Design A5

Chess

HyeonJeong Kwon | Sharon Park | Jae Ho Jung |

Overview

Controller

Our group has used the Model-View-Controller pattern to perform two major tasks: initializing and executing the game after reading the user's command in the form of standard input. Our controller comes in the form of an object of the "Controller" class that calls methods in the board and player class, without seeing or maintaining any of their field values. The controller cannot directly modify these fields, and as such, any of the data regarding the game. In addition, the controller instantiates both a text and graphic display that shows the contents of the chess board and reflects the position of the pieces as they change during the game. In this way, the displays (view) and the model (board, pieces, players) never come into contact, communicating solely through the controller. This helps to maintain high cohesion and low coupling.

During the initialization stage, the controller allows the user to customize the game setting (adding or removing pieces and setting the first player to move) and the level of difficulty if playing with the computer. The program only allows setup while the player is not playing the game. If the player does not set up the chess pieces and the player's turn manually, it automatically sets up the chess pieces and assumes that the white player plays first. The controller class also checks if the board meets the requirement for the initialization (check condition, unique king for each player, valid pawn location). Lastly, the controller initializes the players with the given type human/computer[1-3] according to the user's input.

Finally, in the execution stage, the user may input one of three different commands: undo, move, or resign. The undo function reverts back one step such that the player may re-enter their command if they want to. This can only be done if the most recently executed move was made by a human player.

All exceptions thrown in our data classes for errors such as invalid commands, invalid setup, and invalid moves are handled within the controller with the use of runtime_error and throw-catch.

Design Patterns

Subject (Chess Subject)

The observer pattern was used to notify and update the text and graphic displays (observers) of any modifications on the chessboard (subjects) due to addition or deletion of a chess piece in the setup mode or move or undo during the game.

Board (Chess Board)

All data regarding the game state are stored and modified in the Board class. The Board class contains a chessboard, the 2D array of size 8x8, which contains Piece pointers. This is where special moves such as en passant, castling, capturing are checked to be valid and performed. If the move is invalid, the board class throws an error to the controller.

Move class

The move class stores different features of each move performed by each player to allow for the undo functionality. There are 2 scenarios in which undo can be called: when requested by the player to undo their most recent move and to return pieces to their original place after simulating their movement given a move (this is done internally to determine if the move is valid, for check, stalemate, etc. and is not observed by the player). The initial position, final position, move type (in enumeration class: Castling, Enpassant, Capture, Promotion, Normal), pointer to the piece performing the move, as well as any opponent pieces that were captured (can be empty) are stored in the structure.

Piece

The piece class is divided into six corresponding piece subclasses: king, queen, rook, knight, bishop, and pawn as given in a classic chess game. Each piece contains the color, the current position, and the piece name.

Move input from the controller is to be checked using overwritten virtual functions IsValidMove() in each subclass according to the piece name. Furthermore, the vector of all possible moves for a given piece is returned in a function getposMoves() so that random moves can be generated for different computer levels as well as to check for special cases such as stalemate, check, and checkmate while the game is being played.

Enumeration class for features of Piece

Various enumeration classes are used to store certain defining features such as the colour of each player/piece (black/white) or the piecename used to denote the type of piece (ex. King, Queen, Bishop, etc.).

Player

The Player base class leads to two subclasses: computer and human. Each player object contains its own score as to avoid any confusion or corruption of these values in several rounds of game. Furthermore, the computer class utilizes a function to generate AI moves depending on the level (one of 1, 2, or 3) that has been implemented.

Observer

As mentioned earlier, the observer pattern was used in order to print two different types of display: text and graphic display. There are two concrete classes, TextObserver and the GraphicsObserver, which observe changes to the chessboard. When they are notified of any updates, each observer redisplay the board with updated cell information.

XWindow

The Xwindow class was utilized to display graphics on the board. This class was used to render the board by drawing rectangles and drawing strings to display the coordinates of the pieces on the side of the boards. The chess pieces were displayed by rendering the bitmap files for the pieces.

Design (describe the specific techniques you used to solve the various design challenges in the project)

1. **Observer Pattern**

   Using the observer pattern, we were able to update both text and graphics observers simultaneously when there are any changes/updates to the board or the piece locations. This allows easier rendering for text and graphic display when needed.

2. **Polymorphism**

   Polymorphism is used both in players and pieces. We were able to use inheritance first as a template to reuse for the remaining pieces and player as the code was similar. Also, polymorphism reduces coupling between different functionalities. This also enhanced the debugging process as each module was located in the different file.

3. **Controller and MVC**

   In order to divide the program into smaller parts, our group used the MVC pattern. This allowed us to divide the roles among each other easily as well as to test and debug easily. We were able to debug and modify/add new features to the program as we developed it since we used MVC.

4. **Encapsulation**

   Through encapsulation, we minimized modification of the fields that are outside of the specific class. For instance, we used functions such as getCoords(), getColour(), getType() to extract features of the piece without modifying. Modification was handled separately to minimize confusion and maximize debugging and testing.

5. **Smart Pointers**

   For handling memory allocation, smart pointers are used in order to minimize the memory leak error. Hence, we were able to program only with smart pointers with no memory leak.

6. **Exception handling**

   Exceptions were handled through try and catch blocks in the controller class. Exceptions thrown in board.cc were also handled in the controller as controller calls functions in board.

Resilience to Change: (describe how your design supports the possibility of various changes to the program specification)

1. **Adding/Removing New Chess Piece**

We have utilized polymorphism and inheritance with the Piece superclass (piece types i.e. Rook, Pawn, Queen, etc.) to create different types of pieces as well as checking if the given move is valid. This makes it easier to add a special type of piece if we want to play a different variant of chess or create your own pieces. When adding a new piece, we need to program its movement logic, hence to make it easier.

2. **Adding/Removing New Players**

As we have used the controller, any data regarding the player, its type and its scores are stored and modified only in the player. Hence, adding and removing players can easily be done without modifying any part of the code except for the player class and the controller class (for input modification).

3. **Adding/Removing New Observers**

When a user plays a game, our program only has one Text Display observer and a Graphics Display observer. In a real implementation of an online chess game we may want to have multiple observers when playing the game. In this case, we can add new observers to the subject class and for any changes to the game, the notifyObservers() function (subject.h) will notify all observers. When removing an observer we can simply detach it from the observers.

4. **Adding new commands/ features**

Because we use MVC, modification in any part of the program/game is easier. If we want to modify some features of a board, we would only need to modify the board class. If we want to update some features of the player, then we would only need to modify the player class. The rest of the program is unaffected from the aforementioned modification. they would be fully functioning as expected. We were able to add the “undo” move to the controller due to the benefit of MVC.

Answers to Questions (Specific to Chess):

1) **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.**
We could implement a book of standard openings in the case where the opposing player is a computer. Calling getAIMoves() returns a vector of possible moves. We could create a list of the standard openings as a map of moves and responses. When a standard opening is encountered, we could compare with one of the moves in the map stored and move the pieces according to the book of standard openings. Even knowing what we do now, this seems like the right way to approach the problem.

2) **How would you implement a feature that would allow a player to undo their last move? What about an unlimited number of undos?**
We can store the moves of each player in the stack (as a vector for easier memory handling) in the order they are played, as well as the player the move belongs to. This way, if a player has just moved their piece and wishes to undo, their move will be at the top of the stack (FILO). At this point, the move can be popped off the stack and the player's piece restored to its previous position. If the last move played did not belong to the player, all moves preceding the player's most recent move can be popped off before the respective move is also popped, returning the board to the state before the player's last move. From there, the next moves can be pushed onto the stack as usual. The same method applies when implementing an unlimited number of undos, the moves can be popped until a certain number of the player's moves are popped off the stack. This corresponds with the method we used to implement the undo functionality in our program.

3) **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.**
For starters, we would have to increase the number of players in the controller to 4 (initialize + push 2 more players into the players vector where they are stored), which would be instances of either the Human or Computer class. As each player is given a colour, for distinguishability the enum class Colour would be changed to support 4 players (perhaps adding blue or red).Then, the board would have to be changed to reflect the number of players with an extra row/column of cells on each side. Pawns would be able to become knights at the far right and left cells of any of the other 3 players' sides. Each player would then take one side of the board with their pieces spanning all 8 cells. In addition, turns would rotate in a clockwise direction rather than go back and forth between two players. Once checkmate is called on a player, they are removed from the game. The winner of the round will be the last Player left standing.

Extra Credit Features (what you did, why they were challenging, how you solved them—if necessary)

Extra Credit Feature 1:

For extra credits, we have implemented a new command for the user: “undo”. This allows a human player to undo one move whenever they want to revert back. The next command line received would be considered as the newly modified move command for the same user and will be executed right after the undo.

The reason why this was challenging was because each move requires different action to revert to the latest state. This required a lot of attention to details so that there are no errors in returning back to the previous state. Also, “uncaputring” requires a Move class so that captured pieces can be stored along with the moving piece and their undo-location.

Extra Credit Feature 2:

Another extra credit feature that we used was to use smart pointers only without “new” and “delete” in our code. The entire code does not leak any memory while we don’t explicitly manage our own memory. This was challenging as there are various files used in the game, and we initially implemented the program using new and regular pointers. By switching out this heap allocated pointer to a shared pointer, there were many instances that had to be changed to reflect this (for example, all the pieces contain a pointer to the Board, which needed to be changed for all 32 pieces). We managed to avoid cyclic reference via controller in order to minimize coupling.

Final Questions:

1. What lessons did this project teach you about developing software in teams?

The most important aspect that all the members of the team had learned is the importance of communication. It was important to notify team members for any changes added to the code. If some members are unaware of the changes and start writing new code it is hard to predict what impacts the new code they write may have. It would be lucky if everything works well, but if that is not the case, it is hard to track down and debug the issues especially as the program gets bigger. This is especially true when working on the same module, but also applies when working on modules that seem to be completely independent.

It was also important to discuss which classes must be implemented first. Simply put, we had to set correct priorities. For example, it would be hard to test whether the graphic display works before implementing move. We wouldn't be able to test if piece movements are rendered correctly on graphics. Hence, it was important for the team to understand the dependencies for each module.

Lastly, and possibly most painstaking was the need for unit testing. Due to the pressure of finishing the project by the deadline, tests were not performed after the creation of each function as they should have been. The result was then hours of debugging to find an issue, break something else, look for the reason behind that issue, then rinse and repeat. We handled bugs as they arose by than testing all together rather than checking after each iteration which caused confusion and unnecessary difficulty when it came to not only identifying but also trying not to repeat the same issue over and over.

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

a) Inline Comments

We would have made inline comments a routine whenever we write new code. Even something simple allows other team members to understand what the code does without having to scrutinize it. Additionally, it would have been better if all members were consistently updated of major changes.

b) Creating and Testing Method by Method

When implementing our program it would have been better if we made sure one feature worked perfectly before we move on to implementing other features. Approaching systematically, would prevent any potential errors in the future. For example, it would have been better if we implemented move on the pawns first and tested it thoroughly before moving onto implementing the other pieces.

c) Better Planning before Implementation

When attempting to integrate your code with code written by your team members, If we started again, we would have discussed how the contributions of each member can be integrated with each other rather than blindly writing code independently.

Conclusion:

Through this project we were able to not only implement various object-oriented design patterns, but also experience how to apply these concepts learned to write large scaled programs. Working as a team was not easy. There were many things that worked but also many that didn't. This project gave us the opportunity to learn about how to work as a team to successfully build large programs and we believe that this experience will be valuable in the future when working as a software engineer as a Co-op or full-time.