CHESS

- Cs 246 final project
- Spring 2025

STUDENT NUMBERS: -

- 21155514
- 21121922

Introduction

Both Varnit Sahu and Krithika Kannan are huge fans of chess and when they heard that we had the chance to make a chess game, we jumped right into in. with one of the first teams to finish this project, we loved the entire journey and produced a full-fledged working chess game with proper enhancements working on the project was fun (except the X11 features :/) and was a really fun experience to build something in common together that was so dear to us while trying to incorporate object-oriented programming.

From the empty folder, we worked together fixing bugs, creating classes, having lunch, trying to debug mermaid code and BRAINSTORMING X11 fixes.

This project felt like a grandmaster-level match between code and creativity.

This project was a stage to display our OOP skills along with our chess enthusiasm and we did exactly that. Trying to make the computer [1-3] levels felt like training a little kid how to play chess. Having our AHA moments when we realized we could simplify it so much using inheritance and design patterns was fun. We still remember our first brainstorming session where we aimed for the stars with bonus features.

We are so happy to conclude that we have infact completed the project and added sufficient additional features as per our satisfaction. Utilizing 0 memory management was our biggest goal and we successfully implemented that.

Despite the occasional frustration (and hours lost tweaking X11placements), working on this project together was a journey we'll never forget. Late-night coding sessions turned into impromptu chess matches; whiteboard strategy plans felt like tournament prep.

We hope you are pleased with our work and grade us accordingly.

Preface

This project extensively displays our academic progress within the CS246 Course with a huge focus on object-oriented software development. Both the members of this group aim to display excellent OOP principles and showcase their reediness in a software development team.

The project built entirely in C++ will be an excellent journey for the both of us. We chose chess as the game to develop as it requires keen understanding of the game. Both Varnit and Krithika have been playing chess approximately daily for the past 5 years and love the game passionately.

Rated 1550 and 800 respectively on chess.com we were deeply exited to draft the project once we got to know. Knowing the game inside out, we knew all the edge cases, enhancements and game to play.

This game is aiming to target the chess audience who play the game daily, adding a bunch of customization, enhancements and difference view patterns, we believe we can succeed in building the project on time and score a high grade on the project.

We showcase excellent Design patterns like integrating nearly all five major design patterns taught in the course while maintaining exceptional code quality and memory safety. We utilize the factory method within the piece Factory to produce different pieces needed for the game.

Utilize the template patters by using a piece abstract class and utilize the observer patterns with the game State respectively notifying the text display and graphics Display. We achieve zero delete statements and no manual memory management. We also have no memory leaks within the code (tested with valgrind)

Overview

We implement a layered, component-based architecture that cleanly separates the different responsibilities of the system

The core game runs within the main.cc file. See the UML to better understand the game logic but we start off by parsing all the command line functions from the argv[] c style array. We then understand the specification the user wants to run. We then load the correct variables.

We made the Chess Game class the central orchestrator of my entire application. This class serves as the main entry point and coordinates all other components. The Chess Game class handles starting and stopping games, processing I/O commands, managing player turns, validating input, setting up the environment to load a new game using the setup mode. Placing the pieces, removing the pieces, tracking game scores, storing the variables etc.

We also have player management as the template methods abstraction to handle when to use which player logic. When to use the computers and when to use the human player subclass.

The Board class is the Subject for our Observer pattern are representing the board. We took inspiration from the reversi game we were asked to code for A4, it owns an 8x8 board using unique pointers, handles game sate like checks, checkmate, drawing, stalemate etc. it also has the complete move history. It handles all the move validation and execution logic. It also generates all the legal moves which can be used by the computer later for the different move generations.

We beautifully use the Piece class as our abstract class for all the different piece types in the game. Using inheritance, each concrete piece class (King, Queen, Rook, Bishop, Knight, Pawn) encapsulates its own movement rules. I used the Template Method pattern in the base Piece class to ensure consistent

move validation while allowing each piece type to define its specific movement patterns.

The human player class processes user input from the command line and validates move commands while the computer player Implements algorithms with three difficulty levels (random moves, prefer captures/checks etc as per the assignment).

We also use Observer pattern for my displays as we have a common display getting notified on each change. We implement both text and graphical display for the game to run properly and beautifully. We have amazing UI enhancements and have spent a lot of time for the UI of this project.

The Piece Factory class that we made Implements Factory Method pattern for consistent piece creation from types or character symbols.

We also have a types.h file with a bunch of small Enums/classes for the core game to work like the position structure, piece types, the state of the game, a structure to showcase a move.

We made the decision to use RAII principles throughout my entire system with smart pointers for automatic memory management. This eliminates the need for manual memory management and prevents memory leaks. We have no explicit delete statements anywhere in our code.

To give a small summary, ChessGame receives and validates user commands through the UI, It translate valid commands into actions on the Board,
The Board collaborates with Piece objects to validate and execute moves,
My Board automatically notifies all registered Observer displays whenever the game state changes, the display components render the updated game state independently

Design

We utilized a very straightforward gameflow and introduced a bunch of design principles within the game.

To address the issue of a lot of small changes that can take place, we implemented the Template Method pattern in our ChessGame class. This pattern allowed us to define a stable algorithm for game execution while keeping the specific implementation details of each step separate and modifiable. The game itself follows this consistent pattern, process player input, update game state, check for game-ending conditions, and repeat. This approach gave us the flexibility to modify individual components of the game flow without disrupting the overall structure.

We implemented the Observer pattern to ensure our multiple display components stay synchronized with game state changes. Our Board class serves as the Subject, maintaining a vector of Observer* pointers and providing methods like attach() and detach(). When game changes occur, the board calls specific notification methods, which iterate through all registered observers and call their corresponding event handler methods.

Both our TextDisplay and GraphicalDisplay classes inherit from the Observer interface and implement methods to update their respective presentations. This design allows us to add new display types in the future without modifying any game logic, ensures that all displays receive updates simultaneously and consistently.

We implemented the Factory Method pattern through our PieceFactory class. We use createPiece to create pieces whenever required. This ensures consistent piece initialization. The factory has all the logic for mapping between different representations of pieces and their actual object types. This design makes our system highly extensible, adding new piece types or supporting different chess variants would only require modifications to the factory class rather than changes throughout the entire codebase.

We also used a bunch of class hierarchies and inheritance, polymorphism, abstraction and encapsulation principles as seen from the UML.

Resilience to Change

Our design supports changes through use of design. New piece types only require extending the Piece hierarchy and updating PieceFactory, while new display modes can be added by implementing the Observer interface without modifying game logic. Additional computer difficulty levels or algorithms simply extend our ComputerPlayer strategies, and rule variations like Chess960 or custom board sizes can be accommodated by extending the Board class. This modular approach ensures most changes impact specific components rather than refactoring the entire codebase.

Questions

Question 1)

If we were asked to implement a standard book of opening in a purely OOP principle, we could do it in several ways but the way we both collectively chose was to make use of a combination of design patterns along with a tree data structure. Even though the order of the moves doesn't matter to a high extent in chess, the opening should be played in a specific order to maximize advantage.

Famous openings like the Caro Kann defense and the Reti opening are highly depended on the order of the move. We traverse down the tree, with each node representing a possible next game state, rate it through an evaluating engine like stockfish or a self-written algorithm and create an STL map with 2 elements, (next state and move to reach that state) and use a mix of design patterns like the iterator pattern to quickly printout every piece position in near constant time.

We would also implement the observer pattern for opening analysis with both the graphical Display and the Textual Display. Template Method Pattern for Opening Evaluation is also a good implementation choice that we collectively decided to use as each next move could be another possible board State for an Abstract Board State. We would try our best to reuse code from our original core game as we have already set up the Model in the MVC format and just need to swap the controller for the core board logic to implement this in a strictly observability mode. For the historic win/draw/loss percentages we would try to use an external API if allowed, else just code everything in a 2d array or STLmap.

Question 2)

From a complete OOP point of view, I would use the iterator Pattern for Move History Navigation as it would allow my iterator to go back as well and stop at the right state. We would create a vector for dynamic memory management to save the moves thru which we would iterate in near constant time. Allowing the player to undo would require a little refactor to game principles like leaving a check, (change the time control if we are successful in implementing that) and then just hardcode a function to use the previous move in the undo function.

This would obviously mess up the evaluation algorithm for the computers hence we would notify them as well. It's obvious a player would only like to undo if their evaluation goes down hence this sudden spike is bound to mess up the current evaluation algorithm. We would try to utilize the observer pattern to implement the notify function to notify different microservices all at once like displays, history, eval algo, for unlimited undo's we would have no counter but have a check to prevent the player to undo before the default start position. We would need to make use of the empty piece/ black tile to replace the piece if it isn't captured yet

Question 3)

Since we are already done with the core game setup and coding how the pieces move, where they can go, what's a check, what's a checkmate, draw conditions, movement patterns, legal moves, illegal moves, I think we could reuse this logic so many times. All we would need to do is create another board for each variation. And use the factory method pattern to choose which board we want to run. We can create an abstract Board class and have different versions of it for 2player, 4player etc. all we would need to do it just select the right one during runtime. We could also set up more colors and change the player logic to handle more than one colors.

We could also set up and change the display logic in both the textual and graphical displays. We would need to update the game state logic as that will be our major brain for the project and choose the 4-player game initialize method. We would need to tweak and/or create another setup mode with more valid board logic and team choice if they want to be within teams or free for fall with detailed information regarding that. We would also need to update player management logic within Game State, tweak a little bit of the observer logic in both the displays, change the algorithm logic to evaluate next best moves etc. we need to update game ending and draw logic if all pieces have been traded and 4 kings left. We also need to update the move validation.

Extra Credit Features

We implemented a bunch of extra credit features such as removing all delete statements and removing any manual memory management (tested with valgrind) we achieved Zero Memory Management with just smart pointers. We also can export / import games with a specific FEN notation which can be used to analyze the current game and understand what to be done next. This implementation was completely algorithmic and did not take in any design principles.

We also added a bunch of decorator enhancements within the text display and graphical display too. Adding several colour options (green, brown) to mimic chess.com. we also spent a lot of time for the textdisplay UI.

Final Questions

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

   This project taught us that clear communication and well-defined interfaces between components are essential when multiple people work on the same codebase. Most importantly, we discovered that collaborative debugging and working together, pulling all-nighters is not all that bad and can be made into something fun. The project emphasized the value of regular check-ins, realistic milestone planning and making sure we give A LOT of time for last minute changes.

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

   MORE TESTING! We spent a lot of time trying to merge codes together, planning how everything should interact and a lot of time building the logic that we were not able to test every small class on its own. We regret this and hope to finish it even earlier for proper testing.

CONCLUSION

After the thousands of hours spent interpreting bugs, infinite arguments over pawn movement, and indecision as to whether our knight class was too powerful (it wasn't!), we have finally developed our take on the classic game of chess.

From a text positioning display to a functioning graphical interface, it is coming together bit by bit - literally. Not all of the functionality made it in seamlessly (I see you, en passant), but we are ultimately proud of what we made.

We started with a board and empty classes, and we ended with a game that allows humans and increasingly intelligent computers to compete against each other.The challenge of implementing different levels of difficulty for the computer players simply meant we had to teach a baby to play chess, with the baby either only knowing how to blindly attack (level 1) or run away scared (level 3)! But hey, it works, and that's what matters

There were times when we were feeling trapped in infinite check, but we found one move or another. Whether fixing display bugs, debugging command input, or unintentionally sending a pawn flying across the board, everything taught us something.

And we are better experts on chess rules than we ever thought we would be, including realizing why castling is secretly terrifying to implement!

All in all, this project had its fair mix of brain-numbing logic, rewarding moments and just enough chaos to keep it fun.

We are hoping our code doesn't crash halfway through the demo. We are hoping our kings aren't in illegal positions. And lastly, we hope the TA also grades us favourably.