CS 247 Project: Chess Game Final Report

By Shreyas Peddi and Vidhi Ruparel

Introduction

The project involves development of a chess game with features including standard chess rules, a computer player with five levels of difficulty, score tracking, and both text and graphical board displays. We followed software engineering design principles, including design patterns like Observer, Strategy, Template Method, Iterator, Chain of Responsibility, Builder, Command, State and Mediator along with SOLID principles. The game is also intentionally designed to be extendable for additional features like 4-player chess, move undoing, and advanced computer levels.

To ensure the best reading experience of this report, we recommend reading the Design portion of the document before seeing the UML. The overview section is just a highlight, but the Design section contains a detailed breakdown of OOP principles and design patterns implemented per class. It also includes a thorough discussion of cohesion versus coupling inspired from Shreyas' and mine (Vidhi's) heated debates! Reading this would definitely give a more clear insight into our project and efforts.

Overview

Our project is structured in a way that balances coupling and cohesion. In order to ensure the SOLID principle of Single Responsibility, we divided the major elements of the chess game. In order from high level classes that user interacts with to the lower level implementation classes, we have the following organization:

1. ChessApplication: This contains the command interpreter, much like the test harness provided during assignments. This is where our main method is located. Based on user input, it calls the appropriate ChessGame method.

2. ChessGame: This class mediates all elements of a chess game. It contains a board, scoreboard and players. Acting like a controller, it connects the board model to the text and graphics observers views. From here, the board's movePiece or player's playTurn methods are called. It keeps track of a turn and passes the turn to the next player. After every turn, it checks if the game has ended by checkmate or stalemate and notifies the observer.

3. GameState: This class helps maintain states of the game: SETUP, IN_PROGRESS, WHITE_IN_CHECK, BLACK_WINS, DRAW, etc. It helps ChessGame manage the workflow of the entire game. For instance, ChessGame does not allow setup mode to be activated if GameState is IN_PROGRESS.

4. Colour, ColourUtils: We represent players as colours since Black and White colours are generally represented as players. This proved to us as a very helpful design choice, and is discussed in detail in the design section.

5. Board, Square, Piece: A board contains a grid of Squares and each Square points to a Piece. Board's responsibility is to handle addPiece, removePiece and movePiece methods. Board also keeps track of all moves in a stack of historical moves. This helps in implementing (our of our extra features!) and also for detecting last move in en passant logic or checking if king and rook moved before in our castle logic. Since the logic of checkmate in a 4-handed chess board might be different from a normal board, the logic to detect checkmate is managed by that specific board. It is then easy to make Board abstract and a SimpleBoard with default Board implementation, and a ComplexBoard inherit from it in the future.

6. Pieces: Each of the pieces (King, Queen, Rook, Bishop, Knight, Pawn) have their own classes which inherit from Piece. Piece has a getPossibleMoves() method that finds all possible moves of that piece from its current location. We call this to see if the move made by the user is a possible move or not if it is returned by this method. It also aids the computer strategies to select a suitable move. Specific logic for pawn promotion and en passant is managed by Pawn class, while castling logic is managed by King class.

7. PieceFactory: Creates a new piece based on a given char.

8. Players: Players are only responsible to play their turn, that is they find a move, and give it to Board to execute it.

9. Strategies: Each of the 4 strategies use the random number generator from class. A computer player has-a strategy based on user input. While Strategy is an abstract class, the 4 strategies inherit from it to give customized implementations.

10. Moves: There are 4 types of moves, SimpleMove, EnPassant, Castle and PawnPromotion. Each of them have a different movePiece (and undoPiece) implementation.

11. ScoreBoards: Juicy discussion on this follows in the design section but they keep track of scores of each player. Interestingly, they do not know anything about the players (they only store a map of Colour to double).

12. Observers: Text and Graphics observers use Board as a subject to display the board. Board has a getState() method to get the piece at each square.

Design

Below is a per class breakdown of OOP principles and design choices used. Following these is a list of all design patterns we were inspired by and integrated in our classes.

1. ChessApplication: Initially, for due date 2, we only had a ChessGame class containing the command interpreter, chess board, players and so on. However, in order to follow the Open/Closed principle, we created this ChessApplication class. This class only has a ChessGame object. For instance, if we were to implement a ChessTutorial in the future, an umbrella ChessApplication would be more resilient to change. We would not have to modify ChessGame to have a ChessTutorial object, and it probably wouldn't even make sense to have a tutorial inside a game object.

2. ChessGame: This class, following the Mediator design pattern, mediates all elements of a chess game. It contains a board and scoreboard (unique_ptrs to be precise so it is automatically deleted), players (a vector of unique_ptr<Player> to be precise), and both text and graphics observers. This behaves like a controller connecting the Board model to the Observer views, inspired from MVC design. We also create a Chain of Responsibility style organization since the ChessApplication accesses ChessGame to respond to user input, and in turn the ChessGame accesses Board or Player to move pieces on board or play a turn respectively. For example, if the user says move e1 e2, ChessApplication calls ChessGame's move method, which in turn calls board's addPiece, removePiece or movePiece methods. This ensures cohesion between different elements of the entire application. ChessGame also controls and manages the game state. More on that in the next point.

3. GameState: We did not think of having a game state in due date 2, but it became very important as we started implementing our code. For instance, for performing basic input validation, such as the setup command should not be valid if game is in progress, or game human human should not be a valid command if game is not set up yet. There were many more such instances which called for the use of game states. Thus, we used the State design to manage game progress using states, which also allowed us to keep track of WHITE_IN_CHECK or BLACK WINS. While we did not implement an observer to display such prints on the graphics window (we do on text), in the future, it would be very easy to have an observer that takes the game state as a subject. In that case, we won't have some print statements lying around in the middle of our logic code in future.

4. Colour, ColourUtils: These are utility classes. We represent players as colours since Black and White colours are generally represented as players. However, having an enum class of Colour helps extend the game to more colours, which really means four players, allowing us to extend to 4 hand chess, or chess versions with more players. ColourUtils just converts Colour enum to a string so we don't have if statements in our code. This in our point of view was a good way to ensure the needful cohesion between players and colours. It also removed the hassle of assigning numbers to each player.

5. Board, Square, Piece: The board consists of Squares (unique_ptrs) and Square consists of Pieces (normal pointers). We had a heated debate about having Square own pieces versus Board own pieces. We thought that it is the board's responsibility to own pieces, so the board maintains a grid

(vector<vector>>) of Squares and a list (vector) of Pieces. Each square just points to the piece. Board is only responsible for adding, removing, and moving pieces. It also stores a history of moves in a stack. However, it is not responsible for playing a turn, that is deciding which move to make. In order to follow the Single Responsibility principle, playing turn is Player's responsibility. Board is also not responsible for displaying itself. We made the Board a subject and the Text and Graphics observers do the actual rendering. Moreover, the Square doesn't do anything except storing a piece pointer. It only has 1 method to say if it is empty or not. This ensures low coupling of board with the actual game play, allowing us to have perhaps an 10 x 10 board or a 4 hand chess board or even a hexagonal board. Note that there is needful cohesion between boards, squares and pieces though.

6. Pieces: There is not much to say but each of the pieces (King, Queen, Rook, Bishop, Knight, Pawn) have their own classes which inherit from Piece. We wanted to have a worthy discussion regarding the queen's moves. Although it is essentially a rook's moves plus bishop's moves, having the queen take multiple inheritance from both rook and bishop would lead to a Deadly Diamond of Death situation (since rook and bishop inherit commonly from Piece). Even if it meant some duplicate code, as recommended in Google's C++ style guide and IsoCPP, we decided against multiple inheritance and implemented the queen class from scratch.

7. PieceFactory: It was quite a challenge to create a new piece since the user provides input in chars but our board's grid stores Piece pointers. To efficiently convert a char to Piece, we created this builder class inspired from many design patterns like Factory and Builder design patterns. It is because of the PieceFactory that we ensure low coupling. Every time Board needs to add a Piece, it does not need to create Pieces by itself and interact with Piece objects; PieceFactory does it for Board.

8. Players: Since there are 2 types of Players, Human and Computer, but they behave very similarly, that is they play turn, move pieces, resign, etc, we extended them from the abstract Player superclass. Again, we follow NVI since the implementation details for playing a turn are quite customized to both Humans and Computers. For instance, in humans, we only validate if the user suggested move is valid or not. But, for computers, we have specific strategy implementations which behave in their own specific ways. Thus, we have a private virtual getMove() method in Player class overridden by Human and Computer with customized implementation. But, the public playTurn() method in Player follows some invariants for preconditions and postconditions. For example, after deciding on a move from getMove(), we must move that piece on the board, add it to the stack that stores moves history, get the metadata associated with move (for instance, a capture happened is a metadata), and store the history of each move's metadata too. All these common actions are handled by playTurn().

9. Strategies: We created 4 strategies following the Strategy design pattern, wherein the Computer has a Strategy and chooses the right one as needed. We used the random number generator method of using the system clock as provided to us. We created a StrategyUtils class to abstract the random number generation logic from the chess strategy logic, again ensuring Separation Of Concerns and ensuring low coupling.

10. Moves: Each move stores an old position (pair of ints representing row, col) and a new position. Moves are an integral part of chess. There are four main moves, normal move, en passant, pawn promotion, and castle. We design 4 classes for each since each one's move on the board is different from the other. We also introduced the undo command for the user, and the undo of each move is also different. This implementation style is inspired from the Command pattern. We created a common invoker called the MoveSimulator, so that each specific class (board for example) does not need to worry about calling a specific move. The move simulator decides based on the move and moveMetadata (which has additional details, for example, for a pawn promotion move, we store the originalPawn piece) which move to make or undo on the board. These are static methods in MoveSimulator. This ensures low coupling between Moves and Board, and Moves and Player.

11. ScoreBoards: Sky's the limit when it comes to scoreboards. Someone might want to give 1 point for winning a game, while another person might want to assign points totalling the value of the captured opponent pieces. So, we wanted to keep the board class open to extension yet closed to modification, following the Open/Closed principle. Thus, we have a ScoreBoard abstract class and SimpleScoreBoard concrete class which assigns points based on project requirements. If need be in the future, complex scoreboard concrete classes can be created. We had a map of Colours to scores, since Colours represent players. We were able to use the toString method of ColourUtils to display the player colour. Note that without abstraction of Colour and ColourUtils, we would need if statements to display colours. Let's say you extended 4 handed chess but forgot the 2 extra colours, ScoreBoard would break. Having ColourUtils gives a default UNKOWN_COLOUR so that the scoring does not break, and you can realize your mistake and update ColourUtil's toString() method to include the new colours. Notice how having Colour also removes coupling of Players with ScoreBoard entirely! Scoring works without knowing any information about Players :)

12. Observers: Similar to assignment 3, there are both Text and Graphics observers managed by ChessGame.

Apart from the above, it is noteworthy to see the design patterns we integrated and were inspired from:

- Iterator: In order to iterate over the Board class, we implemented a templated (const and non const) iterator. This removed the hassle of going over row and column every time we wanted to go through the pieces on our board. We thought this was a good idea when a for loop that we had was doing i++ (of outer variable) in the inner for loop, instead of j++!

- Observer: To display the board, we used the Board as subject and the Text and Graphics observers. In order to ensure that the graphics observer does not rerender the entire board after each move, we were inspired by a similar practice that React follows to load up specific parts of the page instead of the entire UI. We kept track of the previous state of each square on the board and if after a new move, it differed from its previous state, we only re-rendered that specific square. There was also a getState() method in the subject, which is Board, giving the character (piece) at a specific square, given its coordinates.

- Strategy: We created 4 strategies following the Strategy design pattern, wherein the Computer has a Strategy and chooses the right one as needed.

- Template Method: We followed NVI from class by giving our Piece a public getPossibleMoves method, that as its name suggests, gives the square that the piece can move to from its current square. However, since each piece's movement behavior is quite customized and specific to it, it calls for a classic case of following NVI and utilizing the Template Method pattern. A private virtual getPossibleMovesImpl method in each class gives its custom implementation. After that, getPossibleMoves has a postcondition, rather an invariant, that none of the possible moves are valid if it puts the king in check. Hence, such common behavior is taken care of in the general Piece getPossibleMoves method.

- Chain of Responsibility and Mediator: We also create a Chain of Responsibility style organization since the ChessApplication accesses ChessGame to respond to user input, and in turn the ChessGame accesses Board or Player to move pieces on board or play a turn respectively. This was discussed above in the detailed design of the ChessGame class. As discussed above, ChessGame is the mediator in this case.

- Builder/Factory: We created PieceFactory as discussed above.

- Command: This was used for undoing logic as discussed above with the Moves class.

- State: This was used by GameState as discussed above.

Updated UML and Difference From Due Date 2

Our updated UML is attached separately in uml.pdf. Noteworthy changes from due date 2 include:

1.) We created a ChessApplication class to decouple the ChessGame instance with the main method which has the public interface to interact with the application.
2.) We created the Square class to separate the logic of storing a piece in the grid from the Piece class itself.
3.) Iterator class inside Board to facilitate iterating over the squares in the Board.
4.) GameState class to keep track of the state of the game.
5.) We created AbstractMove, Move, EnPassant, Castle, and PawnPromotion classes to differentiate between different kinds of moves and to allow simulating and undoing a move dynamically.

Resilience to Change

We designed almost every class with resilience to change in mind. We followed the Open/Closed principle thoroughly throughout our design. These were discussed in the design section, but to highlight:

1. ChessApplication allows us to integrate other aspects of the chess game, such as Chess Tutorials, Chess Book of Opening Moves, Puzzles, and so on.

2. Since the logic of checkmate in a 4-handed chess board might be different from a normal board, the logic to detect checkmate is managed by Board class instead of ChessGame. It is then easy to make Board abstract and a SimpleBoard with default Board implementation, and a ComplexBoard inherit from it in the future.

3. Strategy class follows NVI so any new Strategy added will have a customized implementation but Strategy's invariants (that is getting a move which does not put your king in check) is maintained.

4. We have a ScoreBoard abstract class and SimpleScoreBoard concrete class which assigns points based on project requirements. If need be in the future, complex scoreboard concrete classes can be created that assign let's say 10 points instead of 1. We had a map of Colours to scores, since Colours represent players. We were able to use the toString method of ColourUtils to display the player colour. Note that without abstraction of Colour and ColourUtils, we would need if statements to display colours. Let's say you extended 4 handed chess but forgot the 2 extra colours, ScoreBoard would break. Having ColourUtils gives a default UNKOWN_COLOUR so that the scoring does not break, and you can realize your mistake and update ColourUtil's toString() method to include the new colours.

5. While we did not implement an observer to display such prints on the graphics window (we do on text), in the future, it would be very easy to have an observer that takes the GameState as a subject. It stores states such as BLACK_IN_CHECK or WHITE_WINS. In that case, we won't have some print statements lying around in the middle of our logic code in future, but a GameState observer will be responsible for displaying notifications.

6. We have a has-a relationship between Strategy and Computer, so in future, even HumanPlayer can use Strategy. In fact, a move recommender using Strategy 4 for HumanPlayer is also possible by just returning the move selected by Strategy 4.

7. We use vectors everywhere. For example, having vector<Player> in ChessGame allows app to be extendable to any number of players just by adding new Colours to Colour enum, and without updating ChessGame class.

8. Our board does not need a specific width or height. As long as ChessGame's constructor sets the width and height of the Board when initialized, we can even right now support a 10 x 10 or 100 x 100 board. We used iterators everywhere and did not say (while i < 8) while iterating for instance as well.

In general, not storing integers like 8 for 8 x 8 board or having an array of players helps us a lot. For more advanced features like complicated scoring, having abstracted classes and following the Template method, which allows for more customization is also beneficial even if there is no customization required now. Having a Stateful app, having enums and factory classes are also helpful.

Answers to Questions

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.

Our actual design for undoing is eerily similar to this! We just store 2 stacks, 1 for Moves and 1 for MoveMetadata (which is HistoryItem without move). One different thing we would say is that we would implement the Command pattern since the 4 move types (normal, pawn promotion, en passant and castle) have jarringly different logic.

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.

Extra Credit Features

1. Move undo feature: We added support to undo any move during the game, including special moves such as pawn promotion, en passant, castling, and capturing. Client can simply type “undo” during the game to undo the last move. Our undo feature works for both human and computer player moves, which required careful tracking of removed and restored pieces. For instance, undoing an en passant move involves tracking the captured pawn and its original location. Similarly, for a pawn promotion move, we revert the new piece back to a pawn. Numerous edge cases had to be addressed to ensure the feature's robustness. To manage this, we implemented two data structures in our Board class: a stack of metadata and a stack of move history. The move history stack stores both the old and new positions of pieces, allowing us to revert a piece to its original location. The metadata stack includes information about any captured pieces and the nature of special moves, identified by types such as Move, EnPassant, Castle, and PawnPromotion. Designing this feature was particularly challenging, but our approach enabled us to extend its use to simulating moves and undoing simulated moves, which is crucial for implementing our level 2-5 computer strategies.

2. We incorporated the famous Minimax algorithm along with Alpha-Beta pruning for our Level 5 strategy. Minimax is a backtracking algorithm that simulates moves for both players and evaluates the board after a certain depth to determine the quality of a move. Each node in the Minimax tree represents a game situation, with two players: the maximizer (the current player) and the minimizer (the opponent). The maximizer aims to maximize the total score, while the minimizer aims to minimize it. The total score is calculated by evaluating the board at the base case of the algorithm. To determine this score, we use a simple algorithm that sums the predetermined values of the maximizer's pieces. Alpha-Beta pruning optimizes the Minimax algorithm by reducing the number of nodes that need to be evaluated. The most challenging part of implementing this algorithm was making it effective at lower depths. Due to limited CPU resources, it wasn’t feasible to use high recursion depths. However, the algorithm performed poorly at the initial stages of the game when using lower depths, as every move evaluated to similar positions. To address this, we incorporated our Level 3 strategy during the initial stages of the game. This hybrid approach allowed us to develop a comprehensive computer strategy that performs well throughout the game.

3. Displaying possible moves: We introduced a feature to highlight possible moves for a piece during the game, aimed at making our chess application more accessible to beginners. This feature helps users learn and understand when and how to apply moves, including complex ones like en passant and castling, which have specific conditions. For instance, castling requires that the king is not in check, there are no pieces between the king and the rook, and none of the squares between are under attack.To use this feature, users type "help <location>" where <location> represents the piece's position. The most challenging aspect was designing the implementation. Initially, we considered modifying our getState() implementation, but this approach required tight coupling with the ChessGame and Piece classes, making it complicated and inflexible for future features. Instead, we introduced a new state variable to store the possible moves of the requested piece. This approach allowed us to continue using the Observer pattern, enabling the graphical observer to check other state variables for more information. For the TextObserver, highlighting squares did not make sense, as it needs to display moves in parallel with the pieces. The Observer pattern allowed us to omit this display option for the TextObserver. We also enforced input validation to ensure that a piece exists at the specified location. In the future, we plan to integrate a separate tutorial section and a book of opening moves to further help users improve their chess skills.
4. We completed the entire project without leaks and without explicitly managing our own memory. We managed all memory via smart pointers and vectors.

Final Questions

1.) What lessons did this project teach you about developing software in teams? If you worked alone, what lessons did you learn about writing large programs?

This project was very valuable in teaching us to apply object oriented design in making software, coordinating in a team, resolving challenges, and planning ahead for a better output. The requirements of the application forced us to think about how to write extensible code which allows for others to collaborate and add-on new functionality in the future. We realized the importance of writing code that follows design patterns and OOP principles, as a large program like a chess application has problems that have common solutions. For instance, when we wrote the code for the computer strategy class, we understood the importance of loose coupling as computer strategy would evolve as more levels are added in the future. Another important lesson we learned was about planning. When developing applications, we were used to getting started by coding on day 1. This usually led to bad design choices as a strong starting is needed to make a large-scale application. For this project, Due Date 2 required us to submit a UML and a plan of action which forced us to think about the big picture and plan about how classes interact with each other. In the end, this allowed us to create a more functional and extensible application which has a strong base. Lastly, we also learned about coordinating changes, delegating tasks amongst ourselves, and planning when to achieve it.

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

We are proud of our end product and what we have achieved in the last two weeks. That being said, there are some things which we would do differently. Firstly, we would emphasize on testing every feature that

we implement as we go. Testing early was important as we realized that we spent a lot of time debugging errors which could have been tested and resolved easily early on. We would develop a simple testing suite that would allow us to check the functionality of the program as we add new features and make changes. Secondly, we would have emphasized more on getting the base version of the application set up early. In the initial stage of the project, we started by developing functionality instead of making a skeleton version of the application. A working skeleton version of the application would have been nice as it would have facilitated early testing. As we started by developing features, it was difficult to debug the errors once we integrated the changes. Lastly, we would have regular code reviews through PRs on github. This would be beneficial in getting another perspective on introducing a change and would encourage team members to stay up to date on various parts of the application.

Conclusion

The development of our chess game was an opportunity for us to show off the knowledge gained in class. It was a comprehensive exercise in software engineering, design principles, and collaborative teamwork. We adhered to fundamental design patterns like Observer, Strategy, Template Method, and many others, ensuring our code was modular, maintainable, and extensible. By focusing on SOLID principles, we divided responsibilities clearly among classes, enhancing cohesion and reducing coupling.

Throughout this project, we encountered numerous challenges, from designing the correct architecture to handling complex chess rules and edge cases. Our heated debates and discussions led to thoughtful decisions that strengthened our design, such as using an abstract Board class for future extensions like 4-player chess or integrating utility classes for colors and piece factories. Our implementation of features like move undo, multiple difficulty levels for computer players, and displaying possible moves demonstrated our commitment to creating a fun and detail-oriented application.

Hope you liked our game and efforts!