CS 246 Final Design Document

Overview

Overall structure of project:
Follows the MVC (Model View Controller) design pattern by assigning our board class as the model, our chess class as the observer, and the player class as the controller. Once the program receives one of five player types: human, robot1, robot2, robot3, robot4, a game is initiated by our chess class. Based on the project specifications, human player takes in moves from standard input while robot1 - 4 makes moves based on their respective algorithm and move priorities. Once the input is handled by our controller, the moves themselves are made by our board and respective piece classes. We are able to check for a valid moves in each piece subclass based on their movement patterns. As such, we can give appropriate errors when a user enters an invalid move while ensuring all potential moves that our robot player classes select from are indeed valid. After every move is made, we notify our observers with our chess class being the concrete subject. We attach the appropriate concrete observers (textrender and graphics render) which creates the appropriate displays.

Changes from original UML:

- Added additional fields to our board class including functions that check the current board state for checkmate, check, stalemate, and two king draw. We added this functionality to the board class instead of checking for this within our pieces which we were originally planning
- Added additional parameters to our piece class including type, moved, and canEnpassant. These were added to accommodate special moves and functionality such as castling, and en passant for the king and pawn subclasses. We also included additional getters to get the type and team of a particular piece along with whether or not the piece has moved.
- Inside the chess class, we added an additional gameRunning parameter to check when we are able to enter setup mode. That is cases where the game has been initialized however, no move has been made. We also included a white win and black win counters to keep track of the total score which will be printed at the end. We also added a general function takeTurn which initializes our move sequence.
- We also made additions to our graphics observer and Xwindows class to accommodate new functions to draw the board to our desired specifications and functions to draw letter symbols

Updated UML

See attached pdf

Design

Problem: Implementing text and graphical display
Solution: Observer class

To create the two displays (text and graphics), we utilized the observer pattern which we explored in class. As such, everytime we needed to print a new board within our viewer (chess), we would notify all observers. As a result, after every move is made, or in setup mode, whenever a new piece is added or deleted, then all observers will also be notified to ensure the user is notified of the new board state.

Problem: Implementing different types of players that make different move decisions:
Solution: Player abstract class

Our abstract player class contains only one field, the team. This class serves as the controller in the Model-View-Control design pattern. It also has three methods: constructor, destructor, and a move function which returns a board type. As such, the class human, robot1, robot2, robot3, and robot4 are all subclasses of our abstract player class. In each player, we override the constructor and more importantly, the move function. As a result, we are able to ensure that the function move has different behavior depending on which kind of player is called. For instance, a human player will read commands from standard input when the move is called.

Furthermore, for classes robot1, robot2, robot3, and robot4, we developed an algorithm to make a move. For each level of robot, depending on the specifications from the assignment, it will generate an array of possible moves and then select one of them at random. As per the assignment specifications, robot1 treats all valid moves as equal and selects one at random while robot2 and robot3 prioritizes moves that check and capture (and escape for robot3). We are able to filter if a move delivers check or captures an enemy piece by a boolean parameter in our validmove() function which is in our piece class. As such, since the assignment does not specify a hierarchy between check, capture, and escape, a move that satisfies any one of those conditions is stored in a higher priority vector. When we are looking at all possible moves, we first look at moves which deliver check, captures an enemy, or escapes being captured first. If none of these moves exist, we then explore a secondary array which contains all other possible moves. For robot4, we included an additional parameter which detects if a move delivers instant checkmate. As such, our level4 robot will always identify a one move checkmate. Note that our computer will always find a valid move because our board class is always checking for potential checkmate and stalemate. As such, our robot class is always guaranteed to find a valid legal move.

Problem: Ensuring every piece type has a unique function to check valid moves, and accommodate special moves if needed (ie: castle and enpassant)
Solution: Piece abstract class:

We also utilized an abstract piece class with all the specific pieces (rook, queen, pawn, etc.) as child classes. As such, we wanted all classes to inherit certain properties such as team, type, whether the piece is currently under capture, and the current position of the piece. Additionally, we are able to override the validmove() function since each piece moves differently resulting in a different set of valid moves. This allows us to detect various boardstates. We are able to utilize the valid-move function of each piece to check for board states such as check-mate, check, or draw.

Problem: allowing En passant/Castling/Promotion
Solution: Adding additional parameters and checks in the valid moves of the King and Pawn subclasses

We were also able to solve the special cases of en passant, castling, and promotion by implementing additional private fields and methods within the pawn and king classes. As such, we are able to keep track of whether or not the piece has moved and whether or not it has moved by two spaces in the case of a pawn. Furthermore, by setting the en-passant field as true in pawn class, it enables movement that would otherwise not be allowed. Similarly with castling, we are able to call upon parameters whether or not the pieces (king and pawn have moved). In the case of promotion, we are able to detect cases of promotion by directly getting the position and destination of the pawn in order to replace the pawn with the desired promotion piece when appropriate.

Problem: Undo function
Solution: two-pointers in our subject

Addressing the undo function, we just needed to store two-pointers to two separate boards. One being the current board and the other being the previous board. As such, as long as a valid game is running, and more than a single move is made, we are able to delete the current board pointer and set our current board to the previous board, and set prev board pointer to be nullptr. In the case where prev board ptr is already nullptr when undo() is called, nothing will happen because this indicates that no previous board exists.

Resilience to change

By following the observer and Model-View-Control design pattern, our program is quite resilient to change if it should occur. As such, many changes will only require us to change small portions of the program while leaving the rest largely the same.

A change to the rules of chess, for instance how certain pieces move can easily be handled by altering that specific piece file. Since functions that control move and valid board are overridden by the subclasses of our abstract class piece, a change to once piece can easily be accommodated by only changing that particular piece.cc file. As such, only the valid-move function for that piece needs to be changed leaving the rest untouched. Furthermore, the addition of new pieces that move in new and unique ways can also be accommodated by merely adding a new subclass that inherits from our abstract piece class. We can override the valid move method to add new moving functionality to that specific piece without impacting existing pieces. Furthermore, only small changes need to be made in the game setup and default board to construct this new piece type.

A change to the input method can also be easily accommodated as reading from standard input is restricted to only the controller (chess.cc) and the human player. As such, any changes to the input syntax require only minor adjustments to how the input is detected and which corresponding commands are called. This is another advantage of utilizing the Model View Controller design pattern as we don’t need to touch any of the pieces or board functionality and logic in order to accommodate this change.

If we were to add additional features such as a new type of AI/computer opponent, this can also be handled once again due to the Model View Controller design. The control of our program is restricted to the Player class as that is where the move command for that player is determined (reading input from standard input for human players and choosing a move based on a preset algorithm for our robot/computer levels). As such, if we wanted to add a new level that has a different hierarchy/priority on moves, we can create the logic and then use our board interface to test and determine all valid moves that satisfy the new conditions we want to impose without ever changing how valid-move is written. As such, we can limit all changes purely to the player portion of our program.

If we wanted to customize user-interface elements and the graphical display of our gameboard, that can also be accommodated by adding and/or changing our current concrete observers without any change to the logic or control of how the program functions. As such, we can limit all changes that we want to make to our graphical display within the graphical observer class. If we wanted to add functions that drew additional graphical elements, that can be added in our observer classes without impacting the rest of our program such as the board and players. Furthermore, if we wanted to change the appearance of our pieces (i.e. from drawing letters to drawing the shape of the actual piece), we just need to change the function within our graphical observer. Therefore, we can accommodate changes to the way we draw pieces, changing colors, etc.

As a result, our implementation ensures low coupling and high cohesion which helps us accommodate new changes.

Answers to Questions

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.

Note that we can represent the current state of the game through a sequence of moves which can be represented as a string. Every move will be a string of length 4 as it contains the starting square of the piece and the ending square of the piece i.e. e2e4 means the piece at e2 moves to e4. Since the starting board is constant, we will always know the board state on any particular move. As such, we can match all openings that follow the current opening sequence and provide the continuation of that opening as suggested moves for the player

To start, we will have two main maps.

The first map will contain the move sequence (string) as the key and and the name of the opening (string) as the value.

The second map will contain the name of the opening (string) as the key and the full sequence of the opening (string) as the value

By keeping count of the number of moves, we know the length of the move sequence. Every move is of length 4. As such, we know at the beginning of the game, our game sequence string is of length 0. After move 1, our game sequence string is of length 4. In general, the game sequence string will be of length 4n where n is the number of moves that have been played.

We will put all opening sequences greater than length 0, 4, 8, … and so on into a vector of maps. Each map will contain a key-value pair where the key is the move sequence and the value is the name of the opening. Index 0 contains all key-value pairs where the move sequence (key) is of length >= 0. Similarly, index 1 contains all key-value pairs where the move sequence (key) is of length >= 4 and so on.

As a result, during the game, when the total step count is 0, we will do vector[0] to access all potential openings. It will return a map where all the opening sequences (keys) are of length 0 or greater. If there are multiple potential openings, it will pick one at random.

Once the step count is 1, we will do vector[1] which returns a map where all the opening sequences (keys) are of length 4 or greater. Once step count is 2, we will do vector[2] which returns a map where all the opening sequences (keys) are of length 4 or greater. We will keep track of the values (opening names) of all elements that match the current sequence of the game. Through this process, we will know which openings the current game is following.

At the same time, each step, we will utilize our second map with the key as the opening name and the value as the entire sequence of moves in that given opening. Additionally, we also have knowledge of the current step count of the game. Thus, we know the exact step we are in the opening sequence and know the next steps that can be recommended to the user.

If the current move sequence is not found in the map, then we know the game does not follow any map opening and thus, we don't need to compute this any further.

If the current opening does not have any further moves, then there might be other potential opening names that extend off the current sequence. Thus, we will search for openings that extend further steps. Otherwise, there are no further suggestions for the user as the game no longer follows a recorded opening sequence.

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

As UML shows, we will set a previous board (starting pointer is nullptr) and the current board pointer in the chess class, if we want to undo, then we will delete the current board pointer, then let the current board pointer = previous board pointer, then let the previous board pointer = nullptr. There are only two cases when the previous board pointer = nullptr , case 1: the first case is when the game has just started, and case 2: the game has already been undo once. Note that: When the previous board pointer = nullptr, undo is no longer legal, in order to satisfy the rule that you can only undo once in real chess. Finally, graphics render and text render the withdrawn board.

If we need to undo unlimited times, we need to use the standard template library - vector<string> , which records each player's move as the index changes, e.g. vector <string> v {"a1 a2", "b1 b2"}, then v[0] ("a1 a2") means that white's first move was from a1 to a2, and v[1] ("b1 b2") means that the black side played b1 to b2, and so on to keep track of how each player played each move. This makes it easy to undo, for example, if we need to undo n times then we just need to remove the last n strings in the vector, and then replay each remaining move in the vector on a new board according to v[0], v[1] and so on, and finally we get the board after undoing n moves. Note: 1 <= n < len(vector) where n is an integer.

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.

- Extend the size of the board
    - Change boundaries for detecting valid move
- Increase player count/computer opponents
- Increase to 4 players and have turns alternate across all 4 players
- Have options for 4 teams instead of just black/white
- Give players choice to work as teams/free for all
- Include game modes (i.e. first to checkmate or play for points)

Extra Credit Features

Board display Xwindow optimization
Everytime the graphics renderer is called, instead of rendering the entire board, we can optimize this process by only re-rendering the squares that were changed. As such, any move will only ever change two squares on the board. As such, by keeping track of the current move, we only need to re-render the square the piece was moved from and the destination square. This makes the program much more efficient because instead of rendering 64 squares every move, we only render 2.

Smart pawn promotion. Robot 4 will promote the correct piece to ensure checkmate when possible. (i.e. will select knight over queen if that guarantees checkmate)

There are scenarios where upon promotion, promoting a pawn to a knight will achieve an instant checkmate however, promoting to a queen will not. As such, we implemented additional checks so that the robot will test if promotion to either a knight or a queen will result in a checkmate (note that testing promotion to other valid pieces, rook and bishop, is irrelevant because any potential checkmate will also be achieved by promoting to a queen and a queen is of much greater value). As a result, in scenarios of pawn promotion for robot4, it will make the appropriate decision of promoting to the correct piece if a checkmate is possible.

Game results in draw if only two kings are left:

When there are only two kings on the board, this state is undefined in the assignment. As such, both teams will continue moving forever since it is impossible for either team to deliver checkmate or stalemate and end the game. As two kings is a proven draw and the feature where only two kings remain is an instant draw appears on numerous platforms such as lichess and chess.com. As such, we have also implemented additional checks in our board function to detect this board state and declare a draw awarding both teams 0.5 points when this occurs.

Blank board during setup.

We added an additional command the wipes the board clean if blank is called during set-up mode. It will wipe the board of all pieces. This was a significant quality of life upgrade as many custom scenarios we wanted to test only required a few pieces. As such, without this command, it requires you to manually delete every piece.

Truly random

Using rand will use the same sequence of random numbers each time; however, using srand(time(0)) generates a new seed each time the program is run. This is because time(0) will be different every time the program is run. As such, we are able to get truly random moves every single game as the generated random numbers are unique. Therefore, we won't repeat the same sequence of random numbers (and thus same move sequence) which results in truly random moves for our robot players.

Final Questions

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?

We learned the importance of clear communication and planning, in particular having a clear understanding and constantly updating the UML and how the program is coupled. There were certain instances where one member would write code based on the current UML however, due to some unforeseen changes, we needed to re-work which methods to put in which classes (i.e. overriding moveto and validboard methods in each child of the piece abstract class OR changing reading from standard input from our main to our chess function to adhere to the MVC design pattern). As such, functions that other members are currently working on or have already completed also needed to be adjusted. As a result, we recognized that clear communication throughout the process was crucial to ensure everyone was aware of the changes and updates and the impact they have on other classes in the program.

We also learned the importance of design patterns as it allowed us all to simultaneously work on various portions of the project without needing the implementation details of every single file. As such, because we already developed an interface for our board, pieces, and player classes, we could divide our work and develop these classes simultaneously by relying on the fact that the interface functions will be completed soon. As such, this afforded us a certain degree of efficiency as we didn't need to all look at the same file before moving to the next.

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

If we were to start over, we would have more carefully considered the special cases of en passant, castling, and promotion along with the implementation of our robot levels. We ended up adding additional parameters to the valid-move function of each piece as we wanted to keep track of things such as if the piece can escape, can capture, or can check an opponent piece. We ended up adding these parameters later on as it allowed us to better filter the valid moves and create a priority for which moves the computer should consider.