CHESS REPORT

By: Varuni Gupta, Sydney Nguyen and Alyona Vishnoi

1. Overview

The project follows a design pattern inspired by the Model-View-Controller (MVC) architecture and employs the Observer pattern to facilitate communication between components. The MVC pattern is evident in the division of classes into distinct roles: the ChessController serves as the *controller*, managing the game flow; the Board acts as the *model*, representing the state of the chessboard; and the TextDisplay and GraphicsDisplay (built using the XWindow class) as *views*, responsible for rendering the board in text and graphical format.

The **ChessController** class takes on the role of the controller, orchestrating the game's progression by managing the player turns, scores, and the game board. It employs Player pointers (*WPlayer* and *BPlayer*) to represent the white and black players, offering methods to initialise the game, execute moves (both human and computer-initiated), check for game termination, retrieve scores, and reset the game state.
The **Board** class functions as the model, encapsulating the chessboard's state with a 2D vector of Cell objects. It keeps track of white and black pieces separately and utilizes TextDisplay and GraphicsDisplay (Concrete Observers) for rendering the board visually.
The **Cell** class represents individual cells on the chessboard. The Observer pattern is employed with the Cell class, acting as the observable subject, and TextDisplay and GraphicsDisplay as the observers. When a Cell undergoes a state change, such as a piece move, it notifies its observers, which then updates the displayed board accordingly
The **Piece** class serves as an abstract representation of chess pieces, defining common attributes and behaviours to all pieces. The implementation of different chess pieces are separated into individual classes such as KingPiece, QueenPiece, PawnPiece, etc. They inherit from the abstract base class Piece
The *Computer* and *Human* classes, derived from **Player**, encapsulate player-related functionalities, distinguishing between human and computer players. These concrete Player classes interact with the Board to make moves.
This separation of concerns enhances modularity and extensibility, allowing for easier maintenance and potential future additions, adhering to the principles of both MVC and the Observer pattern.
Finally, the **desc.h** header file contains essential enumerations that provide symbolic names for key aspects of the chess game. The *Color enum* class defines two constants, White and Black, representing the colours of the chess pieces on the board. This enum is instrumental in distinguishing between pieces controlled by different players.

The *PieceType enum* class enumerates the various types of chess pieces, including Pawn, King, Rook, Queen, Knight, and Bishop. Each constant corresponds to a specific type of chess piece, facilitating clear and readable representation of piece types throughout the codebase. By using enum classes, this design choice enhances code clarity, maintainability, and type safety.

2. Design

Our Chess Game accommodates different kinds of players such as **Human** and **Computer**. For the Human player, its "*makeMove*" method has parameters "start" coordinates, "end" coordinates and a promotion character to validate Pawn promotion moves. Within this method, we determine which piece is at the start coordinate and verify its validity to move to the end coordinate by calling and ensuring the specific piece's "*moveable*" method returns true and the "*moveProduceCheck*" method returns false. If all of these conditions are met, the Board's "*update*" method will be invoked to remove the appropriate captured pieces, update the player's piece to the new coordinates and finally notify our displays.

Then, our Computer's moves algorithm consists of a choice between four levels. **Level 1** randomizes any valid move. **Level 2** prioritizes any moves favouring capturing and checking. **Level 3's** top priority is avoiding capture without endangering our piece post-escape, and if that move isn't present, we randomize checking, capturing, and any valid moves. Lastly, **Level 4** prioritizes moves that put our pieces outside of danger. Simultaneously, if our computer can escape danger by capturing the opponent's piece, our algorithm prefers capturing Queens, Rook, Bishop, Knight and Pawns in descending order. Although if there existed no move like previously explained, we would pick between avoiding capture moves, capturing moves, checking moves while attempting not to put our pieces in danger after the move completion and finally, any valid moves.

After determining the move we want to make, the "makeMove" method calls the specific piece's "moveable" method in the starting cell. A true/false value was returned to determine the validity of the move. Then, coupled with using the "moveProduceCheck" method within each Piece's subclasses, if our pieces are both "moveable" according to chess rules and do not put themselves in check after the move completes, we will call the "doMove" method in the Computer class to begin updating changes on the Board. The "doMove" method determines if the executed move was a special move like EnPassant, Castling or Promotion or a normal piece that only requires updates for 2 Cells. The "doMove" method then determines how Display updates differ according to the move that was executed and utilizes the Board's "update" method to remove appropriate captured pieces, update our piece to its new coordinates and finally calls the "notify" method to display changes on our board.

We faced several challenges while coding this game. Initially, we were torn between either using only the Observer pattern or combining it with the MVC model. We then chose to incorporate both in our implementation. The MVC design pattern would deliver a clean implementation of the game logic (Model, user interface (View) and user inputs (Controller). This approach enabled a more flexible, manageable and extensible design, thus allowing the incorporation of future modifications or additions to different aspects of the chess game without disrupting the overall system architecture.

Another challenge involved determining whether a move would put the player or its opponent in check. Since our “isCheck”, “isCheckMate” and “isStalemate” method in Board depends on the most up to date location of all pieces, we needed a way to execute our potential move and then utilize those checking methods without tampering with the actual Board that is displayed to use. A Board copy constructor was incorporated to avoid explicitly changing Pieces coordinates on our Bated. This constructor allowed us to create a temporary board in the specific Pieces’ “moveProduceCheck” method to simulate the move, checking for potential check/checkmate/stalemate scenarios before applying the move to the actual board. This helped us avoid unwanted move operations.

Additionally, the implementation of specific conditions such as Enpassant captures, Castling, and Pawn Promotion necessitated adjustments across multiple cells. Unlike standard moves that solely involve the starting and ending positions, these scenarios required simultaneous modifications to three or even four cells. Hence, prior to updating the piece coordinates and informing the board, we needed to ascertain whether the piece in question was a Pawn. If so, we called the "isEnpassant" method or the "isPromotion" method. Similarly, if the piece was a King, we invoked the "isCastle" method to determine whether the move fell under one of these special categories. This approach ensured that we properly handled these unique moves before treating them as normal moves that affect only two cells and, at most, two pieces.

Thirdly, the implementation of Level 4 for the Computer class was tricky. Since all kinds of moves that we could have thought of were done in Levels 1-3, we had to do extensive research on what kinds of moves would be more sophisticated than the previous levels. This level focuses on capturing moves to extract pieces from danger. It prioritizes capturing the opponent's Queen, Rook, Bishop, Knight, and Pawns in descending order. If no capturing moves apply, it prefers avoiding capture without compromising safety after move completion.

COUPLING AND COHESION:

In terms of cohesion, the ChessController class demonstrates a clear adherence to the principle of high cohesion by maintaining responsibility solely for the game's state rather than the intricacies of the game rules. This design choice ensures that the ChessController remains focused on orchestrating the flow of the game, keeping track of player turns, scores, and the overall state of the chessboard. The actual rules governing moves, piece behaviours, and game progression are appropriately encapsulated within the Board and Piece classes, promoting modularity and separation of concerns.

The Cell class in the program exemplifies high cohesion by adhering to a focused and singular responsibility, namely the management of individual cells on the chessboard. This class encapsulates the state and behaviour of a single cell, holding a pointer to a Piece object and storing its coordinates (r and c). The primary purpose of the Cell class is to serve as a container for a chess piece, and it manages this aspect with precision.

The Piece class also cohesively manages movements and unique piece behaviours, encapsulating the specific rules governing each piece type.

Moreover, the inclusion of displays, such as TextDisplay and potentially GraphicsDisplay, exhibits high cohesion by strictly adhering to their designated roles. These classes are responsible solely for rendering and presenting the game state in either textual or graphical format. This specialization enhances the clarity and maintainability of the code, as each class has a well-defined purpose and operates within its designated domain.

Regarding coupling, the interactions between our classes/modules are facilitated through function calls, minimizing coupling and promoting a modular design. For instance, the ChessController communicates with the Board to update the game state, but the actual implementation details of moves, captures, and checks are encapsulated within the Board class. This allows for flexibility and extensibility, as changes to the game rules can be localized to the relevant classes without affecting the broader system.

In instances where some coupling exists, such as between the Player classes and the Board, it is deliberately introduced to facilitate necessary interactions. For example, players need to interact with the Board to make moves and check game conditions. This limited coupling is justified as it aligns with the logical flow of a chess game and allows for a streamlined communication process without introducing unnecessary complexity. The design choice carefully balances the need for interaction with the requirement for modular, maintainable code, contributing to the overall effectiveness of the program's architecture.

3. Resilience to Change

The resilience to change in the chess program is inherent in its Model-View-Controller (MVC) architecture. This design enables easy modifications to specific components without affecting others, resulting in lower coupling. For instance, updates to graphical or textual representations can be seamlessly implemented by modifying the GraphicsDisplay or TextDisplay classes, respectively. The Observer pattern further enhances flexibility, ensuring that changes in the game state are automatically reflected in displays without direct links to game logic. The dynamic and loosely coupled nature of the Observer pattern supports the addition of new displays or modifications to existing ones without disrupting core functionalities.

Additionally, the use of abstract classes and polymorphism for representing chess pieces allows for adaptability. The introduction of new piece types is facilitated by creating new classes that extend the abstract Piece class, adhering to the Open/Closed Principle. This design approach makes the program well-prepared for future changes, supporting the incorporation of diverse chess features and variations with minimal impact on existing code. For players, the abstract Player class serves as a blueprint, allowing for the easy introduction of new player types. By creating derived classes, such as Human and Computer, developers can implement various player behaviours without altering existing code. This supports the incorporation of new player strategies or the enhancement of existing ones without causing disruptions. Similarly, the Cell class, responsible for individual cells on the chessboard, is designed with high cohesion. Changes to cell-related functionality, such as additional attributes or behaviours, can be localised within the Cell class, minimising the risk of unintended consequences in other parts of the program. This high-cohesion design contributes to the ease of understanding and modifying cell-related features. The game board, represented by the Board class, also exhibits flexibility in accommodating changes. The separation of concerns within the MVC architecture ensures that modifications to the board, such as adjustments to initialization methods or the addition of new rules, can be made independently. The board's encapsulation of board-related actions, such as piece placement and removal, enhances its adaptability to changes in game rules or variations.

In summary, the program's modular design, coupled with the use of abstract classes and interfaces, facilitates the seamless incorporation of changes to players, cells, and the board. This design philosophy aligns with principles such as encapsulation and separation of concerns, contributing to the program's resilience to modifications and its readiness for future enhancements.

4. 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 implement a book of standard opening moves for our chess program, we propose designing an **OpeningMoves** class with two private fields named *whiteOpeningMoves* and *blackOpeningMoves*. These fields would be vectors of 2D vectors, where each row represents an opening move sequence and contains pairs of the end positions for both white and black pieces. This structure allows for a flexible representation of various opening move scenarios. In the Board class, where the most recent moved piece is tracked, we can leverage this information to determine the current state of the game.
   For the concrete Human player class, we would enhance the functionality to display relevant opening moves based on the player's colour and the opponent's previous move. The display of opening moves might also be influenced by whether it's the human player's first move and if they are playing as white. For the concrete Computer player class, the strategy would involve generating a move based on the opponent's last

move. The Computer player would attempt to find a corresponding move in the opening moves vector to make an informed decision. This approach aligns with the goal of integrating standard opening moves into the program's decision-making process, enhancing the Computer player's strategic capabilities during the opening phase. By encapsulating this functionality within a class (OpeningMoves) we establish a clear link with the rest of the program.This design choice promotes modularity and maintainability, as the opening moves logic is encapsulated within a dedicated class, allowing for easy updates or expansions to the opening book in the future. It also facilitates a clean interface for other components of the program to interact with and leverage the opening moves information when necessary.

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

   Our answer to this question remains mostly the same, in implementing the undo feature, the chessboard can be enhanced with a stack-based mechanism to systematically log the move history. This involves adding two private fields to the Board class - a dynamic vector to store the recent moves made by the players and a structure called MoveDetails. The MoveDetails structure would include details such as a 'captured' (bool) to signify whether a piece was captured during the move and 'capturedType' to record the type (char) of the captured piece. These additions enhance the granularity of the move history, allowing us to track not only the source and destination squares of each move but also whether a capture occurred and details about the captured piece. This information proves valuable when implementing the undo feature, as it enables the precise reversion of captured pieces to their original positions on the chessboard. Since the player “has a” relationship with the board, upon making moves the player is responsible for pushing each move onto the stack and updating the board. Upon completing a move, the Observer (Views) is notified directly for synchronised updates. When a player requests an undo from the command line, the controller calls the Board method undo. The system (board) checks the stack; if empty, the user is informed. Otherwise, the system pops the last state and calls the move function to revert the state of the game. To manage memory usage, a cap on the stack size can be implemented. The system allows unlimited undo capability by facilitating multiple pops until the stack is emptied.

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 transform our chess program into a four-handed chess game, several fundamental changes are required. This expansion requires adjustments in the program's core structure, specifically in classes like ChessController and Board, where private fields and enumerations would be updated to accommodate a total of four players. For example, the ChessController class, which manages the overall game flow, would be extended to accommodate two additional player pointers, representing the third and fourth players, each with their own distinct colours. The playerTurn variable would need to be adjusted to handle the turns of all four players in a cyclic manner. Whereas, the Board class, responsible for managing the game board, would be expanded to now accommodate 160 squares (cells). This requires resizing the board grid and adjusting the initialization and setup procedures accordingly. The vectors storing whitePieces and blackPieces would be extended to encompass pieces for all four players. Piece movement and capture logic would be updated to consider the presence of four sets of pieces The command interpreter in the client main.cc file would also undergo modifications to support commands unique to four-handed chess, ensuring that players can interact seamlessly with the game. Additionally, both the TextDisplay and GraphicDisplay components require adjustments to accurately represent the expanded board and visualise the movements of all four players. From a gameplay perspective, critical functions, especially those assessing whether a king is in check, must be reworked to consider the presence of multiple kings on the board. Moreover, the game-ending conditions would be revised to align with the rules of four-handed chess. In this variant, the game concludes when a player checksmates an opponent, stalemates themselves or an opponent, checks more than one king simultaneously with a queen or another piece, or captures active pieces. These rules introduce a point-based scoring system, and the game ends when a player finishes with more points than their opponents.

5. 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?**

   During the project development, we learnt the importance of planning out what features we wanted to incorporate and how they were to be interlinked with one another before starting to write the code. Making a UML diagram helped in combining all of our ideas in developing the program model and also ensuring that a base model was created that we could build upon. This saved a lot of time while coding as we didn't have to think of features on spot or as we coded. We also had to write some pseudo code to understand the flow of the game and brainstorm any fields/functions that had to be added to the classes for certain features. None of us had worked in a team before, so collaborating and building upon others' code was a new skill that we developed. The use of GitLab and git commands to merge code/ resolve conflicts was initially challenging but a very useful technology that helped us combine our codes efficiently. We also developed strong time-management skills as we had to balance almost daily meetings and coding while also preparing for our finals and attending classes. Dividing tasks according to our strengths and helping each other in fixing bugs in their codes allowed us to complete our work on time.

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 definitely write pseudo code for all the classes to know which all functions we could add at every step to ease our task. A lot of our time was spent making new functions that weren't in the initial UML diagram. We could also take the time to list down all possible test cases/ scenarios of playing the game. It would also have been good to have split the work in a more efficient manner. Even though we initially assigned different parts of the program to different team members, we still ended up needing to work on the same code at some point to add functions/fields that were needed from our classes. Sometimes, we worked on the implementation of certain classes simultaneously - which led to some clashes and confusion since the move functions and base classes were being built differently by 3 people. This could have been resolved if we had thought about making the final UML first with the help of pseudo code. As a result, we would want to have better deadlines set for ourselves to complete our tasks on time so that the other person could work on their side of the code without having to wait for the others to push their changes.