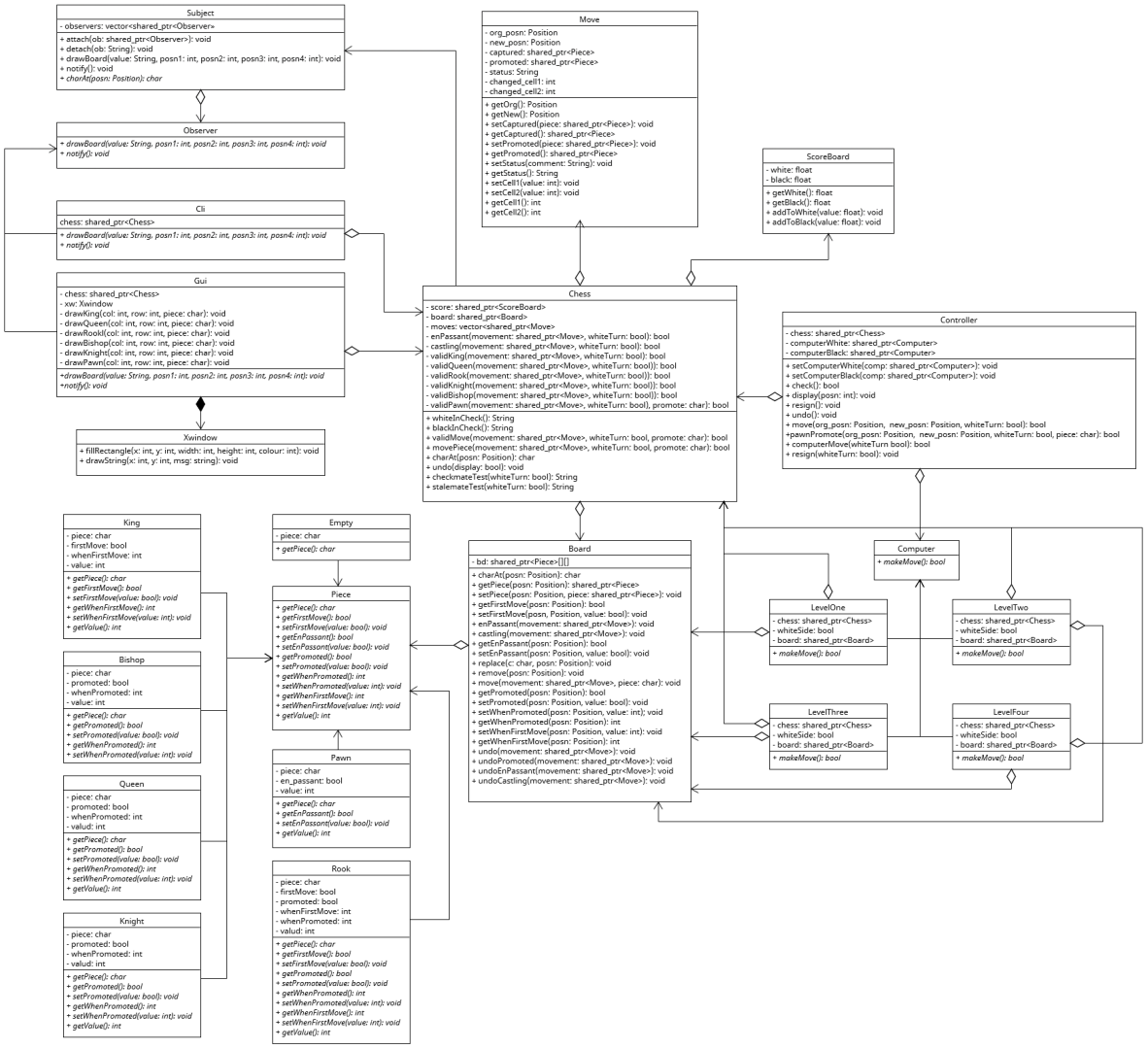
Junsang Yoo, Seung Hyeok Oh

CS246 – A5 Final Documentation DD2 (Chess)

Introduction

In this A5 Chess assignment, our task given is to implement a chess game in C++. Chess is basically a two-player board game where the board is 8 × 8 with a black and white checkered pattern. Each side has a total of 16 pieces(1 King, 1 Queen, 2 Rooks, 2 Bishops, 2 Knights and 8 Pawns). Different type of pieces have different available moves, and it can capture the opponent's piece by moving to the piece's position. The game can end by either a checkmate, stalemate, or resignation. Checkmate happens when one of the players is threatening the other player's king, and it cannot move to any other squares, cannot be protected by another piece and the checking piece cannot be captured. If all of these conditions are met, the attacking player wins via checkmate. A stalemate is when the player to move has no legal moves but is not in check, it is therefore a draw. If a player chooses to resign, then the opposing player wins.

Updated UML



Junsang Yoo, Seung Hyeok Oh

We have several changes to our updated UML diagram compared to the UML diagram from DD1. This is because we have implemented many more methods that we did not account for and think of during DD1. We have also included many more parameters as well as variables for each class. The UML diagram for the chess project has been updated to include new classes and their relationships. The Chess class now has associations with the Board, Scoreboard, Move, and Observer classes. The Board class has associations with the Piece and Observer classes. The Observer class has associations with the Cli and Gui classes, which are new additions to the project. The Cli and Gui classes both have associations with the Chess class. The Controller class has associations with the Chess, Computer, and Move classes. We still maintained the overall flow and logic and still used the same design pattern, just that some classes gained associations to more classes.

Overview

The project is a implementation of the game of chess. It includes classes for representing the chess board, individual pieces, moves, and the overall game. The project also includes classes for implementing computer players at different levels of difficulty, as well as classes for displaying the game using a command line interface or a graphical user interface.

The main class for the chess game is the Chess class, which maintains the state of the game and provides methods for making moves, checking for check or checkmate, and undoing moves. The Chess class also maintains a Scoreboard object, which tracks the score and any special moves (such as castling or en passant) made during the game.

The Pieces class is an abstract base class that defines the common behavior and attributes of all chess pieces. It includes methods for checking if a move is valid, as well as for checking for special moves such as castling or en passant. Each specific type of piece (e.g. King, Queen, Rook, etc.) is represented by a subclass of Pieces, which overrides the methods as necessary to implement the unique rules and behavior of that piece.

The Move class represents a single move made in the game, and includes information about the original and new positions of the piece, any piece that was captured, and any promotion that occurred. The class also maintains a string to track any special status or comments associated with the move.

The Controller class provides a high-level interface for playing a game of chess, including methods for moving pieces, checking the game state, undoing moves, and handling pawn

promotions. The Controller also has references to Computer objects, which can be used to play against the computer at different levels of difficulty.

The Observer classes, Cli and Gui, provide interfaces for displaying the game state and receiving notifications of changes. The Cli class implements a command line interface, while the Gui class uses the XWindow library to draw the game board and pieces using graphical elements.

The Computer class is an abstract base class that defines the common behavior and attributes of computer players. It includes methods for selecting a move, as well as for adjusting the difficulty level. The specific levels of difficulty (e.g. Level1, Level2, etc.) are represented by subclasses of Computer, which override the methods as necessary to implement the desired level of AI.

Our ‘Board’ class simply represents the 8 × 8 chess board by using a 2D array of characters.

The project save the score of all games in the ‘ScoreBoard’ class.

The enumeration class ‘Position’ keeps track of all the pieces position on the board.

Design.

The project involved several design challenges, which were addressed using various techniques.

One of the main challenges was to implement the different types of chess pieces, each with its own unique set of rules for movement and capture. To solve this challenge, we used the object-oriented programming technique of creating separate classes for each type of piece, with each class containing the specific logic for that piece's movement and capture rules. We also used inheritance to allow the pieces to share common attributes and methods, such as the ability to identify the piece's color and position on the board.

Another challenge was to implement the board and its cells, which needed to be able to store and manage the various pieces on the board. To solve this challenge, We used the concept of a two-dimensional array to represent the board, with each cell of the array storing a pointer to the piece occupying that cell on the board. We also used the observer design pattern to enable the board to notify other classes, such as the GUI and CLI, when the state of the board has changed, such as when a piece is moved or captured.

Another challenge was to implement the different game modes, including the ability to play against the computer with different levels of difficulty. To solve this challenge, We used the strategy design pattern to allow the game to easily switch between different modes, such as human vs. human, human vs. computer, and computer vs. computer. We also implemented different computer player classes, each with its own algorithm for determining its next move, allowing the game to adjust the level of difficulty by using different player classes.

Overall, the project employed various techniques from object-oriented programming, design patterns, and data structures to solve the various design challenges and provide a functional and extensible chess game.

Resilience to Change

Our project is resilient to change because it is designed using the MVC model and observer pattern. This allows for a clear separation of concerns, with the 'Chess' class managing the logic and data of the game, the 'Observer' class and its subclasses handling the display of the game, and the 'Controller' class managing interactions and modifying data. This modular design makes it easy to change or add new features without impacting the overall structure of the project. For example, if we wanted to add a new level of computer play, we could simply create a new subclass of the 'Computer' class without modifying the existing code. Similarly, if we wanted to change the way the game is displayed, we could modify the 'Observer' class or create a new subclass without affecting the logic of the game. This design makes the project resilient to change and allows for easy expansion and modification.

Therefore, our project has high cohesiveness because the classes and functions are logically grouped and work together to achieve a common goal. For example, the 'Chess' class manages the logic and data of the game, the 'Observer' class and its subclasses handle the display of the game, and the 'Controller' class manages interactions and modifies data. Each class has a clear and well-defined purpose, and all the classes work together to implement the chess game. This high cohesiveness makes the project easy to understand and maintain.

Our project also has low coupling because the classes are independent and do not depend on each other's implementation details. For example, the 'Chess' class does not need to know how the game is displayed or how the user interacts with it. It only needs to provide the necessary information and functions for the other classes to use. This low coupling makes the project more flexible and allows for easier modification and expansion. For example, if we wanted to change the way the game is displayed, we could modify the 'Observer' class or

create a new subclass without affecting the logic of the game. This low coupling makes the project more resilient to change and allows for easier maintenance and development.

Answers to Questions

Our answers to question 1 and question 3 remains unchanged, but our answer to question 2 has changed as we decided to implement the undo function and the question is therefore relevant to our project.

Question: 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.

If we were required to implement a book of standard openings, we would first need to decide on a format for storing the information. One option would be to use a two-dimensional array, with the first dimension representing the moves and the second dimension representing the responses. Another option would be to use a linked list, where each node represents a move and contains a pointer to the next node, which represents the response to that move. We can then store this information into the 'StandardBook' class and modify the notify() function in each concrete observer classes, 'CLI' and 'GUI' to output suggestions. Then we can edit the main function to contain the 'StandardBook' class, add the function to suggest when the game starts, and create the new text/graphical observers for suggestion, but before the first player makes the first move. Then, the case that the first player get the suggestion is handled. Now, modify the 'Chess' class to contain the 'StandardBook' class and add more functions in the 'Chess' class so that if opponent's move is valid, create a move-applied new board. Then print out the new board with the suggestion through the observers.

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

To implement a feature that allows a player to undo their last move, we would need to add a new function, 'undo', to the 'Chess' class. This function would take the current board and move information as input, and would return the previous board and move information. The 'undo' function would first check if there are any previous moves to undo. If there are, it would retrieve the previous board and move information from a stack data structure (using a

vector), which we would need to add to the 'Chess' class to store the history of moves. Then, the function would update the current board and move information with the previous ones and return them. Because we chose to store just the moves and not the state of the board, we do not know where the piece we have to undo has gone through any specific event (e.g capturing an opponent's piece/castling/enpassant/etc.). That is why we decided to store the events in the 'Board' class creating functions like 'undoPromoted', 'undoEnPassant' and 'undoCastling' in case they happen so that we can call them to undo the event that the move has caused. E.g if a pawn captures an opponent's rook, when we undo the move we will move back the pawn and replace the previous square with the previously captured rook. This feature also allows an unlimited number of undos as well as you just have to pop back the most recent move.

This differs from our previously mentioned answer from DD1, as we said that we would implement the undo feature by representing the board as a stack (vector) and popping off the stack to restore the board to make the previous board to the most recent board. This was not possible as we did not decide to store the whole board as this would affect the efficiency of the project.

Question: 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 make our program into a four-handed chess game, the following changes would be necessary:

1. 'Board' would need to be enlarged to accommodate four players.
2. Modify 'ScoreBoard' class for four players.
3. Modify concrete observer classes to change the way we display the game.
4. The rules of the game would need to be modified to accommodate four players. For example, the rules regarding checkmate and stalemate would need to be modified.
5. There would need to be four sets of pieces, one for each player.
6. Modify the 'Computer' class and its level subclasses for four-player chess. The computer players would need to be programmed to take into account the fact that there are four players. For example, the computer players would need to be programmed to make moves that would consider the other three players.
7. The game would need to be structured so that each player takes a turn, then the next player in clockwise order takes a turn, and so on.

Junsang Yoo, Seung Hyeok Oh

Note : Because our implementation is supports changes easily, then we would not have to modify the majority of the functions.

Extra Credit Features

- Implemented a feature to show in the CLI to display the turn of the player (white or black's turn)
- Implemented a feature to allow players to choose whether to continue the game by sending a prompt at the end of a game ("Do you wish to continue?"). If "yes" the game continues with the same people, if "no" then the user can restart the game with different game settings e.g (from human vs human to human vs computer).
- Improved our text display CLI to show Unicode chess pieces instead of chars for a better visual experience.
- Implemented the "undo" function, which allows users to undo their previously made moves, accommodating unlimited undos until the beginning of the game.
- Allowed users to input their names when starting the game (instead of human human) and the names are also shown when the game has ended or stopped(ctrl-d) in the output of the scoreboard.

Final Questions

Question: 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, collaboration, and time management when working in a team. Working together also gave us the opportunity to learn from each other and to strengthen our skills. Additionally, it taught us the importance of planning and having a clear idea of what we wanted to achieve before beginning the development process. We had to make sure to plan out our tasks and assign them to each team member accordingly. We had to make sure that everyone was on the same page and was aware of everyone else's progress. I think this was made possible by using GitHub to update our project and contribute on our individual devices. Furthermore, we also had to make sure that we had frequent meetings to discuss our project and make sure that everyone was on track. Even though we had a plan of attack, difficulties arose when trying to combine our individual parts into a cohesive whole. There were also issues with trying to

debug our program since the code was spread across multiple files. It was important for us to communicate our ideas, troubleshoot the program, and divide up the tasks so that we could get the project finished in a timely manner.

Question: What would you have done differently if you had the chance to start over?

If we could go back and do the project over, we would have done it with more careful planning, including a more specific timeframe and work breakdown. Additionally, we would have made an effort to employ better coding conventions and practices, such as assigning variable names that are more descriptive and providing more in-depth comments. By dividing classes into more manageable chunks and creating more abstract classes, we would have also attempted to enhance the project's design. In addition, we would have scheduled time for testing and debugging and more precisely defined the project's scope before starting work. This would have allowed us to find any bugs earlier in the development process and fix them, thus saving us time. Additionally, we would have spent more time testing because we discovered that our code did not always function as intended. Instead of rushing particular features to increase time for testing and debugging, we would have tried to pace ourselves consistently and try to solve our difficulties one day at a time. We also came to the realization that because our chess program had so many interconnected classes, it was challenging to correctly code out a class for a certain feature without the completion of the other components. We also believe that we would have developed the chess program first using the CLI text display, ensuring that we addressed all the logic (check, checkmate, etc.) before moving on to the GUI. Not only that, but I believe that implementing setup first, then working on the chess pieces one at a time and making sure they functioned properly before proceeding, would have saved us a great deal of time because we would have known exactly where the errors or bugs came from our later implementations.

Conclusion

In conclusion, the implementation of the Chess game in C++ using the MVC model and observer pattern was successful. The project consisted of various virtual abstract classes and their subclasses, including the model class 'Chess', which managed application data and modified it as needed. The 'Observer' class was the virtual abstract class for the 'CLI' and 'GUI' classes, which handled the command line interface and graphical user interface, respectively. The 'Computer' virtual abstract class had four subclasses, each handling a different level of computer play. The 'Board' class represented the 8x8 chess board, and the

Junsang Yoo, Seung Hyeok Oh

'Pieces' class was the virtual abstract class for the six classes representing each game piece. The 'ScoreBoard' class kept track of game scores, and the 'Position' enumeration class kept track of the positions of all pieces on the board. The 'Controller' class managed interactions and modified data by calling the 'movePiece' function in the 'Chess' class. The main function handled user input and controlled the flow of the game. Overall, this project successfully implemented a functioning chess game with the ability to support human vs computer play at different levels. Although we had many changes and improvements, I think we have completed the project to the best of our abilities, implementing several extra features as well.