CS 246: Object Oriented Programming

Assignment 5: Chess

By: Kriti Sodhi, Jemima Vijayasenan, Malvika Patel

December 6, 2022

Table of Contents

Introduction

As the renowned world chess champion, José Raúl Capablanca once stated, “In order to improve your game, you must study the endgame before everything else.” Provided the three options, our group attempted to create a functioning chess game with text and graphics display for our final project. We used the quote as our motto to consider and implement all the possible cases associated with playing the game. We strived to utilize many of the course concepts such as leveraging polymorphism in our design to create a polished final product. In addition, we implemented the various software engineering design patterns as well as the software development life cycle. The entire project took a considerable amount of time and effort to complete, however, we are extremely proud of our final submission and all the new technical and soft skills we learned and enhanced throughout the process.

Overview

We designed our program to follow the Model-View-Controller (MVC) model so that all user interactions are handled in the controllers, which consist of our main and game class, the view is displayed through the observer pattern in our board class, and the data-related logic is mainly handled in our pieces and player class. To keep our coding environment organized, we divide tasks into 3 main sub-directories: Board, Pieces, and Players. Below we will go into more detail about our controller and each section.

Controller (Game) and Players

To ensure encapsulation, code reusability and maintain abstractness from the client, the main function has minimal code. The main function is responsible for handling two commands from the user: game and set up. If the game command is executed, the game class handles the game between computer vs computer (all four levels), human vs computer and human vs human.

The human vs human aspect handles all user input for the game between two humans. This process starts in the main function where the user chooses between a game command and a setup command. If the user chooses the setup command, the boardSetup function is called from the Game class (our controller). If the game command is prompted, a new game is initiated with a classic board setup of a regular chess game. However, if the user chooses to use a personalized game board setup before starting the game, they are able to set up their game board and use that set gameboard once the game has been initiated. In the implementation of the commands received from user input, we ensured that misspelled commands prevent the program from terminating and instead, the program prompts the user to re-enter their command. Once the game has initiated, the players are created and this invokes the corresponding player classes (human, level one computer, level two computer, level three computer and level four computer).

The overarching player class is the base class for all five types of players. Once in the corresponding player classes, the makeMove method is invoked from the board class based on the user input, provided the player is a human. In the player classes, all possible move cases (two types of castling, pawn promotion, en passant and regular chess moves) are handled considering they are valid moves available on the current game board. In the computer classes, a random legal move is generated using the move generator to make the move. However, these random legal moves have their corresponding hierarchy based on the computer level. For level one random legal moves are generated based on the player’s turn. For those random moves, one move is chosen at random to be executed.

For level two, the move generator randomizes legal moves based on preferring capturing the opponent's pieces over regular legal moves. It checks over other moves to ensure if any capturing moves are available, if so those moves are given first preference, otherwise it executes a random move.

For level three, random legal moves are generated and executed based first ensuring that the move avoids captures from opponent pieces. Once this condition is met, then it prefers moves that capture opponent pieces and avoid checks. Note that avoiding a check is preferred above capturing opponent pieces if both conditions are not met together. If there is no choice and the computer must make a move that causes its piece to get captured, it chooses a random legal move to execute.

Finally, for level four, we decided to enhance the level four computer by adding points to each of the pieces (i.e: queen = 9 points, rook = 5 points, bishop = 3 points, knight = 3 points and pawn = 1 point). When choosing between which piece to capture, the computer prefers to capture the piece that is of higher value. If all the possible capturing moves have the same points value, then the computer chooses a random legal move that captures a piece at random. The remaining conditions and possibilities are implemented as described in level three.

Board

The board directory follows the observer design pattern. It includes our main concrete subject (Board class), subject base class, as well as our observer base class which our two observers (graphic and text) inherit from. Our main Board class hosts the gameboard where the chess game is actually played on. We decided to make our gameboard a 2D vector of pointers to pieces as we believed it would allow for the most efficient access to the game state. All logic concerning the state of the current gameboard is handled in this class. A few examples include checkRow, checkCol, and checkL methods which are then used by our Game and Pieces classes to apply various chess rules and logic. This class also hosts our essential makeMove method which is used whenever a move is made by a human, computer1, computer2, computer3, or computer4 player. All board initialization, setup, adjustments, and rendering calls are handled in this class. As for our Observers, we have graphics and text. Graphics are rendered using the XWindows library and allows for efficient rendering through two types of notify functions. The first allows for full re-rendering of the board, and the second for rendering one specific move at a time. For our XWindow we utilized the files we worked with in assignment four, and made enhancements using documentation found online. We implemented new functions such as the drawLine method which allowed us to enhance the appearance of our graphics.

Pieces

The pieces class is responsible for creating each of the possible pieces, including the king, queen, rook, knight, bishop, and pawn. We implement this by leveraging polymorphism and having multiple types of pieces under one abstraction. Hence, we have a general piece class with overridden methods for function such as validMove in each of the possible concrete piece subclasses. Moreover, the Piece class is also responsible for logic checks for many of the end cases of the current board such as check, checkmate, and stalemate as well as specific move cases such as castling and en passant. For example, to determine whether there is a possible checkmate, our program creates a copy of the board and stimulates the move to determine if the king is in check, has no more valid moves, the opponent's piece can instead capture our piece that called the check, and if the check can be blocked by another piece that can get in the way of the capture. Hence, the Pieces subclass is responsible for handling the main logic of the chess game.

Updated UML

When coming up with the design for our DD1 submission, we discussed the implementation as a group to determine the ideal design using course concepts that would adhere to the assignment requirements while taking into account software engineering strategies such as low coupling and high cohesion. However, once we began coding up our design, we realized that there were methods and logic needed to keep track of our information for each piece that we had not previously considered.

In our Game class, we wanted to give the client (our main function) as little access to the implementation as possible. As a result, we used the game class as the controller for initiating the game and moves for both human and computer players. In the modified Game class, it now takes care of the total score, keeps track of the players currently in the game, whose turn it is, whether there is a current game in progress as well as considering if there is a personalized set up done by the user to play the game. The startGame method initiates the game and creates the players. It then handles the moves based on whether it is a computer or a human player using the humanMove and computerMove methods. The setupBoard calls the boardSetup function in the Board class for a personalized board set up. The endGame method ends the current game in progress and resets all the players and game board for the next game. The showPoints method displays the final score when the program is terminated upon using Ctrl-D. Other methods such as convert and create are used to create a position struct from incoming move. This position struct indicates to the program where to make the moves based on user input if it is a legal move. The create method is used to create players when the startGame method is called at the beginning of each game.

The abstract Player class remains mostly the same with a few methods removed as they are now handled in the controller Game class. For each player, whether it be human, level one computer, level two computer, level three computer or level four computer, they have a getName method which returns the name of the player (human, computer1, computer2, computer3 or computer4). The method kingIsThere returns whether the king exists on the game board for that corresponding player. The hasMoved method returns true if the player has moved and false otherwise. After the player has made a move, it is indicated using the setMoved method. The getInStalemate returns whether the game is in stalemate or not. Finally the playerMove method makes the corresponding move according to the type of player contingent that the move is valid. For the four computer level classes, additional methods are added to help with the generation of random legal moves. For level one to four, a computerPawnPromo method is added to handle the pawn promotion case for computers. For levels two and three, a moveCanCapture method is added for the case that the computer prefers a capturing legal move. For levels three and four, moveCheckOpp, moveAvoidsCapture and makeRandomMove are added to handle the cases if the move can check the opponent's king, avoid capture by opponent and if all else is not possible within the current game board setup, it just generates a random legal move, described respective of the order of the three methods listed.

In our Board class we had initially set up our main game board as a 2D array of "Square" objects. We later realized that this approach would lead to much unnecessary repetition. For example, to access a Square object we would need the coordinates (rank and file), however we had also stored those rank and file integers as fields in the Square object (redundant). Looking back we realize that the adjustment to use a 2D vector was an extremely good decision as it made our code more efficient, less redundant, and overall improved our design. Furthermore, In our original UML we thought we could omit having an abstract subject base class and a separate concrete subject (Board class). Although this design could have still been implemented we decided that we preferred to separate our connections to our observers from

our gameboard logic. Overall we are happy with this change as it made it easier to work and modify our observers and their methods.

Finally, we originally discussed that the Pieces class would be in charge of making the moves on the board. However, after discussing the responsibilities of each of the components in the MVC model, we chose to give this responsibility to the Board class in order to maximize cohesion and minimize coupling. In this case, the pieces class would only be responsible for determining whether a move is allowed with the type of move indicated and the board class would make the move and update the text and graphics observers. Additionally, we chose to override only specific methods such as validMove for each of the pieces instead of the initial decision to override all these methods. Lastly, we also found that adding the Id field for each piece pointer in the Pieces class helped us easily determine the type of Piece when implementing the logic.

Design

The main aspect of our design is the MVC (model-view-controller) implementation. The Game class is the main controller of the game which handles initiating the game and making the moves during an on going game. In order for the Game controller to execute the players' moves, game logic is within the different Pieces classes while the player specific move implementations are within their respective player classes. The views are handled by the board which renders the game board upon initiation of the game as well as during the personalized game board set up phase. This rendering occurs through the text and graphical observers which render the text and graphics move respectively. Both the text and graphics observers render each move on the board while showing the previous board rather than re-rendering the entire game board for each move.

As mentioned in previous sections, another major design aspect is our use of the Observer Design Pattern. By separating our classes in this way, we were able to create a one-to-many dependency between our objects so that all the observers are notified when the state of our board changes. We also implemented the Single Responsibility Principle (SRP) as the class should do exactly one thing! For our project, the communication and game state/logic are handled completely separately. This is seen as we separate the communication responsibility to another class and then communicate via simple function calls, parameter passing and results. The classes contained in the observer design all support high cohesion as everything in the respective classes are very closely related and serve a single purpose.

We also leveraged polymorphism in our design through the Pieces base class and the concrete subclasses for each of the possible pieces. We then override virtual methods such as validMove and moveGenerator in each of these derived classes based on the rules for the respective piece. As discussed in the next section, this allows for a design that is resilient to any changes made in the Piece class or to any type of Pieces. To continue, many helper functions were used to clean up the complicated chess code logic and make it more readable for another person. For example, we made a validMove helper method as well as a validMoveFinal method which calls this helper to determine all the possible conditions where a move is valid. We also make use of private and protected fields such that they are only accessible for the subclasses and implement accessors/getters and mutators/setters to deal with these fields. Most importantly, we considered the software design practice of minimizing coupling and maximizing cohesion as all the elements of the Piece class cooperate to perform exactly one task and has minimal communication with other modules as it only communicates via function calls with basic parameters returns.

Resilience to Change

We attempted to create a design that supports the possibility of various changes to the program specification. At a high level, this was done through the use of abstraction, various helper functions, and the use of design patterns. In the main and Game class, due to the sequence of commands interpreted by the program, more input commands can be accounted for such as a “reset” command, or “pass turn” or even an “undo” move command. In addition, since the player is an abstract class, more different types of players can be added while causing minimal change to the implementation in the main and Game class. In addition, since the computer levels are implemented with minimal reference to the specific two players, adding more players to the game such as implementing a four player chess game for the computers would be possible. This improvement is possible if the computer levels are implemented using good software design practices such that it contains very minimal hard code. Since the levels use the move generator, it is easy to implement specific changes within the players and new special legal moves in addition to those such as castling, pawn promotion and en passant. For example, we can implement a new move such that if there are only two kings left on the board, they are allowed to add one more pawn to the game at a specific square not on the top and bottom rows of the board. In this manner, if the pawn manages to reach the end of the board, they are eligible for a pawn promotion.

For the Board class, we designed it using a 2D vector of piece pointers. This implementation allows for an easy modification of the board size if we wanted to potentially change from the classic 8x8 board. For example further variations of chess include a 4 player board. This change can be easily modified and reflected in our board as a result of the abstraction and high level of cohesion used. Also as reflected in our new UML, the board class is a derived class of our base subject. This allows us to add more concrete subjects if we choose to do so later (i.e have different board setups in a single run - the client can start with an 8x8 board and choose different setups as they would like).

Furthermore, we make sure not to print anything to the output stream in the Board class. Hence, the SRP model as described in the previous section ensures that the Board class is easy to modify if we want to accommodate other input/output or add/remove features. Overall, this implementation allows us to have total freedom to change our method of interaction with the user while keeping the internal chess logic the exact same.

Finally, a major advantage of our design for the pieces is the ability to easily add another type of piece with unique rules to the gameplay. If this was required, we would easily be able to modify the abstract piece class and override each of the virtual methods for the new type of piece while maintaining the same game logic. For example, although we were unable to implement this feature due to time constraints, we did additional research on variants to the classical chess game such as Shogi, the japanese version of chess, where multiple pieces can get promoted. For example, when the bishop is promoted in this version, it becomes a Dragon Horse, a piece that has the combined movements of the king and bishop. Hence, to accommodate for this potential addition, we would just add another derived class called dragonHorse which would override virtual functions such as validMoveFinal and moveGenerator with the specifications for that piece. Then, when methods such as inCheck or the method for checkmate are called during a game, this addition is easily accounted for due to polymorphism as we can just add a Piece pointer to this derived class which will provide us with the specified generated moves.

Answers to Questions

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.**

To implement a book of standard opening move sequences for our chess program, we would refer to the *FCO Fundamental Chess Openings* by Paul Van Der Sterren and insert popular opening move sequences into a map from the STL library. For example, we would have a map<string, string> key, value type that would store potential moves for each of the opening move sequences and potential responses to opponent moves. For example, an important strategy is to occupy as large a portion of the center as possible. Hence, the map would have various opening moves designed for this strategy. We would then have an assesMove function to evaluate opponent moves and call the appropriate key-value pair to respond to the move made by the opponent.

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

In order to implement an undo feature in our game of chess, we would have a list of all player moves stored within a vector. This vector would be created at the beginning of a game and, each time a player completes a valid move, a move object with the respective information about the player who made the move, the piece that was altered, whether the piece was captured or not, the new coordinates of the piece after the move, and the coordinates of the piece before the move would be pushed to the back of the vector. In order to undo the previous move when the player chooses to do so, we would pop back the last move off of the vector. Then, the information described above would be provided to the reverseMove function, which is responsible for undoing the move or potentially calling the uncapturePiece move for the case where the piece was captured before the undo. This solution demonstrates high cohesion and low coupling with each of the reverseMove and uncapturePiece functions completing one task and limited dependencies for the undo process. Finally, if we wanted to allow a player to undo an unlimited number of moves, we would have a for loop within our undo function that would apply the above process of undoing the move by popping off the respective number of moves from the vector one at a time.

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.**

To change the implementation of the chess game to implement the four-handed chess variant, we would need to modify the board, the number of players, as well as the rules of the original game. Since we designed our game to easily plan for change using the MVC model, this task would just require small modifications to the existing model. In order to modify the board so that it changes from an 8x8 board to be extended by three rows of 8 squares on each side, we would just supply these changes in the width and height to our board class. This would allow our observers to get notified at the beginning of the game and change this layout.

Furthermore, the design of our Player class allows us to easily add or remove players in each game. To change the game from 2 players to 4 players, we would simply create 4 instances of our Player class instead of the classical 2-player instances with the additional color enumerations of red, blue, yellow, and green. Finally, since we implemented all the logic of our game within modules of our Model, we would be able to easily add new rules or change existing rules with directives. This will allow us to toggle features that are different between the two variants of the game on and off to save compilation time.

Extra Credit Features

One of our main extra features that we implemented was a computer level four. In level four, the computer generates and executes a move based on three main conditions. First the computer ensures that the move that it wants to make avoids any capture possibility from the opponent in the opponent's next move. If this condition is met, then the computer checks to see if the move that it wants to make can check the opponent's king and capture one of their pieces. If both conditions cannot be satisfied together, then the computer chooses one of the conditions that is satisfied when checking the opponent's king is given first preference. When capturing the opponent's piece, in the case that there is more than one legal move that the computer can execute to capture a piece, it chooses the one that has a higher piece value. This adds more sophistication to our level three implementation. The corresponding piece values are (i.e: queen = 9 points, rook = 5 points, bishop = 3 points, knight = 3 points and pawn = 1 point). Note that if the computer cannot avoid capture by an opponent piece, it attempts to capture their piece based on piece value or put the opponent king in check. If such a legal move is not possible, then the computer chooses a random legal move to execute.

In our graphical display we implemented an extra notify function called GCMove which handles the re-rendering of a single move in contrast to our initial implementation which would rerender the whole graphics window every time a new move was made. This feature allows for faster and more efficient graphical rendering.

We were very excited to get actual images of our pieces on the chess board. However after spending a considerable amount of time conducting research and going through X11 documentation we decided to settle for more elemental graphics functions such as the XDrawLine function. We were able to create more depth i nour graphical display using thick and thin lines along with other strings and rectangles. This allowed for a cleaner and more beautiful display that we were very satisfied with!

Final Questions

1. **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?**

Software development is a thorough process which requires software developers to plan, write, execute and test code rigorously. This project was the first step for the three of us to explore the world of software development. After our first coops, we all had a very basic understanding of software development and its processes. However, developing and deploying this project from scratch allowed us to use the software development life cycle to plan, create, test and deploy our chess implementation. We recognized that software development is an iterative process that consists of several changes, phases for building and running software applications. Since we were working on the project as a group, we used

GitHub to share our code. Only one out of the three of us had previous experience using GitHub. It was a challenging start and nerve wracking experience to see failed pushes and divergent branches. However, in the end it proved to be an extremely valuable experience. GitHub allowed us to learn Git, a tool used for coordinating tasks amongst collaborators when developing source code. This widely used essential skill will follow us throughout our journey in the software development industry. Finally, this project alo allowed us to understand the importance of modularization and its best practices. By implementing many of the design patterns learned in lectures and on previous assignments, we were able to create a strong basis for our program that is resilient to change.

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

After implementing chess, we recognized several areas that we wished we approached differently. If given the chance to start this project over, we would allocate extra time to implement the NVI and plmpl idiom. The NVI would allow subclasses such as the concrete classes for the Piece class to only change what we allow them to instead of having access to customize the entire behavior. This would be done by making sure that all private methods are nonvirtual and that all virtual methods are private or protected. Furthermore, implementing the pImpl idiom would no longer require the client to recompile if we change our private fields in one of the modules. In addition, to ensure better memory management and avoid memory leaks, it would be ideal to implement smart pointers from when we first started coding. Instead, we attempted to complete the entire project first and did not have time to refactor the changes. However, utilizing smart pointers from the start may have also helped to solve many of the memory leaks that took a long time to find and resolve.

In the case that we want to create a four player chess game, we could have avoided hard coding the two players' turns as well as specific moves related to the players such as the two types of castling and pawn promotion. Furthermore, although we created a move generator for our computer levels, by creating a move generator earlier in our implementation, we could have coded the logic for each of the pieces and the special moves within the move generator instead of handling it separately. In this manner, we could just call the move generator to see if a move generated by a human or a computer is valid instead of calling multiple methods from the different piece classes. Deciding on this implementation from the beginning would have also allowed us to improve the efficiency instead of iterating through all the possible locations on the board to generate the possible moves. Finally, we could have implemented better exception handling when communicating inside and in between our modules.

Conclusion

To summarize, we learned quite a lot about the software engineering development process through the implementation of our chess game. We were able to directly apply course contents while exploring various design features to create a fully functional game which we are extremely proud of. Working in a team setting was very intimidating at first, however it turned out to be a great learning experience that not only helped us develop our technical and soft skills but also allowed us to build new friendships that would not have been made otherwise! Although this project was extremely time consuming, stressful, and demanding we can confidently say that the outcome was extremely satisfying. Overall this project and course as a whole was a great experience as an introduction to object oriented software development.