CS247 Chess - Final Report

Introduction:

This chess program was developed as the final project for CS 247. The project, written in C++, was completed between Friday, July 19th, 2024 to Tuesday, July 30th, 2024, by Cynthia Lim, Kiran Mandhane, and Emerson Dang.

Overview:

Our program was built using three main classes.

1. Game Class: This class represents the core structure of the chess game. It is responsible for initializing the game by setting up the board and placing all the pieces in their starting positions. It is also responsible for determining the current state of the game and performing the move selected by the Player.

Major Methods:

- **makeMove**: This method handles various types of moves in a chess game, including normal moves, castling, en passant, and pawn promotions. When the user selects the move they want to perform, this method moves the chosen piece to the user's desired position on the board. It then checks whether the move results in a check, checkmate, or stalemate and updates the game status accordingly. Finally, the method switches the turn to the other player.
- **setUpGame**: This method runs the addPlayersToGame method, creates the Pieces to be placed on the board if they were not created via setup mode, and determines which player starts the game.
- **setup**: This is for the setup mode. This method places the created pieces onto its starting locations on the board and can assign who starts the game.

The Game class also serves as a concrete subject for its observers, which are responsible for generating the game's graphics.

2. Piece Class: This abstract class represents an individual chess piece. Each piece has two main functionalities: lineOfSight and possibleMoves.

Major Methods:

- **lineOfSight**: This method generates all potential positions a piece can move to on the board, without considering whether the move will put its own king in check.
- **possibleMoves**: This method iterates through all of the lineOfSight moves and determines if the move will put the current player's king in check. This is done by creating a simulated copy of the board and calling a helper function "isKingInCheck" on all positions it could go to. This ensures players cannot make moves that leave their king in check.

The Piece class utilizes inheritance to define different piece types. Each subclass overrides the pure virtual lineOfSight method as each type of piece has its own movement capabilities. Only the King overrides the virtual possibleMoves method due to castling and the rest of the pieces inherit the default possibleMoves written in the Piece.h file as they do not have this feature.

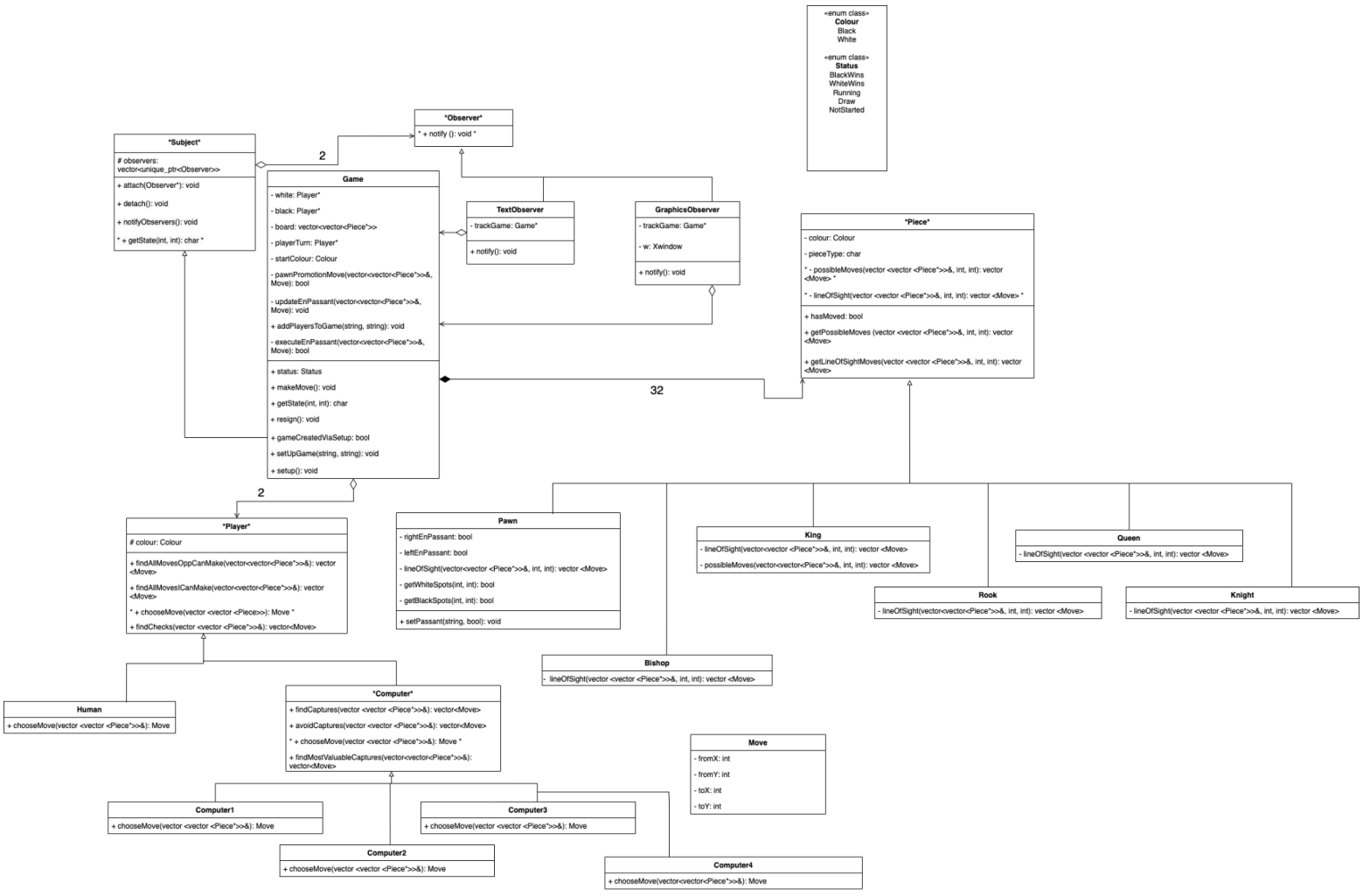
3. Player Class: This abstract class serves as the foundation for creating two types of players: human players and computer-generated players.

Major Methods:

- **findAllMovesICanMake**: This method generates all possible moves for the player's pieces on the board.
- **findAllMovesOppCanMake**: This method generates all possible moves for the opponent's pieces on the board.
- **chooseMove**: This method is overridden by the subclasses. For a human player, this method validates the human's inputted move. For a computer player, this method selects a move from the list of possibleMoves that its pieces can make.
- **findChecks**: This method finds all of the moves a player can make that will put the opponent's king in check

The Player class uses inheritance to define its two subclasses, Human and Computer. The Computer class has three subclasses, each with a different level of difficulty: Computer1 (Level 1), Computer2 (Level 2), and Computer3 (Level 3).

Final UML Diagram



Changes To UML

The overall structure of the new UML remained the same compared to the old UML. The major changes that were done to each class, includes the addition of private fields and public fields/methods.

1. **Game Class Changes**

Added Public Methods + Fields:

- Status: Status
- getState(int, int): char
- gameCreatedViaSetup: bool
- setUpGame(string, string): void

Removed Public Methods:

- getState(int, int): void
- render(): void

Instead of using a string for the game status, we implemented an enum to represent the constant values of the current game state. Using an enum simplifies the process of determining the game's status and eliminates the need for repeatedly updating a string field. This approach enhances code organization and readability.

The getState(int, int) method now returns a char instead of void, allowing us to identify the piece currently located at a specific position on the board. This change helps in determining the type of the piece being dealt with.

We removed the render() method and decided to manually notify the game's observers whenever a change occurs. This approach provides more direct and flexible control over updates.

We added the methods setUpGame and gameCreatedViaSetup to deal with player addition, piece placement on the board, and game initialization. These methods streamline the setup process, ensuring that the game is ready for play.

2. **Player Class Changes**

Added Public Methods:

- findAllMovesOppCanMake<vector<vector<Piece*>>&: vector<Move>
- findAllMovesICanMake<vector<vector<Piece*>>&: vector<Move>
- *findChecks(vector<vector<Piece>>&: Move*

Removed Public Method:

- findAllMoves(vector<vector<Piece>>): vector<Move>

The functionality of findAllMoves still works the same as what we had previously, that being that it finds all of the possible moves a piece can make. However, we encountered an issue where validating human moves (considering checkmates and stalemates) also required consideration of the computer's

potential moves. To address this, we created two separate findAllMoves functions: one for the human player (findAllMovesICanMake) and one for the computer player (findAllMovesOppCanMake).

findChecks() was moved from the Computer class to the Player class as we realized that both human and computer players need access to this information. The functionality remains unchanged, as it still identifies moves that would result in the opposing player's king to be in check.

3. Piece Class Changes

Added Public Methods + Fields:

- hasMoved: bool
- getPossibleMoves(vector<vector<Piece*>>, int, int): vector<Move>
- getLineOfSightMoves(vector<vector<Piece*>>): vector<Move>

Removed Public Method:

- listMoves(vector<vector<Piece>>, int, int): vector<Move>

To determine all the moves a piece on the board can make, we had to consider additional factors. For example, a piece cannot move to every possible square, as this could leave the current player's king in check. To handle this, we categorized moves into two types.

First, the getLineOfSightMoves method functions the same way as listMoves, identifying all possible positions a piece can move to on the board. Second, getPossibleMoves is a subset of getLineOfSightMoves as it identifies the actual moves a human or computer can choose from that do not leave their king in check. This distinction is crucial because if a player makes a move that leaves their king in check, the game would end as the opponent could then attack the king and win the game.

The hasMoved boolean is used for castling purposes (mainly for the King and Pawn). We originally wanted to do this with the pastMoves vector but changed our implementation as we realized that we would have to pass a copy of this vector to every function and it would get very tedious.

4. Pawn Class Changes

Added Public Method:

- setPassant(string, bool): void

This feature was added to the Pawn as it is a valid move which we did not consider when planning it beforehand. Essentially, this method is used to check if a Pawn can perform its enPassant move to eliminate the opponent's Pawn that is beside it when it moves diagonally.

5. Computer Class Changes

Added Public Method:

- findMostValuableCaptures(vector<vector<Piece*>>): Move

This method was added to introduce a new level of difficulty for the Computer player (Computer4) and is used to determine the move that would capture the opponent's highest order precedence piece, if it exists. For example, if a move can be executed such that it captures either the opponent's Queen or Bishop, the Queen would be prioritized and that move would be executed.

6. **Observer Pattern Changes**

Added Public Method

- *getState(int, int): char* (For Subject class)

The getState method was included to keep track of the pieces on the board so that its observers can display it graphically. The Game class is a concrete subject of the Subject class so its getState() is also updated to return a character.

Design Strategies

1. **Strategy Pattern**

To implement varying levels of difficulty for the computer player, we utilized the strategy pattern. This approach follows the open/closed principle, as subclasses are introduced to add functionality to the base class Computer. By defining the Computer class as an abstract class and implementing different difficulty levels through its subclasses, we can introduce new strategies (such as prioritizing moves that could potentially put the opponent's king in check) without altering the base class. This concept also utilizes the idea of polymorphism, as each computer performs its makeMove() in different ways.

2. **Observer Pattern**

To implement the graphics component of the program, we used the observer pattern. In this setup, the game acts as the concrete subject, while the graphical interface components (textObserver and graphicsObserver) serve as the concrete observers. This design allows the observers to track the pieces on the board and update the display whenever a move is made and when pieces are added or removed in setup mode. By decoupling the game logic from the graphical updates, the observer pattern ensures a responsive and maintainable system where the interface always reflects the current state of the game.

3. **Inheritance and Polymorphism**

Throughout our program, inheritance was used to create the different types of Players and Pieces. This technique is crucial to the implementation of the different Pieces, as different Pieces can have different movesets. And since there are different types of players, (computers and humans), inheritance would be necessary to achieve this. Polymorphism also played a significant role. By utilizing polymorphism, we managed to implement various chess Pieces (e.g., rook, bishop, queen) and different difficulty levels for Players. These ideas allowed us to handle these variations effectively within our code.

4. **Coupling and Cohesion**

When implementing our program, we tried to use as low coupling as possible. For example, none of our classes used any friends and no global variables were used. In addition, the Piece class would take in a copy/reference

of the board, but changes to the board itself would be dealt with in the Game class. The Piece class only recalculates its possibleMoves and lineOfSight vectors of the current position the piece is in, but does not actually influence the board. This also applies for the Computer class as findCaptures, avoidCaptures, and chooseMoves take in a copy of the board, but the vectors it returns does not depend on the Game class itself. The only exception for high coupling is in the Game class, as the control flow is constantly being changed to recalculate everything after a Piece is moved on the board.

We also tried to implement our program with as high cohesion as possible. For example, every Piece is in charge of calculating its own respective possibleMoves and lineOfSight. This also applies for the Players as well, as the Human/Computer subclasses are only in charge of determining their own moves to make. For the Game class, it is doing multiple related tasks, as it generates the game logic of chess, performs checks of all possible things that can take place on the game in the moment, and performs the move for the Player when they have chosen one. We also ensured that both graphics and scorekeeping were managed independently.

By utilizing these techniques, we were able to write more readable and maintainable code, which significantly improved collaboration and comprehension within our team.

5. **Dynamic Casting**

Dynamic casting was used to determine whether a pawn promotion is performed by a human or computer. This technique allowed us to differentiate between the two players, as humans have the ability to choose the piece they want their pawn to be promoted to (queen, knight, rook, or bishop), whereas the computer randomly selects one of these four options. By using dynamic casting, we can identify the current player, and perform the changes as necessary. This ensures the correct and expected behavior for pawn promotions in our chess game.

Resilience to Change

1. **Undo Option**

If the program specification stated to include an undo option, we can create a pastMoves vector in the Game class and add each move a Piece made to it. By iterating through the pastMoves vector and reverting the pieces to their previous locations by placing the Piece on its old (x,y) coordinates, this will allow the user to undo any move they made. This feature can be integrated into our program due to its existing structure, as minimal changes would be needed to do this.

2. **Different Moveset**

Since we used inheritance for our Pieces, we can implement a feature that allows each Piece to have a move set that is different from its traditional way of moving. This can be achieved by modifying the Piece's lineOfSight and possibleMoves method to alter its movement patterns.

3. **Multiple Games**

To create a feature which allows the user to play multiple games at once, we would initialize a new vector<Game*>, where each index stores a different game and if the user would want to switch games, we would iterate through this vector to find the game the user would like to play.

4. **Different Game Modes**

To accommodate a variety of game modes beyond the standard chess setup, our program would have to allow users to select from options such as an "all-pawn match" or an "all-rook match" during the setup phase. Implementing these modes involves configuring the initial game state to replace standard pieces with the chosen types. The only critical requirement is that a king must be included in the setup, as it is essential for enforcing the rules of check and checkmate, which determine the end of the game.

5. **Adding New Pieces**

If the program specification required incorporating a new type of piece beyond the standard chess pieces, our design makes this process straightforward. To add a new piece, we simply need to create a new subclass of the Piece class. This subclass would define the unique movement rules and behavior for the new piece while inheriting the common attributes and methods from the Piece class. Because of our current code structure, this approach allows us to extend the functionality of our chess game easily and efficiently. This addition would not affect any of the other classes of the program, as the new piece is only a subclass of the Piece class.

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.**

In order to implement this, we would create a two-dimensional vector of opening moves, *openingMoves*. Each element of *openingMoves* would be a sequence of Moves (stored as a vector) representing a known opening.

At the start of the game, *openingMoves* contains all the existing opening move sequences. When a move is made within the game, it is compared against the corresponding move within each sequence. A variable will be used to determine which sequence's index to check depending on the number of moves that have already occurred. For example, if a pawn was moved first, then a knight, the variable would need to check the knight move against the second move within each sequence. If the executed move does not match the move at the respective index in a sequence, the sequence is removed from *openingMoves*. The remaining sequences are the ones that are partially or completely in effect. With this information, a Player can refer to the next moves within a sequence to determine how to respond accordingly to their opponent's move.

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 create a feature which allows the user to undo a move that they made, we would have:

- A vector of Moves called *PastMoves* inside the Game class
- A vector of Pieces called *DeadPieces* inside the Game class
- A subclass of Move called CaptureMove which inherits from the existing Move class but has the additional field (called *CapturedPiece*) that holds a pointer to the piece that was captured
- Our already existing board, a vector of vectors of Pieces inside the Game class

For every move that is executed, a Move is pushed to the *PastMoves* vector. If the move causes a capture, a CaptureMove is pushed to the *PastMoves* vector instead.

When an undo command is executed, we look at the last element in the *PastMoves* vector.
There are two cases to consider:
1. The element is a regular Move, NOT a CaptureMove. Note that the Move class stores the new (x,y) and the old (x,y) of a piece to track its move. We locate the new (x,y) on the board (which is a vector<vector<Piece>>) and move the piece back to its old coordinates on the board using old(x, y). We then remove the move from the *pastMoves* vector, since it's as if this move never existed. This also accounts for a castling move, even though it involves the movement of two pieces.

Special Cases:
Castle: To recognize a castle, we just need to check whether the king has moved by two squares. The direction of the king's move also informs us about which rook was involved in the castle, so undoing the castle simply requires repositioning the king and appropriate rook to their previous places.

2.The element is a CaptureMove. In this scenario, in addition to rearranging the pointer of the element that was moved in the Move, we also use the *CapturedPiece* field. *CapturedPiece* points to a Piece that was captured, which is no longer on the board and was moved to the *DeadPieces* vector when the capturing move was processed. To undo the capture, we would move *CapturedPiece* out of the *DeadPieces* vector and into its appropriate spot in the "board" vector. Its spot would be the "toX, toY" fields of the CaptureMove object. Once the pieces return to their previous spots, the move is removed from the *pastMoves* vector.

Special Cases:
En passant: To recognize and then undo an en passant move, two pieces of information would need to be checked. Using *pastMoves*, we look two moves back and check whether an opponent's pawn moved forward by 2 squares. We would then check the most recent move, identify whether it was a pawn that was moved, and confirm whether the pawn's new (x,y) is equal to the opponent pawn's new(x, y) from two moves back. If they are equal, it means that the pawn did a normal diagonal move to capture a pawn. If they aren't equal, then it means that the pawn stepped behind the opponent's pawn, implying that en passant was done. No matter the outcome, the opponent's pawn is moved from *DeadPieces* to its proper previous position on the board (same for the pawn that did the capturing) and the move is undone.

Pawn promotion: Pawn promotion will be stored in *DeadPieces* as a captureMove where the pawn is the *CapturedPiece* and the board displays the piece that the pawn was promoted to. Note that it is impossible to capture a pawn when it moves to a square viable for pawn promotion - it will already have been promoted to another piece which is what the current board will see/recognize. So we can use this "impossible case" to classify a pawn promotion move within *pastMoves*. To recognize and then undo pawn promotion, we would first need to check the most recent move and identify whether the piece moved by 1 square, if it moved to a valid pawn promotion square, and whether the *capturePiece* of the CaptureMove is a pawn.

Capturing onto a pawn promotion square: It is possible for a pawn to capture a piece and be promoted within one move. To account for this case, we would need to push two moves to *pastMoves*: 1 move to indicate that the pawn captured a piece followed by another move to indicate the pawn promotion. When undoing this, we would check the most recent move, which would be the pawn promotion. Now any identified pawn promotion move would require an additional check to the move it follows (i.e., in total, look 2 moves back) and determine whether a pawn capture was done at the same pawn promotion square. Once that has been identified, the proper pieces can be returned to their previous positions using the logic above.

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.**

These would be the changes we would make to our program to handle the 4-player chess game, assuming a free-for-all style.

1. We would first have to adjust the size of the board as it is increased. To do this, we would regenerate the *board* vector by generating a 14 x 14 board and we would also have to create some exceptions where specific references to the board cannot be accessed within the vector. For example, the four corners should not be able to be accessed by any of the Pieces on the board. On top of this, we would also require 4 colours instead of 2 (e.g. white, black, yellow, red) which would need to be added in the text-based display (differentiating the players' pieces with symbols), the graphical display, and the game logic.
2. With 2 additional players, we would need to have 4 players per game and generate 2 additional sets of pieces, which would have specific starting positions to be properly placed at the top, bottom and sides of the board vector.
3. 4-player chess games can end in a variety of ways: when 3 players are eliminated or when 2 players remain and one player is ahead by 21 points or more. In addition to recognizing the new ways that a game can end, we would also need to implement a points system via a class with the appropriate methods where points are assigned to players based on captures, checks, and stalemates.
4. We would also have to adjust our *possibleMoves* vector to take into consideration the 4 restricted corners of the 14 x 14 board. The logic for our current methods *findChecks,*

findCaptures, avoidCaptures, etc. would not require any updates to accommodate 4 players aside from the fact of knowing how to account for 4 players at once.
5. Assuming that this game is played with a time component, we would have to create an additional *time* field in our *Player* class, which decreases the starting time by 1 each second while it is a player's turn and eliminates the Player if their time runs out.
6. 4-player chess games can end in a draw when it is amongst 3 players. This would need to change in contrast to the current draw logic that only considers 2 players, as we would have to implement features which considers the draw logic of 3 players.
7. A player's King piece will remain alive if the Player runs out of time or resigns. In that case, our program should prevent the Player from making moves and take control of the King's movement by finding its *possibleMoves* and randomly choosing one of its possible moves to execute until the end of the game.

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?

While developing software in teams, we noticed that every member has their own ideas in mind to solve a particular problem of the program. However, after discussing our ideas with one another, we were able to successfully integrate all of our ideas into one, enhancing the overall effectiveness of the software. To ensure everyone is on the same page, we would review and discuss our implementation strategies with one another before doing the implementation. This collaborative approach helped verify the soundness of our logic, making the debugging process more efficient as everyone was familiar with the work and could provide assistance.

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

If we had the chance to start over, we would focus more on establishing clear communication and collaboration practices from the beginning. This would involve setting up regular meetings to ensure everyone is aligned on project goals, using collaborative tools more effectively to track progress, and encouraging open discussion to integrate diverse ideas early on.

Conclusion

After doing this project, we were able to solidify our understanding of the course concepts and design patterns learned in CS 247 as we applied them directly in our code. Additionally, this experience helped us develop crucial collaboration and teamwork skills to prepare us for careers in the work industry.