Final Design Plan - Chess Game Project

By Ruben Rekhi and Tharsihan Ariyanayagam

Table of Contents

Introduction

The design of the project was not altered much from the initial prototype UML created a few weeks prior. Our original plan of attack encapsulated many of the aspects we would need for the project and covered almost everything we needed to implement the game. We made some slight changes to the UML during the implementation of the project, such as changing the availability of certain methods and fields, or creating getter/setters, due to realizing the need for them during the coding process. The most notable change we made was giving the Player class a reference to the board and the pieces array, as these components were required to make moves, as well as check for validity and legality.

Design Overview

Our program is a demonstration to the principles of object-oriented programming, tailored to create an interactive, board-based game. The architecture is designed to encapsulate the functionality into discrete classes, promoting modularity, reusability, and scalability.

Game Class

At the heart of our program, is the game class, which instantiates objects that represent each game that runs throughout the scope of the program. The game class holds ownership of all other classes in our program, representing the lifetime of an actual chess game, since the board and all the pieces within the game exist within the life of the game (while the game lasts). Other than the Scoreboard (which exists outside the scope of the Game class, since it holds data on *multiple* instances of the class), the main function of the program interacts exclusively with the game class to create games, setup games with custom boards, as well as execute moves and resign games. The game class holds a vector of all the dynamically allocated pieces currently active within the game, information on the state of the game, current turn, dynamically allocated displays, and a dynamically allocated board. Therefore, the main function does not need to hold any information on the state of the game, who's turn it is, etc., and simply interacts with the game class to retrieve this information when necessary. The Game::pieces vector was implemented as a pointer to ChessPiece, which is an abstract class that has concrete implementations for each type of piece, effectively using polymorphism to simplify our program and make it more effective. Similarly, the game class contains pointers to Player classes, which are also abstract, and have concrete implementations for human and computer players. The GameState enumeration, stored as part of the Game as Game::state, is accessed by the main function at the end of every move to determine if the game has ended (a checkmate or a tie), in which case it will destroy the game and continue the program, or if the game is under progress, in which case it continues.

Board and Square Classes

We collectively decided to create our Board class as a vector of Square classes, with the board having an accessor to a square at a certain coordinate. This allowed us to implement cohesive and coupled design throughout our game, as this makes our board abstract and insensitive to change (discussed further in Resilience to Change). Additionally, this allowed us to implement our design choice for each piece to hold a vector of the squares that it can travel to physically with ease and without complicated logic like making the board as a vector of ChessPiece* objects that may contain empty ChessPieces (our initial thought before DD1). This way, we can easily traverse through the board square-by-square wherever needed in the program (ex. to check for checkmate). Each square contains a pointer to a ChessPiece object which points to nullptr if it is empty. We also provided methods Square::getPiece(), Square::getColour(), Square::setPiece(), and Square::isEmpty() which allowed us to utilize the information stored within the squares without the need to determine it again in different parts of the program.

ChessPiece Class(es)

Our ChessPiece class is an abstract class that represents each instance of a piece within a game. The ownership of the class is held by Game, and instances are only created during the start/setup of the game or pawn promotion, and removed when the game is destroyed or a piece is captured (and effectively expelled from the game). The most important component of this class is the ChessPiece::validMoves vector, which as discussed earlier holds a vector of all the squares the piece can travel to. This vector is updated at every state change by the ChessPiece::updateValidMoves() method, which is a pure virtual method implemented within each concrete class to highlight the different movement patterns of each piece. The creation and utilization of this design decision was by far one of the most important components and decisions of our project, and allowed us to design all of our classes in a cohesive manner with low coupling. This will be discussed in detail in the next section.

Player Class(es)

The Player abstract class represents an instance of a player, and is responsible for ensuring the player is moving their own piece, the move does not put them into check and is a valid move, and also provides public methods that let the game determine if that specific player is under checkmate or stalemate. Our checkmate and stalemate logic was almost identical: we check if the player under question has any moves for all its pieces that are valid (the piece can physically move there) and legal (the player will not be under check after the move is executed). The difference is whether the player is under check before that, if yes, then it is a checkmate, if not, it is a stalemate (no check, but no legal moves available). To check if a move is legal, we have the Player::isLegalMove() function which accepts a starting square and ending square, executes the move, and uses Player::isCheck() to determine if the player is under check after the move is executed. The move is then reversed, and the board is set back to its original state.

If the move was legal, the function returns true, and false otherwise. Both functions stop at the first instance of a legal move being found and return false. If the loop executes to the end, there are no moves available to the player and they return true.

Conclusion

In summary, our chess program exemplifies the effective application of object-oriented programming principles, featuring a modular and scalable architecture. Key components like the Game, Board, Square, ChessPiece, and Player classes interact seamlessly to manage the game's mechanics, state, and player actions. This design not only elegantly addresses the complexity of chess but also lays a strong foundation for future enhancements and adaptations, demonstrating the versatility and robustness of object-oriented design.

Design Choices/Challenges

Throughout the project, we faced many design challenges that needed to be tackled, the most obvious being checking to see if a king is under check, determining how we would check if moves are valid, and checking the state of the game for checkmate and stalemate. We were able to solve all these challenges with the design decision of introducing the observer pattern into our program, and making ChessPiece objects both subjects and observers

ChessPiece as Subjects

Whenever a move is made on the game, or a chess piece is initialized (in setup mode or in a normal game), the pieces being changed are considered subjects and the player that owns the piece calls the Player::notifyPieces() function that goes through the Player::pieces vector and calls notify on every piece. This way, every time the board state changes, each piece is made aware of it and can get notified and execute its own behavior.

ChessPiece as Observers

When a piece is notified, it calls ChessPiece:updateValidMoves() for that specific concrete class and uses the movement pattern encapsulated within the class to determine all of the possible squares it can move to. These squares are either empty, or contain an enemy piece, allowing for capture. However, we do not check for whether this will put the king into check to allow for this vector to be used for other functionality, like checking for check, since it doesn't matter if taking the other player's king puts you into check since the game would have ended at that point. Along with the vector, the piece also has a public method ChessPiece::isValidMove() which accepts a square and returns true if that square is part of the piece's valid moves.

Functionality Allowed through Design Decisions

This design choice allowed us to handle a lot of the challenges with ease. To check if a player is under check, we simply iterate through the Player::pieces vector (which is a reference vector to the one in Game) and check if any opponent pieces have the player's KingLocation Square as part of their valid moves. When checking for the validity of a move, we simply call ChessPiece::isValidMove(). When checking for the legality of a move, we execute the move and call isCheck() again, which uses ChessPiece::isValidMove() to determine if the potential move will put the king under check. Additionally, as discussed before, for checkmates and stalemates a combination of these methods is used to determine whether any valid and legal moves exist for the player at that given moment.

Simplification of Computer Algorithms

Nevertheless, this also allowed us to program and design the various computer algorithms in a much simpler manner than we expected at the start of the project. For the level 1 algorithm, we iterate through the computer pieces and attempt to find the first valid move that can be executed. For the level two algorithm, we iterate through the pieces again and try to find a valid move to a square that is not empty (contains enemy square) and attempt to execute that move. If that is not possible, we try to find a valid move that puts the opponent king under check, and if both preferences are not possible, we simply revert back to algorithm one. For the level 3 algorithm, we determine which pieces are under danger by checking the opponents pieces' valid moves, then try to save that piece. If the piece can save itself while capturing a piece, that is preferred. If the piece cannot, then it moves to a safe spot. And if it is not possible to save the piece, we try to sacrifice it to get some value, and if that doesn't work we revert back to algorithm 2 to find a move that checks the opponent or takes a piece. While we did not have time to complete the implementation of the fourth algorithm, our plan was to use the design again to iterate through the valid and legal moves and assign a value to each move based on the points that it will result in (negative points for losing pieces and positive points for taking pieces, based on the point value of each piece). We would then select the move with the highest point value and execute it.

Observer Pattern for Displays

The displays both also utilized the observer pattern, and were notified whenever appropriate. The notify function accepts a vector of updated Squares, which is for the displays, so that they can effectively utilize the information in those squares to only update the squares that have changed in appearance as to not redraw the whole board at every change of state.

Conclusion

In conclusion, the integration of the observer pattern into our chess game design provided an elegant solution to various complex challenges. By treating ChessPiece objects as both subjects and observers, we achieved efficient and dynamic game state management. This approach streamlined the

processes of validating moves, detecting checks and checkmates, and simplified the development of computer algorithms for different levels of gameplay. The observer pattern also enhanced the functionality of our displays, allowing for selective updates and reducing the need for full board redraws. This design choice not only addressed our initial challenges effectively but also opened avenues for future enhancements and sophisticated game strategies.

Resilience to Change

Our design is deeply rooted in object-oriented programming and modularity, which provides us a strong framework for accommodating changes or introducing new features to the game.

Changes in Piece Logic

For instance, if there was a new program specification illustrating that there are new pieces in the game, such as an “Elephant” with unique movements, our design allows us to seamlessly integrate this new piece. All we would have to do is simply create a new class named “Elephant” that inherits the ChessPiece base class. It would contain all the properties a chess piece requires. The only other thing we would have to do is implement the pure virtual ChessPiece::updateValidMoves() method with the specific movement logic of the “Elephant” piece. The unique movement of the “Elephant” is encapsulated within its class, and as discussed before, since the class holds the vector of all squares it can possibly move to at any given moment, nothing else in our program has to be changed due to the low coupling and movement design we have implemented (other than that calling the “Elephant” constructor when the board is created or a pawn is promoted). This example demonstrates how **any** changes to our program relating to the movement of pieces can be simply implemented by altering one function (ChessPiece::updateValidMoves()).

Changes to the Board Design/Pattern

Additionally, since our board class is a separate part of the program within the Game class, which is built with many instances of square classes, any changes to the design, shape, or creation of the board can simply be handled by altering the constructor of the Board to create squares in any pattern and update the Board::getSquare() method to correctly allow access to specific squares (and slightly tweaking the displays to incorporate the new board). For example, for new game modes, such as a "Fog of War" variant where players can see only certain parts of the board, our design's separation of game logic and display logic becomes particularly advantageous. We would update the game state management to track visible squares and modify the display logic to render only these squares, leaving the core game mechanics untouched. We would simply add a field to track the visibility of a square in the Square class and create an accessor to be used by the displays.

Changes to the Overall Logic of the Game

Nevertheless, if there are any logic changes to the overall rules of the game itself, for example, each player getting 2 turns in a row, these changes can easily be incorporated in the private methods and data fields of the Game class since the player board and pieces are separated and do not rely on the specifics of how the game works.

Changes to Legality of Moves or Player State

Any changes to the rules that affect the state of a player, for example introducing a different type of checkmate or additional constraints on how a player can move their pieces (i.e. the queen *has* to be taken out of danger if possible) can easily be implemented due to our abstraction and separation of our player class. Such changes to the rules can be implemented by altering the Player::isLegalMove(), Player::executeMove() and Player::move() methods of the abstract class or add public methods to check for new states of the player (something similar to a checkmate). Additionally, since our Player class has two concrete classes for the Humanplayer and Computer player types, which are responsible for determining an actual move (square1 to square2) before using the Player abstract class's methods to validate the move, any changes on how we want the computer algorithms to generate moves or for the HumanPlayer to accept input on what move to make, can be implemented in the respective files without altering the rest of the program.

Conclusion

In conclusion, our chess program's design exemplifies resilience through its strong foundation in object-oriented programming and modularity. This approach enables effortless integration of new elements, be it pieces, board designs, or rule changes, ensuring adaptability and ease of maintenance. Our architecture not only supports the current game mechanics but is also primed for future enhancements, making the program versatile and future-proof in the evolving landscape of game development.

Q/A

Question: How would you implement a book of standard openings if required?

To integrate a book of standard opening moves into our chess program, we would use text files and vectors. The text file would store opening sequences in a structured format, with each line representing an opening. This line would include the sequence of moves, followed by historical statistics like win/draw/loss percentages. In our program, we'd create a class named OpeningBook. This class would read the text file at the program's start and parse each line to store the opening sequences and statistics in vectors. For instance, one vector could store the sequence of moves as strings, and another could hold the corresponding statistics. The Computer class, a subclass of Player, would be adjusted to use the OpeningBook. At the start of a game or when an opening move is applicable, the Computer class

would iterate through the vector of opening moves to find a match with the current board state. If a match is found, the computer player would choose its next move based on this 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?

To implement an undo feature, we would use a stack data structure in the Game class to store the history of moves. Each move would be represented as an object storing the initial and final positions of the piece, the piece that moved, and any piece that was captured. For a single undo, when a player chooses to undo, the program would pop the last move from the stack and revert the board to the previous state. This would involve moving the pieces back to their original squares and resurrecting any captured piece. For unlimited undos, the stack would keep a record of all moves made during the game. The program would allow popping moves from the stack until it's empty, enabling the player to revert the game back to any previous state.

Question: Outline the changes that would be necessary to make your program into a four-handed chess game.

To modify our program for a four-handed chess game, a few small changes would be necessary. First, the ChessBoard class would need an overhaul to support a larger board suitable for four players, possibly in a cross-shaped configuration. The Game class would be updated to manage four players instead of two and each player would need their set of pieces, requiring adjustments to the Game constructor to initialize these pieces. The additional players would also include changes in our turn management. The Scoreboard class would be changed a little to accommodate the extra players on the scoreboard. Lastly, the user interface, whether textual or graphical, would need to be updated to display the larger, more complex board.

Final Questions

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

Reflecting on our project, we gained several invaluable insights into the dynamics of team-based software development. One crucial realization was the need for proficiency in using different branches in Git. Our initial approach of committing, pushing, and pulling from the same branch led to significant setbacks. We lost a considerable amount of our work, which became evident when we had to restart debugging just a few days before the project deadline. This experience underscored the importance of using separate branches for different features or sections of the project. It would have allowed us to work independently on various aspects of the chess program without the risk of overwriting each other's code,

thereby avoiding the loss of valuable progress. On the other hand, our project was marked by clear communication and well-defined roles. This structured approach facilitated seamless coordination and planning. It allowed us to efficiently divide the workload, align our development goals, and collaborate effectively, ensuring that we were always on the same page regarding the project's progress and objectives. Moreover, proper documentation was a cornerstone of our project. We maintained detailed documentation throughout, which was invaluable in various stages of development. It enabled easy understanding of each other's code, streamlined the integration of our separate contributions, and made the debugging process more efficient.

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

If we had the chance to start our chess project over, we would focus on two main improvements. First, we'd implement regular testing after each class and method creation. This approach would have allowed us to catch and fix issues early on, reducing the extensive debugging we encountered later in the project. Incremental testing would have made our development process more efficient and manageable. Secondly, we'd reconsider our implementation of the castling feature. Initially, we divided the logic into checking the possibility of castling and executing the castling move in separate functions. A more efficient approach would have been to consolidate all castling-related logic into one function. This would have simplified our code, making it more straightforward and less prone to errors, thus streamlining the development and debugging process. These changes, though seemingly small, would have significantly impacted the overall quality and manageability of our project.

Conclusion

In conclusion, this comprehensive report details our journey in developing a chess game, emphasizing the key aspects of our approach. In the Design Overview, we established a robust foundation with object-oriented principles, ensuring modularity and scalability. The Design Choices/Challenges section reflects our strategic responses to complex problems, utilizing the observer pattern for dynamic game state management. Our Resilience to Change section highlights the adaptability of our design, showcasing how new features, such as additional chess pieces or board configurations, can be seamlessly integrated. Lastly, the Final Questions segment reveals our learnings about team dynamics in software development, the importance of regular testing, and how we overcame challenges like castling implementation. This project was not just about building a chess game; it was a journey of growth, innovation, and collaboration, preparing us for future challenges in software development.