CS 246 Project

Introduction

This report outlines the design and implementation of the chess game developed by Rushil Deep, Shubham Patel and Sharang Goel. This game has been developed entirely in C++ using the observer pattern with the MVC methodology. Additionally, it has been implemented using the C++ STL with the RAII idiom for implicit memory management. It can be played through the command line terminal with both a text-based and a graphical output.

Overview and Design

1. Main.cc (Controller) - The chess executable when run, initializes a board and provides the user with a list of options to choose from based on which they would be able to interact with the game. For example, setting up a custom game, a new game, and a help option that provides additional details for accepted inputs. Once the board has been set to a custom game or a default game, two players are initialized using the command line input to either human or computer and an object of the game class is initialized.

2. Game (Controller) - A game has an ownership relationship over a board and two players. After initialization the game sets its players using the setPlayers method and then the chess game is played through a method of the game class (playMove) which takes two arguments; a player and a board reference and it gets the move played by the player (receives an object of the Move class) and then makes that move on the board using the method makeMove which calls a makeMove method on the board.

3. Board (Model) - The board is the main model class of the application which has ownership over n^2 cells (Where n is 8 by default), a TextDisplay and a GraphicsDisplay. It initializes cells based on setup called by main and receives the move to be made from the game class and updates its state based on the move received. It then notifies all the Pieces observing the source and destination cells about the update and returns specific values to board which then updates main about the type of move played (whether the move led to checkmate, resignation, a draw or none). It also consists of a getLastMove and setLastMove method to allow for special moves like En passant as well as a friend in the overloaded output operator (ostream& operator<<) which just prints the TextDisplay that it owns.

4. Cell (Model) - Each cell has a Piece (may or may not be nullptr) and has 2 to 18 observers (As each cell at least has the TextDisplay and GraphicsDisplay as an observer

and may have 0 to 16 Piece observers observing it). The observers of the cell are notified (detached and reattached from cells) every time there is a change in the state of the cell By default, 64 Total cells form the board.

5. Observer (Model and View) - Abstract class which has subclasses in Piece, TextDisplay and GraphicsDisplay. Primarily serves as the View but also a part of the model as Piece is an observer

6. Piece (Model) - Abstract class that consists of subclasses in Rook, Bishop, Knight, Queen, King and Pawn. Has pure virtual methods in attachToCells and detachFromCells that are used by the subclasses to observe and detach from cells. Constructed with a character that determines which type of Piece to create. Each piece subclass has an integer field called points that determine how many points that piece is worth.
    a. Rook - Attaches to all cells horizontal and vertical of itself (5 points)
    b. Bishop - Attaches to all cells diagonal of itself (3 points)
    c. Knight - Attaches to all cells in an L shaped direction from itself (if starting position is x,y; the Knight attaches itself to x+2,y-1 | x+2,y+1 | x-2,y+1 | x-2,y-1 | x-1,y+2 | x-1,y-2 | x+1,y+2 | x+1,y-2)
    d. King - Attaches to all cells in all directions up to 1 move of itself (X points as the game is won/lost when a King can be taken (checkmate))
    e. Queen - Attaches to all cells in all directions from itself (9 points)
    f. Pawn - Attaches to the forward vertical and diagonals up to 1 move of itself (1 point)

7. Player (Model) - Consists of subclasses in human and computers (1-4). Takes input from the user or plays the move on behalf of the computer. Contains the logic for being able to Promote, Castle, En passant, Check and Checkmate. Also owns a move object which is then received by game and board respectively after it is changed.
    a. Human - Takes input from the user for making moves
    b. LevelOne - Makes any valid move randomly using rand() from the cstdlib library
    c. LevelTwo - Makes moves that capture pieces over moves that would not capture pieces
    d. LevelThree- Makes moves that prefer to capture pieces, make checks and avoid capture in this order of preference
    e. LevelFour - Similar to Level 4 but recognizes the value of pieces such as Queen is worth 9 points, Rook 5, Bishop and Knight 3 and Pawn 1. It will look to capture pieces of higher value and protecting it's higher value pieces.
    f. LevelFive- Uses a Minimax algorithm of depth 3 and prefers moves that result in the best position 3 moves out.

8. Move (Model) - Stores the move made by the player and gets passed to the Game and the Board, serving as a data unit of the application. Consists of the source and destination coordinates of the piece being moved as well as whether or not the game was resigned

9. TextDisplay (View) - A universal observer of all the cells of the board, gets notified every time the board state is changed (Add Piece, Remove Piece, Move, Take). Allows for the game to be played purely using the command line by displaying the game state using letters for Pieces and Symbols (‘ ‘ and ‘_’) for black and white pieces.

10. GraphicsDisplay (View) - This is also a universal observer of all the cells of the board which is notified in the same manner as the TextDisplay. Uses X11 Graphics to display output on an XWindow. Includes images of the different types of pieces that are displayed based on the state of the board.

Design - Specific Techniques

One of the biggest questions that came up when we were designing our application was how to make it efficient, which is why our team decided to go above and beyond by making each Piece an observer of cells which allows for fewer checks to be made when a Player makes a move and requires fewer number of checks overall as one only needs to check if a certain cells is observed by a particular piece which is a much faster process as compared to checking for specific conditions like which position did we move to, is that position valid, does it contain a piece, what colour it is, and so on. This, in combination with special moves, would make checking for move validity extremely untidy, which is why we came to the decision of making every Piece an observer of Cells. This way, the code is not only clearer to read and build upon, but it also leverages the Observer pattern to make the application more efficient, requires fewer error checking statements and automatically covers all cases so we do not have to check for edge cases.

Resilience to change

The program developed by the team is highly resilient to change, particularly because it follows the MVC software architecture. If additional piece types were to be added hypothetically, then they would simply become a subclass of the Piece superclass and everything else would mostly remain the same where we would only need to update the TextDisplay based on the character being used for the new Piece and the logic for the Piece itself would go into its implementation file with no arbitrary error checking required since we are heavily focussed on the observer

pattern. Additional levels can also be added very conveniently as they become subclasses of the player superclass and logic can be built upon the pre-existing code for the other classes. Special moves like En-passant and Castling are being handled dynamically, those too can easily be disabled if needed. The board size is also dynamic (Default 8) to incorporate 4 player chess if needed. We also created a Move object for this very purpose so that any changes in the coordinates or resignation requests could easily be handled. Because of the dynamic nature of the Move class, we can also turn off resignation altogether.

Moreover, the Piece objects implement an observer pattern on the Cells of the Board. Hence, every time the state of the Cell changes, as if a piece is moved from or to it, it calls notify on all its observers which dictates where the observers of that cell can go from that point onwards. For example, if a Rook can *observe* cell a4, it can move it. If a piece is moved to a4, Rook's notify is called and its valid moves are reevaluated. This is highly adaptable to changes in the program expectations as observation of cells, attaching, and detaching from cells all happen dynamically, at run-time. This means that any changes to the Board's boundaries only need to change the variables that store the bounds in our code, and the pieces would look all the way to the boundaries. Furthermore, if a piece is to be added, which makes some sort of special move, different from the standard pieces in chess, all one needs to do is implement a Piece subclass to represent the new type, and the integration with the observer pattern is automatic. For example, a popular subvariety of chess is duck chess, where there is a duck that both players must move. We can implement that by adding a duck class that will attach itself as an observer to every cell on the board and will be able to be moved to every empty cell. Since both players can move the duck, it would need an additional property that returns the Colour of the current player, which changes every turn. However, all of this can be accomplished with very few changes to existing code and recompilation, as most of the logic and files will work with this new logic directly.

Answers to Questions

Q1. Chess programs usually come with a book of standard opening move sequences, which list accepted opening moves and responses to opponents' moves, for the first dozen or so moves of the game. See for example https://www.chess. com/explorer which lists starting moves, possible responses, and historic win/draw/loss percentages. Although you are not required to support this, discuss how you would implement a book of standard openings if required.

A: First, we would need to create a tree object containing all the opening moves we would like to support, where each level in the tree would be a board state, and each edge would be a possible move leading to another board state in the opening move tree. We will store each move as 2

coordinates, starting and destination, along with the name of the opening or variation, for example, Queen's Gambit or Sicilian Defense and historic win percentage. This will allow us to store a large number of opening moves efficiently. The game object(controller) will have access to this object through a pointer that would support 2 features. It will classify and name each position of the opening that is played. Also, if this feature is enabled, it will give a list of possible openings and next moves along with the name, possible responses and win rate for each variation.

Q2. How would you implement a feature that would allow a player to undo their last move? What about an unlimited number of undos?

A: To implement a feature to allow undo's, we would only need to modify Game(controller) to store relevant move data, which can be reversed if required. To implement a single redo, we could simply store the previous board object until the next move is performed, at which point the next board position is stored instead. To perform the undo, the controller would simply restore the previous board position. However, to implement unlimited undos, this could get very inefficient as the list of board positions grows large during a long game. Instead, we could create a linked list with 2 coordinates, starting and destination, to represent each move rather than the entire board position, thus being far more resource-efficient. Each move would be inserted at the front of the linked list, so the last move should be first (LIFO) so undoes can be done by the controller performing a "special" move that bypasses the checks for a legal move and simply moves the piece from the destination coordinate to the starting one while deleting that move from the linked list. This can be repeated all the way to the starting position.

Q3. 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.

A: Since our program is constructed in a manner to minimize coupling and increase cohesion, altering our program to support 4-player chess only requires partial changes to only a few modules. Mainly, the board would need to be updated to follow the layout for 4-player chess rather than 2 players, and pieces would need to be initialized for all 4 players. This would be a slight challenge as a 4-player chess board isn't a perfect square, so we would need to indicate null (or invalid) cells in the corners. Additionally, the enumeration of colours would need to be updated to 4 colours (red, blue, yellow, green) to indicate the pieces of 4 players as well as differentiate the output on the graphics display. We would need to ensure that our previous implementation of the text and graphics display continues to work as intended in terms of indicating what piece belongs to what player and scaling to the new board layout. Although we expect our previous implementations to adapt, we may need slight modifications to optimize everything. The controller(game) would need to be updated to accommodate turns for 4 players,

as well as setting up the board properly as well, and creating four player objects instead of two. Finally, the computer player levels would need to be updated to evaluate positions in a 4-player position rather than just 2. This will likely be the most difficult aspect as 4-player chess can be extremely complex and has many more combinations and possibilities, making the computation more resource-intensive and inefficient.

Extra Credit Features

1. Implicit Memory Management and STL - The entire application was developed using smart pointers which follow RAII and manage memory implicitly (e.g. - Each Cell has a unique_ptr<Piece>) . Additionally, we used the C++ Standard Template Library containers like vectors to store a list of items whenever and wherever required. (e.g. - Cell has a vector of observer pointers). This caused us some challenges as we had to transfer ownership of pieces from one cell to another without letting the piece go out of scope as a unique_ptr.

2. Agreed Draw- Our game allows players to agree to a draw in the human vs human mode. This was implemented because it is very common for players to agree to a draw when the game is headed towards an obvious draw which saves time and players can play the next game immediately.

3. Level 5 - A minimax algorithm for determining better moves. This was a challenge as we struggled to efficiently make the move to evaluate the positon and then undo to the original position while returning the best move.

Final Questions

Q. 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?

A: One of the biggest lessons we learned from developing software in teams was the importance of having a depth of knowledge with version control systems such as git (github) as we often came across merge conflicts, problems with pushing, outdated branches and many more issues common among developers who don't have a lot of prior experience with developing in teams. Another big takeaway was the importance of good code communication. Some of our team members were not in the habit of commenting on their code well which made it difficult for others to understand their logic and therefore, leading to a kind of road block. On the other hand,

some of our team members had a habit of documenting their code too heavily, to the point that it was a bit annoying and overwhelming for other members of the team. We also learned the importance of delegation of responsibility which was a part of the Due Date 1 document which served very useful to us as we stuck to that plan of action and this also encouraged a high level of accountability within the team that ensured that team members completed work on time and helped each other out if they were done with their task before hand. Additionally, we come to understand the importance of testing and debugging as when a particular part of the game was 'working' according to the developer, when another team member tested the feature with a different input, it was causing bugs and this process happened often, which is when we realized that we needed to test our code thoroughly instead of having blind confidence in our logic and ending up in a fallacy. In order to actually solve the bugs, we needed to have good knowledge of debugging tools like gdb and valgrind, which most of our team members had and for the others, it was a great learning experience. Lastly, all our team members understood the importance of communicating well, so that the code written does not conflict with that of others and rather works seamlessly with others unless a change is absolutely necessary. This not only ensures that the team is on the same page in terms of development, but it also ensures that the code itself is high in cohesiveness and low in coupling as if communication between members is poor, one may end up importing redundant header files, have cyclic dependencies and so on. Overall, working on this project was a great learning experience for all the members of our team and we hope to bring our best learnings from it into any teams we work in, in the future.

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

A: A few things we would have done differently are that we would have created two different subscription types for observers (Like in Assignment 4). This would make our implementation even more efficient and precise as the TextDisplay and GraphicsDisplay would only be notified about the final state of the board instead of the in-between processes. Another thing we realized which was a problem was that while developing our initial design and UML, we were thinking too ahead of ourselves as we thought about implementing bonus features beforehand (like undo one and undo multiple moves) but we did not focus on the primary task at hand. Even though our UML did not change, we should have made sure that it adheres to the core requirements of the project and thought about additional features later. Thankfully, this lack of judgement did not trouble us much but it served as a good learning to take into account for future projects. Lastly, if we could start over, we would have had a more hands-on approach instead of theorizing as it is easy to think about different classes and their relationships but things only really get clear once ideas are put into action, in this case, code, and therefore it is essential to theorize but not overthink and do more in order to have more clarity.