CS246 Chess Design Document

Name:	User ID:	Student Number:
Adam Muinuddin	azmuinud	21015917
Jiro Kakapovbia	jjkakpov	21019684
Liam Knox	lknox	21066106

Table of Contents

CS246 Chess Design Document **1**
Table of Contents **1**
Introduction **2**
Overview **2**
Design **3**
Answers to Questions **7**
Extra Credit Features **8**
Final Questions **9**

Introduction

This document outlines the design process for our chess project, which was developed in alignment with CS246 principles. While working on the project we strived to achieve a fully functional chess game with both text and graphical interfaces, that support human vs human, human vs computer, and computer vs computer player modes, with different difficulty levels for the computers. The chess game we created follows the standard chess rules.

Overview

In this section, we provide an overview of the overall structure of our project. Our implementation of chess consists of several key classes that integrate to deliver distinct purposes that follow the Model-View-Controller (MVC) architecture pattern.

The model component of the MVC architecture pattern comprises the classes Game, Board, Tile, History, and all of the Tile subclasses (King, Queen, Rook, Bishop, Knight, and Pawn). These classes are responsible for the core functionality of the chess game. They work together to encapsulate the game and directly manage the data and logic of chess, ensuring all rules are enforced whilst providing methods to manipulate the game state.

The view component of the MVC architecture pattern comprises the classes Studio, Subject, Observer, Text, Graphics, and Xwindow. These classes are responsible for displaying the data to the user and presenting the user interface for interaction with the game. They work together to provide textual and visual representations of the game state, ensuring that any model changes update the views accordingly to reflect this new state to the user.

The controller component of the MVC architecture pattern comprises the classes Player, Human, Computer, and the Computer subclasses (Level 1, Level 2, Level 3). These classes are responsible for handling the user inputs, allowing the user to interact with the model. They work together to manage interactions between the user and our chess game, where the controller receives input from the user, updates the model, and ensures that all user input results in changes to the game state that are in alignment with the rules of the model.

By following the MVC architecture pattern, our group has ensured that our chess program is scalable and modifiable to any changes that may need to be made.

Design

The final UML diagram maintained the same MVC structure as the original UML diagram, however, most of the classes had their fields and methods modified or updated. This portion of the design document will outline a bottom-up analysis of how each class and subclass works together to provide functionality to each component of the MVC structure, focusing on OOP principles such as low coupling, high cohesion, modularity, and scalability.

The Model component faced the most modifications and updates to its fields and methods. The abstract Tile class and its subclasses facilitate the validation of moves depending on what character is being stored at a given Tile. For each piece, we have established functions: possibleMoves(), possibleCaptures(), and possibleChecks() which return a vector of pairs of ints (vector<pair<int, int>>) to represent the end coordinates that a Tile can move to. This, of course, depends on the specific piece at a given Tile. possibleMoves() will determine all possible end positions that a piece at the given tile can make that are valid. Similarly, possibleCaptures() will determine all possible end positions that will capture an opponent's piece, and possibleChecks() utilizes both of the previous functions to determine all possible end positions that will put the opponent in check. This design promotes high cohesion by encapsulating logic specifics to each piece within its subclass. To implement possibleMoves() functionality, we used the following logic: since we utilized a vector for start and end coordinates, it makes sense to have a direction vector that stores the directions that a piece can move. We then iterate through each direction and utilize the function from Tile called simulateMove(). If we were not in check when we simulated the move, then we would push back the move. If we encountered a piece that blocked the path, we would determine if it was the opponent’s piece. If it were deduced to be the opponent’s piece, we would push back the move and iterate to the next available direction. If the piece was the user’s, we would iterate to the next available direction. The function simulateMove() was vital in the implementation of possibleMoves(), possibleCaptures(), and possibleChecks() implementation. It allowed for simulating the state of the board by taking a start and end position, a board, and a tile pointer as parameters. After simulating the move, simulateMove() would return true or false depending on if the user’s king was in check. To handle special moves like castling and en passant without increasing coupling, additional boolean fields were added to the relevant subclasses (Pawn, King, and Rook) to “track” move

history, eliminating the need for these classes to access to the History class to check if castling or en passant is a possible move.

The Board class is responsible for managing the data of the tiles, making valid moves, and deciding the winner of a game. The board has fields that manage the active and captured white and black pieces, which makes it easy to undo/ redo moves, and would allow for easy adaptation to point tracking in the future. The current turn field and its accessor function that tracked the current turn was moved from the class Game to Board to lower coupling and increase cohesion, because the Board and Tile classes require access to the current turn in order to set the colour of different pieces, and determine which moves are valid based entirely upon whose turn it is, whereas Game rarely uses it. Within the Board class, an update to the function inCheck was made, originally it was not passed anything, whereas now a board reference (Board &) is passed. This design change was made as the inCheck function was primarily used to determine if temporary boards were in check, when we use simulateMove(). Additionally, the makeMove function in Board is overloaded, depending on if pawn promotion occurs. This design choice was made to simplify the makeMove functionality as pawn promotion is a rare event, overloading simplifies the core functionality by isolating this special case into a separate method.

The History class is responsible for tracking all previous moves for undo functionality, which is an add on, this could possibly be scaled to also incorporate a redo function, by adding another stack similar to moveHistory. This design ensures modularity and scalability, allowing for enhancements to be made without disrupting existing functionality.

The Game class is composed of all of the above classes. It is responsible for managing the overall state and flow of the game. It acts as a central hub ensuring that user inputs from the controller are processed correctly, the view classes reflect changes to the game in real time, games are facilitated with correct adherence to the rules, the win loss record is accurate.

For castling, the bools moved and justMoved were implemented in the king and rook classes. The bools are initially false, but moved becomes true if the piece has been moved at any point in the game, and justMoved becomes true if . This is important when castling, as both the rook and king should not have moved at any point during the game. For en passant, bools moved, justMoved, and justMoved2 were implemented in the pawn class. The bools are initially false, and work identically to how we described above, justMoved2 becomes true if it was the last piece that was moved, and the pawn moves two squares from its initial position. This is important when determining if en passant is available, as the pawn must have moved

2 squares to evade a capture by an opponent pawn. These design choices enhance cohesion and reduce coupling, as it keeps move specific logic within the relevant classes.

The viewer component of the MVC structure that we based our planning around, utilises the subject observer pattern, where the Studio class is the concrete subject, the Subject class is the abstract subject, the Observer class is the abstract observer, both Text and Graphics classes are concrete observers, and the Xwindow class is used by Graphics class to output the board for the user to see. We noticed that the graphical interface was slow to populate and update the display when the program was run as it updated the entire display (all 64 tiles) after each move. To increase the update speed of the graphical display, we tweaked the observer pattern, ensuring that each tile has its own observer object. This change forces the display to only update the Tiles corresponding to the move being made in the makeMove() function. Although this was not implemented, utilization of the observer pattern would easily allow for a decorator pattern to be implemented and change the look of the board and pieces, almost like a skin to change the texture to wood, metal, or frosted glass, giving the player a customizable experience if they do not like the look of the classic chess board and pieces. This pattern ensures low coupling and high cohesion, as the view components are automatically updated whenever there is a change in the model.

The controller component of the MVC structure consists of the classes Player, Human, Computer, and Computers subclasses (Level1, Level2, Level3). The Player class is an abstract class that defines a basic interface and common functionalities for all players, regardless of whether they are human or computer. This promotes low coupling and high cohesion, as both human and computer players can interact with the game. The getMove(Board &) function will accept input from the player. For a human, it will tokenize the input into shorter strings which are the arguments, and those are converted into vector coordinate pairs (pair<pair<int, int>, pair<int, int>>), and for computers, it will utilize static_cast to convert the string that the computer chooses at random from possible moves, and casts the move into a vector coordinate pair. For the different levels of computers, we designed algorithms that will play with different goals. The subclass level 1 will play moves at random, by randomly selecting an active piece for the computer, then randomly selecting a possible move for the randomly selected piece. The subclass level 2 aims to capture and check when possible. To do this we build on top of the level 1 algorithm. Instead of making an entirely random move, we randomly select a piece, search its possible captures and

possible checks simultaneously, and then select a random move from that set. If there are no possible captures or checks, we call possible moves for that piece and randomly select one. The subclass level 3 aims to avoid capture when possible. To do this we build off of level 2. The algorithm randomly selects a piece, iterates through its possible captures and possible checks for that piece, then iterates through all other pieces of the player that can be captured, randomly selects a piece in danger, and selects a random move that avoids capture for the piece in danger. If there are no pieces in danger, the same algorithm that we called in level 2 is used.

By keeping OOP principles in mind during the planning process, we have ensured that the project is modular, scalable, and resilient to change. The MVC architecture, subject observer pattern, and strategic design choices ensure that each component of the game works seamlessly to provide a game of chess that is resilient to change if need be.

Answers to Questions

We have a new answer for the question:
How would you implement a feature that would allow a player to undo their last move? What about an unlimited number of undos?

See extra credit features for answer as we implemented an add on for unlimited undos.

Extra Credit Features

In our chess project, we implemented an extra credit feature called undo. The undo feature can be used anypoint during the game, additionally as the History class uses a stack that holds data representing the previous moves played in a game. The undo feature allows you to undo moves to the very beginning of the match.

We faced a challenge of implementing an efficient and sufficient method to store the history of moves. We knew that we needed to figure out how to store the history of moves in a structure that would let us call them back in the reversed order. We decided to use three stacks, one to hold the moves, one to hold the characters, in the event of a capture, and one to hold a boolean that expresses a pawn promotion. Every valid move played on the board is added onto the stacks. This data structure was sufficient to allow us to reverse engineer any move performed on the board, and set the board to a previous state when undo is called. Although it is sufficient to undo, the reversed moves cannot always be applied by the makeMove function, so it is necessary that history can affect the board directly. This feature allows us to return to a previous board state that was an arbitrary number of moves in the past.

Final Questions

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

Adam: This project highlighted several lessons for me relating to developing software in teams. First it truly showed me how important effective communication is. We held daily meetings and this kept communication channels open between all members. I saw other groups who only met once every few days, and I would say that the difference was enormous. I also learned about the importance of git version control. Taking CS136L we had to learn about git, but I had never used it to this extent. This project showed me the importance of git, and how to use it effectively in teams. I also learned a lot about design choices. There were many instances where we had differing opinions on design choices, when this happened we would listen to one another's ideas, and then start a new brainstorm from scratch to ensure that we thought about things from a logical standpoint, working through the decisions we needed to make as a group. This promoted finding the best idea, rather than using an idea that someone just felt passionately about which I believe enhanced our project.

Jiro: The most important thing that this project taught me was that communication can go a long way, and that there are so many different ways to ensure communication within a group setting. During these past two weeks, we've found numerous ways to communicate and update each other. From our Instagram group chat to Google Meet video chats to meeting in person, I've realized that each of these differing modes of communication comes with its benefits and downsides, none significantly more beneficial than the other. As such, the best way to ensure that one's group is on the same page is to communicate as much as possible, and in as many different ways as possible. Not utilizing all avenues of communication will truly limit oneself from succeeding in a team.

Liam: It makes a very large project far more manageable. While the overall project is large, each individual class is far smaller and more manageable. It lets you focus on a more specific aspect of the program rather than spreading yourself thin trying to understand all the different aspects of it. It also ensures there are people around you all with the same goal, willing to offer help on a difficult bug or algorithm.

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

Adam: If I could start over, I would have started thinking about the movement algorithms for pieces earlier. I felt that the algorithms were easy to understand, but hard to implement in code and ensure that they were following the design principles. Additionally I would prioritise a more detailed initial planning and design so that we would have made less design changes during the assignment. Lastly I would start earlier so that we could build more additional implementations, I would have loved to have coded different textures, or redo, or computer level 4, however we did not have enough time.

Jiro: One thing I would choose to do differently would be to prioritise my sleep, mental, and physical health more. During the past couple of weeks, I've had to dedicate numerous hours to ensure I put my best foot forward for this assignment. As such, some aspects of my life suffered as a consequence (primarily my sleep). As such, I was not nearly as productive as I could've been, and I wasn't consistently in the right state of mind. One way to combat this from happening again would be to be more diligent when it comes to scheduling time to work on group projects, and to simply "lock in" when I'm tasked with working on something at a given time. Instead of procrastinating and tricking myself into believing I have more time than I do to finish my share, I would need to learn to balance my healthy sleep and mental health habits while also ensuring I can complete my work to the best of my abilities.

Liam: If I had the chance to start over, I would put more effort into completing the program earlier. We started with a very hopeful schedule, allotting little time to debugging the program, which caused us to think we were on schedule to finish with plenty of time left over. I now recognize that, especially with a new team, it was foolish to have such high hopes of quick and easy success. With the knowledge I have now, I have worked harder to produce a compiling program faster, and schedule far more time for debugging.