Chess Design Plan | CS246 - A5

Introduction

Through the past two weeks, we have been working on designing and implementing chess, a game known world-wide for its strategic and competitive nature, using Object-Oriented Programming concepts learned in CS246. In this design plan, we will provide an overview of the overall structure of the project, any challenges we encountered and a reflection of the overall project.

Overview

The nature of the project brought upon many challenges when designing the game as there were so many components and functionalities that needed to be considered as well as their integration into the final game. This design challenge was solved by dividing the project up into 4 smaller individual components: the core of the game, the display element, the player/computer interaction (command interpreter) and lastly, the game and its pieces.

1. Core of the game

The core of the game is what handles the games decision making under the Game class. In terms of functionality, the Game class decides if entered moves are valid for the appropriate chess piece, undoing moves, if the game is in stalemate, check, checkmate or if the game should continue as well as handling all the special moves such as castling, en passant and the pawn double-move. In addition, the Game class sets up the default or custom board and handles moving the individual chess pieces.

Since the primary function of the Game class is to manage the game of chess (which will involve moving the chess pieces), naturally the Game class will be coupled relatively highly to the Piece class. In an effort to minimize this coupling, the Game class implements all the necessary checks required to move a piece (e.g. ensure the move is within the board, a player isn't moving themselves into check, etc.) and leaves the responsibility of actually moving the pieces to the Piece class. The Piece class has no knowledge of what pieces are on the board or where they are. This has the benefit of making the game change-resistant since any changes to how pieces should move are localized entirely to the Game class - the Piece class will not change. This also has the advantage of preventing the recompilation of the various piece classes since they do not require changes.

Handling decision making in chess is relatively complicated given the rules of the game. On every move, the Game class will determine the state of the game - the state can be one of stalemate, checkmate, or valid (a game is valid if the current player has at least one valid move).

If the state is either stalemate or checkmate the Game class will return to the Main class with the result of the game. Otherwise, the game attempts to read input and if input is read, it will determine if the input represents a valid move - and perform the move if so, or produce an error message otherwise.

Determining if a move is valid is a relatively complicated process. First, the game class ensures that the input represents a valid move within the 8x8 chess board. If this is satisfied, the game determines if the specified tile contains a piece and the tile that will be moved to contains either no piece or a piece of the opposite team. If both of these are satisfied, the Game class checks to see if the specified piece can actually move to the specified tile (e.g. one tile forward for a pawn, in an L-shape for knight, etc.). Lastly, the Game class checks that performing such a move would not put the player into check. If these are all satisfied, then the piece is moved and the board redisplayed.

If the move is valid, it is performed and then it is pushed onto the move stack. If the command “undo” is called, the top element of the move stack is popped off, read, and the opposite of it performed to undo the previous move. This has the advantage of allowing an unlimited amount of undo moves, up to the point when the move stack is empty and the board is in its default state.

Our implementation of chess has the functionality of setting up a custom board. Additionally, since most games are not played on a custom board we have also implemented a defaultSetup method, which constructs the board to resemble the normal setup most people are familiar with.

The act of determining a checkmate is very similar to checking for stalemate. A player is in checkmate if their king is in check and they have no valid moves, similarly, a game is in stalemate if the current player has no valid moves. To determine if a game is in stalemate, the computer checks all pieces of the current player and generates all moves. If none of these moves are valid, then the game is in stalemate. Likewise, to determine if a player is in checkmate, the Game class determines if the player is in check by checking if the opposite player can make a valid move to the current player's kings’ position. If this is possible, the game then checks for stalemate. If both of these are true the current player is in checkmate and the game ends.

2. Display

The display element of the game is represented through an observer. When a game is initialized, a gameManager class is created which will display the board when desired. This gameManager class inherits from the Subject class which implements the observer pattern. When the gameManager calls the displayBoard() method, each one of the textDisplay and graphicDisplay class will run their display() methods which will display the board using the

information about the pieces from the gameManager class. This has the advantage of easily supporting multiple games such a change was desired.

3. Command interpreter

The entirety of reading in moves and interpreting the results is done in the Human class. When input is expected, the getMove() method is called which listens for one piece of input in the form of <*letter*><*number*> (e.g. g e8, a4, etc.) and returns a pair of integers in the form of a standard pair. If the input is malformed the pair {-1, -1} is returned instead. Input is malformed if it does not contain exactly one letter followed by exactly one number.

4. Chess pieces

The chess pieces were implemented using the decorator pattern. All the chess pieces are defined in their own class type and inherit from the Piece abstract class. Additionally, there is another class Blank that represents a blank tile. Each piece maintains fields for their x and y-values which represent their location on the board, as well as a boolean value to track if they are alive and a pointer to the next piece on the board. When the game desires to perform a specific operation on a specific piece (like moving a piece), the Game class will call that function on the first piece, passing the coordinates of the piece that it wants to perform the action on. If the current piece's coordinates match those that were passed, the action is performed. Otherwise, the piece calls the same function on the next piece using the next field.

In the piece class itself, there is no invariant surrounding the board or dimensions. This means that the board can be expanded to any dimensions without requiring any changes to the Piece classes.

Resilience to Change

The classes of our game have been intentionally designed to be modular and resistant to change. Each class handles one specific aspect of the game and as such the program has very low coupling and high cohesion. Changes to input would be contained in the Player class, changes to the game rules would be contained in the Game class, changes to the displays and output would be contained in the Observer class and its derived classes, and changes to the pieces themselves would be contained in the Piece class and its derived classes. This has the overall effect of making the program resistant to change and when changes are required, recompilation will be minimized.

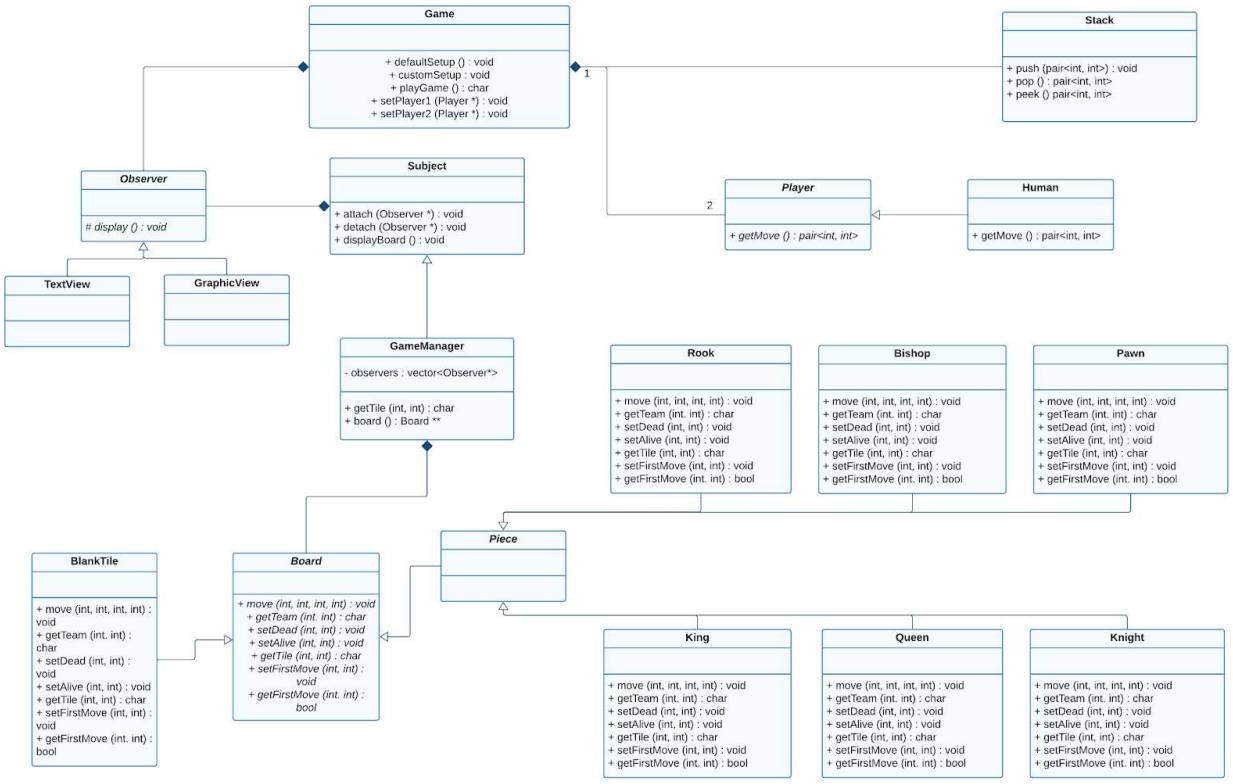
Potential Changes:

- If we decided to add another piece type to the game, we would have to create an implementation for the new piece class which would inherit from the Piece class. Additionally, we would add a new valid move method to the game class to accommodate

the new movement rule that the piece would follow, since this would effectively change the rules of the game. Since our design is modular, these are the only required changes.

- Adding values to pieces would be relatively easy. Since changes to the board are performed in the Game class, we could add two fields to accommodate a score for each player. Additionally, everytime a piece is moved, the information of what piece (if any) would be taken by such a move is already recorded - the player's score can then be incremented as necessary. A value field would not be necessary to add to any piece classes.
- If we decided to change our input to file form, this change would be easily done and localized to the Human class. We have already defined an standard input stream as std::cin in the Human class - changing this input stream to a file would be as easy as modifying this one field.
- Similarly to the potential change above, all output is passed using an output stream define in the game class, which could be easily changed should we not want to output to the standard output.

Updated UML



Design

Some challenges we faced during designing were that our original UML and plan caused unnecessary amounts of recomputation and was generally overengineered, leading to higher coupling. We needed to find a way to reduce the unnecessary functions and have more independence within our code. To do this, we abstracted methods in the Piece classes and increased the cohesion of the Game class. Furthermore, Insead of having the player class obtain the move and then also do the moving - we changed the player class to purely handle input and return that processed input instead. Additionally, many functions that we anticipated needing were actually unneeded (such as *GetX* and *GetY*) so they were removed and their role was filled with a more general purpose function - if the role was still necessary.

Answers to Questions

Q: Discuss how you would implement a book of standard openings if required.
A: To store a single move, we would use a pair data structure, where the first item represents a piece to move and the second represents the tile to move it to. Then, storing these in an array would produce a list of standard moves. We can then store an array of these standard moves to get a book of standard moves (sorted by effectiveness). This can then be implemented in the Computer player class with the level of difficulty corresponding to a single set of standard moves in our array of standard moves. Additionally, if we wished to use the book of moves as a recommendation for the player, we could add a command "help" that recommends a move from the book - of course, this is only effective if the player has made previous moves (somewhat) similar to what is in the book.

Q: How would you implement a feature that would allow a player to undo their last move?
A: To implement an "undo" feature, each move that is made will be stored in a stack data structure. Once the "undo" command is invoked the principal element in the move-stack will be read, the opposite of it performed (to undo the move), and then popped off the moving stack. This requires adding an *undoMove* function in the piece class since not all undo moves will be valid (e.g. a pawn cannot move backward) so they should be handled separately. Additionally, any taken pieces will be returned to life, which is as easy as setting their isAlive field back to true. In this, we can support an unlimited amount of undo move commands, until the stack is empty and the game is in its default state - in which case the notion of undoing a move no longer applies.

Q: Outline the changes that would be necessary to make your program into a four-handed chess game.
A: The four-handed chess board can be represented as a 14x14 board - this modification will be easily done by modifying the dimension fields in the Board class. Additionally, the 3x3 corners of the board are off-limits and so the individual pieces must have their *validMove* functions

modified so that they cannot move into these corners. Lastly, the gameplay logic will remain the same except the Game class will have to check extra cases for the check, checkmate, and stalemate since there are additional players to account for.

Extra Credit Features

The undo feature in chess is a feature that may save a player from losing the game as this feature allows players to *rewind* the game backwards and continue playing from that instance of the game. To enable an "undo" feature, an *undoMove* function must be added to the piece class to handle invalid moves (such as a pawn moving backward). Additionally, any pieces that have been taken will be revived by setting their isAlive field back to true. Each move taken is stored in a stack data structure, so that when the "undo" command is initiated, the principal element in the move-stack is read, the opposite of it is done (to undo the move), and then it is popped off the move stack. This allows an unlimited number of undo moves until the stack is empty and the game is in its initial state, in which case undoing a move is no longer possible.

Another extra feature added was the ability to set up the board as default or customized.

As an extra feature, we also decided to include the "--help" command which can be utilized during the game to view instructions and how to play and interact with the game. This bonus feature was implemented in the main class by utilizing iostreams.

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: When developing software in teams, the planning stage is more important than the coding and debugging stages, additionally it is important that the planned program be as lowly coupled as possible. This is because poorly designed programs will usually require changes to be made during the coding stage, which if the program is not lowly coupled will cause issues with the other team members code, which can easily snowball into a complete redesign of the program. Additionally, communication between the members with regards to the changing design of the code is imperative (programming pun) since poor communication will likely result in modules that do not interact properly. Generally, when developing software in teams we feel low coupling is the most important factor to get a working program in a timely manner.

Q: What would you have done differently if you had the chance to start over?
A: The use of the decorator pattern to handle the pieces was completely overkill and originally was over-engineered as well. In the future, we would implement a single concrete Piece class that had the type of piece as a field, rather than having an abstract Piece class with one concrete child class for every type of piece. This would have the advantage of making the addition of new pieces even easier since it would not even require the Piece class to change, only the Game class.

Conclusion

Chess, known for its strategic gameplay, is also a game that also requires strategic design and planning to build. By utilizing Object-Oriented Programming concepts, we were able to implement a chess game that consists of 4 core components: the core of the game, the display, the command interpreter and the chess pieces. Bringing all these elements together, we produced a chess game where user interaction is done through the command line and through graphical display, the player can see the game unfold in an appealing and aesthetic manner. In future projects, we will make sure to take extra care when designing our software since it has without a doubt been the most important part of this project. All in all, we are ready to go a long time without ever seeing a game of chess.