CS 247 Project: Chess Game Planning Report

By Shreyas Peddi and Vidhi Ruparel

Plan of Attack

Overview

The project involves developing a chess game with features including standard chess rules, a computer player with four levels of difficulty, score tracking, and both text and graphical board displays. We will follow software engineering design principles, including patterns like Observer, Strategy, and Template Method, as well as SOLID principles. The game is also intentionally designed to be extendable for additional features like 4-player chess, move undoing, and advanced computer levels.

Project Breakdown

Our project is broken down in the chronological order for completion of tasks:

1. Initial setup
    a. Tasks involved:
        i. Create blueprints for basic classes (Piece, Board, Player, ChessGame, Color enum, Move), meaning create the header files with class and method declarations but leave the implementation .cc files empty for now.
        ii. Ensure that we are using forward declaration instead of #include wherever possible to make use of separate compilation. Note that we do this every time we create any new file.
    b. Responsibility: Vidhi (just because I am excited to write the first class!)
    c. Estimated completion date: July 21 (we choose to skip July 20 since both group members have a midterm completion on July 20)

2. The Game of Chess piece rules
    a. Tasks involved:
        i. Implement specialized Piece classes (Pawn, Rook, Knight, Bishop, Queen, King) with basic rules of the game of chess, namely the movement logic of each piece.
        ii. Note that the UML diagram follows the Template Method pattern quite strictly, so ensure that each class inheriting from Piece implements a private virtual method to get moves for that piece.
        iii. Ensure that a Piece object should have a non-owning pointer or reference to the board so we do not make duplicate boards.
    b. Responsibility:
        i. Shreyas: Implement Pawn, Rook, Knight classes.
        ii. Vidhi: Implement Bishop, Queen, King classes.
    c. Estimated completion date: July 21

3. Board setup methods
    a. Tasks involved:
        i. Develop board methods for adding and removing pieces, resetting board, checking game state, specifically for SimpleBoard.

        ii. Also, create board methods to determine the winner and check if the game is over by implementing logic to determine checks and checkmate.
    b. Responsibility: Shreyas
    c. Estimated completion date: July 21

4. Recommended Setup mode
    a. Tasks involved:
        i. Since it is recommended to implement the setup mode as it might be useful for testing, we are going to follow it. Create a basic command interpreter (most likely main.cc) that would call ChessGame methods.
        ii. Test that the interpreter does not break down on misspelled words.
        iii. Implement the ChessGame class constructor and setup method.
        iv. Implement other ChessGame methods for setting turns, moving a piece on each turn and accepting resignation. This class helps in game flow.
    b. Responsibility: Shreyas
    c. Estimated completion date: July 22

5. Where are all the players?
    a. Tasks involved:
        i. Implement HumanPlayer and ComputerPlayer that inherit from Player.
        ii. Specifically, write the getMove method that would validate if the user's move is valid and move the piece if so. Follow Template Method Pattern.
        iii. Create a typical *Strategy* pattern setup to get 3 levels of move strategies for the ComputerPlayer.
        iv. Ensure that a Player object should have a non-owning pointer or reference to the board so we do not make duplicate boards.
    b. Responsibility: Vidhi
    c. Estimated completion date: July 22

6. Observers for Displaying Board
    a. Tasks involved:
        i. Implement a typical *Observer* pattern setup with abstract Subject and Observer, and concrete TextObserver and GraphicsObserver.
        ii. Test if the getState() method in SimpleBoard class works correctly and the board is displayed properly on both text and graphics window.
    b. Responsibility: Shreyas
    c. Estimated completion date: July 23

7. Scoreboard
    a. Tasks involved:
        i. Implement the scoring functionality by creating a ScoreBoard class. The SimpleScoreBoard should have an updateScores method that keeps track of player scores. For now, its responsibility is to print scores too since it is a board after all but we might remove it later. Integrate it with the ChessGame class.

b. Responsibility: Vidhi
c. Estimated completion date: July 23

8. Complex logic
    a. Tasks involved: Implement complex game logic including castling, en passant, pawn promotion, and the 4th level of strategy for ComputerPlayer.
    b. Responsibility:
        i. Vidhi: 4th level, en passant
        ii. Shreyas: castling, pawn promotion
    c. Estimated completion date: July 25

9. Extra features! Yes, extra!
    a. Tasks involved:
        i. Plan additional features like move undoing, opening moves file, chess tutorial, or move recommendations.
        ii. We have decided to go for an implementation without explicitly managing memory and move undoing. However, note that other features we want to implement are still TBD.
    b. Responsibility: The task division will be based on the features we end up choosing based on time, but we will be dividing the work equally amongst us.
    c. Estimated completion date: July 27

10. Report writing for due date 3
    a. Tasks involved:
        i. Write the design document, update UML and answers, describe features and resiliency to change, as well as answer final questions.
        ii. Note that we have kept our design resilient (for instance, a board can be any size and not 8x8 based on our UML) to minimize work for due date 3.
    b. Responsibility: Both will divide it. We choose to not micro plan this.
    c. Estimated completion date: July 28 for most of the report (and we leave July 29 for finishing touches)

Answers to posed questions in the project specification

Question 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.

The first step in creating the book of standard opening move sequences is to compile the list of popular openings in a file (from external sources like Kaggle datasets, github, chess.com resources, etc.). This file can be parsed using the FileStream classes and then be represented in a const Tree Data structure owned

by the ChessBoardApplication. Each node in the tree represents the state of the game and each edge represents the possible opening given the current state of the game. The leaf nodes can store the name of the opening. Each path from root to leaf represents a unique opening. Along with the tree structure, we would create a const dictionary which maps the name of the opening to the opening moves in a string separated by commas. Observe that this mimics a database.

To implement a book of standard opening move sequences, we would introduce a ChessBoardApplication class that serves as an abstraction layer for additional features on top of the core game (ChessGame class). This not only allows us to add the book of opening moves, but also other features that are useful for a full-fledged Chess application like - Online Games, Tutorials, User Management, Puzzles, etc. The Tutorial section could include an option to learn opening moves. We would add a command line option to explore the book of standard move sequences (can only be done when a game is not running). A list of all the popular opening names will be displayed on the screen and the user can select the opening that they are interested in viewing. When an opening is selected, we would then create an Opening class which has a non-owning pointer of the Board. This Opening class' responsibility would be to render the Board class x amount of times, where x is the number of moves in the opening. For each move in an opening, the board would be updated such that it represents the move. Since we already have the TextObserver and the GraphicsObserver classes, the render method will display the opening sequence on both the interfaces.

Additionally, we would also incorporate opening moves into the Level 4 strategy of the ComputerPlayer. The algorithm for coming up with a move will include a lookup in the tree to determine if there is any opening sequence given the state of the game. As we stored the openings in a tree, we can simply advance to the child node state if there is an edge for the current move in the tree. If there is no edge in the tree, we stop looking for opening moves in the tree.

Question 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 our current design, we add a stack data structure (stack<HistoryItem>) in the Board class to store a history of moves and a list of captured pieces (vector<Piece>). Each HistoryItem contains the move, a boolean indicating if a piece was captured, a pointer to the piece that was removed from the board, and special conditions (such as en passant, castling, and pawn promotion) stored as a string. This information is sufficient to revert any number of changes. When a piece is removed, we use move semantics to push the piece into the list of captured pieces, ensuring the memory associated with the pieces is not deleted when removed from the board since we will use unique_ptrs. The pieces will be freed when the current game is over.

To undo a capture, we simply use the pointer to the removed piece in the HistoryItem object and add it back to the board. This handles the case of undoing en passant as well since we simply need to revert the move and add the captured pawn back to its position. To undo castling, we need to know if the move was performed by White or Black (available in the piece pointer) and revert the change as there is only one possible location for castling. To undo pawn promotion, we follow the same steps as undoing a captured piece, except it will be for the same color. Additionally, we can maintain an undo count in the Board class to limit the number of undos allowed per turn or per game.

HistoryItem (new class):

- move: Move
- isCaptured: bool
- PieceRemoved: Piece
- specialCond: string

Board (added fields):

- stack<History> movesHistory
- vector<Piece> deletedPieces

We will add a new prompt in the command interpreter called undo. It will be the board's responsibility to perform the undo through its undo and addPiece methods. Note that our decision to have a Move class since the start simplifies this task.

Question 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 adapt our chess program for four-handed chess, we will create a new class called FourHandedBoard, which extends the abstract Board class, representing the four-handed chess variant. This class will initialize a field vector<vector<Piece>> with 160 squares to accommodate the larger board. The isGameDone method in FourHandedBoard will have a specialized implementation to account for the new win and draw conditions. For a win, the method will determine the winner by checking if three players are in checkmate. For a draw, it will include additional cases such as threefold repetition, insufficient material, and the 50-move rule. We do not need to change classes such as Observers, Piece and Player since they are associated with the abstract Board class instead of any concrete class. This follows the Open/Closed Principle.

In the ChessGame class, we will have a new constructor, to initialize 4 players in the player list (vector<Player>). We will add 2 more colours to the Colour enum, enabling turn-taking among all four participants. Furthermore, all of our subclass implementations of the Strategy class support n-player scenarios so no changes are required to modify existing strategies. For any new strategies that are needed specifically for four-handed chess such as alliances, we would simply add another subclass in the Strategy inheritance hierarchy. We might consider implementing the Decorator pattern or the like to avoid creating too many children of Strategy. These changes will ensure a smooth transition from traditional chess to four-handed chess, maintaining the core functionalities while accommodating the additional complexities of the new variant.

Conclusion

This project plan outlines the step-by-step approach for developing a chess game that adheres to software engineering principles. By breaking down the tasks, we aim to complete the project efficiently and effectively, ensuring high cohesion and low coupling throughout the development process. Hope you like our efforts and project!