CS 247: Software Engineering Principles

Spring 2024

Chess Project Report

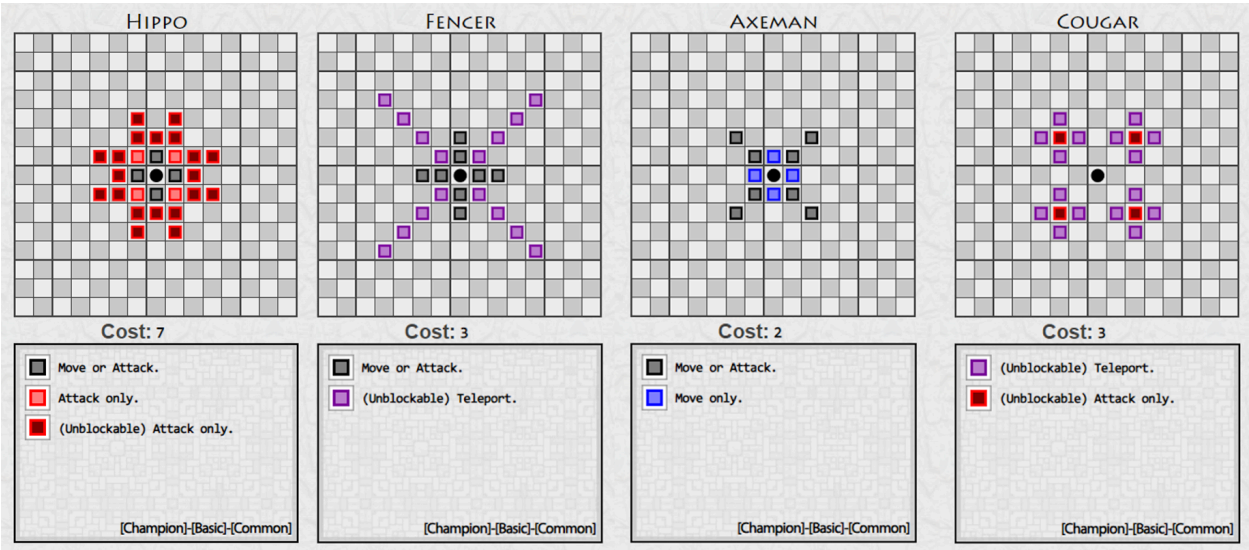
ChessEvolvedOffline

Team Member Full Name	UserName
Max Lu	m22lu
Jason Cheng	j25cheng
Howard Ou	hlou

Introduction

ChessEvolvedOffline is a chess game project developed as part of the CS 247 course on software engineering principles. This project leverages object-oriented design principles and employs effective design patterns to ensure a flexible and maintainable codebase. This document provides an in-depth look at the structure and design of ChessEvolvedOffline, detailing the high-level implementation, design patterns used, and specific techniques employed to address various design challenges.

The game not only includes all six of the standard chess pieces but also introduces twenty extra original, custom pieces with unique moves, such as teleportation.



These custom pieces include innovative designs like the Cougar, which can teleport over other pieces and has additional unblockable attack power, and the Hippo, which has great attacking abilities but very limited mobility. These additions not only enhance the gameplay by introducing new strategies and tactics but also demonstrate the flexibility and extensibility of our design. By incorporating these unique pieces, ChessEvolvedOffline aims to provide a novel twist to the traditional game of chess, making it more dynamic and challenging for players of all skill levels.

Overview

At the top level, ChessEvolvedOffline is structured around several core components that interact to create a functional chess game experience that can easily be modified or built upon. These components are designed to work together to manage game state, player interactions, and piece movements, ensuring a smooth and engaging gameplay through the command line.

Game: Represents the central controller of the entire game, managing player turns, setting up pieces on the board, checking for resignation/checkmate/stalemate, and keeping score.

Board: Represents the chessboard and is responsible for maintaining the state of the board. This includes tracking the positions of all pieces, making and undoing moves, validating board states, generating legal moves, checking threats, placing down pieces, and more.

Player: Represents the players in the game. There are two main types of players: human and computer (with four difficulty levels). Games can happen between any pair of players.

Piece: Represents the various chess pieces used in the game. It defines some basic information for each piece such as color, and whether the piece has moved before (for Pawn and castling rights) while specific details like movesets and value are implemented in their respective subclasses.

The Game class has a Board and manages the interactions between the Board and the Players. Regardless if a human or computer is the one making a move, the information is passed into the Game, which checks if the move is among the Board's generated legal moves, and then prompts the Board to make the move if it is. It will also award points to the colors based on the results of the games that have been played.

The Board is a standard 8x8 chess board that maintains a collection of Piece pointers, each representing a chess piece (or empty space) on the grid. The Board also stores a collection of unique_ptrs to pieces that the grid references, as the Board in our design owns all pieces. Furthermore, it is checking for special moves and conditions such as castling, promotion, and en passant. It is also storing a history of moves that were made to facilitate move undoing.

Finally, the state of the board is displayed in both text-form in the terminal and on a graphics window by observers after every move, ensuring that players have a real-time view of the game.

Final UML Diagram

Click the link above to view our final UML diagram

Design

Designing ChessEvolvedOffline presented several challenges that required thoughtful solutions to ensure a robust and enjoyable gameplay experience. One of the primary challenges was determining legality due to checks, as well as handling checkmate. To solve this, we implemented a method within the Board class that iterates through all possible candidate moves stored in each Piece on the board. By simulating each potential move, temporarily updating the board state, and verifying the king's safety after each move, we could accurately determine if a checkmate condition exists. If no legal moves exist and the king is in check, the game is concluded as a checkmate. Meanwhile if no legal moves exist and the king is not in check, the game is concluded as a stalemate. The Board will generate all legal moves after each makeMove or undoMove, ensuring that the legal moves are always updated at appropriate times to check for stalemate/checkmate.

Another significant challenge was implementing the undo feature, which required maintaining a history of moves and ensuring the board state could be accurately reverted. We addressed this by incorporating an undo stack of Move objects within the Board class. Each Move object in the stack includes details such as the starting and ending positions of the piece, pointers to the original and captured piece, the previous En Passant target square, whether the move was the piece's first move, and promotion piece. When an undo operation is triggered, the game retrieves the last move from the stack and reverts the board state accordingly, allowing both human players to undo their moves, and also allowing much of the Board's and various Computers' move checking logic to simulate moves.

Ensuring that all moves adhere to the rules of chess, including specific movements of different pieces and special moves like castling, en passant, and pawn promotion, was another key challenge. Each piece implements its own method for generating legal moves based on its movement rules. The Board class then validates these moves within the context of the current game state, checking for conditions such as blocked paths, checks, and special rules. This thorough validation process ensures that only legal moves are allowed, maintaining the integrity of the game.

Implementing special moves and conditions required additional rules and checks within the game logic. For example, castling, detected in the Board logic by the king moving two squares, necessitated checks to ensure the king and rook had not previously moved and that no squares between them were under attack. En passant was implemented by tracking an En Passant target square within the Board, to determine whether moves triggered En Passant. Pawn promotion allowed pawns reaching the opposite end of the board to be converted into another piece of the player's choice, typically a queen. The desired piece is then pushed onto the Board's vector of Piece unique_ptrs, and a pointer to the new piece is placed on the grid.

Developing an AI capable of playing at multiple difficulty levels was another design challenge. We created different AI strategies for the ComputerPlayer class. The Level 1 and Level 2 computers only require information stored within the Board's legal moves, while higher difficulty levels require use of the Board's makeMove and undoMove functions to simulate moves to look ahead, with the Level 4 computer maintaining an evaluation function to handle captures, hanging material, centralizing pieces, checkmate, and promotion, making it very significantly stronger than the previous levels. The Level 5 computer uses the minimax algorithm with alpha-beta pruning and the MVV-LVA heuristic, which simulates future moves and assesses board states to minimize the opponent's advantage and maximize the AI's position. The algorithm additionally uses human-written piece evaluations for each individual square of the board, for each type of piece. These are included in the codebase under src/evaluations, and are parsed by the Evaluator class, which facilitates the Level 5 AI to evaluate the strength of every type of piece on different squares. This helps the bot make more optimal moves than the randomly generated moves by many of the lower levels.

For optimization, especially for the higher level AIs, it was critical that generateLegalMoves was called as infrequently as possible. Hence, we store a stack of legal move history in the Board as well, and makeMove and undoMove were split into two additional private functions with testMove and testUndo. When generating legal moves, testMove and testUndo are the ones that are called in order to verify board state, while the public interface has access to makeMove and undoMove, which call testMove and testUndo respectively, call generateLegalMoves and maintain the legal moves stack.

To keep the user interface in sync with the game state, we employed the observer pattern. After every move, the Game class updates the Board, which in turn notifies its observers. These observers update both the text-based and graphical interfaces, ensuring players see the current state of the game in real-time.

Memory management was also a critical consideration. We represented the "owns-a" relationship between the Board and the pieces with a collection of std::unique_ptr, along with standard library

containers like std::vector and std::map, to handle memory management efficiently. This approach minimizes the risk of memory leaks and dangling pointers, ensuring robust and maintainable code.

Finally, we prioritized maintainability and extensibility in our design. Adhering to SOLID principles, we ensured that our code is modular, with low coupling and high cohesion. This makes it straightforward to add new pieces, rules, or game features without disrupting existing functionality. Comprehensive inline documentation and a consistent coding style further enhance the maintainability of the project.

Resilience to Change

Our design for ChessEvolvedOffline incorporates several design patterns that enhance its resilience to change. The Observer Pattern ensures that any changes in the game state are automatically reflected in the user interface, making it easy to add new display components or modify existing ones without altering the core game logic.

Additionally, almost all of the design of our code is designed in a way to make extension as easy as possible. For instance, Colors is an enum class and Board's color-based logic is mostly based on its getNextColor() function, allowing easy extension to adding more players to the chess game. Pieces store their move information in a vector of PartialMoves, which is a struct storing a change in X, change in Y, and a MoveType, which is another enum class. This makes creating dozens of new pieces extremely convenient, as we only need to push the appropriate squares into a new Piece's moveset. We can even easily add new custom move types, such as swap, by simply adding to the enum class and adding logic to handle the new type in the Board. A few helper functions were added in the Piece class to aid its subclasses in populating their movesets, allowing for placement of arbitrary MoveType squares on patterns representing a King, Knight, Bishop, Rook, Valkyrie, or Zebra's squares. The pushBishopMoves and pushRookMoves methods additionally support inner and outer ranges, allowing for even more easy customizability of pieces.

We abstracted out the Player class to handle both human and computer players seamlessly, with both types of players being prompted to produce a MoveInput struct, giving sufficient information to define a move, and with both types of players being provided a Board reference to help them make their decision. We have also abstracted out the Computer class, allowing for easy additions of increasingly complex AI. The ComputerPlayer class extends the Player class, and provides a useful randomization helper function for its children to use.

We defined our own ChessException class, allowing us to throw and safely handle errors gracefully anywhere in our codebase, to be caught and handled in the main Game loop. For critically unsalvageable exceptions, we also have an UnrecoverableChessException class, which serves to allow the program to exit more gracefully without a segmentation fault/memory leaks.

Finally, our code is written to minimize coupling and maximize cohesion, adhering to SOLID principles. A number of our fancy features, as well as our fancy AIs, were not fully planned while writing the majority of the codebase, yet due to the extensibility of our setup, we found adding in many extra features to be maximally painless. The AIs were able to easily manipulate the board state to look ahead, safely undo the board state, and receive important information about threats, captures, and checks, with minimal changes to the main Board code. Implementing highlighting in our custom “select” command to allow threat squares of any piece on the board to be displayed required minimal updates to our codebase, due to the Observer pattern and clear and easy computation and retrieval of all relevant legal move information

---

Project Specification Questions

---

1. How would you implement a book of standard openings?

A book of standard openings can be implemented using a collection/database of moves in PGN format. These standard openings could be collected from reputable sources such as Chess.com and they could be stored in a file to be loaded at startup. (A PGN parser could convert these moves into a tree or graph.) During the game, the computer could consult this database to play a random optimal move based on the current board state. A simple strategy would be to maintain a list of possible openings based on the moves so far, and on each move, this list could be filtered to keep only those openings whose next move matches the current one. The user interface could also display opening moves with their names.

2. How would you implement an undo feature?

An undo feature can be implemented by maintaining a stack of moves played so far. Each move pushes the current state onto the stack, and an undo operation pops the state from the stack, using it to revert to the previous position. Moves store information about a piece’s start square, end square, any possible captures, or special moves that may have occurred (en passant, etc). This approach ensures that any number of moves can be undone, allowing players to backtrack as needed. Additionally, we would

update the display and game status accordingly after each undo operation to reflect the reverted state accurately.

3. How would you modify the program to support four-handed chess?

Supporting four-handed chess would require expanding the board, possibly to 12x12, and modifying the movement logic to account for additional players and pieces. We would need to expand the array of players in the Board class to contain 4 players; the current player's index would also be the move number modulo 4 rather than 2. Each player could additionally store a direction to indicate which way their pawns move and the Color enum would need 2 more values. The legal move validation would also have to account for the new shape of the board. The command interpreter and display logic must be adjusted to manage the increased complexity and additional players' turns, using a turn order such as white, black, red, blue. The promotion logic would also be altered as pawns are promoted in the center of the board. Moreover, multiple players could be checkmated in the same move, so we cannot stop scanning the board state after finding a single checkmate condition at the end of a turn. This also requires changing the checkmate messages to handle multiple players together being checkmated, as well as not announcing a player as the winner unless all 3 other players are out of the game. Implementing new rules for interactions between the four sets of pieces and ensuring the game flow remains smooth and fair would also be necessary: for instance, much of the "other Color" logic would need to be updated accordingly (i.e. undo would now need to call a new getPreviousColor() method)

Extra Features

The biggest extra feature of ChessEvolvedOffline is the inclusion of extra custom pieces, each with unique abilities. These pieces were surprisingly straightforward to implement thanks to the flexible design of our Piece class hierarchy. By establishing a robust abstract base class for pieces, we created a framework where new piece types could be added without significant modifications to the existing codebase. Each custom piece, whether it was the Queen, Hippo, or Ross Evans, was implemented as a subclass of the Piece class and had all of their moves in a vector of possible moves. For example, let's look at how the Queen got all of its possible moves:

```
std::vector<PartialMove> Queen::getPossibleMoves() {
    std::vector<PartialMove> moves;
    pushBishopMoves(moves, 7, MoveType::MoveOrAttack);
    pushRookMoves(moves, 7, MoveType::MoveOrAttack);
    return moves;
}
```

PushBishopMoves took in two customizable parameters, range (1 to 7) and type (move, attack, teleport, etc.) and allowed the Queen to move like a Bishop. Similarly, pushRookMoves utilized these parameters to give the Queen her full movement set. By abstracting the movement logic in this way, we could easily extend or modify the behavior of each piece, ensuring that adding new custom pieces with unique movement patterns was both efficient and straightforward. We even included custom art for all of the pieces, which we worked very hard on.

Additionally, we implemented an undo feature to enhance gameplay flexibility and maintain the integrity of the game. The Board class maintains a history of moves, allowing players to revert to previous states. This history includes details of each move, such as the starting and ending positions of pieces and any captures or special moves like castling or en passant. When a player triggers an undo, the game retrieves the last move from the history stack and reverts the board state accordingly. This feature not only allows players to rethink their strategies and correct mistakes but is also used to revert the state if a player makes an illegal move.

Multiple QOL features have also been added to our game. For instance, the "select" command (and corresponding "deselect" command) highlight possible moves of the selected piece on both the text and graphical display in different ways, which is an extremely useful feature given all our custom pieces. We

also implemented a “default” command in setup mode, which automatically sets the default board state to make setting up faster in some positions.

Finally, we implemented a Level 5 AI, using the minimax algorithm with a-b pruning. Although this AI takes significantly longer to “think”, it is much more competent than the other levels of computers.

Final Questions

---

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

We learned that first dividing tasks clearly among team members allowed each person to focus on their responsibilities, minimizing merge conflicts and streamlining our workflow. We also learned that having team members explain their parts to the group, instead of everyone reading over everything, considerably increased how fast we understood the code.

A less-applicable lesson is also to pick roommates as teammates for easier communication

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

If starting over, we would begin this project earlier. Unfortunately, due to our other courses, this has been quite the challenge this term. Distributing the tasks over a longer period of time and writing rigorous, reusable tests instead of making random chess moves would have greatly improved our workflow. Additionally, we would have allocated more time to explore our computer’s playing capabilities and maybe test how far it could climb on the chess.com elo ladder.

Conclusion

Dear Ross Evans, if you are reading this, please do not sue us for using your name on a chess piece. It was all in good fun, and you have been honored by having your piece be the strongest piece to the game.

It's not every day that a name brings such power to a chessboard, and we hope you appreciate the playful tribute. Just like you’ve been a formidable force in teaching and guiding, your namesake piece dominates the board with unmatched prowess. Thank you for unwittingly being a part of our game and careers!