CS 246, Spring 2024 **Chess Final Design** **Shivum Varshney**
Caroline Kierstead **Tanvir Gahunia**

Project Overview

During the development of our chess program, it was imperative to adhere to our pre-established plan, making necessary adjustments as required. The final product is a sophisticated chess engine capable of displaying graphics both in a dedicated window and on the console. It includes four levels of computer opponents, supports special moves such as en passant and castling, and accurately detects check, checkmate, and stalemate conditions. Additionally, the program features a scoreboard, an undo function, and the ability to configure the board in various ways.

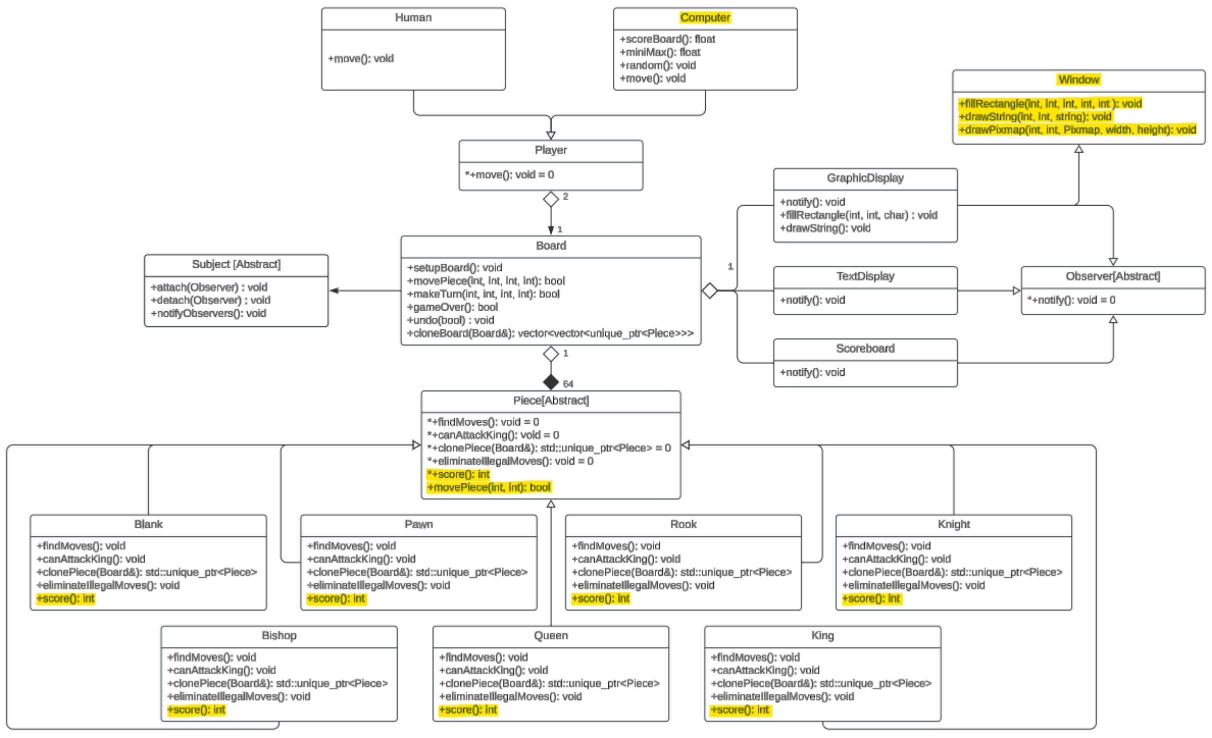
Human
+move(): void
Computer
+scoreBoard(): float
+miniMax(): float
+random(): void
+move(): void
Window
+fillRectangle(int, int, int, int, int): void
+drawString(int, int, string): void
+drawPixmap(int, int, Pixmap, width, height): void
Player
*+move(): void = 0
2
1
GraphicDisplay
+notify(): void
+fillRectangle(int, int, char) : void
+drawString(): void
Subject [Abstract]
+attach(Observer) : void
+detach(Observer) : void
+notifyObservers(): void
Board
+setupBoard(): void
+movePiece(int, int, int, int): bool
+makeTurn(int, int, int, int): bool
+gameOver(): bool
+undo(bool) : void
+cloneBoard(Board&): vector<vector<unique_ptr<Piece>>>
1
TextDisplay
+notify(): void
Observer[Abstract]
*+notify(): void = 0
Scoreboard
+notify(): void
1
64
Piece[Abstract]
*+findMoves(): void = 0
*+canAttackKing(): void = 0
*+clonePiece(Board&): std::unique_ptr<Piece> = 0
*+eliminateIllegalMoves(): void = 0
*+score(): int
+movePiece(int, int): bool
Blank
+findMoves(): void
+canAttackKing(): void
+clonePiece(Board&): std::unique_ptr<Piece>
+eliminateIllegalMoves(): void
+score(): int
Pawn
+findMoves(): void
+canAttackKing(): void
+clonePiece(Board&): std::unique_ptr<Piece>
+eliminateIllegalMoves(): void
+score(): int
Rook
+findMoves(): void
+canAttackKing(): void
+clonePiece(Board&): std::unique_ptr<Piece>
+eliminateIllegalMoves(): void
+score(): int
Knight
+findMoves(): void
+canAttackKing(): void
+clonePiece(Board&): std::unique_ptr<Piece>
+eliminateIllegalMoves(): void
+score(): int
Bishop
+findMoves(): void
+canAttackKing(): void
+clonePiece(Board&): std::unique_ptr<Piece>
+eliminateIllegalMoves(): void
+score(): int
Queen
+findMoves(): void
+canAttackKing(): void
+clonePiece(Board&): std::unique_ptr<Piece>
+eliminateIllegalMoves(): void
+score(): int
King
+findMoves(): void
+canAttackKing(): void
+clonePiece(Board&): std::unique_ptr<Piece>
+eliminateIllegalMoves(): void
+score(): int

Figure 1: UML Diagram with changes from DD1 highlighted

Board

We implemented a Board class to maintain all relevant data for the state of the game. The Board class encompasses features that support and enable various functionalities such as setup mode, move operations, and game-over detection. According to the single responsibility principle, the Board class is not explicitly responsible for managing the legality of moves or what is displayed on the console or graphical interface; it solely manages the state of the game.

The Board acts as a concrete subject with three concrete observers: the console, the graphical display, and a scoreboard. Both the console and graphical display utilize an iterator implemented for the Board object to traverse the current state of the board—a vector of vectors containing unique pointers to an abstract Piece class—and update their displays accordingly. At the start of the game and whenever a move is made, the Board iterates over itself, invoking a method for each piece to recalculate their legal moves. If any move can attack the opponent's king, it sets the relevant boolean in the Board class to true via a mutator, indicating a check has occurred.

The Board also provides a copy constructor, enabling pieces to test potential moves and ascertain if they lead to a check on their own king, thereby eliminating illegal moves. After each turn, the Board calculates these moves and detects the end of the game if the current player has no legal moves, subsequently informing its observers whether the result is a stalemate or a checkmate victory. If the game does not end, the Board indicates if a king is under attack by checking the values of the relevant boolean variables.

Special moves such as en passant and castling are verified for legality by the Piece class, but the Board is notified when a pawn moves diagonally to an empty square or when a king moves more than one square in a single move, allowing it to adjust the board and update additional pieces accordingly. Undo functionality is implemented by maintaining a stack of vectors of vectors of unique pointers to pieces, updated after every move. To undo a move, the top of this stack is assigned to the current board's state and then popped off the stack.

Setup mode allows user input to update the current board by overwriting the exact index in the vector of vectors with a new unique pointer to the piece specified by the user. To ensure the conditions for leaving setup mode are met, the Board iterates over itself to verify that nothing is violated (e.g., exactly one of each king, no immediate threats to a king, no pawns in the final ranks). Observers are notified whenever the board changes due to an undo, a move, or a change in setup mode. Further details on the implementation of pieces are provided below.

Pieces

We implemented an abstract base class, Piece, which encapsulates all relevant data for a chess piece. This includes information about its colour, the file and rank it occupies, a reference to the Board, a vector of legal moves, and methods to be overridden, such as those for populating the vector of legal moves. Subclasses representing each type of chess piece then implement these methods according to the rules of the game.

The three primary methods in the Piece class are for finding moves, eliminating illegal moves, and attacking the king. When the Board iterates through the pieces and calls the findMoves method, the inherited legal moves vector is populated with all potential moves a piece can make, including those that would place its own king in check. During this process, if a piece can move to a square occupied by the opponent's king, a check is declared through a mutator on the Board object.

After populating the legal moves vector, the eliminateIllegalMoves method is called. Each piece simulates its legal moves on a copy of the Board to check if the opponent can capture the current player's king. Moves that result in a check against the player's own king are deemed illegal and removed from the list of legal moves. This process is exemplified in bishop.cc, with similar implementations for other pieces.

Each subclass of Piece also includes a score method that indicates the piece's value at the current state of the game, essential for implementing computer players. The scoring values are as follows: pawns are worth 10 points, bishops and knights 30 points, rooks 50 points, queens 90 points, and the king 10,000 points. Additionally, each piece has an 8x8 integer array serving as a heatmap, indicating its value based on its board position. The piece's score is adjusted according to this heatmap, negated for black pieces and left positive for white pieces.

When a human player makes a move through input, we first check if the move they made is valid by checking if the piece at the indicated file and rank has a valid move to the new indicated file and rank (simple search against the pieces' vector of legal moves). If the move is legal, redirect this command to the board object to update the board. The computer class makes use of the list of legal moves to scan through its options and try to determine the best move according to its level; they will attempt to minimise or maximise score depending on their colour. More on how players were implemented below.

Players

We implemented an abstract base class, Player, which contains a reference to the Board object that hosts the main game (shared by both players) and a single method, move. The implementation of this method varies significantly depending on whether the child class represents a human or a computer player. This abstraction is crucial because it allows our game handler in main to interact with players without differentiating between types, simply calling on either player one or player two to make a move.

For a human player, the move method prompts the console for input, allowing the user to provide a move, signal to undo, or resign. The validity of the move is verified using the methods described in the Piece class. Once a move is confirmed as valid, it is passed to the Board object to update the game state. The undo function calls the Board's method, while the resign signal ends the game via a boolean variable mutator from the Board object.

Alternatively, if the player is a computer, the move method either calls a random move method or the minimax method. The random move method does what one might think it would, it shuffles the list of pieces it has with legal moves, picks one, shuffles its list of legal moves, and picks one to move on the board. If the minimax method is called, it is called with a depth of one less than its level. That is, a level two player searches with a depth of one, a level three player searches with a depth of two, and level four searches with a depth of three. The depth of this search is significant to the call to minimax, it indicates how many moves ahead it should consider before deciding what is the best move.

The computer player will create a copy of the board and play out all of the legal moves until it hits the assigned depth, it will then score the board by iterating over the copy and tallying the score via the methods mentioned in the piece above and the state of the board (if the opponent is in check, the score increases or decreases dramatically to encourage this type of play). If the computer is white, it will make the move that leads to the highest score, and if it is black it will make the move that leads to the lowest score (which is a common scoring scheme seen on popular sites such as chess.com).

As a result, since the level two computer has a depth of one, it will not consider what the opponent can do to retaliate to its moves, it will be aggressive without thought and capture and check whenever possible. The level three computer is a little more thoughtful, it considers what the opponent can do to retaliate and is a bit more passive. It checks and captures only when it knows that it will still be winning. The level four computer is the most advanced, it thinks one moves ahead of level three and is willing to sacrifice pieces for better position.

However, the higher depth comes at a cost, it takes a long time for the level four player to decide its best move to play. There are nearly nine-thousand different ways to play the first three moves in a game of chess, compared to only four-hundred that computer level three has to evaluate. As a result, the level four computer is significantly slower than level three, especially as the game progresses. After some research on how we can improve this run time, we implemented a concept such as alpha beta pruning into our algorithm and move ordering. Alpha beta pruning is quite an advanced concept, it requires passing the scoreboard up the recursive call to eliminate search branches that are significantly worse than already found moves. We also ordered the list of pieces and each move in order of value, and captures, that is the most valuable pieces, and moves that lead to captures are evaluated before any other move. Despite these adjustments, computer level four did improve but not by much; it can take upwards of a minute thirty to decide the best move. You can see computer.cc's minimax method to get a better understanding of the algorithm and what was implemented.

Displays and Main

The main function initialises a Board object, a TextDisplay object, a GraphicDisplay object, and declares two Player objects to be set later based on user input. It then enters a loop that continues until the primary input stream reaches the end of the file. The available commands at this point are setup, game, and q for quit.

The setup command calls the Board's setup method, allowing users to modify the board to their liking. The game command awaits two additional string inputs from the user, indicating the types of players for the game. It then invokes a game handler function, which calls the move methods of player one and player two alternately until the game ends. Upon reaching the end of the game, the user can provide a continue command to return to the main function.

The q command quits the program, consequently invoking the destructors of all objects declared in main, including the Board, displays, and scoreboard.

The Scoreboard updates whenever the game concludes, checking for indications of resignation or checkmate via boolean variables in the Board class. If the game ends in a condition other than resignation or checkmate, it is considered a stalemate, and the

Scoreboard does not update. The Scoreboard's destructor is designed to print the results of the games played to the console.

The TextDisplay object prints the state of the board to the primary output console every time the board state changes. Similarly, the GraphicDisplay object updates the window display for the game. To avoid excessive rerendering, the GraphicDisplay maintains an 8x8 array of its own chars. Each time its notify method is called, it compares its own version of the board with the current board and rerenders only the differences.

Design

We made use of two design patterns for this project, the iterator design pattern and the observer design pattern. The iterator design pattern was implemented on the board class to allow clients to loop over the contents of the state of the board and update and call methods as they see appropriate. The board class uses its own iterator to loop over all pieces it holds and call their find moves method whenever the turn switches. The computer class also uses the board's iterator to recalculate legal moves for all pieces on the board whenever it is testing and searching for the best available move. It also uses it to quickly score the value of all pieces on the board. You can see this being done in the computer.cc implementation file, specifically under the scoreboard method. The observer pattern also made it easy for us to update various displays and classes that need to know when crucial information updates or is made available. Whenever the state of the board changes, such as a move being made, our display observers rerender or print their relative displays as necessary. Whenever the game ends, the scoreboard method is called so it can choose to update the score when necessary. This design pattern made is easy for us to follow the single responsibility principle, as the board class only had to act as a container for all the information about the game, it did not have to deal with reprinting or rerendering the displays or updating the score directly; rather we could create different classes for that task.

Resilience to Change

Our design facilitates easy updates to the program's behaviour and the addition of new features. For instance, we can introduce a new type of piece by declaring it as a subclass of the Piece class and defining its methods in a manner consistent with existing pieces. This modular approach ensures that other components, such as the Computer class, check mechanisms, and end-of-game detections, remain unaffected, demonstrating the low coupling of our modules. This is also a result of following the single responsibility principle, the piece class only needs to define its rules for movement; the remaining code is sufficient for implementing the piece into the game. We can also add higher levels of computer players by implementing new algorithms for move decisions and calling them appropriately from the overridden move method; this is a testament to our modules having high cohesion (everything in a module is closely related to the purpose of the module). For instance the computer class only contains methods that are relevant to algorithms for finding moves. Our code is flexible to change, we can easily add another display, modify the way the display looks, modify the game rules through the board class and make other changes with minimal changes to the code of our program.

Answers to Questions in DD1:

Our undo method was implemented exactly as planned. The only additional information to note is that after every undo we recalculate the legal moves each piece can make, which in turn restores the state of the board back to what it was at that moment in the game. No other revisions are needed to our answers from DD1.

Extra Credit Features

We created our chess program without explicitly managing our own memory. That is, there is not a single delete request in our entire project. All heap memory was managed via unique pointers, and raw pointers were passed around as needed. Our board object holds an eight by eight vector of vectors to unique pointers, and our iterator returns a raw pointer to the unique pointer it is on. Additionally, we had implemented an undo method the exact way it was described to be handled by DD1. That is, the board pushes a deep copy of the state of the current board (a vector of vectors to unique pointers to pieces) before applying a move to a stack called previous moves (a stack of vectors of vectors to unique pointers to pieces). We undo a move by popping the last element of the vector and setting it to our current vector of vectors to unique pointers to pieces. We then recalculate all legal moves to restore the state of the board to its prior state, such as a check.

Questions:

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 was our first time working on a medium sized project as a team and our first experience using version control systems such as git to manage the progression of our project. It was definitely a learning experience; a lot went wrong. There were issues with git that we did not understand and unsynced code leading to miscommunication errors and lost progress. As a result, we had to resort to setting times aside in the day where each of us would work on the code at a time. We later learned about using branches from a friend, but at that point the project was nearly complete. However, looking back at the project and the challenges we had to overcome, it was definitely useful to have a partner to split the work with and bounce ideas off of. Furthermore, having a fresh set of eyes on buggy code helped to debug our code and made the debugging process smoother. This project would have been a lot more difficult given the time constraint if we were forced to work individually.

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

If we could start over, we would make use of branch functionality from git so that we can actually work on the project at the same time leading to more progress. We would also look a little more into how to optimise the computer player's runtime. The algorithm we did use is suitable for smaller games like tic-tac-toe or connect-four where the number of moves either player can do is relatively small. However, for a game like chess there is an exponential blowup in the number of calls to functions and our move search tree. While we were able to optimise this search with numerous enhancements; it is likely a better implementation choice to use a different algorithm that is more suitable for chess rather than try to force something that is not optimised. Additionally, we would look into using a different library to implement our graphics. The X11 library, while easy to use, does not seem to support the transparent pieces we were using in our program forcing us to have two copies of each type of piece with a different coloured background to match the alternating squares. As a result, our initial load time was made significantly slower when using graphics.