful if there were to be a change to the rules of move validity. Let's assume that we wanted to change the criteria for a valid move for some of the pieces. For each specific piece that this change would apply to, only the functions *piece*.isPathValid(), *piece*.generateValidMoves(), and *piece*.generateValidPath() must be changed to account for this (where *piece* is replaced by the actual piece name). *piece*.isPathValid() takes a pair of start and end coordinates, and determines if the path from start to end is a valid path given the piece's move rules. *piece*.generatePossibleMoves() generates a vector of all the cells a piece could theoretically reach (without taking into account possible checks). Lastly, *piece*.generateValidPath() generates a list of the coordinates of the path that the piece would take given a valid move. All 3 of these functions need to know the rules of a piece's valid move, and once these functions are updated, the rest of the program will be able to behave the same way. If new rules are introduced for this move to be valid (for example, if it's only valid if some precondition is met like in en passant), an indicator of this condition must be introduced in the piece's class such as a boolean that keeps track of the state of the piece. This is done since it is not enough for move validation to check if the desired move follows the path of the special move, but it also must know that the piece satisfies those preconditions beforehand. We would also require this in the Move class for the move to be undoable, since we need to know if the state of the piece before any move also satisfies the condition. Altogether, the program nicely accommodates the possibility of changing the rules of move validity for a piece, since each piece has their own polymorphic functions that work together to calculate move validity. Hence, it's organized in overridden functions that are specific to each piece, and so code doesn't need to be changed in too many places.

As another example, there may be a change to the input syntax of the program. This would be a simple change to the accepted input described in the main.cc file. If a new command were to be implemented, a new if statement would be added checking if the input string matches this new command. The logic that would go inside this if statement would depend on the command. An example of this are the "draw" and "undo" commands that we added. By adding an if statement within the if block that starts a game, the user can input these commands and the logic is carried out accordingly. At the end of each command within the game, it is always important to remember to either advance the turn to the correct player, update the number of players if someone is out of the game, or set the current turn back to the initial player and reset the board if a game is over.

There could potentially also be a change to the board size. We came across this example while implementing 4 player chess. To account for this, we implemented a Cell class, and the board is a vector of vectors of Cell objects. This allows us to dynamically adjust the board size, and by setting certain cells

as invalid, we can change the shape of the board. This would be done during the initialization of the board, by feeding it the desired dimension of the height/width.

Finally, there's the case where there is a completely new feature. Given the nature of the context of chess, many of the elements needed for new features should already exist within our code. As examples, we already have functions that check for move validity, generate a list of possible moves for a given piece, check if a piece is in checkmate, etc. So for the implementation for a new feature, aside from the main function that will work with/store the information, the little pieces should mostly all be there. An example of this is the undo feature that we implemented (described below in the "Answers to Questions" section). This was a completely new feature, but it utilizes some of our pre-existing functions such as Board.move(). Thus, by putting our pre-existing functions together in new ways, and adding whatever new methods/attributes that are necessary, the process should be relatively smooth. Minimal to no things should need to be changed, there should be more of an emphasis on adding things. The only files that should be changed would be the file the change is in (whether it's a board function, piece function, game function, etc), and possibly the main file if a new command is needed.

Overall, there are many ways in which our program is resilient to change. If a change to the program specification is presented, minimal files should have to be recompiled.

Answers to Questions: (from the project specification)

Question 1: Standard Opening Move Sequences

Our first objective was the same as in our answer in DD1, except the data structures used are now different and much more cohesive. We created a vector of vectors representing different openings where each subvector is an opening, containing pairs of pairs of integers, with these pairs representing start and end coordinates for a move. From a database of openings, we convert lists of chess coordinates to lists of valid moves in our program and store them to compare moves to later during a game.

We then also keep a list of all Move objects (which is already done for undoing to work), and as moves go on, compare if the last move deviates from any of the provided openings. If it does, do not consider that opening for the next move and move on to the next opening until a match is found. This way, computer player can know what standard openings can still be followed, and each succeeding move, it will check again if the opponent's move fits into any remaining openings

Question 2: Undo Moves

Our first solution for undoing moves was to store the start and end positions of each move in a vector, then "pop" a pair of coordinates each undo, moving the piece located at the end position back to the start position, as reflected in our answer to this question in DD1. The problem was that this initial solution did not account for special moves such as castling and promoting a pawn, nor did it account for taking pieces and how to place those pieces back on the board. Special moves that were made should still be possible after undoing them, meaning the conditions that made a special move possible should still be met after an undo, and other pieces that were taken should reappear in the state they were in prior to the move. To properly implement an undo feature, we had to find a way to save the state of all pieces involved in any given move prior to the move being completed.

Our solution was highly intertwined with the implementation of the Move class, described in the *Design* portion of this document. By using boolean flag attributes and pointers and vectors to modified pieces, we could effectively store the before-state of whatever pieces would be affected by a move, then

look over those details when it came time to undo that move so that the correct changes would be made to revert it. As an example, we can check if the piece is a pawn making a double first move, if it is a promotion move, if it is the piece's first move, etc.

When a piece is taken, we simply move ownership of its shared pointer from the Board and the Player object to the Move object's takenPiece field. This way we avoid having to delete the piece when it is taken, we just move ownership back to the Board and Player, and it remains in the state as when it was taken. Once the ownership is moved back to those two objects, the Board will be able to "see" it is present and render it, and the player has control over it again.

Another hurdle we came across was the fact that in 4 player chess, a checkmate does not necessarily mean the game ends, and so moves that put a player in checkmate should also be undoable. We solved this by including the checkmatePieces vector (containing Piece shared pointers) in the Move class, also described in *Design*. It functions similarly to taking a piece. We do, however, keep it stored in the Board object, since we need the pieces to still be rendered (but as gray, inactive pieces now). When we undo and see that this vector is non-empty, iterate over the elements and move them back into their respective player's pieces vector. So, it is possible to undo a 4 player chess checkmate, giving control back to the removed player and re-activating their pieces.

Question 3: 4 Player Chess

To implement 4 player chess, we closely followed our plan from DD1. The key factor was being able to store/access the information for 4 players using the same logic as for 2 players. Thus, we could not make the assumption anywhere that there would be exactly 2 players. To solve this, we first utilized the Player class that keeps track of its score, pieces on the board, and king position. Instead of hardcoding in attributes for the 2 players at the beginning of a game, which was our original plan, we created a map called playerMap that stored shared pointers to each player object with its designated colour as the key. This way, when we need information from a Player (such as one of their current pieces for a move), we can look up the player's colour in the map and can easily access their information. This solves our problem as the players are now stored together in 1 attribute, allowing for any length of players to suffice.

Another change was the display of the board itself, as 2 and 4 player chess have different board dimensions. To account for this, we created an isFourPlayer boolean in the Board class that checks if the board should be for 4 players, and it gets initialized/rendered accordingly. As mentioned earlier, the board is a vector of vectors of Cell objects, allowing us to dynamically adjust the board size, and by setting certain cells as invalid, we can change the shape of the board. Also, for 4 player chess the pieces are represented by the player's colour's first letter and the piece label, such as Gk for green king. This allows us to make the pieces differentiable when using text based rendering.

When a player gets removed from the board via checkmate or resign, their pieces are removed from the Player class's list of pieces. When only one player in the playerMap still has pieces on the board, they are the winner (the game no longer ends on the first checkmate). One additional detail that we added that we did not discuss in our DD1 answer is what happens to a player's pieces after they are eliminated. We decided to keep the pieces on the board, but set them as inactive (since we removed them all from the player's piece list). This follows what is typically done in 4 player chess games, and it is done because if one were to remove the pieces from the board, another player may suddenly be put under check by clearing a pathway. This also factors into the undo of checkmated moves, which is described above in the Undo section.

Extra Credit Features:

- **4 Player Chess**
    - Along with the standard version of 2 player chess, we also created a feature that allows you to play 4 player chess. All of the same features and rules from 2 player chess are applied here. At the beginning of a game, you are prompted with a message that says "Please enter the number of players (2 or 4)." Once you enter a number, the associated number of players are initialized and the corresponding board is created. At first this task seemed daunting, as introducing 2 more players to an already complicated game does not sound simple. However, we implemented each feature in a way that minimized the amount of changes we would need to make. A detailed description of the problems we ran into and our solutions is provided above under "Answers to Questions: Question 3: 4 Player Chess"
- **Undo feature**
    - As you're playing a game, we implemented a feature that allows you to undo however many moves. By typing in the command "undo x" when it is your turn, where x is replaced by the number of undos you want to make, if all players agree then the game goes back to the state it was in x moves ago. A detailed description of the problems we ran into and our solutions is provided above under "Answers to Questions: Question 2: Undo Moves"
- **Standard Opening Move Sequences**
    - As part of the algorithm for computer player level 4, the computer player uses moves from a list of possible standard opening move sequences, instead of just randomly choosing moves like in the lower difficulty levels. The openings are split up into "reactive" and "setup" openings. Reactive openings branch out from a defined starting point and cause the computer player to make different, predefined moves depending on different opponent moves. Setup moves are the same no matter what the opponent does. This allows the computer player to develop its pieces with structure. This way, once no more openings are available, it is more likely that the computer's pieces will be in a better position to attack and make use of the rest of its decision algorithm. It sort of segues it into a new phase of decisions. A detailed description of the problems we ran into and our solutions is provided above under "Answers to Questions: Question 1: Standard Opening Move Sequences"
- **Draw By Choice + Draw by Insufficient Material**
    - We implemented a feature that allows players to choose to tie the game. Ideally this would be used when the players have reached a point where it is clear it will be a stalemate/draw, so they wish to end the game now to save time. By entering the command "draw", if all players still in the game agree to tie the game, all of their pieces are removed from the board and the game will end in a draw.
    - We also implemented a feature that detects a draw by insufficient material. According to *Chess.com*'s *Rules of Chess*, a draw by insufficient mating material occurs when both players in two-player chess only possess either a lone king, a king and a knight, or a king and a bishop.
- **Zoom Ability**

    - In order to have a better view of the graphical board, we decided to add a zoom feature. To do this, we needed a zoomFactor attribute within the board, which is modified by using board.zoom(). board.zoom() will then set the new value of zoomFactor, and the values of all graphical dimension attributes to their base values multiplied by the given factor. All draw and render functions had to now also include the zoomFactor to multiply x and y values by. This can be seen in board.renderGraphicBoard() and overridden pieceSubclass.draw() functions. board.zoom() only accepts an integer that will not cause the height or width of the board to exceed 500px, the width and height of our Xwindow.
- **Computer Player Level 4**
    - The algorithm for computer player level 4 goes as follows. A vector of collections of moves representing standard openings are stored. Then, the computer player compares all made moves to stored openings until one matches the current move order, and it will make the next move that follows within that opening. If no standard opening matches, it checks to see if any of its pieces are under attack, if it can capture any enemy pieces, or if it can put any enemies in check or checkmate. If so, it uses the algorithm for choosing a move used by computer player level 3. If not, it will calculate all of its opponents' moves one move ahead to see if any enemy moves can cause a check, then will make the move that best prevents this from happening. If no move has yet to be found at this point, it will iterate through all of its own possible moves for each piece, then check which move will allow the most squares on the board to be controlled by its friendly pieces.
- **Memory Management**
    - The entire program is implemented without leaking memory, and without explicitly managing our own memory. Memory management is done via smart pointers and STL containers, as expressed in our UML diagram. In other words, there are no new or delete statements anywhere within our program. The only time raw pointers are used are if they are not meant to express ownership. To prevent having to change a lot of our code, we used this approach right from the beginning of the project.

Final Questions:

a) What lessons did this project teach you about developing software in teams?

This project provided many lessons about developing software in teams. One of the biggest takeaways is how crucial proper communication is between your team members, to ensure that everyone is on the same page. Since different people are in charge of different areas of the code, frequently discussing the functionality of each part of the project can minimize any conflict within the code itself. In other words, if you don't fully understand what code that someone else wrote is supposed to do, you won't be able to use it properly and connect the areas of the project together. On top of communicating about the actual code, it's also important to have group discussions about the big picture of the project. Explicitly planning out how the elements will interact with each other, the order of the function calls, etc, will give everyone an idea of what the end goal will be and what steps to take to achieve this. Time management and organization of ideas go hand in hand with this.

Throughout the course of this project, we developed a routine where we would meet every morning to discuss issues that we came across in the previous day, how to fix them, and what we would

be working on for the current day. This provided some structure to our weeks, and was an easy way to keep everyone in the group up to date with your progress.

Another lesson we learned was the importance of code reviews. We worked on separate branches, and whenever we finished an element/function we would set up a pull request and merge our branches into the main branch. This allowed us to try things out on our branches without making any detrimental errors to the main branch, since there were multiple people working on this project and conflicts could easily arise. The more pull requests we made, the better we got at reviewing each other's code and finding errors before merging to main.

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

If we had the chance to start over, we would place a higher priority on the documentation of code from the beginning. Given that we each worked on different sections of the project, we often had to look at another person's code to connect elements together or troubleshoot errors. However, in most cases, immediately adding comments to our code to explain what they do/what certain lines do was an afterthought for us. This made it difficult to understand how specific pieces of the program worked, and we often had to consult each other for clarifications. If we were to start over, we would add comments describing key parts of the code as soon as we write them, making it easier for the other group members to follow along. Aside from this, having an up-to-date summary of what the bigger, more involved functions should do (written as an algorithm instead of code) would also ensure that everyone is on the same page.

Another thing that we would change is how we organized some of our functions. Given the complexity of the rules of chess, we would often think of new cases that we had to account for after a lot of the code was written. To solve these cases, we would create a new function that carries out the specified logic, and connect it to one of the bigger pieces. The result of this was several functions that may have similar roles to each other, but each was created for a different case. An example of this is our Board.move() function. When moving a piece on the board, we had to create helper functions such as isCheckmate(), isTargeted(), isReachable(), isInCheck(), isOccupied(), etc. While it does make sense to have different functions for each of these goals, a lot of the time the logic carried out within them is similar. Thus, it would be more efficient to have a higher level function that carries out the common logic, and then these functions would manipulate this result accordingly. Therefore, if we were to start over we would try to make some of the functions more abstract so we could reuse them more often, preventing large amounts of our functions from doing similar things.

Conclusion:

Overall, this project was a huge learning experience for all members of the group. By working together and connecting our elements of code, we were able to create a functional chess game. While there were long nights of looking for the tiniest of bugs that would crash our program, there were also many times where we put our brains together and thought critically about how we could implement the next feature while abiding to the best practices of object-oriented design.