Overview

Our chess program could be best described to have been divided into 3 sections, where each controls a specific part of the functionality.

1. Pieces and Board
   The core of our program are the Piece and Board classes. Pieces are the primary objects that are handled in the game. A board hosts all the pieces needed by the Chess game via unique pointers. Thus in its constructor, it creates all 32 pieces that are needed to set up a Chess game. These two classes have a composition relationship.
2. Players
   The players interact with the board in order to take turns making moves during a Chess game. There are two primary types of Players, humans and computers. Humans can choose what move to make by simply typing it in using standard input. Computers however, require access to the state of the board in order to generate all possible moves in order to decide what move to make.
3. Game
   Game is the class that handles most of the control flow involving actually playing the game, such as taking turns to move, checking if there's a winner, etc. In addition, Game also handles if the players want to play with a custom setup interacting with the users to determine what setup they want to use for their game. Game also hosts a vector of Observers, thus the displays for the board are updated whenever the observers are notified within the game.

Design

General Comment

In general, we tried our best to adhere to the MVC structure when creating the program. The "Model" in our program could best be described by the board and pieces, as it is where all the game logic involving the movement of the pieces is handled. In addition, the board is where the "state" of the game is stored. The "View" portion of our program is handled by the Text and Graphical observers. Finally, our Game class could be considered our "Controller", as it links the Board and Observers by acting as the "Subject". This is the main area where the players interact with the program, then the display updates based on the player input.

Implementation of the Pieces

There are three important parts to every piece: the directions they can travel in, how far they can travel, and the next positions they can travel in. Each piece inherits from an abstract class, Piece, that holds this information. Within each child class, if that specific piece has unique behaviour, we override the method provided by the base class. A perfect example of this is how a pawn captures (it captures on the diagonal and not in the direction it travels - which is up). This pawn capture is handled in pawn's overrided generateNextPositions() method. Another one of Piece's important methods is movePutsKingInCheck(), which checks if a player's move puts

their King in check. It creates a copy of the board, modifies it with the move they are attempting to make, and checks if the opponent's pieces can capture their King.

Implementation of the Board

The most important role of the Board is to keep track of the state of the game. This was done via a 2D vector of unique pointers of Pieces. Every time a move is made, the 2D vector would update the index of the Piece pointer. To make these updates, there exists a method called changeBoard() in the class that passes in a Move object. A Move object is simply a wrapper around all the information that is relevant to a move action, such as current position of a piece, new position, what the piece is, etc. Once the Move object is passed in, changeBoard() determines whether or not it is considered a legal move, then updates the 2D vector appropriately.

changeBoard() is also overloaded to enable other behaviours for changing the board. Specifically, for when the board must be changed for customSetup(). The overloaded versions of changeBoard() deal with removing a piece off the board, as well as moving a piece's default position to a new spot on the board.

Finally, another responsibility of the Board is to check whether the current state of the Board is in check, checkmate, or a stalemate. These methods are fairly simple. Firstly in Board, pointers towards both King pieces are stored for easy access, and the pointers are updated whenever the King is moved on the board. Then, we simply need to check the current status of the King (by calling its methods) to check if it is in check or checkmate. For a stalemate, we simply have to check if all the pieces of a player have no legal moves. In the game flow, since we have already checked whether or not the player is in check, this does not need to be checked again in our stalemate method.

Implementation of the Human and Computer Players

The implementation of the human and computer players posed a significant challenge.One strategy we could have used would be to create an abstract class called Player, then have Human and Computer extend it. As the program runs, the number of wins for each colour are continuously being tracked. Ideally, we would be able to save the number of wins for each player as a private field. However, if we were to implement it as such, then we would have to create a new Player pointer for every game that is played. For example, if the white player played the first game as human, but the second as a computer, we would have to delete the Human player to create a Computer player. This would not be ideal as all the data for all the players would be lost. Instead, we need to implement players in a way so that we can change the behaviour of a player between Human and Computer without actually deleting the pointer.

To solve this issue, I took inspiration from the pimpL idiom and separated the behaviour of Player from the actual Player class. Instead, I created a concrete Player class, but an abstract PlayerImpl class. Then, Human and Computer extend PlayerImpl. Player owns a pointer to a PlayerImpl, so that it can freely change its behaviour by simply switching between setting Human or Computer behaviours. The primary purpose for this design was to solve the problem

of inputting a move during a game. Thus, the primary purpose of PlayerImpl is to control the behaviour of the method chooseMove(). In particular, Human implements chooseMove() by taking in a string from standard input in converting it into a Move object. On the other hand, Computer has a private pointer to the board so that it can generate all its possible moves, then choose from the given map.

Furthermore, Computer itself is also an abstract class, as there are different levels of Computer. Our program has 2 levels: Level One and Level Two.

In terms of control flow, in main unique pointers for every type of PlayerImpl are created. Then, Player unique pointers are created and pushed onto the Players vector. Given what types of players the user has inputted for the game, we set the “behaviour” pointer for each player with the appropriate PlayerImpl pointer (using get() to retrieve the actual pointer). Thus when a player makes a move, the pointer will use polymorphism to determine which version of chooseMove() to call. Then for each game, we simply have to change what the Player’s PlayerImpl pointer is pointing to. Thus, we can change the behaviour of our Player’s without ever having to actually delete them, This means we can safely keep track of the number of wins for each winner, and print it when the program ends.

Implementation of the Text and Graphical Observers

One method we could have used to approach the user-interaction portion would be to have all the output that is displayed to the user in one class. However, this approach would cause several issues. Firstly, since much of the output depends on private fields in Board and Piece, this would mean the class would require information from several different classes, thus increasing coupling. Secondly, this approach would not make sense given user-input is conditional on the type of the Player: Human or Computer. Thus, the output has been encapsulated into different classes in order to increase cohesion. Each class is responsible for the output/input that is dependent on their given fields.

Firstly, the output to display the board is divided into the TextObserver and the GraphicalObserver. Since a pointer to Game is passed into both classes, the observers can print out each square on the board in the appropriate manner by calling the Game method getState (which returns the piece or lack thereof at that square).

Secondly, the user interaction for playing a game or customising the board setup are divided into two methods in Game. Thus, depending on whether the user wants to play a game or setup the board, the appropriate game method just has to be called in main.cc (which also has its own command loop).

Finally, the Human is responsible for interacting with the player to receive a Move, as outlined above.

As stated before, the visual display for the game is handled by using the Observer pattern. This was done by having Game extend an abstract Subject class. Thus, Game inherited a vector of

Observer pointers. Although this would decrease encapsulation in the sense that the Observers would have access to information from Game, which had access to information from Board (specifically the “state” of the board), this was necessary to allow for the modularization of our UI.

Resilience to Change

While designing our chess game, we tried to modularize our program as much as possible to ensure low coupling and high cohesion.This way, new features could be implemented without having to change too much code. There are several changes that could potentially be made to our game of chess. This is how our design could potentially the following modifications:

Changes to Piece Behaviour - Legal Moves

If we were to change what is considered legal piece behaviour, we would have to change the vector of Directions a piece is allowed to go in, as well as update the Distance the piece is allowed to go in.

For example, what if we wanted to have bishops behave as rooks and rooks behave as bishops? A change such as this could be implemented very simply. All we would have to do is change the Directions that are in the Bishop and Rook’s directions vector. Since Direction is an enum, this would be a simple change that would require Bishop and Rook to be recompiled.

Another example is if we wanted to define a completely new type of move behaviour. For example, what if we wanted to allow a piece to be able to move one square up, then one square to the left? The first step to implementing this change would be to define a new Direction enum to name this behaviour. Then, we would have to add code to Piece’s allPosInDirection() method so that we can collect all the possible new positions of the Piece given its current position and the new Direction. Once this has all been implemented, we would just have to add the new Direction to our desired piece’s vector of Directions.

All in all, since piece behaviour is centralised into the Piece class, any changes we would like to make to legal moves will not have an effect on other source code. Given the low coupling, we are able to make modifications to piece behaviour without needing to recompile large amounts of code.

Changes to Game Rules

As an example, let’s say that a new rule was implemented that allows players who caught an opponent’s piece to play again. To implement this new rule, a few changes would need to be made in more than one class. However, if these changes are quite simple and if implemented, they would allow for great flexibility for other rule changes as well.

To implement the new rule, we would have to know when a Move causes a piece to be captured. As of now, Piece has a method which generates a map of all the possible moves,

which pair to the corresponding type of Move (capture move, castling, etc). We could simply add a MoveType field into the Move class and add getters/setters. Thus when the possible Moves are generated, we set the appropriate MoveType.

With this change, the other modifications are relatively simple. More specifically, we would only have to modify the playGame() method in Game, which handles the game loop. When Game calls Board's changeBoard(Move move) method, the Move object's MoveType will be implicitly set. Thus, we can simply check using a conditional statement if the MoveType is Capture. If it is, then we can make sure the current Player stays the same by keeping the index of the vector of Players constant.

In general, any changes to the rules of the game would entail mostly changes in Game's playGame() method.

Change in Syntax for Inputting a Move

To change what constitutes valid user input for making a move, the only class that would have to be modified is the Move() class. This is because all the logic for figuring out whether the input made by a Human Player for a move is valid is done within this class.

The methods in Move that would potentially have to be modified are the ones responsible for converting the string into a Position, and the method that is responsible for populating Move's field using the user's input. These methods are mostly just conditional statements to determine whether or not the user input is formatted correctly. Since the verification of the user input is centralised in this one class, the changes made here would not have any effect on the rest of the source code.

Addition of New Commands

Another way that we could modify the program is if we were to add new commands that the user could make. As an example, suppose that we had a command called "switch mode" that allowed for a Player to switch between playing as a Human or Computer during a game. Thanks to how we structured our `Player` classes, no changes would have to be made to those files. Instead, we would have to add a conditional statement in `Game`'s `playGame()` method that would check to see if the user has entered the "switch mode" command. Then, we would use the `Player`'s `setBehaviour()` method to change its `PlayerImpl` pointer. Afterwards, we would ensure the player doesn't lose their chance to make a move by keeping the index for the vectors of Players constant for the next turn.

In general, the type of command would affect which files must be modified and the extent to which they must be modified. However, the way we have designed our classes means that for the most part, only the actual game loop would need to be modified.

Final 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. Although you are not required to support this, discuss how you would implement a book of standard openings if required.*

To allow a list of accepted opening moves and responses to opponent's moves, I would implement 2 maps within the Game class, one for the Black player and the other for the White player. Each map would map a vector of Move objects to a Move object where the keys represent the sequence of Moves for a specific opening and its corresponding value represents the proper opponent response.

Also within the Game class, we would have two fields storing each black's last moves and white's last moves, which are each a vector of Move objects. At a player's turn, we would see if the opponent's lastMoves matches a key in their oppeningMoves map and if so, take the value that's returned and perform that Move.

As soon as a player breaks out of a standard opening, the current sequence of last moves and any future sequence of last moves won't exist as keys of the oppeningMoves map. If this is the case, the game resumes normally and the opponent player selects their next move as they usually would.

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

To allow a player to undo their last move, a fair number of modifications would need to be made to my design. Firstly, the `Move` class would need to be able to store much more information about a Move. For example, on top of what it already stores, `Move` would also need to keep track of whether or not a piece was captured on that move, the pointer to the captured piece, and as stated in the above section, it would also need to store the `MoveType`.

Once Move has been updated to contain all these additional fields, a `Move` pointer field would need to be added to the `Player` class called `lastMove`. This way, each player can keep track of its own last move.

A method called `reverseMove(Move move)` in `Board` would need to be created in order to update the state of the board if the Move was undone. Specifically, `reverseMove()` would move the given piece from its new position to its old and re-add the captured piece back onto the board. Special cases such as castling would also have to be considered, since castling involves needing to move two pieces back into its original position (King and Rook). We would know whether or not we would have to handle castling for that specific Move by checking its

`MoveType`. This would be the same case for Pawn Promotion, so we would have to check if it was a Pawn Promotion move (by checking the boolean field in `Piece` called `promoted`).

Then to actually implement the command to undo functionality, we would create a new command in the game loop within `playGame()` so that if the user enters "undo", `reverseMove()` from `Board` would be called. In addition, after every move a player makes, `lastMove` would be updated.

To be able to perform an unlimited number of undoes, each Player would have a vector of `Move` objects called `pastMoves`, rather than a singular Move pointer. Then every time a move is undone, it gets popped off the vector.

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.*

To be able to handle four-handed chess, there would need to be several changes to many classes. The most notable change that would have to be made is to the dimensions of the board. For four-handed chess, the middle rows are all 2 squares longer on each side in order to accommodate all the pieces. Thus, the 2D vector in `Board` would have to be updated to reflect this.

Secondly, four-handed chess would also mean we would need to add two new `Colour` enums. This will allow us to handle 4 different colours of Players and Pieces. When the user enters the command to play the game, they would have to specify what 4 Players are playing, rather than only specifying 2. Then, `playGame()` should behave as normal, thus it would need any modifications.

The most difficult changes that would need to be made involve the behaviour of the pieces. Given the peculiar shape of the board, we would most likely have to add additional checks in `Piece`'s `allPosInDirection()` to see what row/column we are in, and to ensure that we are not adding any Positions that are beyond the scope of the Board. We would also most likely have to redefine what is considered to be a checkmate, check, and stalemate. While the current implementations of the methods would be suitable for checking for a single Player, we would need wrapper functions around the current implementations to check these states for all three opposing Players. For example, to check for a checkmate, we would have to check if all three opposing Players are in checkmate. In addition, whenever a Player did go into checkmate, we would have to remove their pieces from the board in order to and remove the Player from the vector of Players in order to prevent them continuing to move.

Although there are many of other changes that may need to be made depending on what type of four-handed chess is being implemented (such as the version where each a piece has a

valuation, and each player has a score), the changes stated above would be the ones that must be enacted in order to support four-handed chess.

Although this is a very large change, it can easily be implemented with a couple modifications in key classes. This further demonstrates our program's resilience to change.

Extra Credit Features

An extra-credit feature that was implemented was managing all dynamic memory with vectors and smart pointers. I rarely used shared pointers, opting instead to create unique pointers then use the get() method when the raw pointer was needed. Although raw pointers were used, this did not cause any memory leaks. This was because once the unique pointer would get destroyed once it was out of bounds, all the raw pointers that were derived from the unique pointer using get() would also be deleted. The only instance a shared pointer was used was in the Observers. This was because the `TextObserver` and `GraphicalObserver` both have shared ownership over the subject, `Game`.

Another feature that we added to our text display was to display all the pieces as their respective unicode characters, in order to make the text display more aesthetically pleasing.

Reflection 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?

This project taught us the importance of communication when developing software in teams. We would frequently have to communicate to each other to explain how we implemented certain features, or to discuss changing the implementation. We noticed it was important to discuss even the smallest changes in implementation, as it could greatly impact the code that someone else is writing. In particular, it was crucial we were in constant communication over the implementation of `Move`. Since `Move` was used in many different contexts (generating moves for pieces, creating the move of a player, etc.), we had to make sure we were always keeping each other up to date with what changes we needed to make for the class, and whether these changes would affect the other functions of the program. This was how we decided on using two constructors for `Move`. One constructor would be for dealing with Moves that are generated by `Piece`, the other would be used to deal with Moves created by user interaction.

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

If we could start over, we would spend more time developing a greater understanding of how our different classes would interact with each other on a high level. One of the first problems we

ran into when we started coding was that although we understood how we wanted to implement individual classes, we did not necessarily have a full understanding of their exact relationship. For example, we did not realise that Computer players would require access to the state of the board in order to generate moves until much later into the coding process. Thus, if we had more time we would spend more on the design aspect of the program.