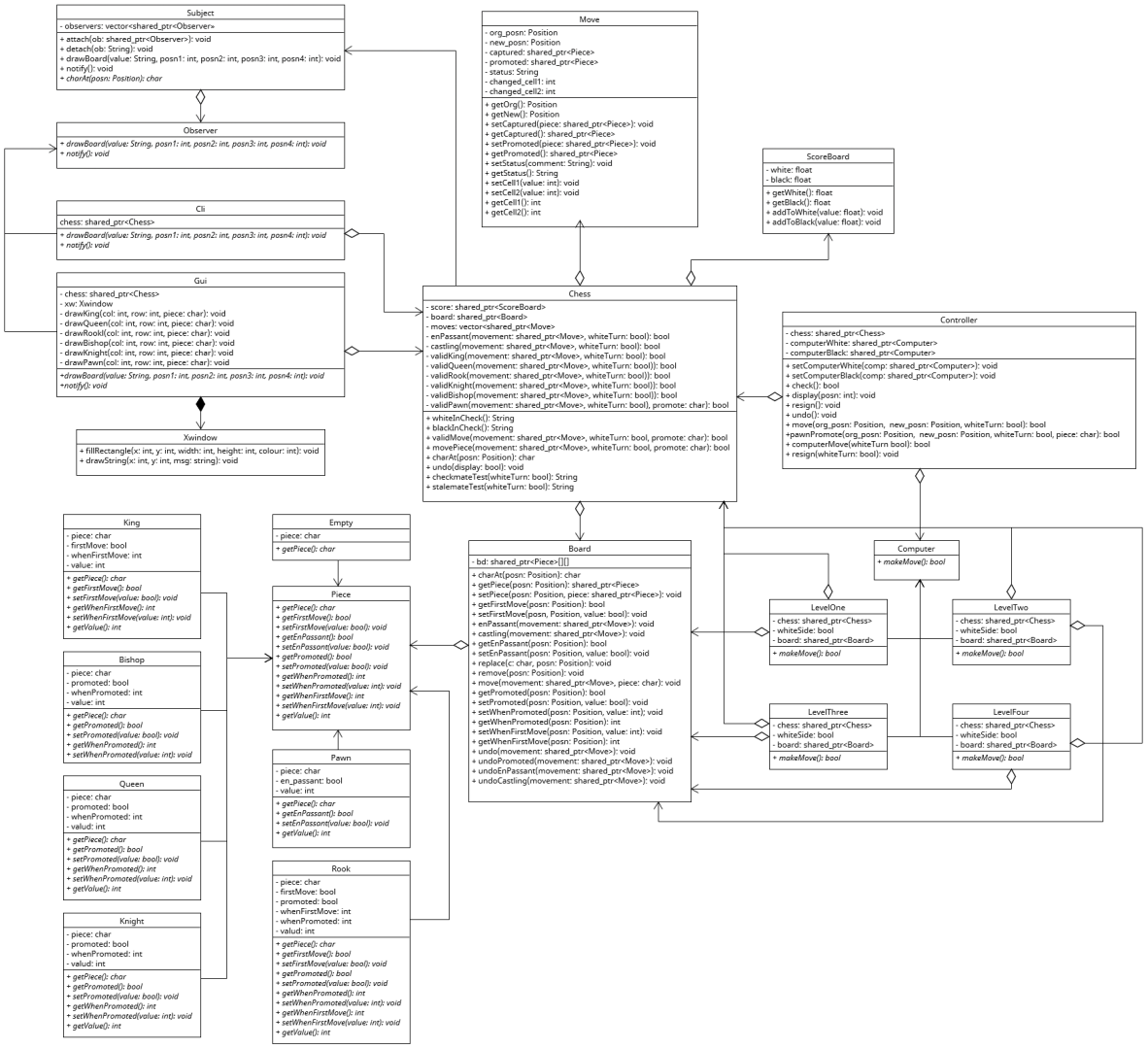
Junsang Yoo, Seung Hyeok Oh

CS246 – A5 Plan of Attack (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 compated 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. Talk about how the updated UML differs from the original UML, Talk about how we added additional classes(specific) and added more methods needed. Talk about how we still maintained the overall flow and logic and still used the same design pattern.

Overview

The overall project consists of several virtual abstract classes and their subclasses. The model class of the project is ‘Chess’ which is the subclass of the ‘Subject’ class. It manages application data and modifies the data.

The ‘Observer’ class is the virtual abstract class and is the virtual abstract class of ‘CLI’ and ‘GUI’ classes. The ‘CLI’ prints out the game through the command line interface, and the ‘GUI’ class displays the game on the graphical user interface through Xwindow.

Since our chess game supports human vs computer with four different levels, there is the ‘Computer’ virtual abstract class with four subclasses, where each class handles the corresponding level.

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

There are six classes representing each game piece (King, Queen, Rook, Bishop, Knight, and Pawn), and their virtual abstract class, ‘Pieces’.

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.

We have decided to create new classes

Design

We applied the MVC model to the observer pattern, where the ‘Chess’ class is the Model in MVC, and is the concrete subject in the observer pattern. Thus, the ‘Chess’ class will take the responsibility of managing all the logic of the chess game, e.g., valid move, check, checkmate, stalemate, castling, etc. The ‘Observer’ class, as well as its subclasses, ‘CLI’

and ‘GUI’ classes, takes the role of the View. The ‘Controller’ class manages interaction and modifies data by calling ‘movePiece’ function from the ‘Chess’ class.

If it is the turn of the human player, the program gets information about a move from the player and the move() function in the ‘Controller’ class is called from the main. Otherwise, if it is the computer's turn, the ‘Controller’ class calls nextMove() function in the chosen level class, and get the position of next move. Then, the ‘Computer’ subclasses

Now, the ‘Chess’ class checks all the required logic based on the given information about the move. If everything is okay, it creates the new board for the move and notifies ‘CLI’ and ‘GUI’ classes. If the game ends, it modifies the score by calling functions of ‘ScoreBoard’ class.

Note: the main.cc is not in the UML, but user can modify starting board by setup command. Also, when the program ends, the main function prints out the overall score.

Then, the concrete observer classes displays in their own way to users.

Resilience to Change

Talk about MVC model + talk about how if we want to change a certain feature we can easily edit the changes in the particular class in the program file. Because our project is modular due to OOP we can just change the class and it will work and not negatively affect the other classes and run smoothly.

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 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 support undoing moves, we decided to store the previous moves of the chess pieces as a vector. We decided to do this as we did not want to store the entire board each time we made a move as that would be very inefficient. But 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 case they happen so that we can undo the move as well as 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:

Junsang Yoo, Seung Hyeok Oh

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.

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 scoreboard output.

Final Questions

Junsang Yoo, Seung Hyeok Oh

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 continuing, 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

Although we had many changes and improvements, I think we have done and completed the project to the best of our abilities, implementing several extra features as well.